# DCT Implementation on FPGA Boards for JPEG Compression

By Discrete Cosine Sabha (Romit, Jaskirat, Soham, Nishchay)

Verilog Files

1. ***Topmodule.v***

`timescale 1ns / 1ps

module topmodule(

input clk,

input recieve, // Start receiving data

input Rx, // UART Rx pin

input transmit,

input dct\_enable, // Start transmitting data

output Tx, // UART Tx pin

output [7:0] dataout\_topmodule, // Output from BRAM storing image data

output imrxcomplete, // Flag for receiving completion

output dct\_done // Flag for dct done

);

wire s\_clk;

slow\_clock\_gen #(.DIV\_FACTOR(1\_000)) slow\_clk (clk, s\_clk);

// BRAM to store the original image

reg ena\_top;

reg wea\_top;

reg [13:0] addr\_top;

reg [7:0] din\_top;

wire [7:0] dout\_top;

blk\_mem\_gen\_0 original(

.clka(clk),

.ena(ena\_top),

.wea(wea\_top),

.addra(addr\_top),

.dina(din\_top),

.douta(dout\_top)

);

// BRAM to store the processed image

reg ena\_processed;

reg wea\_processed;

reg [14:0] addr\_processed;

reg [7:0] din\_processed;

wire [7:0] dout\_processed;

blk\_mem\_gen\_1 processed(

.clka(clk),

.ena(ena\_processed),

.wea(wea\_processed),

.addra(addr\_processed),

.dina(din\_processed),

.douta(dout\_processed)

);

// Image receiver module

wire ena\_imrx;

wire wea\_imrx;

wire [13:0] addr\_imrx;

wire [7:0] din\_imrx;

reg [7:0] dout\_imrx;

imrx uut1(

.clk(clk),

.reset(~recieve),

.ena(1),

.RxD(Rx),

.addr(addr\_imrx),

.dout(dout\_imrx),

.ImRxComplete(imrxcomplete),

.ena\_imrx(ena\_imrx),

.wea\_imrx(wea\_imrx),

.din\_imrx(din\_imrx),

.addr\_imrx(addr\_imrx)

);

reg dct\_start;

wire ena\_dct\_orig,ena\_dct\_proc1;

wire wea\_dct\_orig,wea\_dct\_proc1;

wire [13:0] addr\_dct\_orig;

wire [14:0] addr\_dct\_proc1;

wire [7:0] din\_dct\_orig;

wire [7:0] din\_dct\_proc1;

reg [7:0] dout\_dct\_proc1;

reg [7:0] dout\_dct\_orig;

dct\_top\_2 dct\_inst (

.clk(clk),

.s\_clk(s\_clk),

.reset(~dct\_enable),

.start(dct\_start),

.done(dct\_done),

// Connect the original image BRAM ports.

.orig\_ena(ena\_dct\_orig),

.orig\_wea(wea\_dct\_orig),

.orig\_addr(addr\_dct\_orig),

.orig\_din(din\_dct\_orig),

.orig\_dout(dout\_dct\_orig),

.proc1\_ena(ena\_dct\_proc1),

.proc1\_wea(wea\_dct\_proc1),

.proc1\_addr(addr\_dct\_proc1),

.proc1\_din(din\_dct\_proc1),

.proc1\_dout(dout\_dct\_proc1)

);

// Image transmitter module

wire [7:0] data\_tx;

wire ena\_imtx;

wire wea\_imtx;

wire [14:0] addr\_imtx;

wire [7:0] din\_imtx;

reg [7:0] dout\_imtx;

imtx uut2(

.clk(clk),

.reset(~transmit),

.transmit(transmit),

.data(data\_tx),

.TxD(Tx),

.ena\_tx(ena\_imtx),

.wea\_tx(wea\_imtx),

.addr\_tx(addr\_imtx),

.din\_tx(din\_imtx),

.dout\_tx(dout\_imtx)

);

