ASIC & FPGA HW(2)

Parham Gilani – 400101859

سوال 1)

a)

1. استفاده از Shift Register:

- در Verilog می توانید با استفاده از ساختار shift register یک Register 64 بیتی با عرض 6 بیت پیاده سازی کنید. هر عنصر از ثبات شیفت دارای یک مقدار 6 بیتی است و این مقادیر از طریق Register ها در هر Clock جابجا می شوند.
- داده های ورودی در اولین Register بارگذاری می شوند ([0]reg)، و سپس از طریق Register 63 باقی مانده در clock بعدی منتقل می شوند.
- با اتصال خروجی آخرین رجیستر به پورت خروجی reg_out می توانید مقدار Register 64 بیتی را در هر لحظه مشاهده کنید.

2. استفاده از آرایه ای از Register ها:

- راه دیگر برای ایجاد یک Register 64 بیتی با عرض 6 بیت در Verilog با تعریف آرایه ای از Register 64 است که هر یک دارای عرض 6 بیت است.
- در حین Reset، تمام رجیسترها به صفر می رسند. در هر Clock، داده های ورودی در اولین Register بارگذاری می شوند ([0]reg) و سپس از طریق آرایه Register منتشر می شوند.
- خروجی reg_out همه رجیسترها را به ترتیب معکوس (از [63]reg تا [0]reg) به هم متصل می کند تا مقدار خروجی 64 بیتی را تشکیل دهد.

هر دو توضیح نشان می دهند که چگونه می توانید یک Register 64 بیتی با عرض 6 بیت را در Verilog با استفاده از یک Shift Register یا آرایه ای از Register ها پیاده سازی کنید.

b) یک SRL (Shift Register Lookup-Table) منبعی است که در تراشه‌های Xilinx موجود است که می‌تواند عملکردهای یک شیفت رجیستر را به روشی بهینه ارائه کند. SRL ها انرژی، مساحت کمتری مصرف می کنند و به طور کلی سریعتر هستند. در حالی که محدودیت ها و ملاحظات خاصی برای استفاده از آنها هنگام طراحی ماژول ها با استفاده از زبان های HDL وجود دارد، آنها می توانند طراحی را به روش های مختلفی بهبود بخشند. هر SRL از یک مالتی پلکسر 4-1 و 16 فلیپ فلاپ در تراشه های Xilinx تشکیل شده است. فلیپ فلاپ ها به صورت سری به هم متصل می شوند و از این رو عملکرد یک رجیستر شیفت 16 بیتی را با تمام سیگنال های کنترلی لازم مانند clock داخلی و سیگنال فعال کننده ساعت اجرا می کنند. سیگنال آدرس مالتی پلکسر می تواند برای تغییر طول رجیستر و دسترسی به هر بیت استفاده شود. به منظور cascade کردن این بلوک های 16 بیتی و ایجاد shift registers های بزرگتر، SHIFTOUT SRL اول باید به سیگنال SHIFTIN دومی متصل شود. همچنین خروجی های دو SRL باید با بیت های باقی مانده از سیگنال آدرس مالتی پلکس شوند. برای دستیابی به رفتار مطلوب، ما به سادگی داده های ورودی را به عنوان یک bus با عرض 6 مشخص می کنیم. این به ابزار سنتز جهت استفاده از 6 64 SRL بیتی جداگانه را می دهد.

سوال 3)

a) معماری Xilinx DSP48 یک بلوک تخصصی پردازش سیگنال دیجیتال (DSP) است که در FPGAهای Xilinx یافت می‌شود که برای اجرای کارآمد عملیات حسابی پیچیده که معمولاً در برنامه‌های پردازش سیگنال استفاده می‌شود، طراحی شده است. بلوک DSP48 یک بلوک سخت افزاری اختصاصی و قابل تنظیم است که می تواند برای اجرای توابع حسابی مختلف مانند ضرب، جمع، تفریق، و غیره استفاده شود.

ویژگی های کلیدی معماری Xilinx DSP48 عبارتند از:

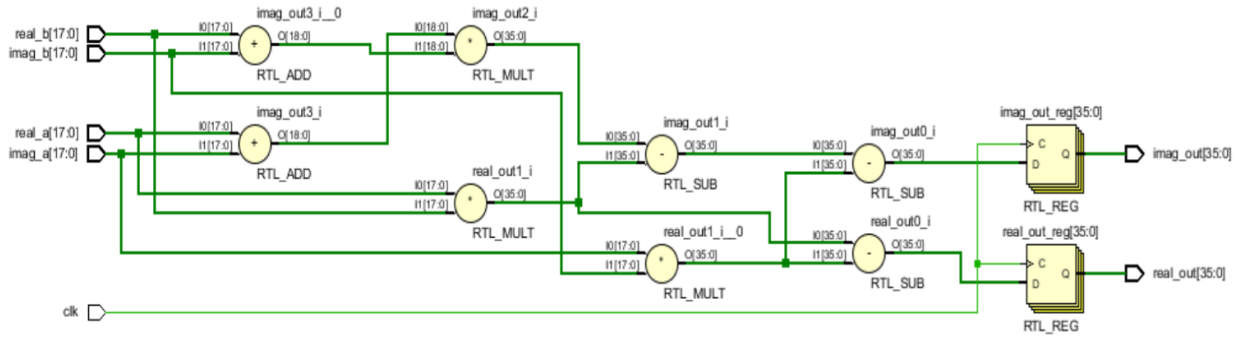
1. قابلیت‌های Multiply-Acumulate (MAC): بلوک DSP48 می‌تواند یک عملیات جمع چند برابری را در یک Clock انجام دهد که آن را برای پیاده‌سازی فیلترها، تبدیل‌ها و الگوریتم های DSP بسیار کارآمد می‌کند.
2. دقت قابل تنظیم: بلوک DSP48 را می توان برای پشتیبانی از عرض های مختلف داده و فرمت های اعداد پیکربندی کرد و به طراحان اجازه می دهد تا عملکرد و استفاده از منابع را برای نیازهای کاربردی خاص خود بهینه کنند.

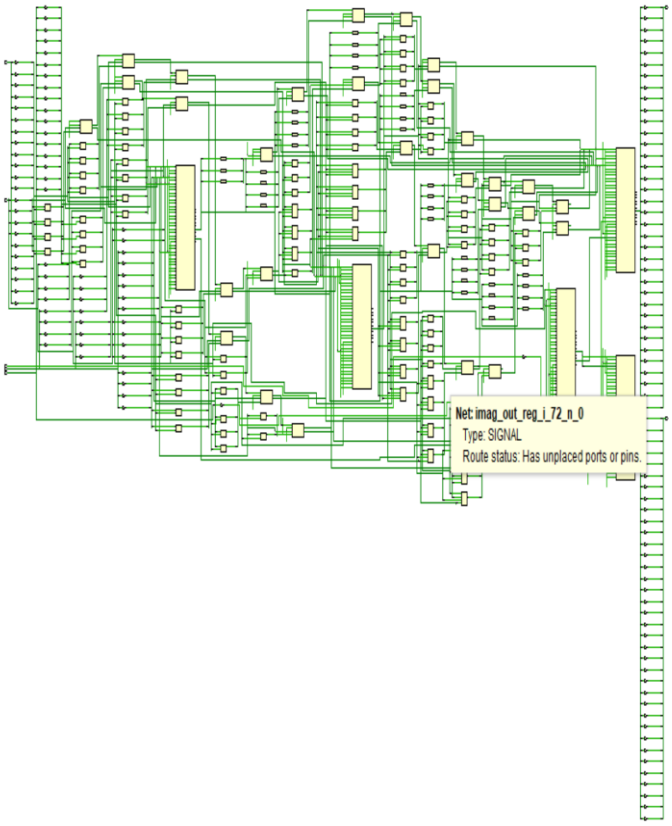
3. معماری Pipeline: بلوک DSP48 با معماری Pipeline طراحی شده است که امکان عملیات با سرعت بالا و پردازش موازی کارآمد چندین عملیات حسابی را فراهم می کند.
4. پشتیبانی از توابع مختلف حسابی: علاوه بر عملیات MAC، بلوک DSP48 را می توان برای اجرای سایر توابع حسابی مانند عملیات جمع، تفریق، شیفت و منطقی نیز مورد استفاده قرار داد.
5. ویژگی های داخلی برای بهینه سازی: بلوک Xilinx DSP48 شامل ویژگی هایی مانند مراحل پیش جمع کننده و پس از جمع، carry chains، Register های ورودی و خروجی است که به بهینه سازی عملکرد و کاهش تاخیرهای critical path کمک می کند.

به طور کلی، معماری Xilinx DSP48 یک بلوک سخت افزاری قدرتمند و کارآمد را برای اجرای عملیات پیچیده حسابی در سیستم های پردازش سیگنال مبتنی بر FPGA در اختیار طراحان قرار می دهد. با استفاده از قابلیت‌های بلوک DSP48، طراحان می‌توانند به پردازش با کارایی بالا و تاخیر کم برای طیف گسترده‌ای از برنامه‌های DSP دست یابند.

b) کد ضرب کننده 18 بیتی مختلط بدین گونه است و نتیجه سنتز نیز امده است.

```
module Q3(
    input clk,
    input [17:0] real_a,
    input [17:0] imag_a,
    input [17:0] real_b,
    input [17:0] imag_b,
    output reg [35:0] real_out,
    output reg [35:0] imag_out
);

    always @(posedge clk) begin
        real_out <= real_a * real_b - imag_a * imag_b;
        imag_out <= (real_a + imag_a) * (real_b + imag_b) - real_a * real_b + imag_a * imag_b;
    end

endmodule
```





همانطور که دیده میشود مدار دارای پایپلاین (به دلیل وجود Clock) و 5 تا DSP است چون با بهینه سازی عمل ضرب به جای 4 تا از 3 تا ضرب بهره بردیم که موجب به کاهش تعداد DSP ها میشود. باید بگویم چون مدار ما دارای clock است پس مدار دارای پایپلاین نیز هست.

