به نام خدا

دانشکده مهندسی برق

گزارش فاز 2 پروژه

درس سیستم های مبتنی بر FPGA/ASIC

محمدامین حاجی خداوردیان

97101518

استاد:دکتر مهدی شعبانی

نیمسال دوم 99-00

در فاز 2 ما قصد داریم که encoder و Viterbi_decoder را پیاده سازی کنیم.

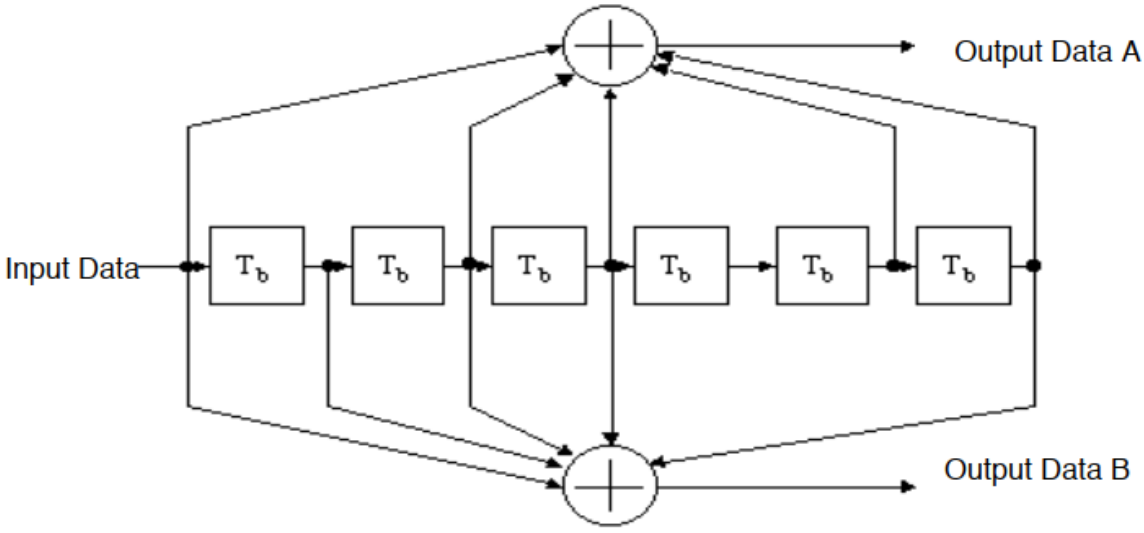
مقدمه

بخش Encoder :

در این بخش با توجه به استاندارد داده شده از دو چند جمله ای مولد با مقادیر 133 و 171(در مبنای 8) استفاده میکنیم.با تبدیل مقادیر بالا به اعداد باینری داریم که:

$$171_8 = 1111001_2 \qquad 133_8 = 1011011_2$$

پس با توجه به روابط بالا تابع مولد ما به شکل زیر خواهد بود:

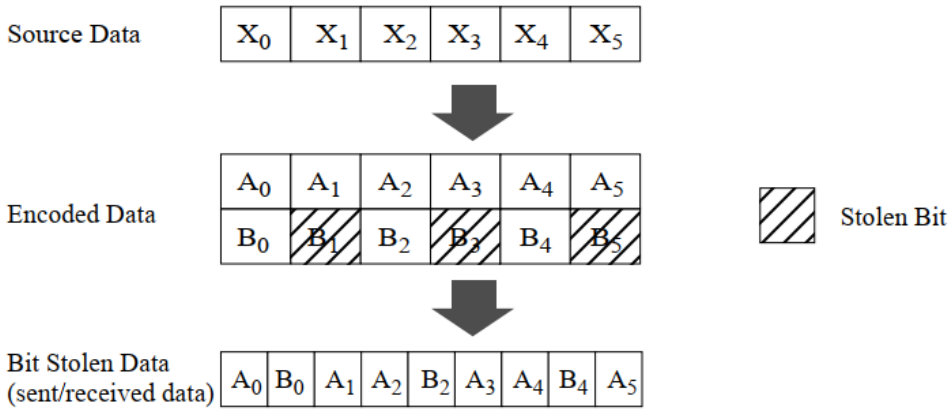


یک حافظه وجود دارد که مقادیر ورودی در آن وارد میشود و در 7 کلاک ساعت در دو خروجی encoder اثر میگذارد.پس با تابع بالا در هر کلاک دو خروجی به دست می آوریم که با استفاده از Coding Rate باید نسبت به نحوه برخورد با آن تصمیم گیری لازم را انجام دهیم.

برای Coding Rate = ½ کافی است در هرکلاک دو بیت به دست آمده از Encoder را خروجی دهیم.

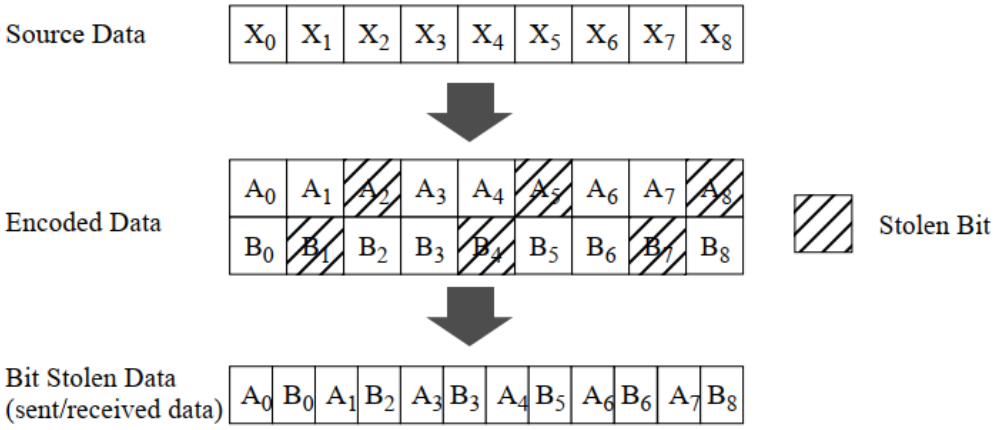
برای Coding Rate = $\frac{2}{3}$ باید یک سری از داده های خروجی Encoder را حذف کنیم و بقیه بیت ها را به عنوان خروجی بدهیم که این تصمیم گیری با توجه به استاندارد به شکل زیر انجام میشود:

Punctured Coding (r = 2/3)



برای Coding Rate = $\frac{3}{4}$ مشابه بالا با توجه به استاندارد باید به شکل زیر عمل کنیم:

Punctured Coding (r = 3/4)



پیاده سازی متلب:

برای این بخش کد متلب پیاده سازی شده در پوشه matlab با نام Encoder.m آورده شده است.

