باسمه تعالی

دانشگاه صنعتی شریف

دانشکده مهندسی برق

طراحی سیستم های مبتنی بر FPGA/ASIC

پروژه نهایی: تشخیص لبه تصاویر با الگوریتم Sobel فاز ۴ + شبیه سازی + امتیازی

دانشجو: سیدسعید جزائری - علی خدنگی

شماره دانشجویی: ۹۸۱۰۴۸۸۵ - ۹۸۱۰۱۴۹۳

فهرست مطالب

فهرست تصاویر

۱ مقدمه

در فاز چهارم به سه کار اصلی پرداخته ایم. اولا که سه فاز اولیه را به نحوی با هم تجمیع کرده ایم که بتوانیم در ارائه، آن را پروگرم کرده و خروجی بگیریم. به این نحو که داده ها به صورت سریال با پروتکل UART از کامپیوتر به سمت FPGA ارسال میشوند، این داده ها که اطلاعات تصویر هستند از ابتدا در مموری ذخیره میشوند، سپس از rgb2gray و sobel عبور میکنند و لبه های تصویر یافت میشوند. حال تصویر رنگی اصلی به همراه لبه های آن به همراه یک تایتل بر روی مانیتور با Interface موجود یعنی VGA بهن نمایش در خواهند آمد.

تمام موارد فوق و کد ها، پین اساینمنت ها و ... در پوشه phase4 موجود است. در این پروژه، فایل های پیشین بهمراه فایل های UART Receiver و همچنین فایل های متلبی که یک تصویر را انکود کرده و از طریق COM Port میفرستند آورده شده اند.

اما برای شبیه سازی این فاز، با توجه به اینکه برد در دسترس نیست، ماژول های VGA و lpmram بی کاربرد میشوند. برای همین باید چاره دیگری اندیشید.

برای حل این مشکل، فرایند زیر را انتخاب کرده ایم: ابتدا در ماژول تست بنچ، داده های یک تصویر را -که قبلا در اسکریپت متلب آن را به صورت ۳بیت تک رنگی به همراه crc تولید کرده ایم- را میفرستیم. در ماژول receiver این داده ها را پردازش کرده و بیت های حاوی اطلاعات آن را استخراج کرده و به ماژول rgb2gray میدهیم و سپس خروجی این ماژول به خروجی sobel پاس داده میشود تا لبه ها به دست آیند. در این فرایند اطلاعات تصویر خاکستری و تصویر لبه را در فایل هایی مینویسیم تا بتوانیم به نوعی جبران نبود VGA را با پلات کردن آن ها کنیم.

۲ شبیه سازی

برای شبیه سازی، کد وریلاگ زیر نوشته شده است:

```
1
2  module TOP_tb();
3
4  parameter CLK_to_BAUD = 10;
5
6
7  reg CLOCK_50;
8  reg resetn = 0;
9  reg UART_RXD = 1;
10 reg[7:0] clk_cntr = 0;
11 reg[7:0] UART_BUFF[199:0][159:0];
12
13 integer fd;
14 integer i, j, k;
15
16 initial CLOCK_50 = 1;
17 always @(CLOCK_50)
18     CLOCK_50 <= #5 ~CLOCK_50;
19
20 initial begin
21       $readmemh("C:/Users/padidar/Desktop/Semester7/FPGA/Project/XilinxSimulation/phase4Simulation/
22                 rgb1color.mem", UART_BUFF);
23 end
24
25 initial begin
26     resetn = 0;
27     @(posedge CLOCK_50);
28     @(posedge CLOCK_50);
29     @(posedge CLOCK_50);
30
31     resetn = 1;
32     for(i=0; i<200; i=i+1) begin //row cntr
33         for(j=0; j<160; j=j+1) begin //col cntr
34             for(k=0; k<10; k=k+1) begin   // bit data cntr
35
36                 for(clk_cntr=0; clk_cntr<CLK_to_BAUD; clk_cntr=clk_cntr+1) begin
37                     @(posedge CLOCK_50);
38                 end
39
40
41                 case(k)
42                     0:  UART_RXD = 0;
43                     1:  UART_RXD = UART_BUFF[i][j][0];
```

```
                    2:  UART_RXD = UART_BUFF[i][j][1];
                    3:  UART_RXD = UART_BUFF[i][j][2];
                    4:  UART_RXD = UART_BUFF[i][j][3];
                    5:  UART_RXD = UART_BUFF[i][j][4];
                    6:  UART_RXD = UART_BUFF[i][j][5];
                    7:  UART_RXD = UART_BUFF[i][j][6];
                    8:  UART_RXD = UART_BUFF[i][j][7];
                    9:  UART_RXD = 1;
                endcase
            end
        end
    end

    while (TOP_uut.my_sketch.ready_gray != 1) begin
        @(posedge CLOCK_50);
    end

    for (i=0; i<200; i=i+1) begin
        for (j=0; j<160; j=j+1) begin
            TOP_uut.my_sketch.my_sobel.ROM_Gray[199-i][159-j] =
            TOP_uut.my_sketch.my_rgb2gray.ROM_Gray[i][j];
      end
   end

    $finish;
end


TOP TOP_uut(
  .CLOCK_50(CLOCK_50),
  .resetn(resetn),
  .UART_RXD(UART_RXD)

);

endmodule
```

ماژول های TOP و زیر ماژول های آن را توضیح نمیدهیم. کلیت عملکرد آن ها همان است که در فاز های قبلی توضیح داده ایم بغیر از تغییراتی که بواسطه نحوه شبیه سازی ممکن است در ایشان ایجاد کرده باشیم.