KARATSUBA'S TRICK

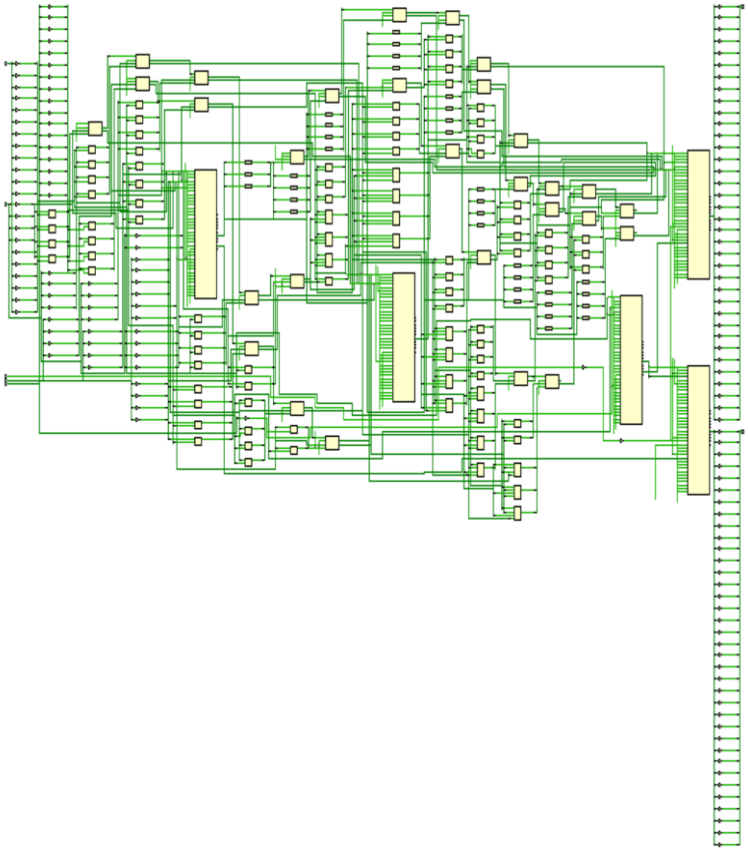
$$\text{end result} = (ac)10^{n} + (ad + bc)10^{n/2} + (bd)$$

ac & **bd** can be recursively computed as usual

ad + bc is equivalent to **(a+b)(c+d) - ac - bd**

$$= (ac + ad + bc + bd) - ac - bd$$
$$= ad + bc$$

So, instead of computing **ad** & **bc** as two separate subproblems, let's just compute **(a+b)(c+d)** instead!



c) همان کد را برای 19 بیتی انجام دادیم. همانطور که دیده میشود تعداد DSP ها تغییری نمیکند و چون در اصل تعداد عمل های ضرب ما تغییری نمیکند و همان 3 باقی میماند و صرفا به جای محاسبه 18 بیتی 1 بیت اضافه کردیم.

d) نتیجه شبیه سازی نیز اینگونه است.

```
# //
# Loading project test
ModelSim> vsim work.Q3_tb
# vsim work.Q3_tb
# Start time: 23:22:30 on Mar 19,2024
# ** Note: (vsim-8009) Loading existing optimized design _opt7
# Loading work.Q3_tb(fast)
VSIM 2> run -all
# (0000000000000011 + 1111111111111100 * j) * (1111111111111100 + 1111111111111100 * j) = 0000000000000010101111111111100100 + 1111111111110101100000000000100100 * j
# (0000000000000010 + 1111111111111111 * j) * (0000000000000010 + 1111111111111100 * j) = 0000000000000001010000000000000000 + 1111111111111100111111111111111110 * j
# (0000000000000001 + 0000000000000011 * j) * (0000000000000100 + 0000000000000010 * j) = 1111111111111111111111111111111110 + 0000000000000000000000000000011010 * j
# (0000000000000100 + 0000000000000100 * j) * (0000000000000010 + 0000000000000000 * j) = 0000000000000000000000000000001000 + 0000000000000000000000000000001000 * j
# (1111111111111100 + 0000000000000000 * j) * (1111111111111100 + 1111111111111111 * j) = 1111111111111100000000000000010000 + 1111111111111101000000000000000100 * j
# (1111111111111110 + 0000000000000001 * j) * (0000000000000100 + 1111111111111111 * j) = 0000000000000001011111111111111001 + 1111111111111111000000000000000100 * j
# (1111111111111101 + 0000000000000100 * j) * (1111111111111101 + 1111111111111101 * j) = 1111111111111101100000000000000010101 + 0000000000000001011111111111100101 * j
# (1111111111111110 + 1111111111111101 * j) * (1111111111111110 + 0000000000000000 * j) = 1111111111111110000000000000000100 + 1111111111111101100000000000000110 * j
# (1111111111111101 + 0000000000000100 * j) * (0000000000000001 + 0000000000000000 * j) = 0000000000000000011111111111111101 + 0000000000000000000000000000000100 * j
# (1111111111111100 + 0000000000000000 * j) * (0000000000000000 + 1111111111111101 * j) = 0000000000000000000000000000000000 + 1111111111111100100000000000001100 * j
# ** Note: $stop    : C:/Users/gilan/OneDrive/Desktop/ASIC & FPGA_HW(2)_400101859/Q3/Q3_tb.v(29)
#    Time: 400 ns  Iteration: 1  Instance: /Q3_tb
# Break at C:/Users/gilan/OneDrive/Desktop/ASIC & FPGA_HW(2)_400101859/Q3/Q3_tb.v line 29

VSIM 3>
```