برای پیاده سازی Encoder در متلب در سه بخش پیاده سازی انجام شده است که برای Coding Rate های متفاوت در نظر گرفته شده است.

برای Rate = $\frac{1}{2}$ به صورت زیر پیاده سازی شده است:

```
%%%tx
f = fopen('In.txt');
x = fscanf(f , '%d');
%encode
rate = 0;
if(rate == 0)%%0: rate = 1/2
o = fopen('Out_rate0.txt' , 'w+');
encode_state = [0 , 0 , 0 , 0 , 0 , 0 , 0];
encode_data = zeros(50 , 1);
for i=1:25
       encode_data(2*i-1) = mod(encode_state(2)+ encode_state(3)+ encode_state(5)+encode_state(6) + x(i) , 2);
       encode_data(2*i) = mod(encode_state(1)+encode_state(2) + encode_state(3) + encode_state(6) + x(i) , 2);
        encode_state(2:6) = encode_state(1:5);
        encode_state(1) = x(i);
        disp(encode_state);
end
fprintf(o , '%d \n' , encode_data );
fclose(o);
end
```

همانطور که دیده می شود مقادیر حافظه در ابتدا با 0 مقدار دهی اولیه شده است و سپس تابع داده شده در استاندارد آورده شده است که شامل XOR بیت های مشخصی از حافظه است.

در ابتدا از یک فایل به نام In.txt ورودی های تابع خوانده میشود.

با توجه به اینکه Rate = $\frac{1}{2}$ پس هر دو خروجی تابع معتبر و قابل استفاده است و آن را ذخیره میکنیم تا در تست پیاده سازی وریلاگ از آن استفاده شود.

خروجی ها را در فایلی با نام Out_rate0.txt ذخیره میکنیم تا در تست پیاده سازی وریلاگ از آن استفاده شود.

برای $\text{Rate} = \frac{2}{3}$ به صورت زیر پیاده سازی شده است:

```
rate = 1;
encode_data = zeros(length(x) , 1);
if(rate == 1) %% 1: rate = 2/3
  o = fopen('Out_rate1.txt' , 'w+');
   encode_state = [0 , 0 , 0 , 0 , 0 , 0 , 0];
   for i=1:length(x)
        if mod(i, 2) == 1
            encode_data((3*i-1)/2) = mod(encode_state(2)+ encode_state(3)+ encode_state(5)+encode_state(6) + x(i) , 2);
            encode_data((3*i+1)/2) = mod(encode_state(1)+encode_state(2) + encode_state(3) + encode_state(6) + x(i) , 2);
        end
        if mod(i,2) == 0
            encode_data((3*i)/2) = mod(encode_state(2)+ encode_state(3)+ encode_state(5)+encode_state(6) + x(i) , 2);
        end

        encode_state(2:6) = encode_state(1:5);
        encode_state(1) = x(i);
   end
   fprintf( o , '%d\r\n' , encode_data);
   fclose(o);
end
```

در کد بالا مشابه قبل با آمدن اولین ورودی هر دو خروجی تابع معتبر است اما با ورود دومین داده فقط خروجی A تابع درنظر گرفته میشود و به اصطلاح خروجی B، Stole میشود.خروجی ها نیز در فایلی ذخیره میشود تا از آن استفاده کنیم.

برای $\text{Rate} = \frac{3}{4}$ به صورت زیر پیاده سازی شده است:

```
  encode_state = [0 , 0 , 0 , 0 , 0 , 0 , 0];
  for i=1:length(x)
       if mod(i, 3) == 2
           encode_data((4*i+1)/3) = mod(encode_state(2)+ encode_state(3)+ encode_state(5)+encode_state(6) + x(i) , 2);
       end

       if mod(i, 3) == 1
           encode_data((4*i-1)/3) = mod(encode_state(2)+ encode_state(3)+ encode_state(5)+encode_state(6) + x(i) , 2);
           encode_data((4*i+2)/3) = mod(encode_state(1)+encode_state(2) + encode_state(3) + encode_state(6) + x(i) , 2);
       end

       if mod(i, 3) == 0
           encode_data((4*i)/3) = mod(encode_state(1)+encode_state(2) + encode_state(3) + encode_state(6) + x(i) , 2);
       end

       encode_state(2:6) = encode_state(1:5);
       encode_state(1) = x(i);
  end
  fprintf( o , '%d\r\n' , encode_data);
  fclose(o);
  fclose(f);
end
```

با توجه به شرح پروژه ما فقط نیاز به قسمت کد $\text{Rate} = \frac{1}{2}$ داریم و باقی بخش تحت عنوان بخش های امتیازی و اختیاری در نظر گرفته شده است.

پیاده سازی وریلاگ:

برای این بخش یک پیاده سازی انجام شده که تنها برای Rate = $\frac{1}{2}$ است، در پوشه Extra پیاده سازی انجام شده است که تمامی Rate های مدنظر را پوشش میدهد

ابتدا به توضیح کد اصلی میپردازیم.

```
`timescale 1ns / 1ps
module encoder(
    data_in,
    data_out,
    Clk,
    reset,
    rate,
    en
);
    input data_in;
    input Clk;
    input reset;
    input rate;// 0: rate = 1/2
    input en;
    output reg [1:0]  data_out;
    reg [1:0] counter=2'b0;

    reg [6:1]data_reg = 6'b000000;
```

ماژول طراحی شده شامل ورودی های زیر است:

Data_in : ورودی سریال داده ها

Clk

Reset

En: برای فعال سازی بخش انکودر

Rate: که این بخش با توجه به اینکه تنها برای Rate = $\frac{1}{2}$ این کد پیاده سازی شده است اهمیتی ندارد.

و خروجی نیز به صورت داده های دو بیتی پشت سر هم تحت عنوان data_out داده میشود. بخش data_reg همان حافظه پیاده سازی شده در کد متلب است که با توجه به مقادیر موجود در آن تابع مدنظر را پیاده سازی میکنیم.

```
wire A;
wire B;
reg  f;
assign A = data_in ^ data_reg[2] ^ data_reg[3] ^ data_reg[5] ^ data_reg[6];
assign B = data_in ^ data_reg[1] ^ data_reg[2] ^ data_reg[3] ^ data_reg[6];

always @(posedge Clk , negedge reset)
begin

    if(!reset) begin
        data_reg <= 6'b000000;
        counter = 2'b00;
        f <= 0;
        end
    else if(en)
                begin
                data_reg <= {data_reg[5:1] , data_in};
                data_out = {B , A};

        end
end
endmodule
```

دو بخش A و B تابع های 133_8 و 171_8 است که با توجه به توضیحاتی که داده شد آورده شده است.

