//////////////////////////////////////////////////////////////////////////////////

// Company: Compute\_seniorDesignProject\_ee490

// Engineer: Gaston Regazzo, Alvaro Valera, and Richard Proppe

//

// Create Date: 9:06:46 11/21/2017

// Design Name: final draft, 12/12/2017

// Module Name: Top Level before Digital to Analog Converter

// Project Name: MIDI Sound Module Piano Synthesizer

// Target Devices: NEXYS4 DDR

// Tool versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module test(input SYSTEM\_CLOCK,

input IN, // UART Receive (1N4)

// input IN\_PI,

input RST,

input wire [3:0] SW,

output reg [10:0]status,

output [3:0]out,

output DAC,

output DAC2); //input for the dac

//wire [7:0] temp0; //first output register

//wire [7:0] temp1; //second output register

//wire [7:0] temp2; //third output register

wire [15:0] pa1,pa2,pa3,pa4,pa5,pa6,pa7,pa8;

//wire [15:0] pa1,pa2,pa4,pa8;

wire [15:0] temp7;

wire [15:0] temp8;

wire [15:0] Lut2Demux;

wire clk2;

wire clk3;

wire [10:0] led;

wire [10:0] o\_lut;

wire [10:0] o\_env;

wire [1:0] sel\_Demux;

wire channel\_mux\_out;

wire [7:0] out1,out2,out0;

wire nRST = ~RST;

parameter clk\_freq=50000000;

parameter idle\_timeout=1;

parameter idle\_level=1;

///////////// Testing Mux ////////////

always @(posedge SYSTEM\_CLOCK) begin

case (SW)

0: begin

status = out0;

end

1: begin

status = out1;

end

2: begin

status = out2;

end

3: begin

status = led;

end

4: begin

status = pa1;

end

5: begin

status = o\_lut;

end

6: begin

status = temp8;

end

default: begin

status = ~0;

end

endcase

end

// assign out[3] = CLK;

uart\_rx #(.baudRate(3200))

UART(.iClock(SYSTEM\_CLOCK),.rxSerialInput(IN),.o\_Byte0(out0),.o\_Byte1(out1),.o\_Byte2(out2),.testBit(out));

// UART(.iClock(clk3),.rxSerialInput(IN),.o\_Byte0(out0),.o\_Byte1(out1),.o\_Byte2(out2),.testBit(out));

//uart\_rx2 #(.baudRate(3200))

// UART2(.iClock(SYSTEM\_CLOCK),.rxSerialInput(IN\_PI),.o\_Byte0(out0),.o\_Byte1(out1),.o\_Byte2(out2),.testBit(out));

//Phase\_Accumulator PHACC (.SYSTEM\_CLOCK(clk2),.message1(out1),.message2(out2),.led(led),.pa\_out1(pa1),.pa\_out2(pa2),.pa\_out3(pa3),.pa\_out4(pa4),.pa\_out5(pa5),.pa\_out6(pa6),.pa\_out7(pa7),.pa\_out8(pa8));

//Phase\_Accumulator PHACC (.SYSTEM\_CLOCK(clk190),.message1(out1),.message2(out2),.led(led),.pa\_out1(pa1),.pa\_out2(pa2),.pa\_out4(pa4),.pa\_out8(pa8));

Phase\_Accumulator PHACC (.SYSTEM\_CLOCK(clk2),.message1(out1),.message2(out2),.led(led),.pa\_out1(pa1),.pa\_out2(pa2),.pa\_out3(pa3),.pa\_out4(pa4),.pa\_out5(pa5),.pa\_out6(pa6),.pa\_out7(pa7),.pa\_out8(pa8));

clkdiv #(.FBIT(10)) CDIV (.mclk(SYSTEM\_CLOCK), .clr(RST), .clk48(clk2),.clk190(clk3));

LUT UTT\_02(.SYSTEM\_CLOCK(SYSTEM\_CLOCK),.addr1(pa1),.addr2(pa2),.addr3(pa3),.addr4(pa4),.addr5(pa5),.addr6(pa6),.addr7(pa7),.addr8(pa8),.DAC\_out(Lut2Demux),.o\_lut(o\_lut));

//LUT UTT\_02(.SYSTEM\_CLOCK(clk3),.addr1(pa1),.addr2(pa2),.addr3(pa3),.addr4(pa4),.addr5(pa5),.addr6(pa6),.addr7(pa7),.addr8(pa8),.DAC\_out(Lut2Demux),.o\_lut(o\_lut));

//LUT UTT\_02(.SYSTEM\_CLOCK(clk2),.addr1(pa1),.addr2(pa2),.addr3(pa3),.addr4(pa4),.addr5(pa5),.addr6(pa6),.addr7(pa7),.addr8(pa8),.DAC\_out(temp7),.o\_lut(o\_lut));

//LUT UTT\_02(.SYSTEM\_CLOCK(clk2),.addr1(pa1),.addr2(pa2),.addr4(pa4),.addr8(pa8),.DAC\_out(temp7),.o\_lut(o\_lut));

//envelope UTT\_03(.SYSTEM\_CLOCK(clk3),.message2(out2),.LUT\_in(temp7),.result(temp8));