در کد تست بنچ واضح است که ابتدا یک مموری 160x120 را مقدار دهی اولیه کرده ایم. این مموری با داده های یک تصویر پر شده است. اسکریپت متلب زیر این کار را (تبدیل یک تصویر به مموری فایل با ۳ بیت رنگی به

همراه crc) را انجام میدهد:

```
clear;
clc;

input_image = imread('my_pic.jpg');


r = input_image(:,:,1);
g = input_image(:,:,2);
b = input_image(:,:,3);
file_id = fopen ('rgb1color.txt', 'w');


for i=1:200
   for j=1:160
       rv = r(i,j) > 127;
       gv = g(i,j) > 127;
       bv = b(i,j) > 127;
       numToWrite = [rv, gv, bv, 0];
       packet = crc_encoder(numToWrite);
       t = 7:-1:0;
       x = sum(packet .* (2.^t));
       formatSpec = '%c%c ';
       fprintf(file_id, formatSpec, dec2hex(x,2)); %or * 16
   end
       fprintf(file_id ,'\n');

end
```

حال همانطور که در کد تست بنچ نیز واضح است، این داده ها با Baudrate دلخواه به صورت سریال از تست بنچ به TOP فرستاده میشوند. در آن جا ماژول Receiver داده ها ی سریال را دریافت میکند و با بررسی CRC آن ها خروجی داده های معتبر را برای ما تولید میکند.

این خروجی ها دارای ۳ بیت مهم، یعنی تک بیت متناظر با R، تک بیت متناظر با G و تک بیت متناظر با B است. پس از این سایر زیر ماژول ها عملیات خودشان را مطابق پیش انجام میدهند.

از آنجا که این یک فایل شبیه سازی است و استفاده از System Task ها در آن مجاز و منطقی است، در جای جای آن موارد مورد نیاز را در فایل Dump کرده ایم تا از آن ها استفاده کنیم.

خروجی ها را پلات میکنیم و با خروجی های قبلی مقایسه میکنیم:

الف: تصویر خاکستری شده:

شکل ۱: تصویر خاکستری شده با داده های ۱ بیتی در وریلاگ

شکل ۲: تصویر خاکستری شده با داده های ۸ بیتی در وریلاگ

شکل ۳: تصویر خاکستری شده با داده های ۸ بیتی در متلب

واضح است که تصویر خاکستری شده در وریلاگ با ۱ بیت داده رنگی، افت کیفیت قابل توجهی به نصب وریلاگ ۸ بیتی یا شبیه سازی متلب دارد.

ب: تصاویر لبه:

شکل ۴: تصویر لبه با داده های ۱ بیتی در وریلاگ

شکل ۵: تصویر لبه با داده های ۸ بیتی در وریلاگ

شکل ۶: تصویر لبه با داده های ۸ بیتی در متلب

واضح است که تصویر لبه در وریلاگ با ۱ بیت داده رنگی، افت کیفیت چندان قابل توجهی به وریلاگ ۸ بیتی ندارد اما به نسبت متلب که محاسبات دابل دارد این اختلاف چشمگیر است.

۳ برنامه برای پیاده سازی

در توضیحات فوق، شبیه سازی کامل را در حالتی که داده های ما ۱ بیت به ازای هر رنگ، و به همراه CRC کد شده است را انجام دادیم.

پروژه ای که در پوشه phase4 قرار دارد، پیاده سازی دقیقا همین شبیه سازی بر روی برد DE2 میباشد که امیدواریم در روز ارائه بدون مشکل پیاده سازی شود.

همچنین یک فایل دیگر از پروژه ساخته شده است که در آن داده های سریالی که ارسال میکنیم، به صورت ۲ بیت به ازای هر رنگ میباشد که واضحا دقت کار ما را بسیار بیشتر میکند. اما در این پروژه CRC ارسال نمیکنیم و هدف صرفا افزایش کیفیت تصاویر خروجی با توجه به افزایش تعداد بیت هر رنگ میباشد.

در صورت موفقیت آمیز بودن هر دو قسمت قبل، در همان جلسه آزمایشگاه ارسال داده ۶ بیتی به همراه CRC را بررسی میکنیم اما واضح است که خواسته های پروژه در همان دو قسمت قبلی براورده شده اند.

۴ Viterby

در این قسمت الگوریتم Viterby که قابلیت Error Correction دارد را پیاده سازی کرده ایم.