حافظه data_reg هر بار ورودی در ابتدای آن قرار میگیرد و باقی مقادیر یک شیفت میخورند. خروجی نیز چون $\text{Rate} = \frac{1}{2}$ است هر دوی خروجی های انکودر معتبر هستند و در خروجی قرار میگیرند.

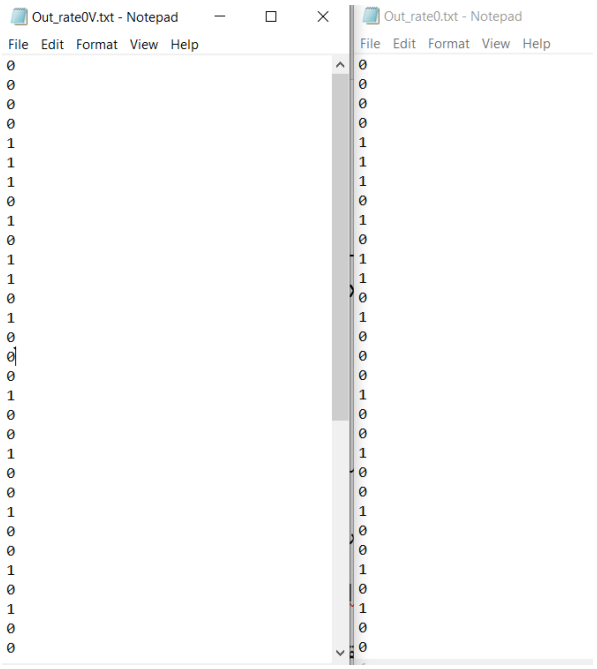
کد تست بنچ برای تست قطعه کد بالا به شکل زیر زده شده است:

```
module encode_tb();
    reg  clk;
    reg  en;
    reg reset;
    wire [1:0]rate;
    reg   in;
    wire [1:0] out;
    reg  des;
    integer e = 0;
    integer i = 0;
    integer  InData, OutData, f1, f2 , OutData2;
    assign rate = 0;
    initial
        begin
        clk = 0;
        en  = 1;
        reset = 1;
        #4 reset = 0;
        #4 reset = 1;
        InData  = $fopen("In.txt", "r");
        OutData2 = $fopen("Out_rate0V.txt", "w");
        f1 = $fscanf(InData, "%b\n", in);
        for (i=0 ; i<26 ; i = i+1)
            @(posedge clk);
        $stop;
    end
```

```
always #10 clk = ~clk;

always@(posedge clk)
begin

        f1 = $fscanf(InData, "%b\n", in);
        $fwrite(OutData2,"%b\n%b\n",out[0] , out[1]);

    end

encoder enc(.Clk(clk), .reset(reset), .en(en), .rate(rate), .data_in(in), .data_out(out));
endmodule
```

همانطور که دیده میشود در اینجا نیز همان داده های ورودی In.txt که به کد متلب داده شد به عنوان ورودی encoder در نظر گرفته شده است.

ورودی ها به encoder داده میشود و یک فایل با نام out_rate0V.txt تحت عنوان خروجی وریلاگ در نظر گرفته میشود که با بررسی آن با مقادیر خروجی متلب که با عنوان Out_rate0.txt ذخیره شده است متوجه میشویم عملکرد کد کاملا درست بوده است.(به علت reset در ابتدای فایل خروجی وریلاگ دو 0 بیشتر نوشته میشود)



حال به سراغ بخش دیگر که در آن تمامی Rate ها را میتوان پیاده سازی کرد میرویم:

در داخل پوشه Verilog/encoder/extra فایل های زیر وجود دارد:

Encode.v , encoder_tb.v , encoder_tb1.v , encoder_tb2.v

که تست بنچ ها مشابه تست بنچ توضیح داده شده در بالا است و با مقایسه فایل های ذخیره شده متلب و وریلاگ میتوان درستی کد را متوجه آن شد.

در بخش اصلی کد نیز ورودی ها مشابه بالا است تنها تفاوت در این است که این بار Rate ها اهمیت دارند و به عنوان ورودی در مسیر پیاده سازی تاثیر گذار هستند

```
assign A = data_in ^ data_reg[2] ^ data_reg[3] ^ data_reg[5] ^ data_reg[6];
assign B = data_in ^ data_reg[1] ^ data_reg[2] ^ data_reg[3] ^ data_reg[6];

always @(posedge Clk , negedge reset)
begin

    if(!reset) begin
        data_reg <= 6'b000000;
        counter = 2'b00;
        end
    else if(en)
            begin
                if(counter)begin
                    data_reg <= {data_reg[5:1],data_in};
                end

                if((counter<=rate))begin
                    counter <= counter + 1 ;
                end

                else
                begin
                    counter <= 0;
                end

            end
end
```

```
always @(*) begin
    case ({rate , counter})

        //rate = 1/2
        {2'd0 , 2'd0}: data_out = A;
        {2'd0 , 2'd1}: data_out = B;
        //rate = 2/3
        {2'd1 , 2'd0}: data_out = A;
        {2'd1 , 2'd1}: data_out = B;
        {2'd1 , 2'd2}: data_out = A;
        //rate = 3/4
        {2'd2 , 2'd0}: data_out = A;
        {2'd2 , 2'd1}: data_out = B;
        {2'd2 , 2'd2}: data_out = A;
        {2'd2 , 2'd3}: data_out = B;

    endcase
end
```

در کد بالا همانطور که دیده میشود یک شمارنده وجود دارد که با توجه به Rate ریست میشود.