//envelope UTT\_03(.SYSTEM\_CLOCK(clk2),.message2(out2),.LUT\_in(temp7),.result(temp8));

demux1to2 UTT\_03(.Data\_in(Lut2Demux),.sel(sel\_Demux),.Data\_out\_0(temp7),.Data\_out\_1(temp8));

dac\_dsm2\_top UTT\_04(.din(temp8),.dout(DAC),.clk(SYSTEM\_CLOCK),.n\_rst(nRST));

//dac\_dsm2\_top UTT\_04(.din(temp8),.dout(DAC),.clk(clk3),.n\_rst(nRST));

//dac\_dsm2\_top UTT\_04(.din(temp8),.dout(DAC),.clk(clk2),.n\_rst(nRST));

//dac\_dsm2\_top UTT\_04(.din(temp8),.dout(DAC),.clk(clk3),.n\_rst(nRST));

dac\_dsm2\_top\_2 UTT\_05(.din(temp7),.dout(DAC2),.clk(SYSTEM\_CLOCK),.n\_rst(nRST));

//dac\_dsm2\_top\_2 UTT\_05(.din(temp7),.dout(DAC2),.clk(clk3),.n\_rst(nRST));

//channel\_mux #(.clk\_freq(clk\_freq), .idle\_timeout(idle\_timeout), .idle\_level(idle\_level))

// UTT\_07(.in\_0(IN),.in\_1(IN\_PI),.clk(SYSTEM\_CLOCK),.out(channel\_mux\_out));

endmodule

module clkdiv #(parameter FBIT=10)

(input wire mclk,

input wire clr,

output wire clk190,

output wire clk48);

reg [24:0] q;

always @(posedge mclk or posedge clr) begin

if (clr == 1 || q == 2083)

// if (clr == 1)

q <= 0;

else

q<=q+1;

end

assign clk48 = q[FBIT];

assign clk190=q[1];

endmodule

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

/\*

\* channel\_mux - Vaughn Ganem Haka

\* Usage: Automatically prioritize an active communication channel, between two

\* competing channels.

\* Input:

\* clk - The clock used. For safety, it should be >= 2\*channel\_freq

\* Assuming a shared channel frequency.

\* in\_0, in\_1 - Input communication channels.

\* Output:

\* out - Output value of the current active channel. Idles at `idle\_level` if

\* no active channel.

\* Parameters:

\* clk\_freq - The apparent clock frequency, used for determining the timeout.

\* idle\_timeout - An integer scalar, can represent seconds before a timeout.

\* idle\_level - The level which the channels idle. Assumes both channels have a shared `idle\_level`

\*/

module channel\_mux #(parameter clk\_freq=50000000, parameter idle\_timeout=1, parameter idle\_level=1) (

input in\_0, in\_1,

input clk,

output reg out

);

reg active\_channel\_num = 1'bZ;

reg [31:0] timeout\_counter = 0;

localparam ACTIVE = !idle\_level;

always @(posedge clk) begin

if (active\_channel\_num == 1'bZ) begin // Assign channel, if unassigned

if (in\_0 == ACTIVE) begin // implicitly prioritizes `in\_0` activity over `in\_1`

active\_channel\_num <= 0;

out <= in\_0;

end else if (in\_1 == ACTIVE) begin

active\_channel\_num <= 1;

out <= in\_1;

end

end else if (active\_channel\_num) begin // Assign output, check if idle. Code is semi-redundant

out <= in\_1;

if (in\_1 == out) begin // if value has not changed

timeout\_counter <= timeout\_counter + 1; // inc timeout counter (towards timeout)

end else begin

timeout\_counter <= 0; // reset timeout

end

end else begin // Same, for channel 0

out <= in\_0;

if (in\_0 == out) begin

timeout\_counter <= timeout\_counter + 1;

end else begin

timeout\_counter <= 0;

end

end

if (timeout\_counter >= (clk\_freq \* idle\_timeout)) begin // timeout current channel

timeout\_counter <= 0;

active\_channel\_num = 1'bZ;

out <= idle\_level; //idle while channel is inactive

end

end

endmodule

////////////////////////////////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////

// Company: Compute\_seniorDesignProject\_ee490

// Engineer: Alvaro Valera, Richard Proppe and Gaston Regazzo

//

// Create Date: 9:06:46 11/21/2017

// Design Name: final draft, 12/12/2017

// Module Name: Three Byte MIDI UART Receiver

// Project Name: MIDI Sound Module Piano Synthesizer

// Target Devices: NEXYS4 DDR

// Tool versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module uart\_rx

#(parameter baudRate = 3200) //paramterized to easily change baud rate

// 100 MHz / 31250 = 3200

(

input iClock,

input rxSerialInput,

// output reg [7:0] led,

output reg [7:0] o\_Byte0,o\_Byte1,o\_Byte2,

output reg [2:0] testBit

);

//Declared FSM states as local parameters

localparam idleState = 3'b000;

localparam checkStartBitState = 3'b001;

localparam receiveDataBitsState = 3'b010;

localparam receiveStopBitState = 3'b011;

reg [2:0] state = 3'b000; //holds FSM state

reg [2:0] bitCount = 3'b000; //counts received bits

reg [2:0] regCount = 3'b000;

reg [13:0] clockCounter = 0;

reg [7:0] tmpreg = 0;

reg [7:0] o\_RX\_statusByte [2:0];