// Control logic for BRAM access

assign dataout\_topmodule = dout\_top;

always @(posedge clk) begin

if (recieve) begin // Receive operation

ena\_top <= ena\_imrx;

wea\_top <= wea\_imrx;

addr\_top <= addr\_imrx;

din\_top <= din\_imrx;

dout\_imrx <= dout\_top;

end else if (dct\_enable) begin

dct\_start <= ~dct\_done;

ena\_top <= ena\_dct\_orig;

wea\_top <= wea\_dct\_orig;

addr\_top <= addr\_dct\_orig;

din\_top <= din\_dct\_orig;

dout\_dct\_orig <= dout\_top;

ena\_processed <= ena\_dct\_proc1;

wea\_processed <= wea\_dct\_proc1;

addr\_processed <= addr\_dct\_proc1;

din\_processed <= din\_dct\_proc1;

dout\_dct\_proc1 <= dout\_processed;

end else if (transmit) begin // Transmit operation

ena\_processed <= ena\_imtx;

wea\_processed <= wea\_imtx;

addr\_processed <= addr\_imtx;

din\_processed <= din\_imtx;

dout\_imtx <= dout\_processed;

end

end

endmodule

1. ***dct\_top\_2.v***

`timescale 1ns / 1ps

module dct\_top\_2 (

input clk,

input s\_clk,

input reset,

input start,

output reg done,

// Original BRAM interface (read-only in this design)

output reg orig\_ena,

output reg orig\_wea,

output reg [13:0] orig\_addr,

output reg [7:0] orig\_din,

input [7:0] orig\_dout,

// Processed BRAM interface (write)

output reg proc1\_ena,

output reg proc1\_wea,

output reg [14:0] proc1\_addr,

output reg [7:0] proc1\_din,

input [7:0] proc1\_dout

);

// Parameters: 256 blocks, each 8x8 pixels (64 pixels per block).

parameter NUM\_BLOCKS = 256;

parameter BLOCK\_PIXELS = 64;

parameter TOTAL\_PIXELS = NUM\_BLOCKS \* BLOCK\_PIXELS; // 16384

// FSM states.

reg [7:0] wait\_time;

localparam IDLE = 3'd0,

ENABLE = 4'd9,

WAIT\_FOR\_READ = 4'd8,

READ = 3'd1,

WAIT\_FOR\_PROCESS = 3'd6,

PROCESS = 3'd2,

WAIT\_FOR\_WRITE = 3'd7,

WRITE = 3'd3,

NEXT = 3'd4,

DONE = 3'd5,

WAIT\_TIME = 3'd7,

PROCESS\_TIME = 'd80;

// Counters.

reg [15:0] overall\_pixel = 0; // overall pixel address: 0 to TOTAL\_PIXELS-1

reg [5:0] pixel\_count = 0; // pixel counter within a block (0 to 63)

reg [7:0] block\_count = 0; // block counter (0 to NUM\_BLOCKS-1)

// Block registers (each block is 64 pixels, each pixel is 16-bit).

reg [1023:0] dct\_block\_in;

reg [1023:0] dct\_block\_out;

reg dodct\_start = 0;

wire dodct\_done;

wire [1023:0] dct\_result;

// Temporary registers.

reg [15:0] conv\_pixel; // Converted pixel value (16 bits)

reg [7:0] pixel\_data; // latched orig\_dout

reg write\_phase; // 0: write upper 8 bits, 1: write lower 8 bits

reg [13:0] read\_addr; // Computed read address (14 bits)

reg [14:0] temp\_addr; // Computed base write address (15 bits)

reg process\_reset;

do\_dct dct\_process (

.clk(clk),

.s\_clk(s\_clk),

.reset(process\_reset),

.start(dodct\_start),

.block\_in(dct\_block\_in),

.block\_out(dct\_result),

.done(dodct\_done)

);

reg [3:0] state = IDLE;

always @ (posedge s\_clk)

begin

case(state)

IDLE: begin

overall\_pixel <= 0;

pixel\_count <= 0;

block\_count <= 0;

done <= 0;

write\_phase <= 0;

orig\_ena <= 0;

orig\_wea <= 0;

orig\_addr <= 0;

orig\_din <= 0;

proc1\_ena <= 0;

proc1\_wea <= 0;

proc1\_addr<= 0;

proc1\_din <= 0;

// Clear block registers.