سوال 5)

a) با توجه به 11 بیتی بودن داده نیاز به 4 بیت پریتی داریم که همانطور که در کد مشاهده میشود بیت های پریتی باقی مانده بر 2 حاصل جمع تعدادی بیت داده است که در عمل XOR داده ها است و پس از Random Generate کردن تعدادی عدد 11 بیتی و حساب کردن 4 بیت پریتی به ان عدد های 11 بیتی و 15 بیتی را در فایل های مناسب ذخیره کردم.

```
%% Q5_a
clc; clear; close all;

n = 20; % Number of sets of 11-bit data to generate

% Save 11-bit & 15-bit file
fid11 = fopen('data_11bit.txt', 'w');
fid15 = fopen('data_15bit.txt', 'w');

for i = 1:n
    % Generate 11-bit random data
    data_11bit = randi([0,1],1,11);

    % Calculate parity bits
    p1 = mod(sum(data_11bit([1,3,5,7,9,11])),2);
    p2 = mod(sum(data_11bit([2,3,6,7,10,11])),2);
    p4 = mod(sum(data_11bit([4,5,6,7])),2);
    p8 = mod(sum(data_11bit([8,9,10,11])),2);

    % Create 15-bit data with parity bits
    data_15bit = [p1 p2 p4 p8 data_11bit];

    % Save 11-bit data
    fprintf(fid11,'%d',data_11bit);
    if (i~=n)
        fprintf(fid11,'\n');
    end

    % Save 15-bit data
    fprintf(fid15,'%d',data_15bit);
    if (i~=n)
        fprintf(fid15,'\n');
    end

end
```

b) در این قسمت هم کد مورد نظر را نوشتم که در هر مرحله Parity ها را حساب میکند و اگر انها با 4 رقم MSB ورودی یکی نبود Valid را 1 میکند و بیت های داده اصلی را خروجی میدهد در غیر این صورت Valid را 0 و Error_count را یک مقدار زیاد میکند.

```
module receiver (
    input wire clk,
    input wire [14:0] input_data,
    output reg [10:0] output_data,
    output reg valid,
    output reg [4:0] error_count
    );

    wire [3:0] parity;
    assign parity[3] = input_data[0] ^ input_data[2] ^ input_data[4] ^ input_data[6] ^ input_data[8] ^ input_data[10];
    assign parity[2] = input_data[0] ^ input_data[1] ^ input_data[4] ^ input_data[5] ^ input_data[8] ^ input_data[9];
    assign parity[1] = input_data[4] ^ input_data[5] ^ input_data[6] ^ input_data[7];
    assign parity[0] = input_data[0] ^ input_data[1] ^ input_data[2] ^ input_data[3];

    always @(posedge clk) begin

        if (parity == input_data[14:11]) begin
            output_data <= input_data[10:0];
            valid <= 1;
        end else begin
            error_count <= error_count + 1;
            output_data <= 0;
            valid <= 0;
        end

    end

    initial begin
        output_data <= 0;
        error_count <= 0;
        valid <= 0;
    end

endmodule
```

