

به نام خدا



دانشگاه صنعتی شریف

دانشکده مهندسی برق

طراحی سیستم‌های مبتنی بر FPGA/ASIC

تمرین سری ۱

استاد درس: دکتر شعبانی

مهران مظاهری

۹۸۱۰۲۳۴۶

۲۰ آبان

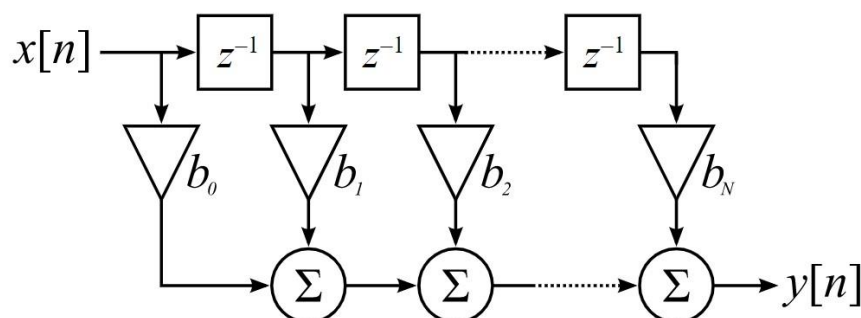
۱. فیلتر FIR

A. ساختار کلی فیلتر و پیاده‌سازی یک فیلتر با 10 Tap

فیلتر FIR فیلتری است که خروجی آن در هر مرحله، تنها تابعی خطی از ورودی همان مرحله و ورودی‌های قبلی باشد:

$$Y[n] = b_0x[n] + b_1x[n-1] + b_2x[n-2] + \dots + b_Nx[n-N]$$

با توجه به رابطه بالا، دیاگرام بلوکی این فیلتر به شکل زیر خواهد بود:



خروجی از جمع (#تعداد tapها) حاصل ضرب دو عدد (#عرض هر ضریب) بیتی و (#عرض عدد ورودی) بدست می‌آید. پس عرض خروجی برابر با $\text{Coef_length} + \text{Input_length} + \text{clog2}(\text{numberOfTaps})$ است.

پس در سوال ما خروجی $20 = 4 + 8 + 8$ بیتی خواهد بود.

ماژول این بخش در فایل Q1/RTL/tenTapFIRFilter.v پیاده‌سازی شده است. این ماژول کاملاً پارامتریزه شده است و قابلیت تغییر TAP_NUMBER، INPUT_LENGTH و COEF_LENGTH را دارد.

B. ضرایب symmetric

در فیلترهای با ضرایب متقارن رابطه زیر برقرار است:

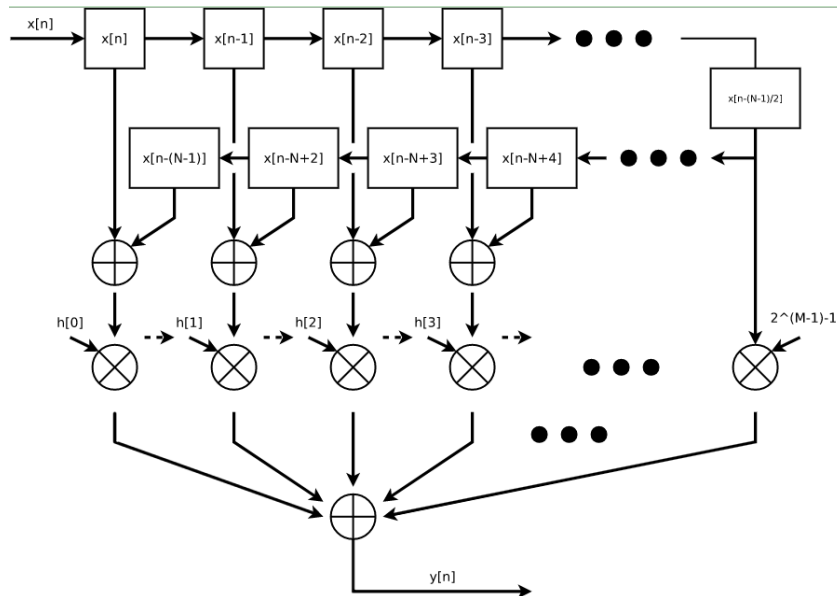
$$h[k] = h[N-1-k]$$

پس رابطه خروجی بر حسب دنباله ورودی نیز به شکل زیر قابل بازنویسی است:

$$y[n] = \sum_{k=0}^{N-1} h[k] x[n-k]$$

$$= h[M] x[n-M] + \sum_{k=0}^{\frac{1}{2}(N-1)-1} h[k] (x[n-k] + x[n+k-(N-1)])$$

و بلوک دیاگرام فیلتر به شکل زیر خواهد بود:



با توجه به بلوک دیاگرام بالا، برای طراحی فیلتری با ۹ ضریب باید دنباله ورودی به اندازه ۱۷ ورودی را ذخیره کنیم؛

با توجه به این مورد تعداد بیت خروجی که باید در نظر بگیریم برابر است با

$$\$clog2(\#numberOfCoefs) + InputLength + CoefLength + 1$$

ماژول این بخش در فایل Q1/RTL/nineCoefSymmetricFilter.v پیاده سازی شده است. این ماژول کاملاً

پارامتریزه شده است و قابلیت تغییر COEF_NUMBER، INPUT_LENGTH و COEF_LENGTH را دارد.

C. مزیت ماژول قسمت B نسبت به A

در ماژول قسمت B در حالت تعداد tapهای برابر با ماژول قسمت A از نصف تعداد ضرب کننده استفاده می کند؛ به عبارت دیگر به ازای ماژول های یکسان ظرفیت فیلتر قسمت B تنها با هزینه اضافه کردن تعدادی جمع کننده، دو برابر فیلتر قسمت A خواهد بود.

D. ماژول با ضرایب فقط {0, -1, 1}

در این مازول ضرب کننده با یک مالتی پلکسر ۳ به ۱ جایگزین میشود که ساده تر از پیاده سازی ضرب کننده است.

کد این مازول نیز در فایل های این سوال آمده است.

۲. مازول Debouncer

مبنای کار Debouncer به این شکل است که اگر ورودی با خروجی متفاوت باشد شروع به شمردن counter میکند و اگر این تفاوت مانا بود (counter پیوسته در سیکل های متوالی در حال شمارش بود) و ناشی از نویز نبود، خروجی را تغییر میدهد.

ماژول این بخش در فایل Q2/RTL/Debouncer.v پیاده سازی شده است.

۳. صحت سنجی با استفاده از matlab

A. موج سینوسی fixed point

برای این منظور از کد متلب زیر برای ساختن فایل باینری مموری استفاده میکنیم:

```
clc
clear
t = linspace(0,2*pi,1024);
%generate function:
x1 = sin(t);
x2 = cos(t);
%make functions' output fixed-point:
x1_f = fi(x1,1,16, 14, 'RoundingMethod', 'Floor');
x2_f = fi(x2,1,16, 14, 'RoundingMethod', 'Floor');
%convert to binary:
x1_bin = bin(x1_f);
x2_bin = bin(x2_f);
%save binary data to file:
x1_file = fopen('sin.mem','w');
x2_file = fopen('cos.mem','w');
fwrite(x1_file,x1_bin)
fwrite(x2_file,x2_bin)
fclose(x1_file);
fclose(x2_file);
```

کد بالا در فایل Q3/Matlab/fi_sin.m آمده است. فایل های حاوی مقادیر باینری sin و cos در فولدر

Q3/Matlab/sin.mem و Q3/Matlab/cos.mem آمده است. نمونه چند مقدار اولیه این دوفایل در زیر قابل

مشاهده است:

sin.mem:

0000000000000000
0000000001100100
0000000011001001
0000000100101101
0000000110010010
0000000111110111
0000001001011011
0000001011000000
0000001100100100
0000001110001001
0000001111101101
0000010001010010
0000010010110110
0000010100011010
0000010101111111
0000010111100011
0000011001000111
0000011010101011
0000011100001111
0000011101110011
0000011111010111
0000100000111011
0000100010011111

Cos.mem:

0100000000000000
0011111111111111
0011111111111110
0011111111111101
0011111111111011
0011111111111000
0011111111110100
0011111111110000
0011111111101100
0011111111100110
0011111111100001
0011111111011010
0011111111010011
0011111111001011
0011111111000011
0011111110111010
0011111110110000
0011111110100110
0011111110011011
0011111110010000
0011111110000100
0011111101110111
0011111101101010

این فایل‌ها در مازول لود خواهند شد و به عنوان LookUpTable در مازول ضرب‌کننده فرکانسی استفاده میشود.
ماژول frequency_multiplier در زیر قابل مشاهده است:

```
module FrequencyMultiplier(  
    input wire clk,  
    input wire reset,  
    input wire [2:0]frequency_select,  
    output reg [15:0]sin_signal,  
    output reg [15:0]cos_signal  
);  
//----- Internal Signals  
Declaration  
    reg [9:0] counter;  
    reg [15:0]sinus[0:1023];  
    reg [15:0]cosine[0:1023];  
//----- initialize memmory  
    initial begin  
        $display("Loading memmory.");  
        $readmemb("sin.mem", sinus);  
        $readmemb("cos.mem", cosine);  
    end  
//----- Sequential Logic  
    always @(posedge clk) begin  
        if (reset) begin  
            counter <= 0;  
            sin_signal <= sinus[0];  
            cos_signal <= cosine[0];  
        end else begin  
            case (frequency_select)  
                3'd1: counter <= counter + 1'b1;  
                3'd2: counter <= counter + 2'b10;  
                3'd3: counter <= counter + 3'b100;  
                3'd4: counter <= counter + 4'b1000;  
                3'd5: counter <= counter + 5'b10000;  
                default: counter <= counter;  
            endcase  
            sin_signal <= sinus[counter];  
            cos_signal <= cosine[counter];  
        end  
    end  
endmodule
```

فایل مازول بالا در Q3/RTL/FrequencyMultiplier.v آمده است. مازول بالا قابلیت تولید سیگنال‌های خروجی با فرکانس برابر، دوبرابر، چهاربرابر، هشت برابر و شانزده برابر سیگنال اصلی را دارد که توسط سیگنال ورودی frequency_select قابل تنظیم است.

B. تست‌بنچ برای مازول ضرب‌کننده فرکانسی

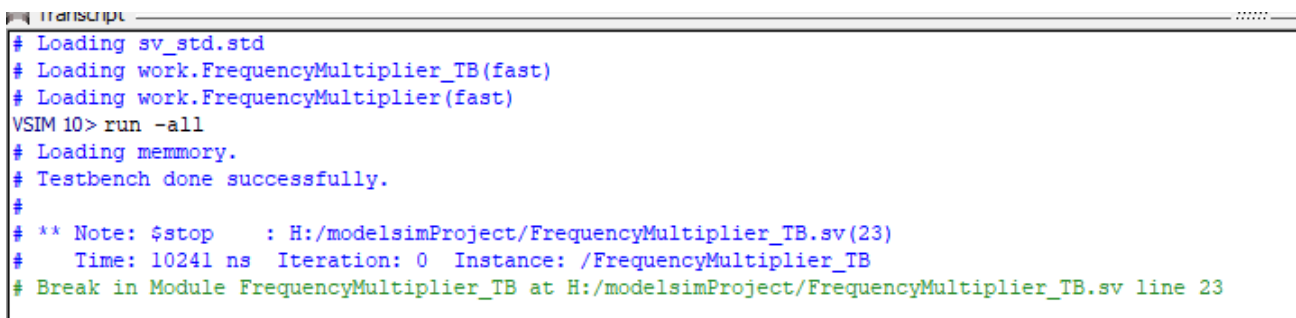
فایل تست‌بنچ زیر برای حالت فرکانس دو برابر پیاده‌سازی شده است که در فایل Q3/TB/FrequencyMultiplier_TB.sv آمده است.

```

`timescale 1ns/1ns
module FrequencyMultiplier_TB;
    reg clk = 1;
    always #5 clk = ~clk;
    reg reset;
    reg [2:0] frequency_select = 3'd2;
    integer sin_file, cos_file;
    initial begin
        sin_file = $fopen("sin.txt", "w");
        cos_file = $fopen("cos.txt", "w");
        reset = 1;
        #2
        reset = 0;
        for (int i=0; i<1024; ++i) begin
            @(posedge clk);
            #1
            $fwrite(sin_file, "%x\n", uut.sin_signal);
            $fwrite(cos_file, "%x\n", uut.cos_signal);
        end
        $fclose(sin_file);
        $fclose(cos_file);
        $display("Testbench done successfully.\n");
        $stop;
    end
    FrequencyMultiplier uut(.clk(clk), .reset(reset),
        .frequency_select(frequency_select), .sin_signal(), .cos_signal());
endmodule