dct\_block\_in <= 0;

dct\_block\_out <= 0;

dodct\_start <= 0;

process\_reset <= 1;

if (start) begin

wait\_time <= WAIT\_TIME;

state <= ENABLE;

end else

state <= IDLE;

end

ENABLE: begin // enable reading

orig\_ena <= 1;

orig\_wea <= 0;

orig\_addr <= overall\_pixel[13:0];

state <= WAIT\_FOR\_READ;

end

WAIT\_FOR\_READ: begin

wait\_time <= wait\_time - 3'd1;

if (wait\_time == 0)

state <= READ;

else

state <= WAIT\_FOR\_READ;

end

READ: begin

pixel\_data = orig\_dout; // get the latest value then move ahead

conv\_pixel = ($signed({1'b0, pixel\_data}) - 9'sd128) <<< 5;

dct\_block\_in[pixel\_count\*16 +: 16] = conv\_pixel; // all blocking statements

// Prepare for next pixel.

overall\_pixel <= overall\_pixel + 1;

if (pixel\_count == BLOCK\_PIXELS-1) begin

pixel\_count <= 0;

wait\_time <= PROCESS\_TIME;

state <= WAIT\_FOR\_PROCESS;

process\_reset <= 0;

end else begin

pixel\_count <= pixel\_count + 1;

wait\_time <= WAIT\_TIME;

state <= ENABLE;

end

end

WAIT\_FOR\_PROCESS: begin

if (wait\_time == PROCESS\_TIME)

dodct\_start <= 1;

else if (wait\_time == 0)

state <= PROCESS;

else

state <= WAIT\_FOR\_PROCESS;

wait\_time <= wait\_time - 3'd1;

end

// PROCESS state: dummy DCT processing (pass-through).

PROCESS: begin

dct\_block\_out <= dct\_result;//dct\_block\_in; //dct\_result;

dodct\_start <= 0; // Clear the start for next block.

process\_reset <= 1;

// Reset pixel counter for write.

write\_phase <= 0;

pixel\_count <= 0;

wait\_time <= WAIT\_TIME;

state <= WAIT\_FOR\_WRITE;

end

WAIT\_FOR\_WRITE: begin

wait\_time <= wait\_time - 3'd1;

if (wait\_time == 0)

state <= WRITE;

else

state <= WAIT\_FOR\_WRITE;

end

WRITE: begin

// Compute the base address for the pixel (same order as read).

// Note: overall\_pixel for write is derived from block\_count and pixel\_count.

temp\_addr = (block\_count \* BLOCK\_PIXELS) + pixel\_count;

if (write\_phase == 0) begin

// Write upper 8 bits at the same address.

proc1\_addr <= temp\_addr;

proc1\_din <= dct\_block\_out[pixel\_count\*16 + 15 -: 8];

proc1\_ena <= 1;

proc1\_wea <= 1;

write\_phase <= 1;

end else begin

// Write lower 8 bits to the address offset by 16384.

proc1\_addr <= temp\_addr + 15'd16384;

proc1\_din <= dct\_block\_out[pixel\_count\*16 + 7 -: 8];

proc1\_ena <= 1;

proc1\_wea <= 1;

write\_phase <= 0;

// If we have written both phases for this pixel, move to the next.

if (pixel\_count == BLOCK\_PIXELS-1) begin

pixel\_count <= 0;

state <= NEXT;

end else begin

pixel\_count <= pixel\_count + 1;

end

end

end

// NEXT state: proceed to next block.

NEXT: begin

if (block\_count == NUM\_BLOCKS-1) begin

state <= DONE;

end else begin

block\_count <= block\_count + 1;

state <= ENABLE;

end

end

DONE: begin

done <= 1;

state <= DONE; // Remain here.

end

default: state <= IDLE;

endcase

end

endmodule

1. ***do\_dct.v***

`timescale 1ns / 1ps

module do\_dct (

input clk,

input s\_clk,

input reset,

input start,

input [1023:0] block\_in,

output reg [1023:0] block\_out,

output reg done

);

// Behavioral BRAMs (64 words of 16 bits each)

reg [15:0] bram\_X [0:63]; // Input X (and final result overwritten here)

reg [15:0] bram\_B2 [0:63]; // Intermediate storage: result of D\*X

reg [15:0] bram\_D [0:63]; // DCT coefficient ROM

integer o;

// Wires for connecting to the multiplier modules

reg [1023:0] mat\_A;

reg [1023:0] mat\_B;

wire [1023:0] mat\_out;

wire mult\_done;

// For the second multiplication

reg [1023:0] inter\_result;

wire [1023:0] final\_result;

wire mult2\_done;