۱.۴ مقدمه

در این فاز یک ماژول ViterbiDecoder پیادهسازی کردهایم. و از ماژول transmitter و ماژول receiver که در فاز سوم پیادهسازی کرده بودیم استفاده کردیم تا آن را تست کنیم. همچنین با اندکی تغییر از تست بنچ نوشته شده در فاز قبل tb receiver نیز در این جا برای تست ماژول Viterbi استفاده کردیم. توضیحات مربوط به ماژولهای receiver و transmitter که در فاز قبلی بود را نیز در اینجا آوردهایم

۲.۴ ماژول ViterbiDecoder

این ماژول همانند ماژول دیکودر CRC که در فاز سوم پیادهسازی کردیم، یک سیگنال start و یک دیتای ۸ بیتی ورودی میگیرد. همچنین بر خالف CRC که فقط کار تشخیص خطا را انجام میداد، Viterbi میتواند خطا را تصحیح کند. به همین دلیل در اینجا سیگنال خروجی valid دیگر وجود ندارد و فقط خروجی outputData و ready را برای ماژول قرار داده ایم. کلیت کار ماژول به این شکل است که هر state را (۴ state داریم) به صورت یک راس در هر مرحله در نظر میگیریم. و با توجه به کودین گ کانولوشنی که در متلب روی دیتا اعم ال کردهایم، یالهای گراف را میسازیم و توسط الگوریتم Viterbi سعی میکنیم که دیتا را استخراج کنیم. در هر

مرحله cost هر راس را حساب کرده و state قبلی که این راس در این مر حله از روی آن آپ دیت شده است را در رجیستر lastState قرار میدهیم. به این شکل در آخر پس از ۴ کال ک سایکل دیتای خروجی را از روی این مقادیر میسازیم.

```
module ViterbiDecoder(
    input clk,
    input start,
    input [7:0] inputData,
    output reg [3:0] outputData,
    output reg ready
);
reg [1:0] error [1:0][1:0];
reg [3:0] cost [3:0];
reg [1:0] lastState [3:0][3:0];
reg[7:0] inputTemp;
reg[2:0] counter;
integer i;
integer j;
always @(posedge clk) begin
    if(!start) begin
    outputData <= 0;
    inputTemp <= 0;
    counter <= 0;
    ready <= 0;
    for(i = 0; i < 4; i = i + 1)
    for(j = 0; j < 4; j = j + 1)
    lastState[i][j] <= 0;
    cost[0] <= 0;
    for(i = 1; i < 4; i = i + 1)
    cost[i] <= 4'b111;
end else begin
    if(counter == 0) begin
    counter <= counter + 1;
    inputTemp <= inputData;
end else if(counter < 5) begin
    counter <= counter + 1;
    inputTemp <= inputTemp >> 2;
    if((cost[0] + error[0][0]) < (cost[2] + error[1][1])) begin
        cost[0] <= cost[0] + error[0][0];
        lastState[0][counter-1] <= 0;
    end else begin
        cost[0] <= cost[2] + error[1][1];
        lastState[0][counter-1] <= 2;
    end
    if((cost[0] + error[1][1]) < (cost[2] + error[0][0])) begin
        cost[1] <= cost[0] + error[1][1];
        lastState[1][counter-1] <= 0;
```

```
        end else begin
            cost[1] <= cost[2] + error[0][0];
            lastState[1][counter-1] <= 2;
        end
        if((cost[1] + error[0][1]) < (cost[3] + error[1][0])) begin
            cost[2] <= cost[1] + error[0][1];
            lastState[2][counter-1] <= 1;
        end else begin
            cost[2] <= cost[3] + error[1][0];
            lastState[2][counter-1] <= 3;
        end
        if((cost[1] + error[1][0]) < (cost[3] + error[0][1])) begin
            cost[3] <= cost[1] + error[1][0];
            lastState[3][counter-1] <= 1;
        end else begin
            cost[3] <= cost[3] + error[0][1];
            lastState[3][counter-1] <= 3;
        end
    end else begin
        ready <= 1;
        if(cost[0] < cost[1]) begin
            if(cost[0] < cost[2]) begin
                if(cost[0] < cost[3]) begin
                    outputData <= {
                    lastState[lastState[lastState[0][3]][2]][1][0],
                    lastState[lastState[0][3]][2][0],
                    lastState[0][3][0],
                    1'b0};
                end else begin
                    outputData <= {
                    lastState[lastState[lastState[3][3]][2]][1][0],
                    lastState[lastState[3][3]][2][0],
                    lastState[3][3][0],
                    1'b1};
                end
            end else begin
                if(cost[2] < cost[3]) begin
                    outputData <= {
                    lastState[lastState[lastState[2][3]][2]][1][0],
                    lastState[lastState[2][3]][2][0],
                    lastState[2][3][0],
                    1'b0};
                end else begin
                    outputData <= {
                    lastState[lastState[lastState[3][3]][2]][1][0],
                    lastState[lastState[3][3]][2][0],
                    lastState[3][3][0],
```

```
                    1'b1};
                end
            end
        end else begin
            if(cost[1] < cost[2]) begin
                if(cost[1] < cost[3]) begin
                    outputData <= {
                    lastState[lastState[lastState[1][3]][2]][1][0],
                    lastState[lastState[1][3]][2][0],
                    lastState[1][3][0],
                    1'b1};
                end else begin
                    outputData <= {
                    lastState[lastState[lastState[3][3]][2]][1][0],
                    lastState[lastState[3][3]][2][0],
                    lastState[3][3][0],
                    1'b1};
                end
            end else begin
                if(cost[2] < cost[3]) begin
                    outputData <= {
                    lastState[lastState[lastState[2][3]][2]][1][0],
                    lastState[lastState[2][3]][2][0],
                    lastState[2][3][0],
                    1'b0};
                end else begin
                    outputData <= {
                    lastState[lastState[lastState[3][3]][2]][1][0],
                    lastState[lastState[3][3]][2][0],
                    lastState[3][3][0],
                    1'b1};
                end
            end
        end
    end
end
end
always @(*) begin
error[0][0] = {0, inputTemp[1]} + {0, inputTemp[0]};
error[1][1] = {0, ~inputTemp[1]} + {0, ~inputTemp[0]};
error[0][1] = {0, inputTemp[1]} + {0, ~inputTemp[0]};
error[1][0] = {0, ~inputTemp[1]} + {0, inputTemp[0]};
end
endmodule
```

