به نام خدا

دانشکده مهندسی برق

گزارش فاز 1 پروژه

درس سیستم های مبتنی بر FPGA/ASIC

محمدامین حاجی خداوردیان

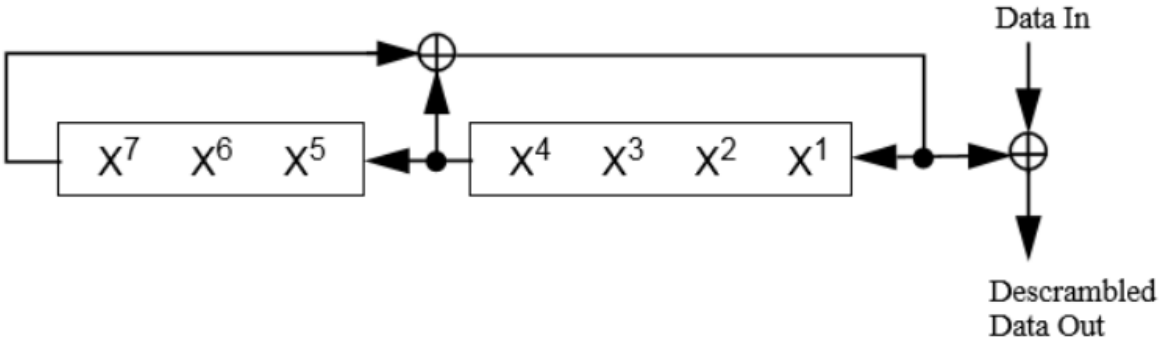
97101518

استاد:دکتر مهدی شعبانی

نیمسال دوم 99-00

مقدمه:

در فاز اول پروژه قصد داریم پیاده سازی scrambler و descrambler را انجام دهیم. طراحی Scrambler با توجه به استانداردی که در اختیار ما قرار دارد به شکل زیر است:



همانطور که در شکل دیده میشود داده خروجی ما به صورت زیر است:

Data_out = Data_in ^ scrambler_seed[7] ^ scrambler_seed[4]

که عبارت بالا را برای سادگی به صورت چندجمله ای نیز میتوان نمایش داد و نمایش آن به صورت زیر است:

$$S(x) = x^7 + x^4 + 1$$

حال با توجه به عبارت بالا باید scrambler_seed مقداری مخالف 0 داشته باشد که داده خروجی به درستی scramble شود.

عملیات descramble نیز مشابه همین scrambler کردن است و باید برای descramble از همان مقدار اولیه ای استفاده کنیم که برای scramble کردن داده های ورودی استفاده کرده بودیم زیرا:

D = data

S = scrambler_seed

F = output of descrambler X = output of scrambler

X = S ^ D

$$F = X \wedge S = (D \wedge S) \wedge S = D$$

همانطور که از روابط بالا برمیاید در این صورت خروجی descrambler همان داده های ابتدایی ما خواهد بود.

نکته دیگری که باید به آن توجه کرد این است که فریم ورودی ما به صورت زیر خواهد بود:

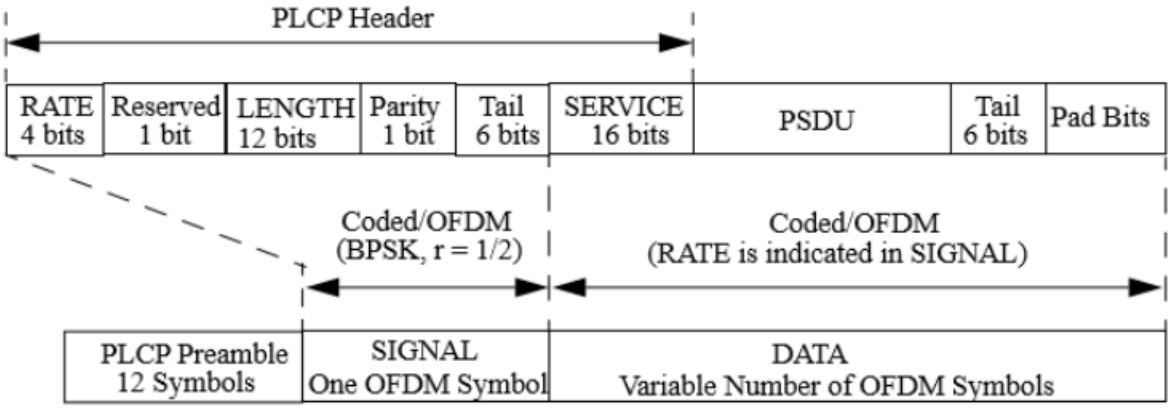


Figure 107—PPDU frame format

با توجه به استاندارد میدانیم که فقط بخش DATA باید scramble بشود و با بخش signal ما کاری نخواهیم داشت.

با استفاده از بخش preamble در گیرنده متوجه آمدن frame جدید خواهیم شد که پیاده سازی آن توضیح داده شده است.

پیاده سازی متلب:

در کد متلبی که در پوشه matlab قرار دارد، در یک فایل tx_and_rx.m هر دو بخش scrambler و descrambler کد آن زده شده است که توسط کامنت های %tx و %rx این دو بخش از هم جدا شده اند.

کد بخش فرستنده:

```
%%%tx
f = fopen('In.txt' , 'w+');
o = fopen('Out.txt' , 'w+');
x = fopen('N_pad.txt' , 'w+');
rate = dec2bin(13 , 4);
rate_param = SET_RATE_PARAMS(6);
length_n = dec2bin(20 , 12);
N_SYM = ceil((16 + 8*20+6)/rate_param.NDBPS);
N_DATA = N_SYM * rate_param.NDBPS;
N_PAD = N_DATA - (16 + 8 * 20 + 6);
n_pad = dec2bin(N_PAD , 6);
fprintf(x ,'%c' , n_pad(:));
scramble_state = [1 ,0 ,1 ,1 ,0 ,1 ,1 ];
psdu = floor(2.*rand(1 ,bin2dec([length_n,'000'])));
signal_preamble = ['111111111111'  , rate ,'0' ,length_n ,'0' ,'000000'];
for i=1:length(signal_preamble)
    fprintf(f ,'%c \t' , signal_preamble(i));
    fprintf(o ,'%c \t' , signal_preamble(i));
end
data = [zeros(1,16) ,psdu ,zeros(1,6) ,  zeros(1 , N_PAD)];
dataout = [zeros(1,16) ,psdu ,zeros(1,6) , zeros(1 , N_PAD)];
```

```
for i=1:length(data)
    fprintf(f ,'%d \t' , data(i));
    scramble_data = xor(scramble_state(1),scramble_state(4));
    scramble_state = [scramble_state(2:7) ,scramble_data];
    dataout(i) = xor(data(i) ,scramble_data);
    fprintf(o ,'%d \t' , dataout(i));
end
send = [signal_preamble , dataout];
fclose(f);
fclose(o);
fclose(x);
```

همانطور که در کد بالا دیده میشود ابتدا سه فایل برای ساختن ورودی و خروجی های مناسب برای تست کردن پیاده سازی توسط HDL و یک فایل برای تعداد بیت های N_pad برای مشخص کردن طول N_pad باز شده است.

در اینجا بخش rate و length که جزوی از فریم signal هستند به ترتیب 6(با توجه به استاندارد برای رسیدن به rate برابر با 6Mbits/s باید ورودی 4’b1101(13) باشد) و 20 انتخاب شده اند در واقع عدد length تعداد 8 بیتی های ورودی را مشخص میکند بنابراین 8*20 داده ورودی را به صورت رندوم تعیین میکنیم.

سپس فریم سیگنال را میسازیم و در فایل های مدنظر آن هارا ذخیره میکنیم.

در ابتدای فریم signal_preamble به اندازه 12 بیت 1 قرار داده شده است که با استفاده از آن در گیرنده قرار است تشخیص دهیم که فریم جدید آغاز شده است.