// Top-level state machine

reg [1:0] state;

localparam IDLE = 2'b00,

MUL1 = 2'b01,

MUL2 = 2'b10,

DONE = 2'b11;

integer i;

// Instantiate multiplier for stage 1: D \* X (normal multiplication)

matrix\_mult\_8x8\_dsp #(.TRANSPOSE\_B(0), .QUANTIZE(0)) mult1 (

.clk(clk),

.s\_clk(s\_clk),

.reset(reset),

.enable(state == MUL1),

.A(mat\_A), // will carry D (from bram\_D)

.B(mat\_B), // will carry X (from bram\_X)

.C(mat\_out),

.done(mult\_done)

);

// Instantiate multiplier for stage 2: (D\*X) \* D^T

matrix\_mult\_8x8\_dsp #(.TRANSPOSE\_B(1), .QUANTIZE(0)) mult2 (

.clk(clk),

.s\_clk(s\_clk),

.reset(reset),

.enable(state == MUL2),

.A(inter\_result), // intermediate result from stage 1

.B(mat\_B), // will carry D (from bram\_D)

.C(final\_result),

.done(mult2\_done)

);

// Top-level state machine: control the two multiplications and the memory transfers.

always @(posedge s\_clk or posedge reset) begin

if(reset) begin

state <= IDLE;

done <= 0;

// values for DCT Coefficients Matrix

bram\_D[0] <= 16'h000B; bram\_D[1] <= 16'h000B; bram\_D[2] <= 16'h000B; bram\_D[3] <= 16'h000B; bram\_D[4] <= 16'h000B; bram\_D[5] <= 16'h000B; bram\_D[6] <= 16'h000B; bram\_D[7] <= 16'h000B;

bram\_D[8] <= 16'h0010; bram\_D[9] <= 16'h000D; bram\_D[10] <= 16'h0009; bram\_D[11] <= 16'h0003; bram\_D[12] <= 16'hFFFD; bram\_D[13] <= 16'hFFF7; bram\_D[14] <= 16'hFFF3; bram\_D[15] <= 16'hFFF0;

bram\_D[16] <= 16'h000F; bram\_D[17] <= 16'h0006; bram\_D[18] <= 16'hFFFA; bram\_D[19] <= 16'hFFF1; bram\_D[20] <= 16'hFFF1; bram\_D[21] <= 16'hFFFA; bram\_D[22] <= 16'h0006; bram\_D[23] <= 16'h000F;

bram\_D[24] <= 16'h000D; bram\_D[25] <= 16'hFFFD; bram\_D[26] <= 16'hFFF0; bram\_D[27] <= 16'hFFF7; bram\_D[28] <= 16'h0009; bram\_D[29] <= 16'h0010; bram\_D[30] <= 16'h0003; bram\_D[31] <= 16'hFFF3;

bram\_D[32] <= 16'h000B; bram\_D[33] <= 16'hFFF5; bram\_D[34] <= 16'hFFF5; bram\_D[35] <= 16'h000B; bram\_D[36] <= 16'h000B; bram\_D[37] <= 16'hFFF5; bram\_D[38] <= 16'hFFF5; bram\_D[39] <= 16'h000B;

bram\_D[40] <= 16'h0009; bram\_D[41] <= 16'hFFF0; bram\_D[42] <= 16'h0003; bram\_D[43] <= 16'h000D; bram\_D[44] <= 16'hFFF3; bram\_D[45] <= 16'hFFFD; bram\_D[46] <= 16'h0010; bram\_D[47] <= 16'hFFF7;

bram\_D[48] <= 16'h0006; bram\_D[49] <= 16'hFFF1; bram\_D[50] <= 16'h000F; bram\_D[51] <= 16'hFFFA; bram\_D[52] <= 16'hFFFA; bram\_D[53] <= 16'h000F; bram\_D[54] <= 16'hFFF1; bram\_D[55] <= 16'h0006;

bram\_D[56] <= 16'h0003; bram\_D[57] <= 16'hFFF7; bram\_D[58] <= 16'h000D; bram\_D[59] <= 16'hFFF0; bram\_D[60] <= 16'h0010; bram\_D[61] <= 16'hFFF3; bram\_D[62] <= 16'h0009; bram\_D[63] <= 16'hFFFD;