در کد بالا Rate به صورت زیر تعریف شده است:

Rate=0 → Rate = $\frac{1}{2}$

Rate = 1 → Rate = $\frac{2}{3}$

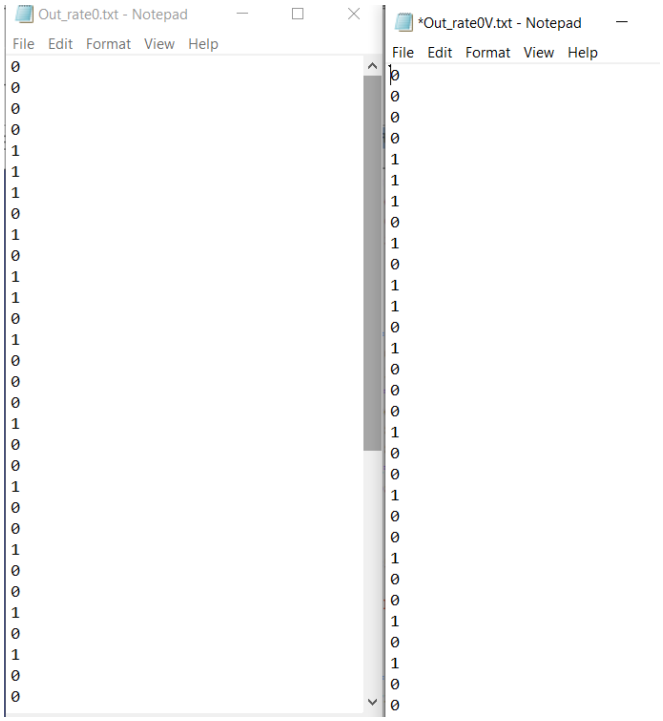
Rate = 2 → Rate = $\frac{3}{4}$

با توجه به بخش encode این را درمیابیم که داده اول در همگی Coding Rate های مختلف هردو خروجی تابع XOR معتبر است بنابراین با استفاده از شمارنده در ابتدا اجازه تغییر حافظه را فقط برای اولین بار نمیدهیم.

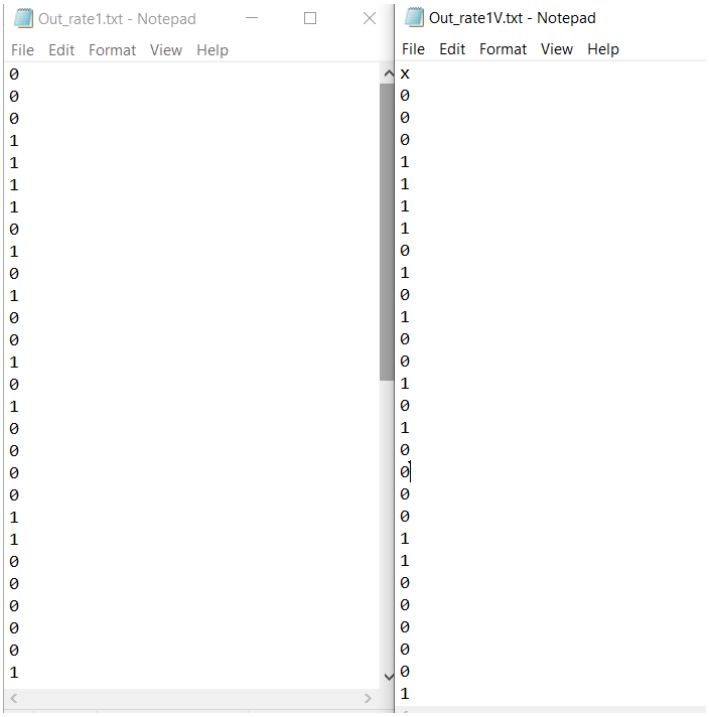
در کد پیاده سازی شده این نکته اهمیت دارد که برای Rate های مختلف با استفاده از یک کنترل کننده خارج از این ماژول داده های ورودی را کنترل کنیم که هر داده تا چند کلاک ساعت معتبر باشد. به عنوان مثال برای Rate = $\frac{1}{2}$ باید هر داده دو کلاک به encoder ورودی داده شود تا به درستی کار کند یا برای Rate = $\frac{2}{3}$ داده اول باید دو کلاک و داده دوم یک کلاک معتبر باشد.

در بخش دوم کد بالا دیده میشود که با توجه به Rate و شمارنده یک بیت یک بیت خروجی های معتبر داده میشود.(در هنگام چاپ مقادیر در تست بنچ ها در کلاک اول چون هنوز خروجی آماده نشده است یک x ظاهر میشود)

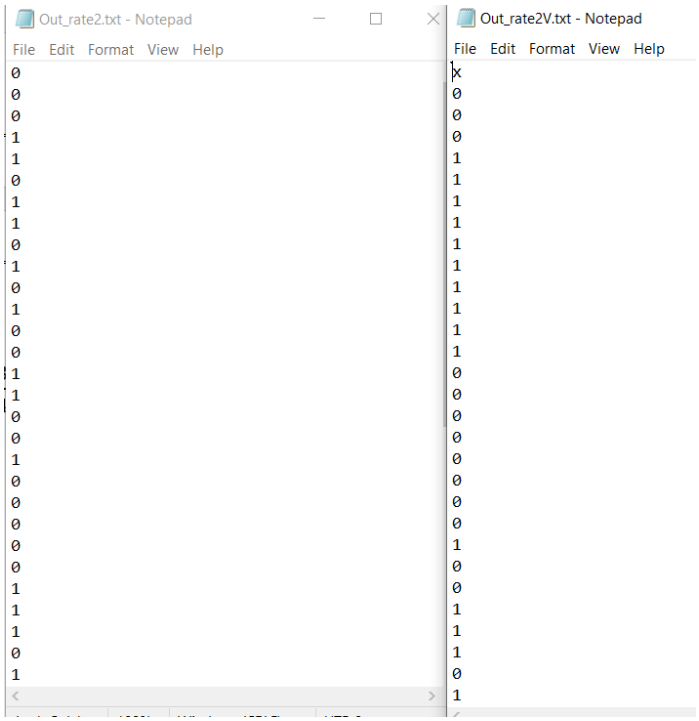
Rate = $\frac{1}{2}$:



Rate = $\frac{2}{3}$:



Rate = $\frac{3}{4}$:

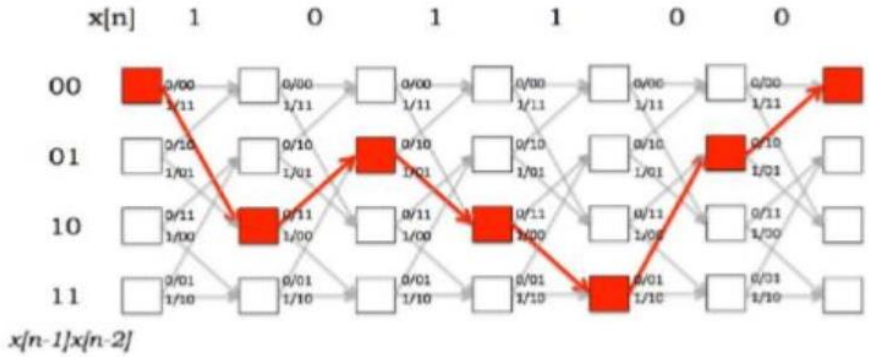


همانطور که دیده میشود برای بررسی درستی کد ها کافی است خروجی های Out_rate0.txt (خروجی متلب) و Out_rate0V.txt(خروجی وریلاگ) همچنین برای بقیه آن ها نیز با یکدیگر مقایسه شود تا از درستی کد اطمینان حاصل کنیم.