c,d) در این مرحله صرفا تست بنچی نوشتم که داده های 15 بیتی تولید شده توسط کد قسمت قبل را به ماژول بدهد و خروجی متناسب با ان را در فایلی بنویسد و سپس توسط متلب محتوای فایل ها را مقایسه کردم تا از صحت عملکرد انها مطمئن شوم. درقسمت اضافه کردن داده های noisy نیز مقادیر رندم به فایل اضافه کردم و در خروجی به ازای همان ورودی ها مقدار Valid برابر 0 شد.(فایل ها پیوست شده اند).

```
%% Q5_c
clc; clear; close all;

fid1 = fopen('data_11bit.txt', 'r');
fid2 = fopen('receiver_output.txt', 'r');

data1 = fscanf(fid1, '%s');
data2 = fscanf(fid2, '%s');

if (data1 == data2)
    disp("Input and Output are the same");
end
if (data1 ~= data2)
    disp("Input and Output are not the same");
end

fclose(fid1);
fclose(fid2);
```

```
module receiver_tb();

    reg [14:0] datas [19:0];
    reg clk;
    integer i , receiver_output;

    always @(clk) #10 clk <= ~clk;

    receiver uut (.clk(clk),.input_data(datas[i]),.output_data(),.valid(),.error_count());

    initial begin
        $readmemb("errordata_15bit.txt",datas);
        receiver_output = $fopen("receiver_output2.txt", "w");
        clk <= 1;
        for ( i = 0 ; i < 20 ; i = i + 1) begin
            @(posedge clk)
            #10 $display("i = %0d , input_data = %b : output_data = %b , valid = %b , error_count = %0d" ,
                i+1 , datas[i] , uut.output_data , uut.valid , uut.error_count);
            $fwrite(receiver_output, "%b", uut.output_data);
            if (i!=19) $fwrite(receiver_output , "\n");
        end
        $fclose(receiver_output);
        $stop;
    end

endmodule
```

سوال 4)

a,b) در این قسمت صرفا یک FSM کشیدم و تعدادی حالت را در ان گرفتم و با توجه به current_state و W تشخیص دادم مقدار خروجی و next_state چیست و در هر Clock مقدار current_state را برابر next_state گذاشتم. در تست بنچ نیز 20 بار به ماژول مقدار دادم و خروجی را مشاهده کردم.

```
module Sequence_Detector_Moore_tb();

  reg clk;
  reg W , reset , valid;
  integer i;

  always @(clk) #10 clk <= ~clk;

  initial begin
    clk = 1;
    valid = 1;
    reset = 0;
    #10
    reset = 1;
    for(i = 0; i<20 ; i = i + 1) begin
      W = $random;
      @(posedge clk);
      #10 $display ("W = %b , valid = %b , reset = %b : z = %b"
        , W , valid , reset , uut.z);
    end
    $stop;
  end

  Sequence_Detector_Moore uut (
    .clk(clk),
    .valid(valid),
    .reset(reset),
    .W(W),
    .z()
  );

endmodule
```

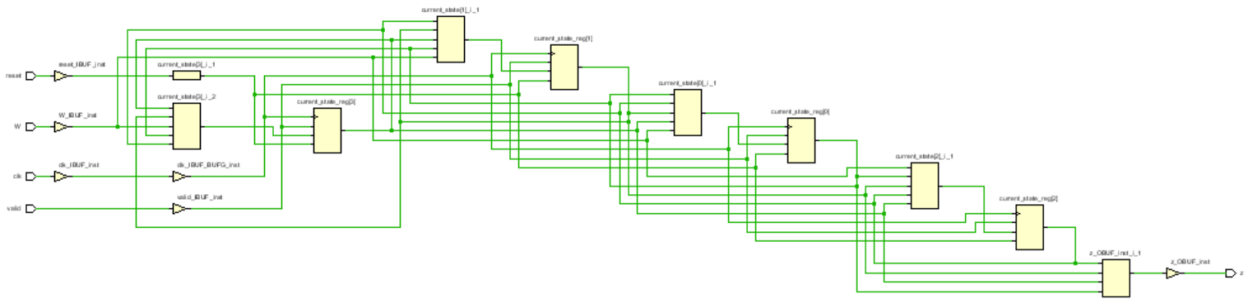
```
module Sequence_Detector_Moore(
    input clk,
    input valid,
    input reset,
    input W,
    output z
    );

    parameter S_0 = 0 , S_1 = 1 , S_10 = 2 , S_101 = 3 , S_1011 = 4 , S_10110 = 5 , S_101101 = 6 , S_1011011 = 7 , S_10110110 = 8;
    reg [3:0] current_state;
    wire [3:0] next_state;

    assign next_state = (current_state == S_0)? ((W)? S_1 : S_0) :
                        (current_state == S_1)? ((W)? S_1 : S_10) :
                        (current_state == S_10)? ((W)? S_101 : S_0) :
                        (current_state == S_101)? ((W)? S_1011 : S_10) :
                        (current_state == S_1011)? ((W)? S_1 : S_10110) :
                        (current_state == S_10110)? ((W)? S_101101 : S_0) :
                        (current_state == S_101101)? ((W)? S_1011011 : S_10) :
                        (current_state == S_1011011)? ((W)? S_1 : S_10110110) :
                        (current_state == S_10110110)? ((W)? S_101101 : S_0) : S_0;
    assign z = (current_state == S_10110110);

    always @(posedge clk) begin
        if(!reset) current_state <= S_0;
        else if(valid) current_state <= next_state;
    end

endmodule
```