// reg [7:0] tmp\_Byte0,tmp\_Byte1,tmp\_Byte2;

// initial begin

// led = 8'b01010101;

// end

always @ (posedge iClock) begin

case(state)

//waiting for start bit

idleState : begin

//set the test bit to bit number 8 out of 8 total

testBit<=3'b111;

// tmp\_Byte0 <= 8'b11111111;

// tmp\_Byte1 <= 8'b11111111;

// tmp\_Byte2 <= 8'b11111111;

clockCounter <= 0;

//if the rx line is high, 5V, stay in idle state

if(rxSerialInput) state <= idleState;

//if the rx line goes low, 0V, change to next state

else state <= checkStartBitState;

end //idleState

//confirming actual receipt of start bit

checkStartBitState : begin

//check middle of start bit

if(clockCounter < (baudRate/2) - 1 ) clockCounter <= clockCounter + 1;

else begin

clockCounter <= 0;

//if rx line is high, not in the start bit return to idle

if(rxSerialInput) state <= idleState;

//if rx line is low, the start bit is confirmed

else begin

//reset bit counter to 0 out of 7

bitCount <= 3'b000;

//change to next state

state <= receiveDataBitsState;

// led = 8'b00000000; //reset led

end

end

end //checkStartBitState

//start receiving data bits

receiveDataBitsState : begin

//begin counting up to 3200, to find the middle of the next bit

if(clockCounter < baudRate - 1) clockCounter <= clockCounter + 1;

else begin

//begin writing data to the temporary storage register, o\_RX\_statusByte

clockCounter <=0;

// led[bitCount] <= rxSerialInput;

o\_RX\_statusByte[regCount][bitCount] <= rxSerialInput;

testBit[regCount] <= rxSerialInput;

if(bitCount < 7)begin

bitCount <= bitCount + 1;

end

else begin

state <= receiveStopBitState;

if (regCount == 2) begin

regCount <= 0;

end

else begin

regCount <= regCount+1;

end

end

end

end //receiveDataBitsState

//confirm stop bit

receiveStopBitState : begin

case (regCount)

0: begin

o\_Byte2 <= o\_RX\_statusByte[2];

end

1: begin

o\_Byte0<= o\_RX\_statusByte[0];

end

2: begin

o\_Byte1 <= o\_RX\_statusByte[1];

end

default: begin

o\_Byte0 <= o\_RX\_statusByte[0];

end

endcase

// if (tmp\_Byte2 == 8'b00000000 || (tmp\_Byte0 == 8'b11111111&&tmp\_Byte1 == 8'b11111111&&tmp\_Byte2 == 8'b11111111)) begin

// {o\_Byte0,o\_Byte1,o\_Byte2} <= {o\_Byte0,o\_Byte1,o\_Byte2};

// end

// else begin

// {o\_Byte0,o\_Byte1,o\_Byte2} <= {tmp\_Byte0,tmp\_Byte1,tmp\_Byte2};

// end

//o\_Byte2 <= tmp\_Byte2;

//o\_Byte0 <= tmp\_Byte0;

//o\_Byte1 <= tmp\_Byte1;

bitCount <= 0;

if(clockCounter < baudRate - 1) clockCounter <= clockCounter + 1;

else begin

//go back to idle if stop bit received, rx line is high 5V

if(rxSerialInput) state <= idleState;

else begin //if no stop bit, then transmission may have been misidentified, so dump data

// led <= 8'b00000000;

state <= idleState;

end

end

end //receiveStopBitState

default : state <= idleState;

endcase

end

endmodule

//////////////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////

// Company: Compute\_seniorDesignProject\_ee490

// Engineer: Alvaro Valera, Richard Proppe, Gaston Regazzo, Isai Hernandez, and Abel Flores

//

// Create Date: 9:06:46 11/21/2017

// Design Name: final draft, 12/12/2017

// Module Name: Phase Accumulator Register

// Project Name: MIDI Sound Module Piano Synthesizer

// Target Devices: NEXYS4 DDR

// Tool versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module Phase\_Accumulator(input SYSTEM\_CLOCK,

// input [7:0] message0, //first output register

input [7:0] message1, //second output register

input [7:0] message2, //third output register

output reg [10:0] led,

output [15:0] pa\_out1, //);

output [15:0] pa\_out2,

output [15:0] pa\_out3,

output [15:0] pa\_out4,

output [15:0] pa\_out5,

output [15:0] pa\_out6,

output [15:0] pa\_out7,

output [15:0] pa\_out8);

wire [31:0]temp2 = 0;

wire [31:0]temp3 = 0;

wire [31:0]temp4 = 0;

wire [31:0]temp5 = 0;

wire [31:0]temp6 = 0;

wire [31:0]temp7 = 0;

wire [31:0]temp8 = 0;

reg [23:0] pa\_message;

// assign pa\_out1 = pa\_message [21:13];

assign pa\_out1 = pa\_message [23:7];

// assign temp2 = 2\*pa\_out1;

// assign pa\_out2 = temp2[31:15];

// assign temp3 = 3\*pa\_out1;

// assign pa\_out3 = temp3[31:15];

// assign temp4 = 4\*pa\_out1;

// assign pa\_out4 = temp4[31:15];

// assign temp5 = 5\*pa\_out1;

// assign pa\_out5 = temp5[31:15];