مقدمه

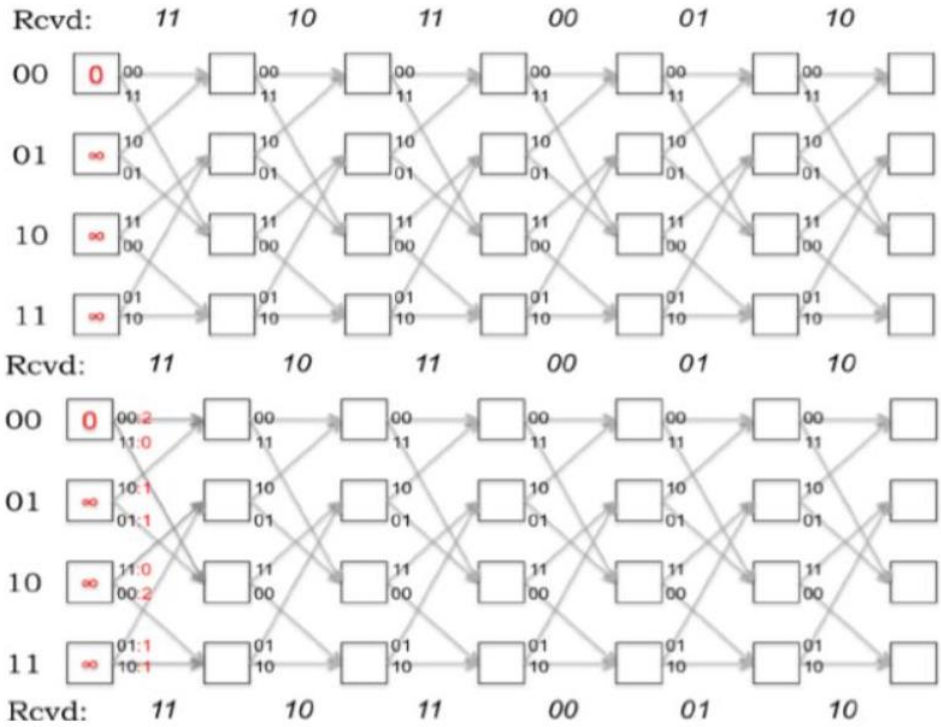
بخش Viterbi_decoder :

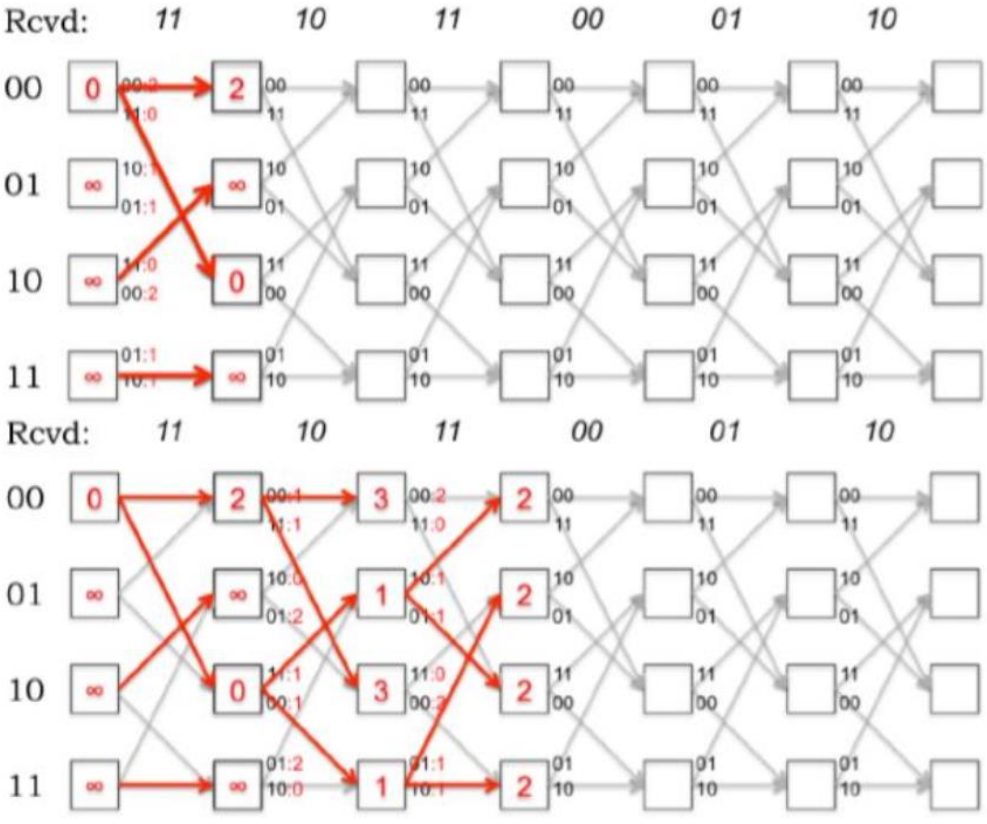
برای درک بهتر این بخش فرستنده یا همان encoder را مورد بررسی قرار میدهیم و نمودار trellis را برای آن رسم میکنیم.