c) برای Mealy بودن نیز مقدار z را وقتی در 1011011هستیم و مقدار ورودی برابر 0 است و valid است 1 کردم و تغییر خاصی در کلیت کد ایجاد نشده است.

```
module Sequence_Detector_Mealy_tb();

  reg clk;
  reg W , reset , valid;
  integer i;

  always @(clk) #10 clk <= ~clk;

  initial begin
    clk = 1;
    valid = 1;
    reset = 0;
    #10
    reset = 1;
    for(i = 0; i<20 ; i = i + 1) begin
      W = $random;
      #10
      @(posedge clk)
      $display ("W = %b , valid = %b , reset = %b : z = %b"
        , W , valid , reset , uut.z);
    end
    $stop;
  end

  Sequence_Detector_Mealy uut (
    .clk(clk),
    .valid(valid),
    .reset(reset),
    .W(W),
    .z()
  );

endmodule
```

```
module Sequence_Detector_Mealy(
    input clk,
    input valid,
    input reset,
    input W,
    output z
    );

    parameter S_0 = 0 , S_1 = 1 , S_10 = 2 , S_101 = 3 , S_1011 = 4 , S_10110 = 5 , S_101101 = 6 , S_1011011 = 7 , S_10110110 = 8;
    reg [3:0] current_state;
    wire [3:0] next_state;

    assign next_state = (current_state == S_0)? ((W)? S_1 : S_0) :
                        (current_state == S_1)? ((W)? S_1 : S_10) :
                        (current_state == S_10)? ((W)? S_101 : S_0) :
                        (current_state == S_101)? ((W)? S_1011 : S_10) :
                        (current_state == S_1011)? ((W)? S_1 : S_10110) :
                        (current_state == S_10110)? ((W)? S_101101 : S_0) :
                        (current_state == S_101101)? ((W)? S_1011011 : S_10) :
                        (current_state == S_1011011)? ((W)? S_1 : S_10110110) :
                        (current_state == S_10110110)? ((W)? S_101101 : S_0) : S_0;
    assign z = (current_state == S_1011011) && (W == 0);

    always @(posedge clk) begin
        if(!reset) current_state <= S_0;
        else if(valid) current_state <= next_state;
    end

endmodule
```

d) در هر 2 حالت Mealy و Moore خروجی سنتز یکسان است چون در یک وقتی که به حالت 10110110 رسیدیم خروجی 1 میشود و در دیگری وقتی که در حالت 1011011هستیم و ورودی 0 است خروجی 1 میشود که معادل این است که به حالت 10110110 برسیم. در حالت کلی سنتز مدار در حالت Moore ساده تر از Mealy است در مدار های بزرگتر و ممکن است که منجر به پیچیدگی مدار در حالت Mealy شویم.



سوال 2)

a) طبق کد بالا که کد متلب است ابتدا بین 0 و 2π 1024 مقدار تولید کردم و با استفاده از quantizer ان را کوانتیزه کردم و با استفاده از num2bin اعداد را به باینری تبدیل کردم و سپس اعداد باینری را در فایل های sin.txt و cos.txt ذخیره کردم.
در فایل وریلاگ نیز علاوه بر ورودی و خروجی در initial صرفا همان داده ها را لود کردم و در هر clock مقدار خروجی را با توجه به ادرس و توان فرکانس مقدار دهی کردم. نحوه چند برابر کردن فرکانس را چون در اصل 2 به توان است میتوان با shift پیاده سازی کرد.
بدیهی است که چون رجیستر pow را 3 بیتی در نظر گرفتم نمیتوان بیشتر از 128 فرکانس را بیشتر کرد که برای این کار باید تعداد بیت های رجیتر های pow و address را زیاد کرد. با مقدار دهی بیشتر از اندازه رجیستر اروری نمیدهد و overflow رخ میدهد که منجر به جواب اشتباه میشود.