// assign temp6 = 6\*pa\_out1;

// assign pa\_out6 = temp6[31:15];

// assign temp7 = 7\*pa\_out1;

// assign pa\_out7 = temp7[31:15];

// assign temp8 = 8\*pa\_out1;

// assign pa\_out8 = temp8[31:15];

// assign pa\_out1 = pa\_message;

assign temp2 = 2\*pa\_out1;

assign pa\_out2 = temp2;

assign temp3 = 3\*pa\_out1;

assign pa\_out3 = temp3;

assign temp4 = 4\*pa\_out1;

assign pa\_out4 = temp4;

assign temp5 = 5\*pa\_out1;

assign pa\_out5 = temp5;

assign temp6 = 6\*pa\_out1;

assign pa\_out6 = temp6;

assign temp7 = 7\*pa\_out1;

assign pa\_out7 = temp7;

assign temp8 = 8\*pa\_out1;

assign pa\_out8 = temp8;

initial begin

pa\_message = 24'd0;

led = 11'b11001100111;

end

always @(posedge SYSTEM\_CLOCK) begin

if(message2) begin

led = 11'b10101010101;

case (message1)

// 8'd9 : pa\_message = pa\_message + 14'd9612;

// 8'd10 : pa\_message = pa\_message + 14'd10184;

// 8'd11 : pa\_message = pa\_message + 14'd10789;

// 8'd12 : pa\_message = pa\_message + 14'd11431;

// 8'd13 : pa\_message = pa\_message + 14'd12110;

// 8'd14 : pa\_message = pa\_message + 14'd12830;

// 8'd15 : pa\_message = pa\_message + 14'd13593;

// 8'd16 : pa\_message = pa\_message + 14'd14402;

// 8'd17 : pa\_message = pa\_message + 14'd15258;

// 8'd18 : pa\_message = pa\_message + 14'd16165;

// 8'd19 : pa\_message = pa\_message + 15'd17127;

// 8'd20 : pa\_message = pa\_message + 15'd18145;

// 8'd21 : pa\_message = pa\_message + 15'd19224;

// 8'd22 : pa\_message = pa\_message + 15'd20367;

// 8'd23 : pa\_message = pa\_message + 15'd21578;

// 8'd24 : pa\_message = pa\_message + 15'd22861;

// 8'd25 : pa\_message = pa\_message + 15'd24221;

// 8'd26 : pa\_message = pa\_message + 15'd25661;

// 8'd27 : pa\_message = pa\_message + 15'd27187;

// 8'd28 : pa\_message = pa\_message + 15'd28803;

// 8'd29 : pa\_message = pa\_message + 15'd30516;

// 8'd30 : pa\_message = pa\_message + 15'd32331;

// 8'd31 : pa\_message = pa\_message + 16'd34253;

// 8'd32 : pa\_message = pa\_message + 16'd36290;

// 8'd33 : pa\_message = pa\_message + 16'd38448;

// 8'd34 : pa\_message = pa\_message + 16'd40734;

// 8'd35 : pa\_message = pa\_message + 16'd43156;

// 8'd36 : pa\_message = pa\_message + 16'd45722;

// 8'd37 : pa\_message = pa\_message + 16'd48441;

// 8'd38 : pa\_message = pa\_message + 16'd51322;

// 8'd39 : pa\_message = pa\_message + 16'd54373;

// 8'd40 : pa\_message = pa\_message + 16'd57607;

// 8'd41 : pa\_message = pa\_message + 16'd61032;

// 8'd42 : pa\_message = pa\_message + 16'd64661;

// 8'd43 : pa\_message = pa\_message + 17'd68506;

// 8'd44 : pa\_message = pa\_message + 17'd72580;

// 8'd45 : pa\_message = pa\_message + 17'd76890;

// 8'd46 : pa\_message = pa\_message + 17'd81468;

// 8'd47 : pa\_message = pa\_message + 17'd86312;

// 8'd48 : pa\_message = pa\_message + 17'd91445;

// 8'd49 : pa\_message = pa\_message + 17'd96882;

// 8'd50 : pa\_message = pa\_message + 17'd102643;

// 8'd51 : pa\_message = pa\_message + 17'd108747;

// 8'd52 : pa\_message = pa\_message + 17'd115123;

// 8'd53 : pa\_message = pa\_message + 17'd122064;

// 8'd54 : pa\_message = pa\_message + 17'd129322;

// 8'd55 : pa\_message = pa\_message + 18'd137012;

// 8'd56 : pa\_message = pa\_message + 18'd145160;

// 8'd57 : pa\_message = pa\_message + 18'd153791;

// 8'd58 : pa\_message = pa\_message + 18'd162936;

// 8'd59 : pa\_message = pa\_message + 18'd172625;

// 8'd60 : pa\_message = pa\_message + 18'd182890;

// 8'd61 : pa\_message = pa\_message + 18'd193765;

// 8'd62 : pa\_message = pa\_message + 18'd205287;

// 8'd63 : pa\_message = pa\_message + 18'd217494;

// 8'd64 : pa\_message = pa\_message + 18'd230426;

// 8'd65 : pa\_message = pa\_message + 18'd244128;

// 8'd66 : pa\_message = pa\_message + 18'd258645;

// 8'd67 : pa\_message = pa\_message + 19'd274025;