// Pack the contents of the coefficient ROM (bram\_D) into a 1024-bit bus

// This bus is used as the A operand for mult1.

for(i = 0; i < 64; i = i + 1)

mat\_A[i\*16 +: 16] = bram\_D[i];

end else begin

case(state)

IDLE: begin

done <= 0;

if(start) begin

state <= MUL1;

// For the B operand: In stage MUL1 we use X (our data matrix)

mat\_B = block\_in;

end

end

MUL1: begin

if(mult\_done) begin

// Capture the intermediate result into bram\_B2 and into a register for the next multiplier

for(i = 0; i < 64; i = i + 1)

bram\_B2[i] <= mat\_out[i\*16 +: 16];

inter\_result <= mat\_out;

state <= MUL2;

// For the B operand: In stage MUL2 we use D (from bram\_D)

for(i = 0; i < 64; i = i + 1)

mat\_B[i\*16 +: 16] = bram\_D[i];

end

end

MUL2: begin

if(mult2\_done) begin

block\_out <= final\_result;

state <= DONE;

end

end

DONE: begin

done <= 1;

state <= DONE;

block\_out <= block\_out;

end

endcase

end

end

endmodule

1. ***matmul.v***

`timescale 1ns / 1ps

module matrix\_mult\_8x8\_dsp #(

parameter TRANSPOSE\_B = 0,

QUANTIZE = 0

) (

input clk,

input s\_clk,

input reset,

input enable,

input [1023:0] A, // 8x8 matrix, each element 16-bit Q11.5

input [1023:0] B,

output reg [1023:0] C,

output reg done

);

// 2D arrays for the matrices (signed Q11.5 numbers)

reg signed [15:0] matA [0:7][0:7];

reg signed [15:0] matB [0:7][0:7];

// 32-bit accumulators in Q22.10 format (result of Q11.5\*Q11.5 multiplication)

reg signed [31:0] accum [0:7][0:7];

// Control signals and states

reg [3:0] k; // dot product index

reg [2:0] state;

localparam IDLE = 3'b000,

LOAD = 3'b001,

COMPUTE = 3'b010,

DONE\_ST = 3'b011,

READY = 3'b100;

integer i, j;

// Quantization Matrix (Q)

reg [2:0] Q\_mat [0:7][0:7];

// Initialize Q\_mat

initial begin

Q\_mat[0][0] = 3'd4; Q\_mat[0][1] = 3'd3; Q\_mat[0][2] = 3'd3; Q\_mat[0][3] = 3'd4;

Q\_mat[0][4] = 3'd5; Q\_mat[0][5] = 3'd5; Q\_mat[0][6] = 3'd6; Q\_mat[0][7] = 3'd6;

Q\_mat[1][0] = 3'd3; Q\_mat[1][1] = 3'd3; Q\_mat[1][2] = 3'd4; Q\_mat[1][3] = 3'd4;

Q\_mat[1][4] = 3'd5; Q\_mat[1][5] = 3'd6; Q\_mat[1][6] = 3'd6; Q\_mat[1][7] = 3'd6;

Q\_mat[2][0] = 3'd4; Q\_mat[2][1] = 3'd4; Q\_mat[2][2] = 3'd4; Q\_mat[2][3] = 3'd5;

Q\_mat[2][4] = 3'd5; Q\_mat[2][5] = 3'd6; Q\_mat[2][6] = 3'd6; Q\_mat[2][7] = 3'd6;

Q\_mat[3][0] = 3'd4; Q\_mat[3][1] = 3'd4; Q\_mat[3][2] = 3'd4; Q\_mat[3][3] = 3'd5;

Q\_mat[3][4] = 3'd6; Q\_mat[3][5] = 3'd6; Q\_mat[3][6] = 3'd6; Q\_mat[3][7] = 3'd6;

Q\_mat[4][0] = 3'd4; Q\_mat[4][1] = 3'd4; Q\_mat[4][2] = 3'd5; Q\_mat[4][3] = 3'd6;

Q\_mat[4][4] = 3'd6; Q\_mat[4][5] = 3'd7; Q\_mat[4][6] = 3'd7; Q\_mat[4][7] = 3'd6;