۳.۴ ماژول transmitter

این ماژول را برای تست این فاز پیادهسازی کردهایم. متغیر COUNTER OF END تعداد کالک الزم برای دریافت دیتا را مشخص میکند برای مثال برای این که با 115200 rate baud دیتا را دریافت کنیم و کالک 50M داشته باشیم باید این پارامتر را 434 قرار دهیم. در این فاز برای تست این پارامتر را روی ۱۰ قرار میدهیم. دیتای ورودی روی سیگنال data که ۸ بیتی است قرار میگیرد و سیگنال start نیز برای شروع کار باید یک شده و یک بماند. پس از اتمام کار سیگنال done یک میشود. همچنین دیتا بر روی سیگنال TX فرستاده میشود.

```
module transmitter(
    input clk,
    input start,
    input [7:0] data,
    output reg TX,
    output reg done
    );
    parameter END_OF_COUNTER = 10;

    reg [3:0] i;
    wire [9:0] temp = {1'b1 , data , 1'b0};
    reg [7:0] counter; //determines Transmitting Rate ( in this code : 115200)
    reg running;

    always @(posedge clk)begin
        if(!start) begin
            counter <= 0;
            TX <= 1;
            i <= 0;
            done <= 0;
            running <= 0;
        end else if(!done) begin
            running <= 1;
        end

        if(running) begin
            counter <= counter + 1;
            if(counter == END_OF_COUNTER) begin
                i <= i + 1 ;
                TX <= temp [i];
                counter <= 0;
                if(i == 9)begin
                    i <= 0;
                    done <= 1;
                    running <= 0;
                end
            end
```

```
        end
    end

endmodule
```

۴.۴ ماژول receiver

این ماژول اصلی این فاز است. این ماژول یک سیگنال start دارد برای شروع به کار این ماژول باید این سیگنال را یک کنیم . همچنین ورودی این ماژول از سیگنال rx میآید. همان پارامتر COUTNER OF END در این ماژول نیز وجود دارد و همان موارد ماژول transmitter در اینجا هم صدق میکند. یک instance از ماژول CRC نیز در این ماژول گرفته شده است و سیگنال start این ماژول را با crcStart مشخص کردهایم. پس از دریافت کامل پکت، این سیگنال را یک میکنیم. سپس چند کالک طول میکشد که ماژول CRC دیتای خروجی را حساب کند. سیگنالهای ☒valid ready و outputData از ماژول CRC در خروجی ماژول receiver قرار گرفته اند

```
module receiver(
    input clk,
    input start,
    input rx,
    output ready,
    output[3:0] outputData
    );
    parameter END_OF_COUNTER = 10;

    reg state;
    reg [8:0] data;
    reg [7:0] counter;
    reg [3:0] i;

    reg crcStart;

    always@(posedge clk) begin
        if(!start) begin
            data <= 0;
            counter <= 0;
            i <= 0;
            state <= 0;
            crcStart <= 0;
        end else begin
            if(!state) begin
```

```
                if(rx == 0)
                    state <= 1;
            end if(state) begin
                counter <= counter + 1;
                if(counter == END_OF_COUNTER) begin
                    i <= i + 1;
                    data[i] <= rx;
                    counter <= 0;
                    if(i == 8) begin
                        state <= 0;
                        i <= 0;
                        crcStart <= 1;
                    end else
                        crcStart <= 0;
                end
            end
        end

    ViterbiDecoder decoder(
        .clk(clk),
        .start(crcStart),
        .inputData(data[7:0]),
        .ready(ready),
        .outputData(outputData)
    );

endmodule
```

۵.۴ شبیه سازی

برای شبیهسازی ماژول tb receiver که در فاز قبل پیادهسازی کردهبودیم را اندکی تغییر دادیم تا به صورت خودکار خطای پیادهسازی را از روی مقاسیه data raw و دیتای خروجی بدست آمده، پیدا کند.

```
`define N 100

module receiver_tb;

    reg clk = 0;
    reg startT;
    reg startR;
    reg[7:0] inputData;