// 8'd68 : pa\_message = pa\_message + 19'd290319;

// 8'd69 : pa\_message = pa\_message + 19'd307582;

// 8'd70 : pa\_message = pa\_message + 19'd325872;

// 8'd71 : pa\_message = pa\_message + 19'd345249;

// 8'd72 : pa\_message = pa\_message + 19'd365779;

// 8'd73 : pa\_message = pa\_message + 19'd387529;

// 8'd74 : pa\_message = pa\_message + 19'd410573;

// 8'd75 : pa\_message = pa\_message + 19'd434987;

// 8'd76 : pa\_message = pa\_message + 19'd460853;

// 8'd77 : pa\_message = pa\_message + 19'd488256;

// 8'd78 : pa\_message = pa\_message + 19'd517290;

// 8'd79 : pa\_message = pa\_message + 20'd548049;

// 8'd80 : pa\_message = pa\_message + 20'd580638;

// 8'd81 : pa\_message = pa\_message + 20'd615165;

// 8'd82 : pa\_message = pa\_message + 20'd651744;

// 8'd83 : pa\_message = pa\_message + 20'd690499;

// 8'd84 : pa\_message = pa\_message + 20'd731558;

// 8'd85 : pa\_message = pa\_message + 20'd775059;

// 8'd86 : pa\_message = pa\_message + 20'd821146;

// 8'd87 : pa\_message = pa\_message + 20'd869974;

// 8'd88 : pa\_message = pa\_message + 20'd921705;

// 8'd89 : pa\_message = pa\_message + 20'd976513;

// 8'd90 : pa\_message = pa\_message + 20'd1034579;

// 8'd91 : pa\_message = pa\_message + 21'd1096099;

// 8'd92 : pa\_message = pa\_message + 21'd1161276;

// 8'd93 : pa\_message = pa\_message + 21'd1230329;

// 8'd94 : pa\_message = pa\_message + 21'd1303488;

// 8'd95 : pa\_message = pa\_message + 21'd1380998;

// 8'd96 : pa\_message = pa\_message + 21'd1463116;

8'd9 : pa\_message = pa\_message + 14'd9612 ;

8'd10 : pa\_message = pa\_message + 14'd10183 ;

8'd11 : pa\_message = pa\_message + 14'd10789 ;

8'd12 : pa\_message = pa\_message + 14'd11430 ;

8'd13 : pa\_message = pa\_message + 14'd12110 ;

8'd14 : pa\_message = pa\_message + 14'd12830 ;

8'd15 : pa\_message = pa\_message + 14'd13593 ;

8'd16 : pa\_message = pa\_message + 14'd14401 ;

8'd17 : pa\_message = pa\_message + 14'd15258 ;

8'd18 : pa\_message = pa\_message + 14'd16165 ;

8'd19 : pa\_message = pa\_message + 15'd17126 ;

8'd20 : pa\_message = pa\_message + 15'd18144 ;

8'd21 : pa\_message = pa\_message + 15'd19223 ;

8'd22 : pa\_message = pa\_message + 15'd20367 ;

8'd23 : pa\_message = pa\_message + 15'd21578 ;

8'd24 : pa\_message = pa\_message + 15'd22861 ;

8'd25 : pa\_message = pa\_message + 15'd24220 ;

8'd26 : pa\_message = pa\_message + 15'd25660 ;

8'd27 : pa\_message = pa\_message + 15'd27186 ;

8'd28 : pa\_message = pa\_message + 15'd28803 ;

8'd29 : pa\_message = pa\_message + 15'd30516 ;

8'd30 : pa\_message = pa\_message + 15'd32330 ;

8'd31 : pa\_message = pa\_message + 16'd34253 ;

8'd32 : pa\_message = pa\_message + 16'd36289 ;

8'd33 : pa\_message = pa\_message + 16'd38447 ;

8'd34 : pa\_message = pa\_message + 16'd40734 ;

8'd35 : pa\_message = pa\_message + 16'd43156 ;

8'd36 : pa\_message = pa\_message + 16'd45722 ;

8'd37 : pa\_message = pa\_message + 16'd48441 ;

8'd38 : pa\_message = pa\_message + 16'd51321 ;

8'd39 : pa\_message = pa\_message + 16'd54373 ;

8'd40 : pa\_message = pa\_message + 16'd57606 ;

8'd41 : pa\_message = pa\_message + 16'd61032 ;

8'd42 : pa\_message = pa\_message + 16'd64661 ;

8'd43 : pa\_message = pa\_message + 17'd68506 ;

8'd44 : pa\_message = pa\_message + 17'd72579 ;

8'd45 : pa\_message = pa\_message + 17'd76895 ;

8'd46 : pa\_message = pa\_message + 17'd81468 ;

8'd47 : pa\_message = pa\_message + 17'd86312 ;

8'd48 : pa\_message = pa\_message + 17'd91444 ;

8'd49 : pa\_message = pa\_message + 17'd96882 ;

8'd50 : pa\_message = pa\_message + 17'd102643 ;

8'd51 : pa\_message = pa\_message + 17'd108746 ;

8'd52 : pa\_message = pa\_message + 17'd115213 ;

8'd53 : pa\_message = pa\_message + 17'd122064 ;

8'd54 : pa\_message = pa\_message + 17'd129322 ;