برای محاسبه تعداد PADbits با توجه به جدول داده شده در استاندارد یک تابع جدید زده شده است که با توجه به DataRate ورودی به ما متغیر های لازم برای ساخت تعداد بیت های PAD میدهد.(Coding rate برای این بخش ها برابر ½ است)

```
function RATE_PARAMS = SET_RATE_PARAMS(rate_tmp)
    switch(rate_tmp)
        case 6
            nbpsc = 1;
            ncbps = 48;
            ndbps = 24;
        case 12
            nbpsc = 2;
            ncbps = 96;
            ndbps = 48;
        case 24
            nbpsc = 4;
            ncbps = 192;
            ndbps = 96;
        otherwise
            disp('Error - RATE');
    end
    RATE_PARAMS = struct('NBPSC', nbpsc, 'NCBPS', ncbps, 'NDBPS', ndbps);

end
```

Table 78—Rate-dependent parameters

Data rate (Mbits/s)	Modulation	Coding rate (R)	Coded bits per subcarrier (N_{BPSC})	Coded bits per OFDM symbol (N_{CBPS})	Data bits per OFDM symbol (N_{DBPS})
6	BPSK	1/2	1	48	24
9	BPSK	3/4	1	48	36
12	QPSK	1/2	2	96	48
18	QPSK	3/4	2	96	72
24	16-QAM	1/2	4	192	96
36	16-QAM	3/4	4	192	144
48	64-QAM	2/3	6	288	192
54	64-QAM	3/4	6	288	216

همانطور که مشخص است با توجه به جدول خروجی های مدنظر را توسط این تابع دریافت خواهیم کرد.

بخش DATA نیز در ابتدا از یک بخش service تشکیل شده است که 16 بیت است آن را با توجه به استاندارد 0 قرار میدهیم و سپس داده اصلی و سپس 6 بیت 0 برای بخش Tail و به تعداد N_pad که توسط تابع نشان داده شده در این صفحه و روابط موجود در کد صفحه قبل محاسبه شده است در انتهای آن 0 قرار میدهیم.

در ادامه تابعی که گفته شده بود با seed مشخص که 7'b1011011 است به ترتیب درایه های 4 و 7 با هم XOR میشوند و سپس این مقدار در ابتدای scrambler قرار میگیرد و برای ساخت خروجی با داده ورودی نیز XOR میشود.

کد بخش گیرنده:

```
flag = 1;
counter = 0;
while(flag)
    preamble(i) = send(i);
    if(preamble(i) == 1)
        counter = counter + 1;
    end
    if(counter == 12)
        flag = 0;
    end
end
for i=1:length(signal_preamble)-counter
    signal_d(i) = send(i+counter);
end
for i=25:length(send)-12
    scramble_data = xor(scramble_state(1),scramble_state(4));
    scramble_state = [scramble_state(2:7) ,scramble_data];
    data_d(i-24) = xor(send(i+counter) ,scramble_data);
end
```

برای بخش گیرنده چون گیرنده از آغاز نمیداند که چه زمانی فریم آغاز شده است بنابراین با استفاده از بخش preamble که 12 بیت 1 بوده است آن بخش را شبیه سازی میکنیم زمانی که در ورودی گیرنده 12 بیت 1 دریافت شود متوجه میشویم که فریم آغاز شده است و به سراغ باقی پردازش ها میشویم.

بخش SIGNAL نیازی به scramble شدن ندارد و فقط باید بخش DATA را با ساخت seed اولیه scramble کنیم تا به درستی داده های ورودی بازیابی شود.

برای مقایسه درستی خروجی کافی است که متغیر های data و data_d را با هم مقایسه کنیم، که کاملا یکسان هستند و عملکرد کد متلب درست است.

پیاده سازی وریلاگ:

ماژول اصلی به نام tx ساخته شده است که ورودی و خروجی های آن به شکل زیر است:

```
module tx(
    data_in,
    n_pad,
    Clk,
    reset,
    data_out,
    ready
    );
    input data_in;
    input Clk;
    input reset;
    input [5:0] n_pad;
    output data_out;
    output reg ready;
```

که ورودی ها به صورت سریال وارد میشوند.