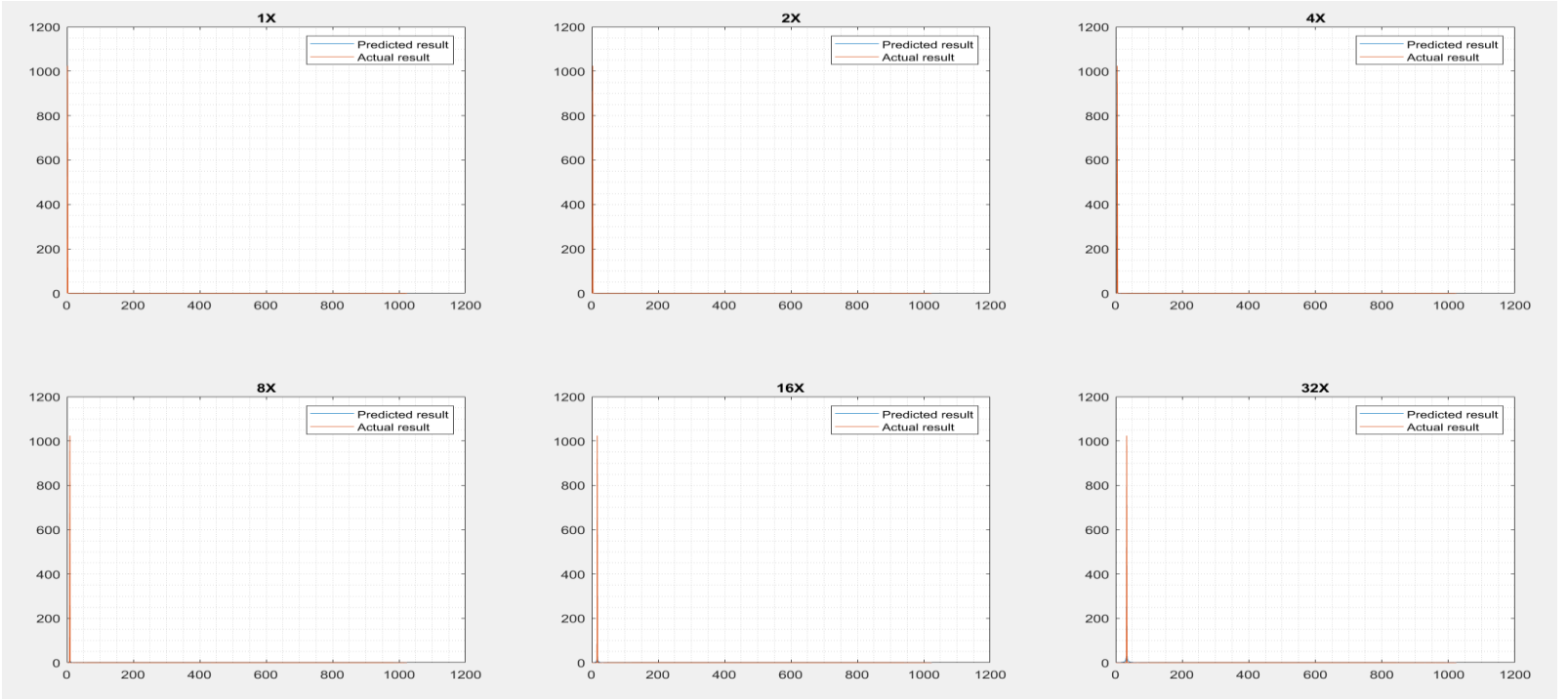
```
%% Q2_a
clc; clear; close all;

mem_len = 1024;

x = linspace(0, 2*pi, mem_len);
q = quantizer([16,14]);

y_sin = num2bin(q, sin(x));
y_cos = num2bin(q, cos(x));

sin = fopen('sin.txt', 'w');
cos = fopen('cos.txt', 'w');

for i = 1 : mem_len
    fprintf(sin, '%s', y_sin(i,:));
    if (i ~= mem_len)
        fprintf(sin, '\n');
    end
    fprintf(cos, '%s', y_cos(i,:));
    if (i ~= mem_len)
        fprintf(cos, '\n');
    end
end
```

```
module Q2(
    input wire clk,
    input wire [2:0] pow,
    input wire [15:0] address,
    output reg [15:0] sin_output,
    output reg [15:0] cos_output
    );

    reg [15:0] sin [1023:0];
    reg [15:0] cos [1023:0];

    initial begin
        $readmemb("sin.txt", sin);
        $readmemb("cos.txt", cos);
    end

    always @(posedge clk) begin
        sin_output <= sin[(address << pow) % 1024];
        cos_output <= cos[(address << pow) % 1024];
    end

endmodule
```