8'd55 : pa\_message = pa\_message + 18'd137012 ;

8'd56 : pa\_message = pa\_message + 18'd145159 ;

8'd57 : pa\_message = pa\_message + 18'd153791 ;

8'd58 : pa\_message = pa\_message + 18'd162936 ;

8'd59 : pa\_message = pa\_message + 18'd172624 ;

8'd60 : pa\_message = pa\_message + 18'd182889 ;

8'd61 : pa\_message = pa\_message + 18'd193764 ;

8'd62 : pa\_message = pa\_message + 18'd205286 ;

8'd63 : pa\_message = pa\_message + 18'd217493 ;

8'd64 : pa\_message = pa\_message + 18'd230426 ;

8'd65 : pa\_message = pa\_message + 18'd244128 ;

8'd66 : pa\_message = pa\_message + 18'd258644 ;

8'd67 : pa\_message = pa\_message + 19'd274024 ;

8'd68 : pa\_message = pa\_message + 19'd290319 ;

8'd69 : pa\_message = pa\_message + 19'd307582 ;

8'd70 : pa\_message = pa\_message + 19'd325872 ;

8'd71 : pa\_message = pa\_message + 19'd345250 ;

8'd72 : pa\_message = pa\_message + 19'd365779 ;

8'd73 : pa\_message = pa\_message + 19'd387530 ;

8'd74 : pa\_message = pa\_message + 19'd410573 ;

8'd75 : pa\_message = pa\_message + 19'd434987 ;

8'd76 : pa\_message = pa\_message + 19'd460852 ;

8'd77 : pa\_message = pa\_message + 19'd488257 ;

8'd78 : pa\_message = pa\_message + 19'd517290 ;

8'd79 : pa\_message = pa\_message + 20'd548050 ;

8'd80 : pa\_message = pa\_message + 20'd580638 ;

8'd81 : pa\_message = pa\_message + 20'd615165 ;

8'd82 : pa\_message = pa\_message + 20'd651744 ;

8'd83 : pa\_message = pa\_message + 20'd690499 ;

8'd84 : pa\_message = pa\_message + 20'd731558 ;

8'd85 : pa\_message = pa\_message + 20'd775059 ;

8'd86 : pa\_message = pa\_message + 20'd821146 ;

8'd87 : pa\_message = pa\_message + 20'd869974 ;

8'd88 : pa\_message = pa\_message + 20'd921706 ;

8'd89 : pa\_message = pa\_message + 20'd976513 ;

8'd90 : pa\_message = pa\_message + 20'd1034580 ;

8'd91 : pa\_message = pa\_message + 24'd1096099 ;

8'd92 : pa\_message = pa\_message + 24'd1161276 ;

8'd93 : pa\_message = pa\_message + 24'd1230329 ;

8'd94 : pa\_message = pa\_message + 24'd1303489 ;

8'd95 : pa\_message = pa\_message + 24'd1380998 ;

8'd96 : pa\_message = pa\_message + 24'd1463116 ;

8'd97 : pa\_message = pa\_message + 24'd1550117 ;

8'd98 : pa\_message = pa\_message + 24'd1642292 ;

8'd99 : pa\_message = pa\_message + 24'd1739948 ;

8'd100 : pa\_message = pa\_message + 24'd1843410 ;

8'd101 : pa\_message = pa\_message + 24'd1953025 ;

8'd102 : pa\_message = pa\_message + 24'd2069158 ;

8'd103 : pa\_message = pa\_message + 24'd2192197 ;

8'd104 : pa\_message = pa\_message + 24'd2322552 ;

8'd105 : pa\_message = pa\_message + 24'd2460658 ;

8'd106 : pa\_message = pa\_message + 24'd2606976 ;

8'd107 : pa\_message = pa\_message + 24'd2761995 ;

8'd108 : pa\_message = pa\_message + 24'd2926232 ;

8'd109 : pa\_message = pa\_message + 24'd3100235 ;

8'd110 : pa\_message = pa\_message + 24'd3284584 ;

8'd111 : pa\_message = pa\_message + 24'd3479896 ;

8'd112 : pa\_message = pa\_message + 24'd3686821 ;

8'd113 : pa\_message = pa\_message + 24'd3906051 ;

8'd114 : pa\_message = pa\_message + 24'd4138317 ;

8'd115 : pa\_message = pa\_message + 24'd4384394 ;

8'd116 : pa\_message = pa\_message + 24'd4645104 ;

8'd117 : pa\_message = pa\_message + 24'd4921316 ;

8'd118 : pa\_message = pa\_message + 24'd5213953 ;

8'd119 : pa\_message = pa\_message + 24'd5523991 ;

8'd120 : pa\_message = pa\_message + 24'd5852464 ;

default: pa\_message = 24'b0000000000000000000000;

endcase

end

else pa\_message = 24'b0000000000000000000000;

end

endmodule

/////////////////////////////////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////

// Company: Compute\_seniorDesignProject\_ee490

// Engineer: Alvaro Valera, Richard Proppe, Gaston Regazzo, Abel Flores, and Isai Hernandez

//

// Create Date: 9:06:46 11/21/2017

// Design Name: final draft, 12/12/2017

// Module Name: Phase to Amplitude Look Up Table

// Project Name: MIDI Sound Module Piano Synthesizer

// Target Devices: NEXYS4 DDR

// Tool versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module LUT #(parameter addWidth = 16 , dataWidth = 16)