یک ورودی n_pad به صورت موازی برای آن در نظر گرفته شده است که تعداد بیت های بخش PAD را که در متلب با توجه به تابع گفته شده در استاندارد حساب کردیم به عنوان ورودی به اسکرمبلر میدهد که مشخص شود که به چه تعداد باید برای بخش آخر که scramble کردن بیت های tail و pad است باید صبر کنیم.

یک ماژول دیگر تحت عنوان scramble وجود دارد که برای بخش های DATA دیتافریم آن را فعال میکنیم تا به scramble کردن داده ها بپردازد.

ابتدا به سراغ بخش tx میرویم.

به ازای هر کلاک یک خروجی scramble شده خارج خواهد شد.

```
parameter signal = 0 , service = 1   , psdu = 2 , tail_pad = 3;
wire [6:0] scrambler_seed = 7'b1011011;
reg [2:0] C_state;
```

همانطور که دیده میشود scramble_seed = 7’b1011011 است. همچنین برای بخش های مختلف دیتا فریم 4 استیت در نظر گرفته شده که با توجه به هرکدام تصمیم گیری ها را انجام دهیم.

```
case(C_state)

    signal_preamble:begin
        if(Counter < 35)begin
            out <= data_in;
            Counter <= Counter + 1'b1;
            signal_field <= {signal_field[22:0] , data_in};
            ready <= 0;
            en_scrambler <= 1'b0;
            rst_scrambler <= 1'b1;
        end
        else begin
            out <= data_in;
            Counter <= 0;
            signal_field <= {signal_field[22:0] , data_in};
            en_scrambler <= 1'b1;
            rst_scrambler <= 1'b0;
            in_scrambler <= data_in;
            C_state <= service;
        end
    end
```

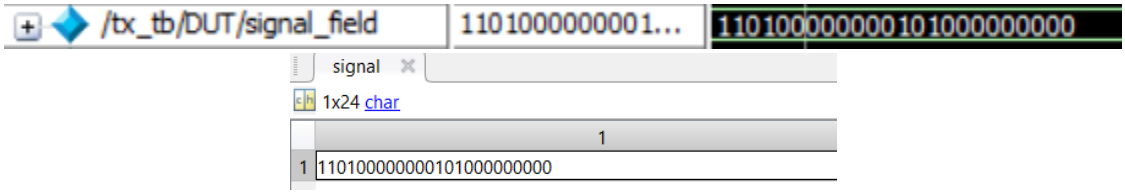
```
service:begin
    if(Counter < 15)begin
        out <= out_scrambler;
        Counter <= Counter + 1'b1;
        in_scrambler <= data_in;
        en_scrambler <= 1'b1;
        rst_scrambler <= 1'b0;
    end
    else begin
        out <= out_scrambler;
        Counter <= 0;
        in_scrambler <= data_in;
        en_scrambler <= 1'b1;
        rst_scrambler <= 1'b0;
        C_state <= psdu;
    end
end
```

در ورودی بخش preamble اهمیتی ندارد و تنها در ابتدای فریم قرار است که فرستاده شود پس کافی است در فرستنده کاری با آن بخش نداشته باشیم.

بخش signal نیازی به scramble ندارد بنابراین داده ای را scramble نمیکنیم.

یک شیفت رجیستر وجود دارد که داده های ورودی سیگنال در آن ذخیره میشود تا بتوانیم بخش های rate و length را از آن بدست بیاوریم.

در شکل های زیر کاملا مشخص است که مقادیر بخش سیگنال دقیقا برابر داده ورودی ساخته شده توسط متلب است:



در بخش های بعد ماژول scramble را enable میکنیم و بخش DATA دیتافریم را به ترتیب شروع به scrambleکردن میکنیم.

با توجه به کد آورده شده برای بخش های psdu و tail_pad بخش های مربوط به این 3 بخش از DATA فریم Scramble میشود.