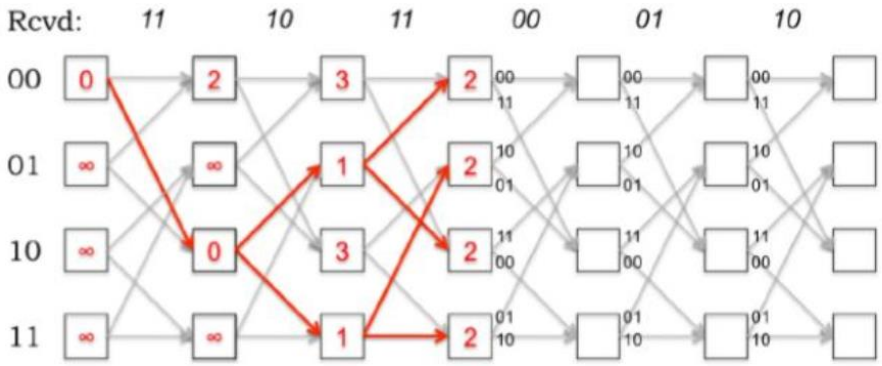


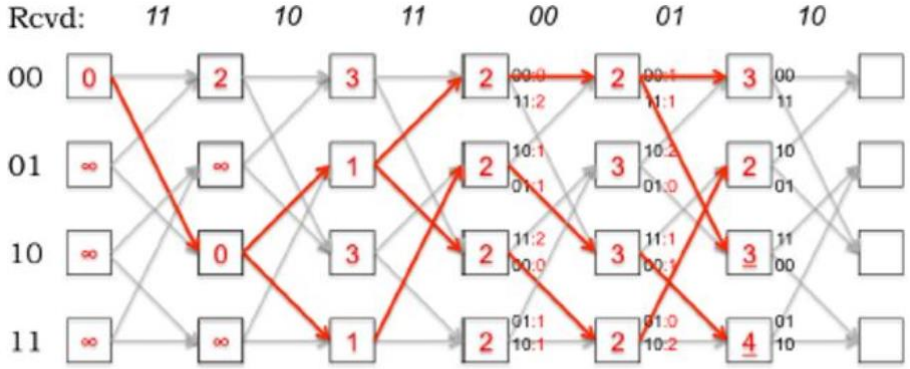
در گیرنده از decoder با عنوان Viterbi استفاده میکنیم که با توجه به نمودار بالا طراحی شده است به این صورت که در هر گام با توجه به جفت بیت های دریافت شده برای تمام حالت های ممکن بررسی می کنیم که از چه حالت هایی در مرحله قبلی میتوان به آن وارد شد؟ این حالت ها را a و b می نامیم(دو حالت وجود دارد)

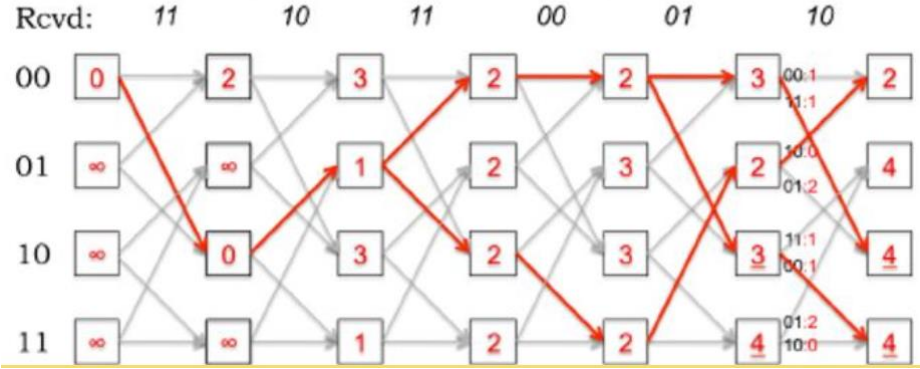
سپس برای هرکدام، بررسی میکنیم که با کدام جفت بیت ها به حالت مورد نظر میرسیم و فاصله hamming جفت بیت های دریافتی با این جفت بیت ها را حساب میکنیم. در آخر باید حالت minimum را با هزینه انباشته تا این مرحله حساب کنیم و اینگونه هزینه کلی هر مسیر را حساب میکنیم. در آخر مسیری که هزینه ی کمتری دارد را انتخاب میکنیم و گام به گام به عقب باز می گردیم تا پیام دریافتی را کدگشایی کنیم.مثالی در این باره در ادامه میبینیم:

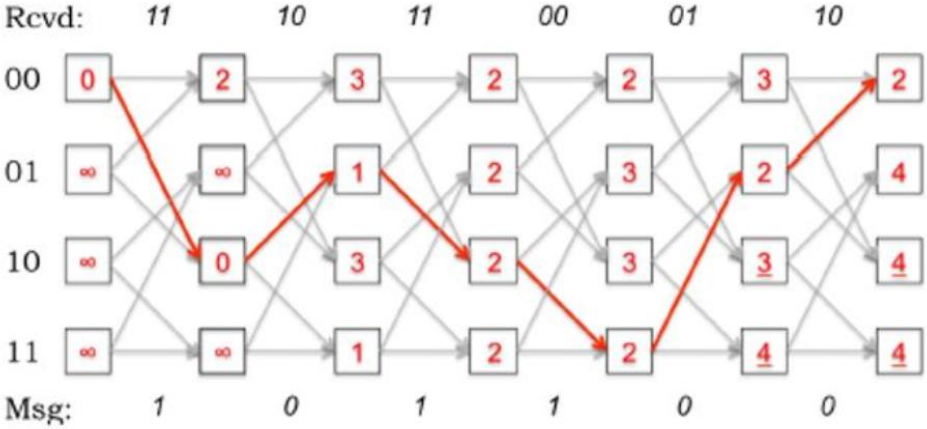

Rcvd: 11 10 11 00 01 10
00
01
10
11
Rcvd: 11 10 11 00 01 10
Rcvd: 11 10 11 00 01 10


Rcvd: 11 10 11 00 01 10
00
01
10
11
Rcvd: 11 10 11 00 01 10


Rcvd: 11 10 11 00 01 10
00
01
10
11


Rcvd: 11 10 11 00 01 10
00
01
10
11


Rcvd: 11 10 11 00 01 10
00
01
10
11


Rcvd: 11 10 11 00 01 10
00
01
10
11
Msg: 1 0 1 1 0 0

پیاده سازی متلب:

```
Num_data_in = 72;
state_Num = 64;
state = 1:64;
node_Cost = zeros(state_Num , Num_data_in/2+1);
node = zeros(state_Num , Num_data_in/2+1);
f = fopen('Out_rate0.txt');
in = fscanf(f , '%d');
for i=1:Num_data_in/2
    a = dec2bin(in(2*i-1));
    b = dec2bin(in(2*i));
    data(i) = bin2dec([a , b])+1;
end
node(1,1) = 1;%%Mark node as visited
trace = zeros(state_Num , Num_data_in/2 );
trace_back = zeros(state_Num , Num_data_in/2 );
```

در کد بالا ابتدا هزینه هر کدام از گره ها را در ابتدا مقدار دهی میکنیم و همچنین داده های سریال ورودی را به صورت دوبیت دوبیت تبدیل میکنیم.