(input SYSTEM\_CLOCK,

input [addWidth-1:0] addr1,

input [addWidth-1:0] addr2,

input [addWidth-1:0] addr3,

input [addWidth-1:0] addr4,

input [addWidth-1:0] addr5, // address passed in by PC

input [addWidth-1:0] addr6,

input [addWidth-1:0] addr7,

input [addWidth-1:0] addr8,

output [dataWidth-1:0] o\_lut,

output reg [dataWidth-1:0] DAC\_out ); // value to pass out

reg [dataWidth-1:0] LUT[0:(2\*\*addWidth)-1];

assign o\_lut = DAC\_out;

//assign div2 = addr2 >>1;

//assign div4 = addr4 >>2;

//assign div8 = addr8 >>3;

assign div2 = LUT[addr2] >> 1;

assign div3 = LUT[addr3] >> 1;

assign div4 = LUT[addr4] >> 2;

assign div5 = LUT[addr5] >> 2;

assign div6 = LUT[addr6] >> 2;

assign div7 = LUT[addr7] >> 2;

assign div8 = LUT[addr8] >> 3;

initial begin

$readmemb("LUT\_16.txt", LUT);

end

always @(posedge SYSTEM\_CLOCK)begin

// DAC\_out <= LUT[addr1];

//DAC\_out <= (1/1)\*LUT[addr1] + (1/2)\*LUT[addr2] + (1/3)\*LUT[addr3] + (1/4)\*LUT[addr4] + (1/5)\*LUT[addr5]+ (1/6)\*LUT[addr6] + (1/7)\*LUT[addr7] + (1/8)\*LUT[addr8];

DAC\_out <= LUT[addr1] + div2 + div3 + div4 + div5 + div6 + div7 + div8;

end

endmodule

//////////////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////

// Company: Compute\_seniorDesignProject\_ee490

// Engineer: Gaston Regazzo

//

// Create Date: 9:06:46 11/21/2017

// Design Name: final draft, 12/12/2017

// Module Name: Demux 1:2

// Project Name: MIDI Sound Module Piano Synthesizer

// Target Devices: NEXYS4 DDR

// Tool versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

//Verilog module for 1:2 DEMUX

module demux1to2(input [15:0] Data\_in,

input [1:0] sel,

output reg [15:0] Data\_out\_0,

output reg [15:0] Data\_out\_1);

//always block with Data\_in and sel in its sensitivity list

always @(Data\_in or sel)

begin

case (sel) //case statement with "sel"

//multiple statements can be written inside each case.

//you just have to use 'begin' and 'end' keywords as shown below.

2'b00 : begin

Data\_out\_0 = Data\_in;

Data\_out\_1 = 0;

end

2'b01 : begin

Data\_out\_0 = 0;

Data\_out\_1 = Data\_in;

end

endcase

end

endmodule

////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////

// Company: Compute\_seniorDesignProject\_ee490

// Engineer: Richard Proppe, Gaston Regazzo, and Alvaro Valera

//

// Create Date: 9:06:46 11/21/2017

// Design Name: final draft, 12/13/2017

// Module Name: ADSR Envelope Module

// Project Name: MIDI Sound Module Piano Synthesizer

// Target Devices: NEXYS4 DDR

// Tool versions:

// Description:

//

// Dependencies:

//

// Revision: 2.01 - File Modified

// Revision 0.01 - File Created

// Additional Comments: This module is incomplete as a better clock divider or

// counter scheme is still needed. This design originally used

// decimal integers to scale the signal amplitudes and the

// system clock. However this added distortion to the signal.

// This version right shifts the incoming data in order to

// divide and reduce the signal am,plitude.

//

//////////////////////////////////////////////////////////////////////////////////

//declare STATE MACHINE variables

`define IDLE 3'd0

`define ATTACK 3'd1

`define DECAY 3'd2

`define HOLD 3'd3

`define FADE 3'd4

//declare counter variables, M100 = 100Mhz = 1 second

`define M400 30'd400000000// count up to 400 Mega clock cycles from zero, takes 4 seconds

`define M100 30'd100000000// count up to 100 Mega clock cycles from zero, takes 1 seconds

//used to right shift 16bits in two seconds

`define M12 30'd12500000// count up to 12.5 Mega clock cycles from zero, takes 1/8 seconds

module envelope(input SYSTEM\_CLOCK,

input [7:0] message2, //dataByte2

input [15:0] LUT\_in, //input from LUT(DAC\_out)

output reg [15:0] result);

reg [2:0] current\_state = `IDLE;

reg [2:0] next\_state;

//declare and initialize state time counters

reg [30:0] a\_count = 30'd0;

reg [30:0] d\_count = 30'd0;

reg [30:0] t\_count = 30'd0;

reg [30:0] f\_count = 30'd0;

//added 12-12-2017

//declare and initialize state loop variables

reg[4:0] a = 16;

reg[4:0] d = 0;

reg[4:0] h = 8;

reg[4:0] f = 8;

always @(posedge SYSTEM\_CLOCK) begin

current\_state = next\_state;

end

always @\* begin

case (current\_state)