```

```
10      wire doneT;
11
12
13      wire data_wire;
14      wire readyR;
15      wire[3:0] outputData;
16
17      always@(clk)
18          clk <= #5 ~clk;
19
20      integer numberOfErros = 0;
21      integer i;
22      integer file;
23      reg[`N-1:0] raw_data;
24      reg[2*`N-1:0] encoded_data;
25
26      initial begin
27          file = $fopen("raw_data.mem", "r");
28          $fscanf(file, "%b\n", raw_data);
29          $fclose(file);
30
31          file = $fopen("encoded_data.mem", "r");
32          $fscanf(file, "%b\n", encoded_data);
33          $fclose(file);
34      end
35
36      initial begin
37          startT = 0;
38          startR = 0;
39          inputData = 0;
40          @(posedge clk);
41
42
43          for(i = 0; i < 2 * `N - 8; i = i + 8) begin
44              startT = #2 1;
45              startR = 1;
46              inputData[7] = encoded_data[i+7];
47              inputData[6] = encoded_data[i+6];
48              inputData[5] = encoded_data[i+5];
49              inputData[4] = encoded_data[i+4];
50              inputData[3] = encoded_data[i+3];
51              inputData[2] = encoded_data[i+2];
52              inputData[1] = encoded_data[i+1];
53              inputData[0] = encoded_data[i];
54              @(posedge doneT);
55
56              @(posedge readyR);
```

```
            startT = #2 0;
            $write("output data = %b\n", outputData);
            if(raw_data[3:0] !== outputData) begin
                numberOfErros = numberOfErros + 1;
                $write("ERROR: expected %b but the output is %b\n", raw_data[3:0], outputData);
            end
            raw_data = raw_data >> 4;

            @(posedge clk);
        end

        if(numberOfErros > 0)
            $write("Number of erros = %d\n", numberOfErros);
        else
            $write("All tests passed\n");

        #20;
        $finish;
    end

    receiver receiverModule(
        .clk(clk),
        .start(startR),
        .rx(data_wire),
        .ready(readyR),
        .outputData(outputData)
    );

    transmitter transmitterModule(
        .clk(clk),
        .start(startT),
        .data(inputData),
        .TX(data_wire),
        .done(doneT)
    );

endmodule
```

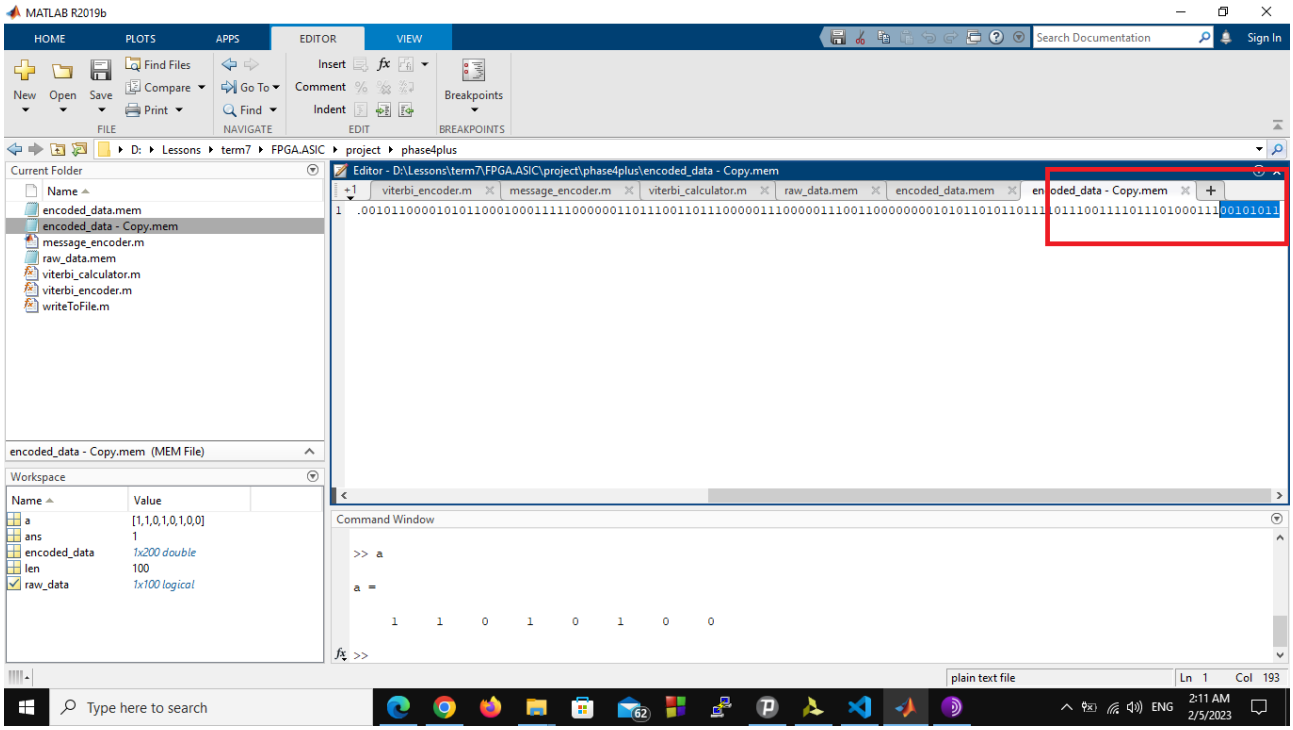
۶.۴ متلب

در متلب یک دیتای ۱۰۰ بیتی را به صورت تصادفی ایجاد کردهایم و Viterbi را برای هر دیتای ۴ تایی حساب میکنیم و به هم میچسبانیم. سپس این دو دیتا را در فایل مینویسیم.