Q\_mat[5][0] = 3'd5; Q\_mat[5][1] = 3'd5; Q\_mat[5][2] = 3'd6; Q\_mat[5][3] = 3'd6;

Q\_mat[5][4] = 3'd6; Q\_mat[5][5] = 3'd7; Q\_mat[5][6] = 3'd7; Q\_mat[5][7] = 3'd6;

Q\_mat[6][0] = 3'd6; Q\_mat[6][1] = 3'd6; Q\_mat[6][2] = 3'd6; Q\_mat[6][3] = 3'd6;

Q\_mat[6][4] = 3'd7; Q\_mat[6][5] = 3'd7; Q\_mat[6][6] = 3'd7; Q\_mat[6][7] = 3'd7;

Q\_mat[7][0] = 3'd6; Q\_mat[7][1] = 3'd6; Q\_mat[7][2] = 3'd6; Q\_mat[7][3] = 3'd6;

Q\_mat[7][4] = 3'd7; Q\_mat[7][5] = 3'd7; Q\_mat[7][6] = 3'd7; Q\_mat[7][7] = 3'd7;

end

// Main state machine using DSP slices for MAC operations

always @(posedge s\_clk or posedge reset) begin

if (reset) begin

state <= IDLE;

done <= 0;

k <= 0;

for (i = 0; i < 8; i = i + 1)

for (j = 0; j < 8; j = j + 1)

accum[i][j] <= 32'd0;

end else begin

case (state)

IDLE: if (enable) state <= LOAD;

LOAD: begin

for (i = 0; i < 8; i = i + 1) begin

for (j = 0; j < 8; j = j + 1) begin

matA[i][j] <= A[(i\*8+j)\*16 +: 16];

matB[i][j] <= B[(i\*8+j)\*16 +: 16];

end

end

state <= COMPUTE;

k <= 0;

end

COMPUTE: begin

// When multiplying two Q11.5 numbers, the product is Q22.10.

for (i = 0; i < 8; i = i + 1) begin

for (j = 0; j < 8; j = j + 1) begin

if (TRANSPOSE\_B)

accum[i][j] <= accum[i][j] + (matA[i][k] \* matB[j][k]);

else

accum[i][j] <= accum[i][j] + (matA[i][k] \* matB[k][j]);

end

end

if (k == 7) begin

state <= DONE\_ST;

k <= 0;

end else begin

k <= k + 1;

end

end

DONE\_ST: begin

// Convert from Q22.10 to Q11.5 by discarding 5 fractional bits.

// Here we select bits [20:5] of the accumulator for each element.

for (i = 0; i < 8; i = i + 1) begin

for (j = 0; j < 8; j = j + 1) begin

if (QUANTIZE)

C[(i\*8+j)\*16 +: 16] <= $signed(accum[i][j][20:5]) >>> Q\_mat[i][j];

else

C[(i\*8+j)\*16 +: 16] <= accum[i][j][20:5];

end

end

state <= READY;

end

READY: begin

done <= 1;

state <= READY;

end

endcase

end

end

endmodule

1. ***imrx.v***

`timescale 1ns / 1ps

module imrx(

input clk, //input clock

input reset, //input reset

input ena,

input RxD, //input receving data line

input [13:0] addr, //address to infer

input [7:0] dout,

output reg ImRxComplete,

output ena\_imrx,

output wea\_imrx,

output [7:0]din\_imrx,

output [13:0]addr\_imrx

);

//internal variables

reg [7:0] din;

reg wea;

reg shift; // shift signal to trigger shifting data

reg state, nextstate; // initial state and next state variable

reg [3:0] bitcounter; // 4 bits counter to count up to 9 for UART receiving

reg [1:0] samplecounter; // 2 bits sample counter to count up to 4 for oversampling

reg [13:0] counter; // 14 bits counter to count the baud rate

reg [13:0] addr1;

reg [9:0] rxshiftreg; //bit shifting register

reg [7:0]RxData;

reg clear\_bitcounter,inc\_bitcounter,inc\_samplecounter,clear\_samplecounter,inc\_bytecounter; //clear or increment the counter

//reg [7:0] addr1;