همانطور که دیده میشود در بخش آخر به تعداد (6 + n_pad) کلاک عملیات scramble انجام میشود که 6 کلاک مرتبط با بخش tail و به تعداد n_pad برای بخش PAD بیت است که این n_pad از قبل محاسبه شده و به عنوان ورودی داده میشود.

```
psdu:begin
    if(Counter < (length , 3'b000)) begin// 8bit * Num of frames
        out <= out_scrambler;
        Counter <= Counter + 1'b1;
        in_scrambler <= data_in;
        en_scrambler <= 1'b1;
        rst_scrambler <= 1'b0;
    end
    else begin
        out <= out_scrambler;
        Counter <= 0;
        in_scrambler <= data_in;
        en_scrambler <= 1'b1;
        rst_scrambler <= 1'b0;
        C_state <= tail_pad;
    end
end

tail_pad:begin
    if(Counter < (6 + n_pad))begin
        out <= out_scrambler;
        Counter <= Counter + 1'b1;
        en_scrambler <= 1'b1;
        rst_scrambler <= 1'b0;
    end
    else begin
        ready <= 1;
    end
end
```

در بخش اصلی کد Tx یک instance از ماژول scramble میگیریم و ورودی را به آن میدهیم و منتظر میمانیم تا ورودی en آن فعال شود تا شروع به کار کند.

ماژول scramble:

```
module scrambler(
    in_data,
    Clk,
    scrambler_seed,
    reset,
    en,
    out_data
);
    input in_data;
    input reset;
    input Clk;
    input en;
    input[6:0]scrambler_seed;
    output out_data;

    wire feedback;
    reg [6:0]seed;

    assign feedback = seed[6] ^ seed[3];
    assign out_data = in_data ^ feedback;

    always@(posedge Clk)
    begin

        if(reset)begin
            seed <= scrambler_seed;
        end
        else if(en)begin
            seed <= {seed[5:0] , feedback};
        end

    end


endmodule
```

همانطور که دیده میشود یک شیفت رجیستر برای scrambler_seed وجود دارد که دائما درحال آپدیت شدن است.

خروجی ها نیز با توجه به طراحی که در ابتدا گفته ایم با دو XOR مشخص است.

برای تست کردن بخش خروجی نیز تست بنچ زیر نوشته شده است:

```
module tx_tb();

    reg Clk;
    reg reset;
    reg in;
    reg [5:0]pad_in;
    wire out;
    integer op1, op_out ,op2;
    integer i , k=0 , temp;
    wire ready;
    reg inreg[220:0];

tx DUT(
    .data_in(in),
    .n_pad(pad_in),
    .Clk(Clk),
    .reset(reset),
    .data_out(out),
    .ready(ready)
    );
```

```
    initial
    begin
        Clk = 0;
        reset = 1;
        $readmemb("In.txt" , inreg);
        op2 = $fopen("N_pad.txt" , "r");
        op_out = $fopen ("Out_v.txt" , "w");
        #20;
        reset = 0;
        for(i=0 ; i < 229 ; i = i+1)
            @(posedge Clk);
        $stop;
    end

    always #10 Clk = ~Clk;

    always@(posedge Clk)
    begin
        temp = $fscanf(op2 , "%b" , pad_in);
        in = inreg[k];
        #5 k = k+1;
        $fwrite(op_out,"%b \t",out);
    end
```

که ورودی های تولید شده توسط کد متلب را به ماژول ما میدهد و خروجی هارا در یک فایل با نام Out_V.txt ذخیره میکند.

خروجی های حاصل از متلب نیز در فایل Out.txt ذخیره شده است.

با مقایسه این دو متوجه میشویم نتایج کاملا یکسان هستند.

توجه: در ماژولی که در وریلاگ زده شده است یک 0 در ابتدا بیشتر وجود دارد که دلیل آن ریست کردن در ابتدای کار است.