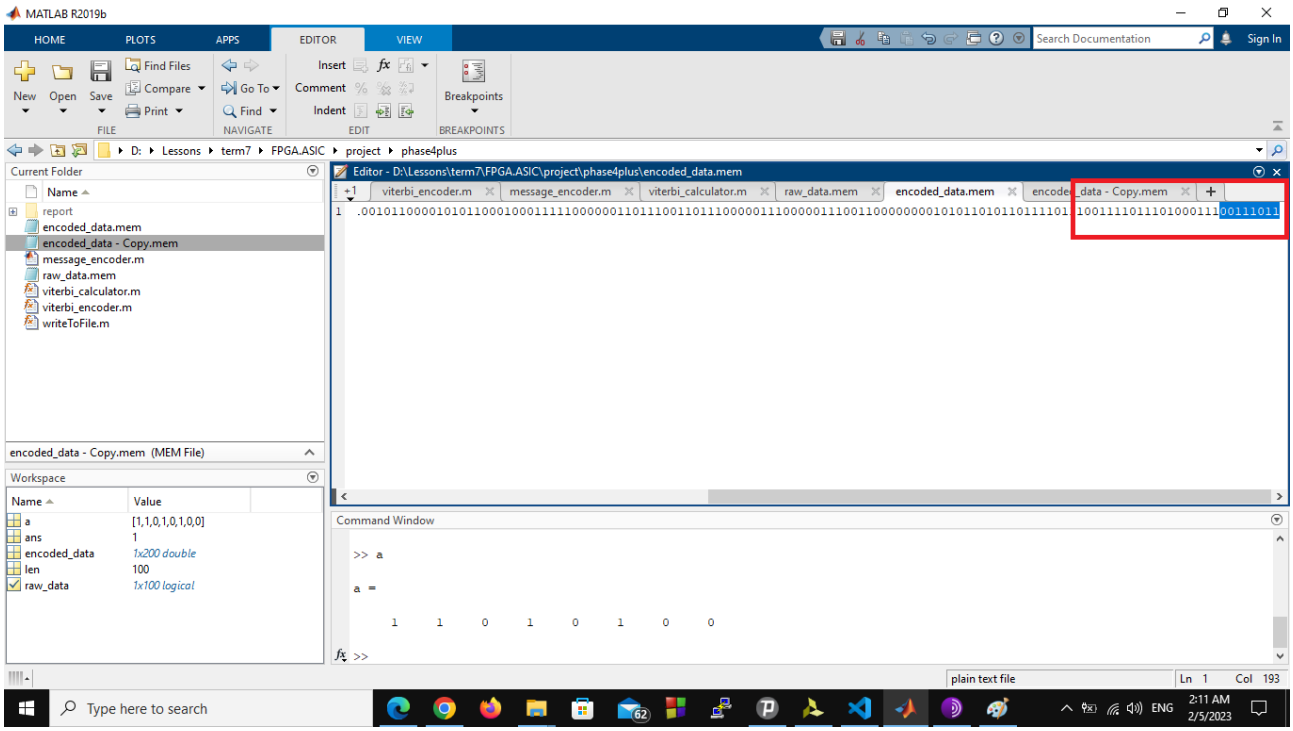
```
%% FPGA project phase 4+ viterbi
clear;
clc;

len = 100;

raw_data = rand(1, len) > ;5.0
writeToFile(raw_data, 'raw_data.mem');

encoded_data = viterbi_encoder(raw_data);
writeToFile(encoded_data, 'encoded_data.mem');



function viterbi = viterbi_calculator(packet)
    viterbi = zeros(1, 8);
    s0 = 0;
    s1 = 0;
    for i = 1:4
        bit = xor(packet(i), s1);
        viterbi(2*i-1:2*i) = [xor(bit, s0), bit];
        s1 = s0;
        s0 = packet(i);
    end
end



function writeToFile(raw_data, file_name)
    x_bin = char(double(raw_data) + 48);
    %x_bin = upsample(x_bin, 2);
    %x_bin(x_bin == 0) = newline;
    bin_data = x_bin;
    file = fopen(file_name, 'wt');
    fwrite(file, bin_data);
    fclose(file);
end



function encoded_data = viterbi_encoder(raw_data)
    raw_data = double(raw_data);
    len = length(raw_data);
    encoded_data = zeros(1, 2*len);
    for i = 1:len/4
        packet = raw_data(4*i-3:4*i);
```

```
        encoded_data(8*i-7:8*i) = fliplr(viterbi_calculator(packet));
    end
end
```