// constants

parameter clk\_freq = 100\_000\_000; // system clock frequency

parameter baud\_rate = 9\_600; //baud rate

parameter div\_sample = 4; //oversampling

parameter div\_counter = clk\_freq/(baud\_rate\*div\_sample); // this is the number we have to divide the system clock frequency to get a frequency (div\_sample) time higher than (baud\_rate)

parameter mid\_sample = (div\_sample/2); // this is the middle point of a bit where you want to sample it

parameter div\_bit = 10; // 1 start, 8 data, 1 stop

//uncomment the following bram to run imrx as a standalone module

//blk\_mem\_gen\_0 uut(.clka(clk), .ena(ena), .wea(wea), .addra(addr1), .dina(din), .douta(dout));

//the following assignment is the controlling logic for the bram

assign ena\_imrx = ena;

assign wea\_imrx = wea;

assign addr\_imrx = addr1;

assign din\_imrx = din;

//assign dout = dout;

//assign RxData = rxshiftreg [8:1]; // assign the RxData from the shiftregister

//assign data = file[address];

//UART receiver logic

always @ (posedge clk)

begin

if (reset)begin // if reset is asserted

state <=0; // set state to idle

bitcounter <=0; // reset the bit counter

counter <=0; // reset the counter

addr1 = addr;

samplecounter <=0; // reset the sample counter

end else begin // if reset is not asserted

counter <= counter +1; // start count in the counter

if (counter >= div\_counter-1) begin // if counter reach the baud rate with sampling

counter <=0; //reset the counter

state <= nextstate; // assign the state to nextstate

if (shift)rxshiftreg <= {RxD,rxshiftreg[9:1]}; //if shift asserted, load the receiving data

if (clear\_samplecounter) samplecounter <=0; // if clear sampl counter asserted, reset sample counter

if (inc\_samplecounter) samplecounter <= samplecounter +1; //if increment counter asserted, start sample count

if (clear\_bitcounter) bitcounter <=0; // if clear bit counter asserted, reset bit counter

if (inc\_bitcounter)bitcounter <= bitcounter +1; // if increment bit counter asserted, start count bit counter

if (inc\_bytecounter)addr1 <= addr1 +1;

end

end

end

//state machine

always @ (posedge clk) //trigger by clock

begin

shift <= 0; // set shift to 0 to avoid any shifting

clear\_samplecounter <=0; // set clear sample counter to 0 to avoid reset

inc\_samplecounter <=0; // set increment sample counter to 0 to avoid any increment

inc\_bytecounter <=0;

// ImRxComplete<=0;

wea<=0;

clear\_bitcounter <=0; // set clear bit counter to 0 to avoid claring

inc\_bitcounter <=0; // set increment bit counter to avoid any count

nextstate <=0; // set next state to be idle state

case (state)

0: begin // idle state

if (RxD) // if input RxD data line asserted

begin

nextstate <=0; // back to idle state because RxD needs to be low to start transmission

end

else begin // if input RxD data line is not asserted

nextstate <=1; //jump to receiving state

clear\_bitcounter <=1; // trigger to clear bit counter

clear\_samplecounter <=1; // trigger to clear sample counter

end

end

1: begin // receiving state

nextstate <= 1; // DEFAULT

if (samplecounter== mid\_sample - 1) shift <= 1; // if sample counter is 1, trigger shift

if (samplecounter== div\_sample - 1) begin // if sample counter is 3 as the sample rate used is 3

if (bitcounter == div\_bit - 1) begin // check if bit counter if 9 or not

if (addr1<16383)begin

ImRxComplete<=0;

nextstate <= 0; // back to idle state if bit counter is 9 as receving is complete

din<=rxshiftreg [8:1];

wea<=1;

inc\_bytecounter <= 1;

end else begin ImRxComplete<=1; end

end

inc\_bitcounter <=1; // trigger the increment bit counter if bit counter is not 9

clear\_samplecounter <=1; //trigger the sample counter to reset the sample counter

end else inc\_samplecounter <=1; // if sample is not equal to 3, keep counting

end

default: nextstate <=0; //default idle state

endcase

end

endmodule

1. ***imtx.v***

`timescale 1ns / 1ps

module imtx(

input clk, //UART input clock

input reset, // reset signal

input transmit, //btn signal to trigger the UART communication

output [7:0] data, // data transmitted

output reg TxD, // Transmitter serial output. TxD will be held high during reset, or when no transmissions are taking place

output reg done, // Signal to indicate transmission complete

output ena\_tx,

output wea\_tx,

output [14:0] addr\_tx,

output [7:0] din\_tx,

input [7:0] dout\_tx

);

//internal variables

reg [14:0] address = 0; // Updated from 14-bit to 15-bit register

reg donef;

reg inc\_addr;

reg [3:0] bitcounter; //4 bits counter to count up to 10

reg [13:0] counter; //14 bits counter to count the baud rate, counter = clock / baud rate

reg state, nextstate; // initial & next state variable

// 10 bits data needed to be shifted out during transmission.