دو آرایه دیگر برای ذخیره سازی مسیر و برگشت آن نیز در نظر میگیریم. از آنجایی که میدانیم مقدار دهی اولیه encoder با 6’b000000 است پس گره اول را تحت عنوان دیده شده در نظر میگیریم.

با توجه به encoder ما 64 حالت مختلف داریم.

```
for j=1:state_Num
    if(node(j , i) == 1)
        [Next_state0 , in0 , out0 , Next_state1 , in1 , out1] = Calc_state(state(j));
        if(cost_calc(data(i) , out0) + node_Cost(j , i) < node_Cost(Next_state0 , i+1)  || node(Next_state0 , i+1)==0)
            node(Next_state0 , i+1) = 1;
            node_Cost(Next_state0 , i+1) = cost_calc(data(i) , out0) + node_Cost(j , i);
            if(i==1)
                trace(Next_state0 , i) = in0;
            else
                trace(Next_state0 , 1:i) = [trace_back(j , 1:i-1) , in0];
            end
        end
        if(cost_calc(data(i) , out1) + node_Cost(j , i) < node_Cost(Next_state1 , i+1)  || node(Next_state1 , i+1)==0)
            node(Next_state1 , i+1) = 1;
            node_Cost(Next_state1 , i+1) = cost_calc(data(i) , out1) + node_Cost(j , i);
            if(i==1)
                trace(Next_state1 , i) = in1;
            else
                trace(Next_state1 , 1:i) = [trace_back(j , 1:i-1) , in1];
            end
        end
```

در قسمت بالا با توجه به ورودی ها ابتدا با استفاده از تابع cost_calc هزینه hamming را محاسبه میکنیم سپس برای هر گره میتوان از دو مسیر به آن رسید:مسیر اول اگر ورودی به encoder 0 بوده باشد و مسیر دوم اگر ورودی به encoder 1 بوده باشد.

حال با توجه به هزینه های انباشته تصمیم گیری را انجام میدهیم.(in0 = 0 و in1 = 1)

سپس به سراغ مرحله بعد و ستون بعدی می رویم.

در ابتدا باید تشخیص دهیم که به چه گره و حالتی خواهیم رفت که آن نیز با یک تابع تحت عنوان calc_state بدست می آید.

قطعه کد calc_state:

```
function[state0 , in0 , out0 , state1 , in1 , out1]= Calc_state(in)
    CS = dec2bin(in-1 , 6);
    in0 = 0;
    in1 = 1;
    state0 = 1 + bin2dec(['0',CS(1,1:5)]);
    state1 = 1 + bin2dec(['1',CS(1,1:5)]);
```

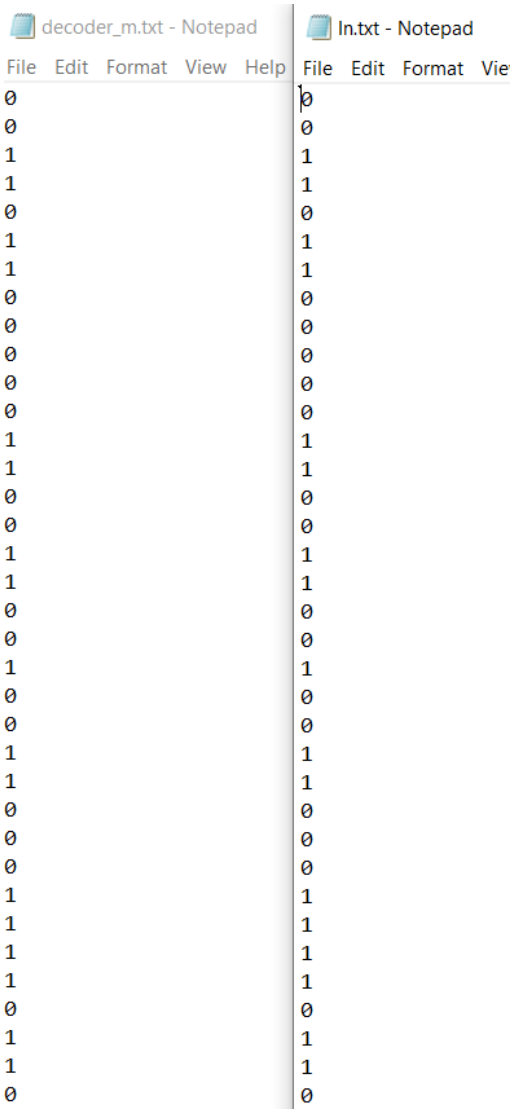
با توجه به کد بالا حالت گره های بعدی در صورت ورودی 0(state0) و ورودی 1 (state1) است. خروجی های out0 و out1 نیز از طریق XOR حالت گره ها بدست می آید.

قطعه کد cost_calc:

```
function [out]= cost_calc(in1,in2)
    if(in1==in2)
      out=0;
    end
    if(in1==1&&in2==2||in2==1&&in1==2)
      out=1;
    end
    if(in1==1&&in2==3||in2==1&&in1==3)
      out=1;
    end
        if(in1==1&&in2==4||in2==1&&in1==4)
      out=2;
        end
        if(in1==2&&in2==3||in2==2&&in1==3)
      out=2;
        end
    if(in1==2&&in2==4||in2==2&&in1==4)
      out=1;
        end
     if(in1==3&&in2==4||in2==3&&in1==4)
      out=1;
    end
    end
```

کد بالا نیز با توجه به ورودی های دوبیتی که داده میشود هزینه hamming را محاسبه میکند.

برای تست کردن درستی کد بالا خروجی encoder را با عنوان Out_rate0.txt به عنوان ورودی به decoder میدهیم و خروجی decoder را با ورودی encoder مقایسه میکنیم.



پیاده سازی وریلاگ:

کد های موجود در این بخش 5 کد وریلاگ است که شامل 4 ماژول به همراه یک تست بنچ است.

ماژول Viterbi_decoder از چند زیر ماژول تشکیل شده است.