```

نتیجه اجرای تست‌بنچ بالا در تصویر زیر آمده است و دو فایل txt خواسته شده در Q3/TB/(sin,cos).txt آمده است که به شکل اعداد hex ذخیره شده‌اند.



```

Transcript
# Loading sv_std.std
# Loading work.FrequencyMultiplier_TB(fast)
# Loading work.FrequencyMultiplier(fast)
VSIM 10> run -all
# Loading memory.
# Testbench done successfully.
#
# ** Note: $stop      : H:/modelsimProject/FrequencyMultiplier_TB.sv(23)
#   Time: 10241 ns  Iteration: 0  Instance: /FrequencyMultiplier_TB
# Break in Module FrequencyMultiplier_TB at H:/modelsimProject/FrequencyMultiplier_TB.sv line 23

```

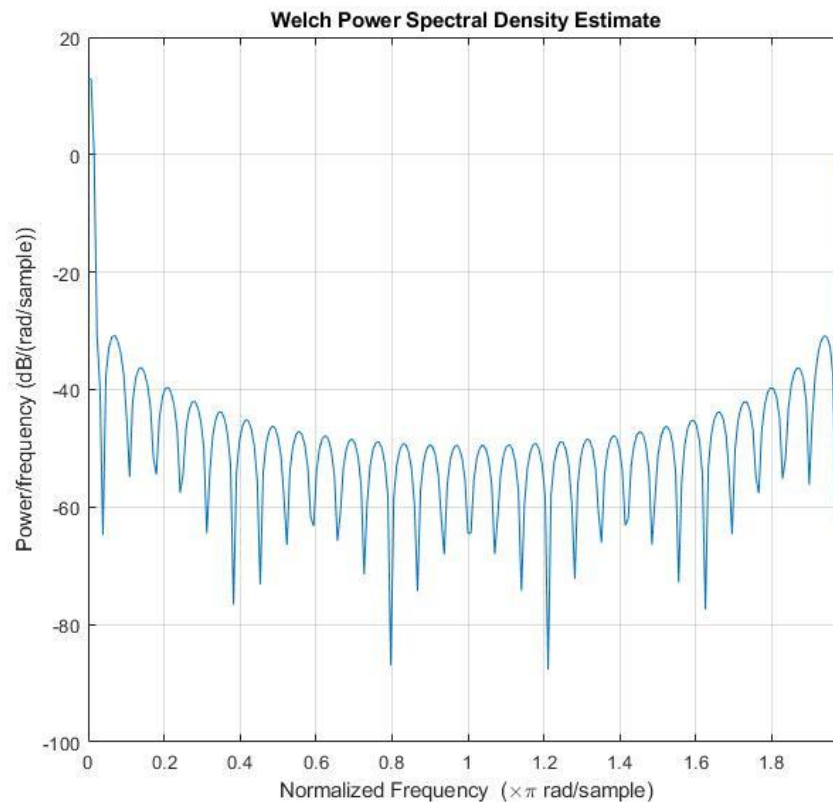
در مرحله بعدی فایل‌های متنی در متلب بارگذاری شده‌اند و مورد بررسی قرار گرفته‌اند. کد متلب این بخش به صورت زیر است که در فایل Q3/Matlab/txtFileRead.m آمده است.

```

clc;
clear;
sin_file = fopen('sin.txt');
cos_file = fopen('cos.txt');
x_sin = fscanf(sin_file, '%x', Inf);
x_cos = fscanf(cos_file, '%x', Inf);
fclose(sin_file);
fclose(cos_file);
x_sin_fi = reinterpretcast(fi(x_sin,0,16,0),numerictype(1,16,14));
x_cos_fi = reinterpretcast(fi(x_cos,0,16,0),numerictype(1,16,14));
x_sin_double = double(x_sin_fi);
x_cos_double = double(x_cos_fi);
x = x_cos_double + 1i * x_sin_double;
pwelch(x)