// The least significant bit is initialized with the binary value 0 (a start bit)

// A binary value 1 is introduced in the most significant bit

reg [9:0] rightshiftreg;

wire [7:0] dout;

reg shift; //shift signal to start bit shifting in UART

reg load; //load signal to start loading the data into rightshift register and add start and stop bit

reg clear; //clear signal to start reset the bitcounter for UART transmission

//uncomment the following bram to use imtx as a standalone image transmitter module

//blk\_mem\_gen\_0 uut3(.clka(clk), .ena(1), .wea(0), .addra(address), .dina(0), .douta(dout));

//the following assignment is for the controlling logic of the bram

assign ena\_tx = 1;

assign wea\_tx = 0;

assign addr\_tx = address;

assign din\_tx = 0;

assign dout = dout\_tx;

assign data = dout;

//UART transmission logic

always @(posedge clk) begin

if (reset) begin // reset is asserted (reset = 1)

state <= 0; // state is idle (state = 0)

counter <= 0; // counter for baud rate is reset to 0

bitcounter <= 0; // counter for bit transmission is reset to 0

done <= 0;

end else begin

counter <= counter + 1; // counter for baud rate generator start counting

if (counter >= 10415) begin // if count to 10416 (from 0 to 10415)

state <= nextstate; // previous state change to next state

counter <= 0; // reset counter to 0

if (load) rightshiftreg <= {1'b1, data, 1'b0}; // load the data if load is asserted

if (clear) bitcounter <= 0; // reset the bitcounter if clear is asserted

if (inc\_addr) address <= address + 1;

// if (donef) done<=1; else done<=0;

if (shift) begin // if shift is asserted

rightshiftreg <= rightshiftreg >> 1; // right shift the data as we transmit the data from lsb

bitcounter <= bitcounter + 1; // count the bitcounter

end

end

end

end

//state machine

always @(posedge clk) begin

load <= 0; // set load equal to 0 at the beginning

shift <= 0; // set shift equal to 0 at the beginning

clear <= 0; // set clear equal to 0 at the beginning

TxD <= 1; // set TxD high during no transmission

inc\_addr <= 0;

done <= 0; // default done is 0

case (state)

0: begin // idle state

if (transmit) begin // assert transmit input

nextstate <= 1; // move to transmit state

load <= 1; // set load to 1 to prepare to load the data

shift <= 0; // set shift to 0 so no shift ready yet

clear <= 0; // set clear to 0 to avoid clear any counter

end else begin // if transmit not asserted

nextstate <= 0; // next state is back to idle state

TxD <= 1;

end

end

1: begin // transmit state

if (bitcounter >= 10) begin // check if transmission is complete or not. If complete

nextstate <= 0; // set nextstate back to idle state

clear <= 1; // set clear to 1 to clear all counters

if(address < 32767) begin

inc\_addr <= 1;

end else begin

// When the last address is reached, signal done for the top module.

done <= 1;

end

end else begin // if transmission is not complete

nextstate <= 1; // set nextstate to 1 to stay in transmit state

TxD <= rightshiftreg[0]; // shift the bit to output TxD

shift <= 1; // set shift to 1 to continue shifting the data

end

end

default: nextstate <= 0;

endcase

end

endmodule

1. ***slow\_clock.v***

module slow\_clock\_gen #(

parameter DIV\_FACTOR = 1\_000 // Set to divide input clock frequency

)(

input wire clk, // Fast input clock (e.g., 100 MHz)

output reg slow\_clk // Slower output clock

);

reg [$clog2(DIV\_FACTOR)-1:0] counter = 0;

always @(posedge clk) begin

if (counter == DIV\_FACTOR - 1) begin

counter <= 0;

slow\_clk <= ~slow\_clk;

end else begin

counter <= counter + 1;

end

end

endmodule