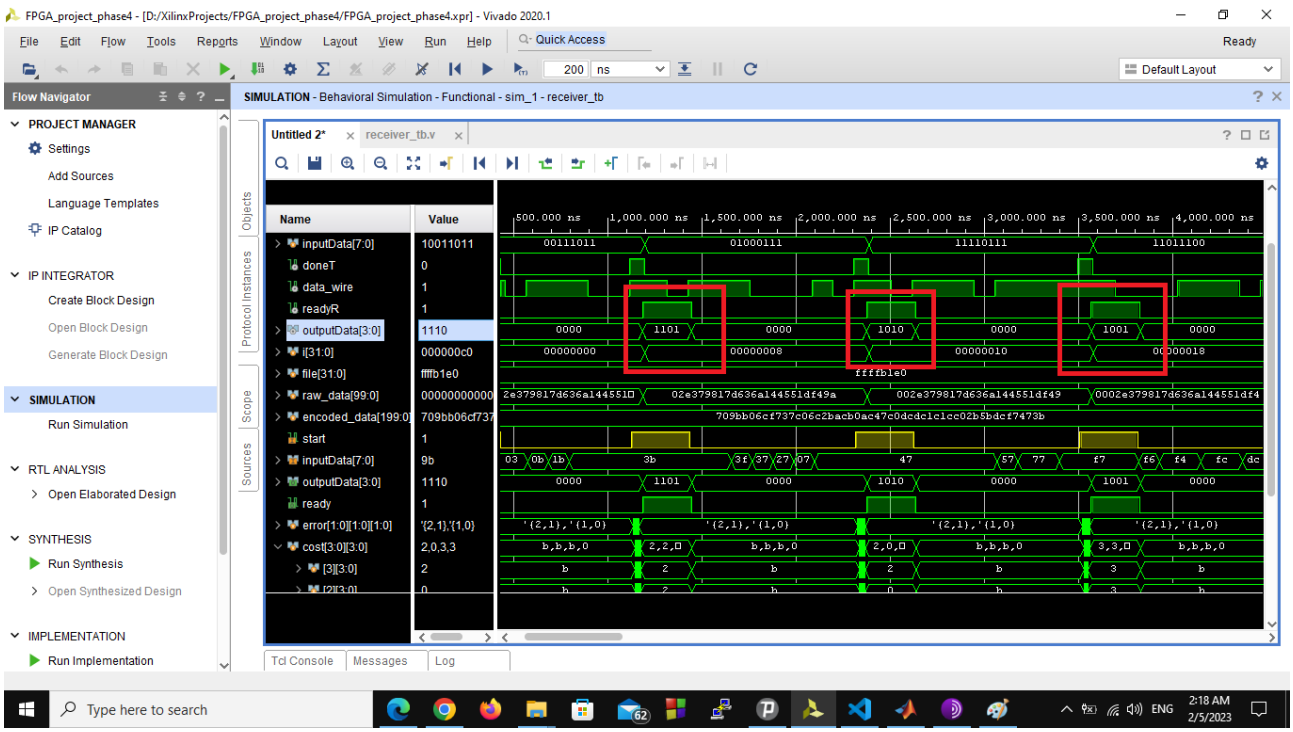
تصویر شبیهسازی را در شکل زیر مشاهده میکنیم. همانطور که میبینید در فایل encoded.mem data مقدار پکت اول را تغییر دادهام تا ببینیم که آیا ماژول میتواند خطا را تصحیح کند یا خیر. سپس همانطور که در تصویر سوم میبینیم، خروجی درست بوده است و پکت اول درست شناسایی شده است و خطای آن تشخیص داده شده و تص حیح شده است. تصویر چهارم خرو جی چاپ شده را نشان میدهد که همه تستها پاس شدهاند. تصویر پنجم دیتا پیش از code شدن توسط Viterbi را نشان میدهد. raw data