بخش گیرنده:

برای بخش گیرنده در ابتدا بیت های preamble وارد میشوند. در ابتدا وارد یک بخش به نام preamble میشویم و تا زمانی که 12 بیت 1 دریافت نشده است در آن باقی میمانیم ولی اگر دریافت شود وارد بخش Signal خواهیم شد.تا زمانی که در بخش preamble هستیم خروجی گیرنده برابر 0 خواهد بود و پس از آن و وارد شدن به بخش Signal فریم فرستاده شده را درخروجی میبینیم.

```
preamble:begin
    preamble_check <= {preamble_check[10:0] , data_in};
    mem <= 7'd0;
    scrambler_seed <= 7'd0;
    if(preamble_check[10:0] == 11'b11111111111)
        C_state <= signal;
    else
        C_state <= preamble;

end
```

```
signal:begin
    if(Counter < 23)begin
        out <= data_in;
        Counter <= Counter + 1'b1;
        signal_field <= {signal_field[22:0] , data_in};
        ready <= 0;
        en_scrambler <= 1'b0;
        rst_scrambler <= 1'b1;
    end
    else begin
        out <= data_in;
        Counter <= 0;
        signal_field <= {signal_field[22:0] , data_in};
        en_scrambler <= 1'b1;
        rst_scrambler <= 1'b0;
        in_scrambler <= data_in;
        C_state <= service;
    end
end
```

گیرنده نیز ورودی های آن مشابه فرستنده است و داده های ورودی را به صورت تک بیت دریافت میکند.

مشابه ورودی 24 بیت signal دریافت خواهیم کرد که نیازی به scramble کردن ندارد.

برای دریافت دیتافریم DATA باید ابتدا scramble_seed اولیه را با توجه به اینکه در فرستنده بخش service همگی 0 است به راحتی با یک تابع میتوان آن را دریافت کرد.

باید ابتدا به اندازه 7 کلاک صبر کرد که داده های ابتدایی service دریافت شود.سپس scramble_seed از این مرحله به بعد با تابع زیر بدست می آید:

```
scrambler data_scramble (
.in_data(data_in),
.Clk(Clk),
.scrambler_seed({mem[5:0],mem[6]^mem[3]}),
.reset(rst_scrambler),
.en(en_scrambler),
.out_data(out_scrambler)
);
```

بنابراین کافی است برای اینکه از این مرحله به بعد درست کار کند scrambler را Reset کنیم تا scrambler به درستی مقدار دهی شود که این را در این بخش میبینیم:

```
service:begin
    if(Counter < 7)begin
        out <= 0;
        Counter <= Counter + 1'b1;
        in_scrambler <= data_in;
        mem <= {mem[5:0] , data_in};
        en_scrambler <= 1'b0;
        if(Counter == 6)
        rst_scrambler <= 1'b1;
        else
        rst_scrambler <= 1'b0;
    end
    else begin
        out <= 0;
        Counter <= 0;
        in_scrambler <= data_in;
        en_scrambler <= 1'b1;
        rst_scrambler <= 1'b0;
        C_state <= service1;
    end
end
```

با این بخش scramble_seed اولیه به راحتی یافت میشود و سپس ادامه کار مشابه بخش فرستنده خواهد بود.

برای تست کردن این بخش خروجی فرستنده را به ماژول rx میدهیم و خروجی را با فایل ورودی in.txt که داده های ورودی ما هستند مقایسه خواهیم کرد.

با مقایسه دو فایل In.txt و In_v.txt متوجه خواهیم شد که ماژول به درستی کار میکند.

با مقایسه این دو متوجه میشویم که زمانی که در فرستنده بخش preamble 12 بیت 1 میفرستد در گیرنده آن را برابر 0 در نظر میگیرد و پس از آن وارد بخش signal میشود و فریم ها دقیقا یکسان هستند و پس از اتمام فریم مجدد به بخش preamble باز میگردیم.

خروجی به صورت سریال خواهد بود که دقیقا مشابه فریم ورودی ابتدا بخش سیگنال و سپس بخش service و سپس داده ها در خروجی قرار خواهند گرفت.

توجه: در ماژولی که در وریلاگ زده شده است یک 0 در ابتدا بیشتر وجود دارد که دلیل آن ریست کردن در ابتدای کار است.