b) در تست بنچ نیز صرفا در initial به ازای فرکانس های 1 و 2 و 4 و 8 و 16 و 32 و همه ادرس ها که 1024 تا است (طول پریود سینوسی بود) داده های سینوسی را در sin_output.txt و کسینوسی را در cos_output.txt ذخیره کردم. در متلب نیز در اول همان کار قسمت قبل را انجام دادم که از فایل خواندم در یک لوپ انها را با مقادیر مورد انتظار plot کردم که دقیقا همپوشانی داشتند.

```
module Q2_tb();
    reg clk;
    reg [2:0] pow;
    reg [15:0] address;

    integer i, j, sin_txt, cos_txt;

    initial begin

        sin_txt = $fopen("sin_output.txt", "w");
        cos_txt = $fopen("cos_output.txt", "w");
        clk <= 1;

        for(i = 0; i < 6 ; i = i + 1) begin
            pow <= i;
            for ( j = 0 ; j < 1024 ; j = j + 1 ) begin
                address <= j;
                @(posedge clk)
                #10
                $fwrite(sin_txt, "%b", $signed(uut.sin_output));
                $fwrite(cos_txt, "%b", $signed(uut.cos_output));
                if (j != 1023 || i != 5) begin
                    $fwrite(sin_txt, "\n");
                    $fwrite(cos_txt, "\n");
                end
            end
        end

        $fclose(sin_txt);
        $fclose(cos_txt);
        $stop;

    end

    always @(clk) #10 clk <= ~clk;

    Q2 uut(
        .clk(clk),
        .pow(pow),
        .address(address),
        .sin_output(),
        .cos_output()
    );

endmodule
```

```
%% Q2_b
clc; clear; close all;

mem_len = 1024;
x = linspace(0, 2*pi, mem_len);
q = quantizer([16,14]);

sin_output_string = readlines('sin_output.txt');
cos_output_string = readlines('cos_output.txt');

sin_output = cell2mat(bin2num(q, sin_output_string));
cos_output = cell2mat(bin2num(q, cos_output_string));

for n = 0:5
    predicted_values = exp(i*pow2(n)*x);
    actual_values = cos_output(n*mem_len+1:(n+1)*mem_len) + i*sin_output(n*mem_len+1:(n+1)*mem_len);
    subplot(2, 3, n+1);
    plot(abs(fft(predicted_values)));
    hold on
    plot(abs(fft(actual_values)));
    title(string(pow2(n)) + 'X');
    legend(["Predicted result","Actual result"]);
    grid minor
end
```



c) FPGA ها معمولاً شامل هسته های نرم افزاری و سخت افزاری هستند تا پیاده سازی های طراحی انعطاف پذیر و کارآمد را ممکن کنند.

هسته های نرم افزاری در FPGA ها معمولاً با استفاده از ابزارهای سنتز سطح بالا که الگوریتم های نرم افزار را به توضیحات سخت افزاری تبدیل می کنند، پیاده سازی می شوند. این هسته‌ها برای پیاده سازی عملکردهای پیچیده مانند processors، memory controllers یا communication interfaces استفاده می‌شوند. نمونه هایی از هسته های نرم افزاری عبارتند از MicroBlaze و Zynq Processing System در Xilinx FPGA.

هسته های سخت افزاری در FPGA بلوک های IP از پیش طراحی شده ای هستند که می توانند در FPGA پیاده سازی شوند. این هسته ها برای عملکردهای خاص بهینه شده اند و می توانند به راحتی در یک طراحی ادغام شوند. نمونه‌هایی از هسته‌های سخت‌افزاری عبارتند از Ethernet MAC، PCI Express یا DDR memory controllers.

DDS تکنیکی است که در پردازش سیگنال دیجیتال برای تولید شکل موج های آنالوگ با دقت و پایداری بالا استفاده می شود. DDS معمولاً در کاربردهایی مانند سنتز فرکانس، تولید سیگنال و سنتز شکل موج استفاده می شود.

Xilinx هسته های IP DDS را ارائه می دهد که می توانند در FPGA های خود برای پیاده سازی عملکرد DDS استفاده شوند. دیتاشیت هسته IP Xilinx DDS اطلاعاتی در مورد ویژگی‌های هسته، interfaces، configuration options و ویژگی‌های عملکرد ارائه می‌دهد. با مطالعه دیتاشیت، می توان درک کرد که چگونه می توان هسته IP DDS را در یک طراحی FPGA ادغام کرد و آن را برای برآوردن نیازهای خاص پیکربندی کرد.

به طور کلی، DDS یک تکنیک قدرتمند برای تولید شکل موج های آنالوگ دقیق است و Xilinx هسته های IP را برای تسهیل اجرای آن در طرح های FPGA ارائه می دهد.

از DDS می توان برای افزایش فرکانس سیگنال با فرکانس های بالا با کنترل دقیق استفاده کرد. با DDS پیاده سازی شده در FPGA، کاربران به راحتی می توانند سیگنال های فرکانس بالا را با سنتز شکل موج در فرکانس مورد نظر تولید کنند.

با استفاده از DDS در یک FPGA، طراحان می توانند به تولید سیگنال با فرکانس بالا با وضوح فرکانس عالی و پایداری دست یابند. انعطاف پذیری FPGA ها امکان تنظیم فرکانس خروجی، فاز و دامنه سیگنال تولید شده را فراهم می کند و آن را برای برنامه هایی که نیاز به تنظیم فرکانس سریع دارند ایده آل می کند.

علاوه بر این، ماهیت قابل برنامه ریزی FPGA به طراحان امکان می‌دهد تا تکنیک‌های پردازش سیگنال پیشرفته را در کنار DDS پیاده سازی کنند تا کیفیت و عملکرد سیگنال را افزایش دهند.