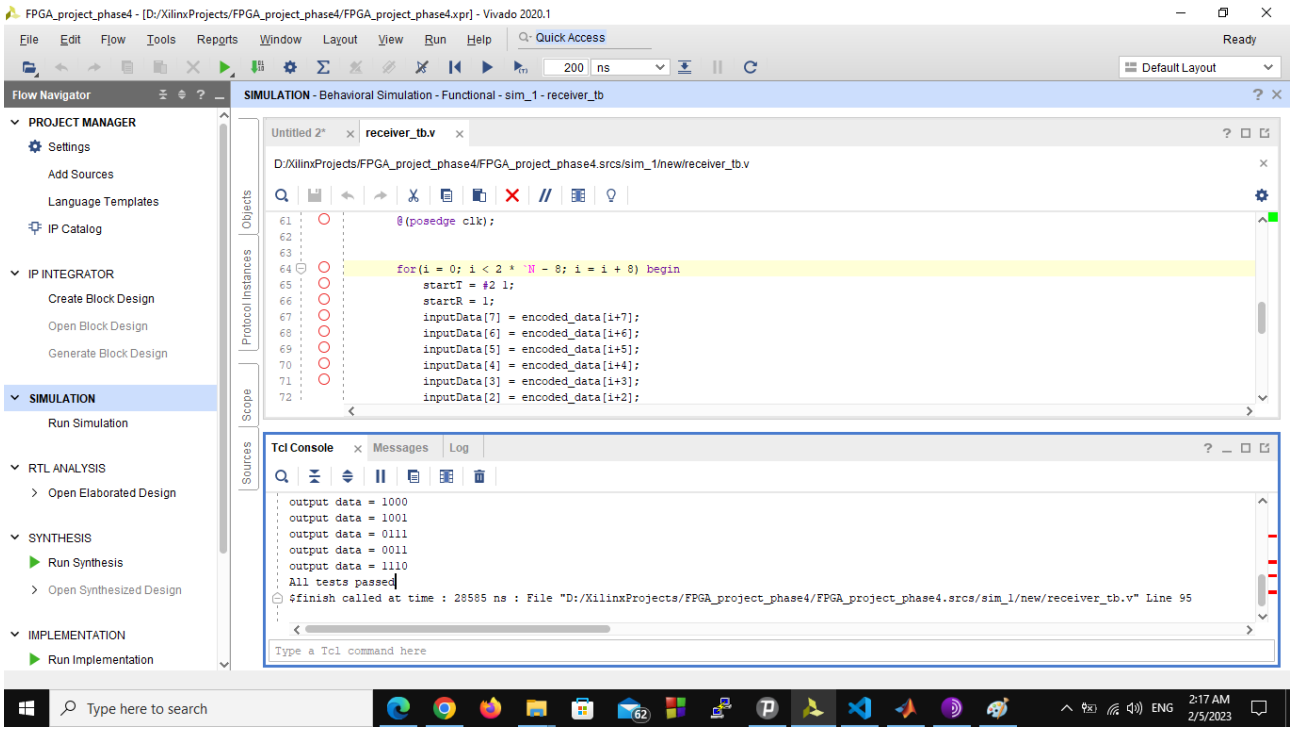
شکل ۷: encoded data copy



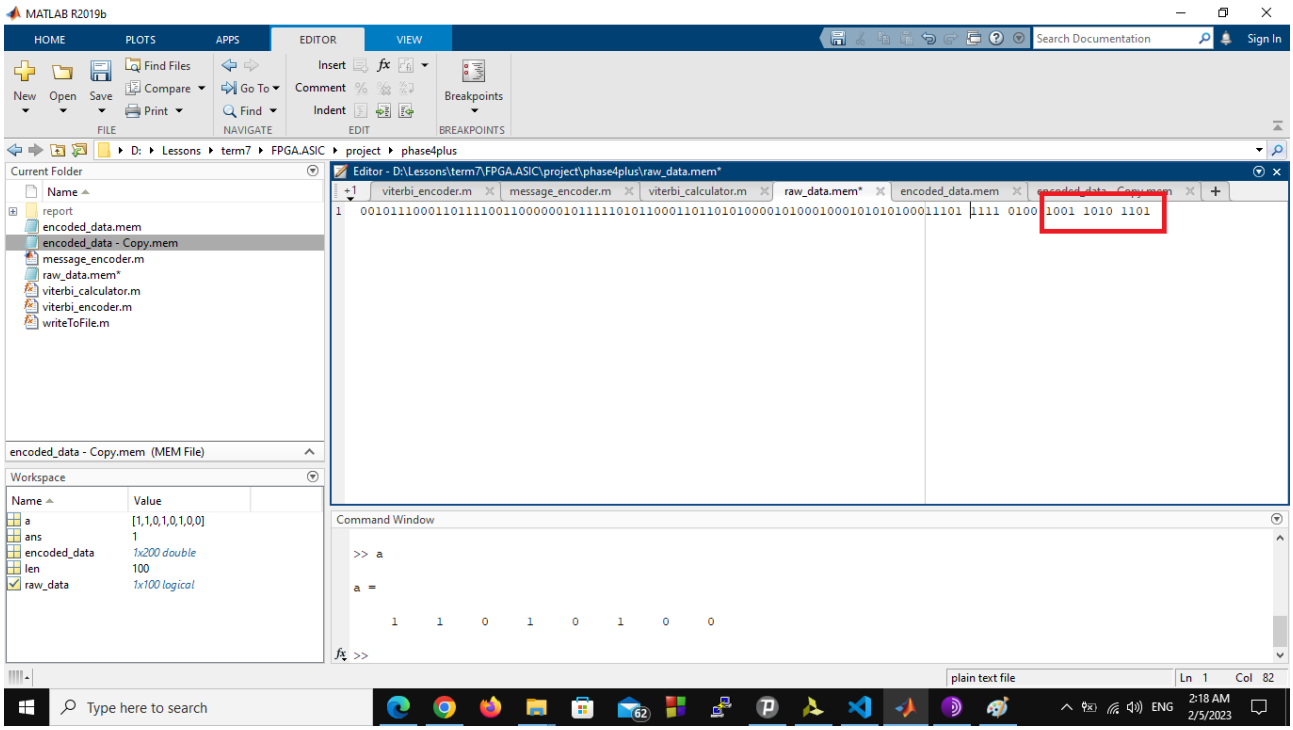
شکل ۸: encoded data



شکل ۹: شبیه سازی



شکل ۱۰: خروجی کنسول



شکل ۱۱: raw data