`IDLE: begin

//reset all counters and loop variables

a\_count = 30'd0;

d\_count = 30'd0;

t\_count = 30'd0;

f\_count = 30'd0;

a = 16;

d = 0;

h = 8;

f = 8;

//if a key was presses change to next state

if (message2 != 0) begin

next\_state = `ATTACK;

end

//remain in idle state

else begin

next\_state = `IDLE;

end

end

`ATTACK: begin

//no key pressed, off

if (message2 == 0) begin

next\_state = `IDLE;

end

//key is pressed, on

else begin

//count up to 1/8 second, 12.5M clock cycles

if (a\_count != `M12) begin

a\_count = a\_count + 1;

end

else if (a\_count == `M12) begin

//increase signal amplitide by shifting bits

if (a != 0) begin

//reduce the number of bits being right shifted by one

a <= a - 1;

//reset counter to zero, go to top of state

a\_count <= 0;

end

else if (a == 0) begin

//change to next state

next\_state = `DECAY;

end

else

//return to idle state

begin

next\_state = `IDLE;

end

end

end

end

`DECAY: begin

//no key pressed, off

if (message2 == 0) begin

next\_state = `IDLE;

end

//key is pressed, on

else begin

//count up to 1/8 second, 12.5M clock cycles

if (d\_count != `M12) begin

d\_count = d\_count + 1;

end

else if (d\_count == `M12) begin

//decrease signal amplitide by shifting bits

if (d != 8) begin

//increase the number of bits being right shifted by one

d <= d + 1;

//reset counter to zero, go to top of state

d\_count <= 0;

end

else if (d == 8) begin

//change to next state

next\_state = `HOLD;

end

else

//return to idle state

begin

next\_state = `IDLE;

end

end

end

end

`HOLD: begin

//no key is pressed, off

if (message2 == 0) begin

next\_state = `IDLE;

end

//key is pressed, on

else begin

//hold note for 400M clock cycles, 4 seconds

if (t\_count != `M400) begin

t\_count = t\_count + 1;

next\_state = `HOLD;

end

else begin

//change to next state

if (t\_count == `M400) begin

next\_state = `FADE;

end

end

end

end

`FADE: begin

//no key pressed, off

if (message2 == 0) begin

next\_state = `IDLE;

end

//key is pressed, on

else begin

//count up to 1/8 second, 12.5M clock cycles

if (f\_count != `M12) begin

f\_count = f\_count + 1;

end

else if (f\_count == `M12) begin

//decrease signal amplitide by shifting bits

if (f != 16) begin

//increase the number of bits being right shifted by one

f <= f + 1;

//reset counter to zero, go to top of state

f\_count <= 0;

end

else if (f == 0) begin

//change to idle state, end of ADSR loop

next\_state = `IDLE;

end

else

//return to idle state

begin

next\_state = `IDLE;

end

end

end

end

default: begin

//reset all counters and loop variables

a\_count = 30'd0;

d\_count = 30'd0;

t\_count = 30'd0;

f\_count = 30'd0;

a = 16;

d = 0;

h = 8;

f = 8;

//if key is pressed go to attack state

if (message2 != 0) begin

next\_state = `ATTACK;

end

//return to idle state

else begin

next\_state = `IDLE;

end

end

endcase

end

always @(current\_state) begin

case (current\_state)

`IDLE: begin

result = 0;

end

`ATTACK: begin

result = LUT\_in >> a;

end

`DECAY: begin

result = LUT\_in >> d;

end

`HOLD: begin

result = LUT\_in >> h;

end

`FADE: begin

result = LUT\_in >> f;

end

default: begin

result = 0;

end

endcase

end

endmodule

//////////////////////////////////////////////////////////////////////////////////

// Company: Compute\_seniorDesignProject\_ee490

// Engineer: Gaston Regazzo

//

// Create Date: 9:06:46 11/21/2017

// Design Name: final draft, 12/12/2017

// Module Name: Digital to Analog Converter DAC Module

// Project Name: MIDI Sound Module Piano Synthesizer

// Target Devices: NEXYS4 DDR

// Tool versions:

// Description:

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.numeric\_std.all;

entity dac\_dsm2\_2 is

generic (

nbits : integer := 16);

port (

din : in signed((nbits-1) downto 0);

dout : out std\_logic;

clk : in std\_logic;

n\_rst : in std\_logic);

end dac\_dsm2\_2;

architecture beh1 of dac\_dsm2\_2 is

signal del1, del2, d\_q : signed(nbits+2 downto 0) := (others => '0');

constant c1 : signed(nbits+2 downto 0) := to\_signed(1, nbits+3);

constant c\_1 : signed(nbits+2 downto 0) := to\_signed(-1, nbits+3);

begin -- beh1

process (clk, n\_rst)

begin -- process

if n\_rst = '0' then -- asynchronous reset (active low)

del1 <= (others => '0');

del2 <= (others => '0');

dout <= '0';

elsif clk'event and clk = '1' then -- rising clock edge

del1 <= din - d\_q + del1;

del2 <= din - d\_q + del1 - d\_q + del2;

if din - d\_q + del1 - d\_q + del2 > 0 then

d\_q <= shift\_left(c1, nbits);

dout <= '1';

else

d\_q <= shift\_left(c\_1, nbits);

dout <= '0';

end if;

end if;

end process;

end beh1;