Calc_state:

در این ماژول با توجه به استیت ها(حالت های گره) تصمیم گیری شده است و خروجی های گذر از یک iteration به بعدی نیز در آن مشخص میشود و تمامی حالت های آن به صورت مجزا نوشته شده است.

این مقادیر از روی جدول زیر برداشته شده است:

Previous state	*Next state and 2output-bit With Input 0*	*Next state and 2output-bit With Input 1*	*Previous state*	*Next state and 2output-bit With Input 0*	*Next state and 2output-bit With Input 1*
000000	00000000	10000011	100000	01000010	11000001
000001	00000011	10000000	100001	01000001	11000010
000010	00000101	10000110	100010	01000111	11000100
000011	00000110	10000101	100011	01000100	11000111
000100	00001000	10001011	100100	01001010	11001001
000101	00001011	10001000	100101	01001001	11001010
000110	00001101	10001110	100110	01001111	11001100
000111	00001110	10001101	100111	01001100	11001111
001000	00010011	10010000	101000	01010001	11010010
001001	00010000	10010011	101001	01010010	11010001
001010	00010110	10010101	101010	01010100	11010111
001011	00010101	10010110	101011	01010111	11010100
001100	00011011	10011000	101100	01011001	11011010
001101	00011000	10011011	101101	01011010	11011001
001110	00011110	10011101	101110	01011100	11011111
001111	00011101	10011110	101111	01011111	11011100
010000	00100011	10100000	110000	01100001	11100010
010001	00100000	10100011	110001	01100010	11100001
010010	00100110	10100101	110010	01100100	11100111
010011	00100101	10100110	110011	01100111	11100100
010100	00101011	10101000	110100	01101001	11101010
010101	00101000	10101011	110101	01101010	11101001
010110	00101110	10101101	110110	01101100	11101111
010111	00101101	10101110	110111	01101111	11101100
011000	00110000	10110011	111000	01110010	11110001
011001	00110011	10110000	111001	01110001	11110010
011010	00110101	10110110	111010	01110111	11110100
011011	00110110	10110101	111011	01110100	11110111
011100	00111000	10111011	111100	01111010	11111001
011101	00111011	10111000	111101	01111001	11111010
011110	00111101	10111110	111110	01111111	11111110
011111	00111110	10111101	111111	01111100	11111111

Table 4.1 the next state metric

Hamming_calc:

در این ماژول مشابه کد متلب مقادیر ورودی را برای ورودی های 0 و 1 به encoder با خروجی های استیت های مختلف تعداد بیت خطا را با استفاده از hamming میابیم.

```
    always @(x , y) begin
        if(x==y)
            z=0;
        if(x==1&&y==2||y==1&&x==2)
            z=1;
        if(x==1&&y==3||y==1&&x==3)
            z=1;
        if(x==1&&y==4||y==1&&x==4)
            z=2;
        if(x==2&&y==3||y==2&&x==3)
            z=2;
        if(x==2&&y==4||y==2&&x==4)
            z=1;
        if(x==3&&y==4||y==3&&x==4)
            z=1;
//in = 1
         if(x==m)
            z1=0;
        if(x==1&&m==2||m==1&&x==2)
            z1=1;
        if(x==1&&m==3||m==1&&x==3)
            z1=1;
        if(x==1&&m==4||m==1&&x==4)
            z1=2;
        if(x==2&&m==3||m==2&&x==3)
            z1=2;
        if(x==2&&m==4||m==2&&x==4)
            z1=1;
        if(x==3&&m==4||m==3&&x==4)
            z1=1;
    end

endmodule
```

که ورودی های آن داده های دوبیتی ورودی، خروجی ورودی 0 به استیت بعدی(y)، خروجی ورودی 1 به استیت بعدی(m) و خروجی های آن z و z1 است که یکی برای ورودی 0 و دیگری برای ورودی 1 است.

:Compare

با توجه به اینکه به هرکدام از استیت ها دو ورودی وارد میشود برای حساب کردن هزینه کمینه و محاسبه هزینه انباشته گره(استیت) های مختلف کافی است این مقادیر را به عنوان ورودی به این ماژول بدهیم.

همچنین شماره مرحله ای که در آن قرار داریم را به این ماژول میدهیم تا مسیر درست را در خود ذخیره کند.

```
    always@(*)
        begin
        if(cost1+hamming1<cost2+hamming2)
        begin
            outputcost=cost1+hamming1;
            for (i=1;i<column;i=i+1)
                outputtrace[i]=trace1[i];
            outputtrace[column]=input1;
        end
        else
        begin
            outputcost=cost2+hamming2;
            for (i=1;i<column;i=i+1)
                outputtrace[i]=trace2[i];
            outputtrace[column]=input2;
        end
        end
endmodule
```

ورودی های هزینه انباشه استیت قبلی و فاصله hamming از دو مسیر مختلف با هم مقایسه میشود و هزینه انباشته استیت بعدی به همراه مسیر که برای ساخت داده اولیه نیاز است ذخیره میشود.

:Decoder

در این ماژول چون 64 استیت مختلف داشتیم از ماژول های قبل 64 نمونه گرفته شده است و مطابق با نیاز به یکدیگر وصل شده اند.

با هر کلاک یک مرحله و گام به جلو میرویم و در نهایت به تعداد 36 بیت خروجی با هم آماده میشود و در خروجی قرارداده میشود.

```
always@(posedge Clk, negedge reset)
begin
if(!reset)
begin
    counter<=1;
    for(i=1;i<=state_Num;i=i+1)
    begin
        for(j=1;j<=data_Num/2+1;j=j+1)
        begin
        if(i==1&&j==1)
        cost[i][j]<=11'd0;
        else
        cost[i][j]<=11'd300;
        end
        for(j=1;j<data_Num/2+1;j=j+1)
        begin
        if(i==1)
        trace[i][j]<=1'd0;
        end
    end
end
```

```
    else
    begin
    counter<=counter+1;
    for(i=1;i<=state_Num;i=i+1)
    begin
    trace[i]<=next_trace[i];
    cost[i][counter+1]<= cost_wire[i];
    end
    if(counter==data_Num/2)
    begin

        for(i=2;i<=state_Num;i=i+1)
        begin
            begin
                if(cost[i][data_Num/2+1]<cost[index][data_Num/2+1])
                index=i;
            end
        end
        data_out<=next_trace[index];
    end
    end
end
```

در باقی کد نیز نمونه های ماژول های قبلی که توضیحات آن داده شده است نمونه برداری شده و به یکدیگر متصل شده اند.