```

خروجی کد بالا نیز به شکل است :

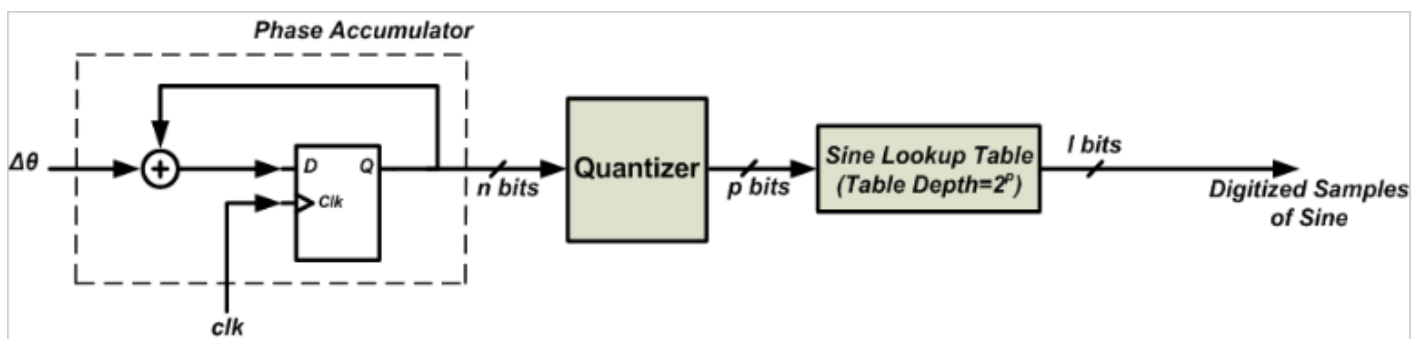


با توجه به اینکه در تست بنچ ما، فرکانس دوبرابر برای مازول انتخاب شده است، انتظار داشتیم که در 2π رادیان بر نمونه peak داشته باشیم که مشاهده می شود.

C. هسته های نرم افزاری و سخت افزاری

این هسته‌ها هر کدام مشابه ماژول خاصی هستند که کار خاصی را انجام می‌دهند که قابلیت تنظیم دارند، در حالتی بهینه طراحی شده‌اند و با بیشترین هماهنگی برای دستگاه هدف ساخته میشوند که کسانی که ماژول‌های خود را طراحی میکنند، میتوانند در صورت نیاز از این زیرماژول‌ها استفاده کنند. برخی از این هسته‌ها تنها پیاده‌سازی نرم‌افزاری دارند که به این معناست با گیت و ماژول‌های مرسوم پیاده‌سازی شده‌اند و وظیفه خود را انجام می‌دهند، در مقابل ماژول‌هایی که به شکل سخت‌افزاری پیاده‌سازی شده‌اند، شامل المان سخت‌افزاری مجزا به منظور عملکرد درست خود خواهند بود. از این هسته‌ها میتوان به هسته‌های کارت‌های SFP+ (پورت نوری شبکه 10Gb)، هسته‌های رمزنگاری معروف، میکروبلیز (پیاده‌سازی یه میکروکنترلر بر روی FPGA)، انواع مموری‌ها و FIFO Arrays و هسته‌های رابط انتقال صدا و تصویر مرسوم اشاره کرد.

هسته‌های DDS: این هسته که متعلق به شرکت Xilinx میباشد وظیفه تغییر فاز سیگنال سینوسی و کسینوسی خروجی از خودش را با تغییر ورودی خودش را دارد و همچنین میتواند میتواند با ورودی متفاوت، سیگنال خروجی با فرکانس متفاوت را این ایجاد کند؛ بلوک‌دیagram این هسته در شکل زیر رسم شده است:



عملکرد این هسته کاملاً مشابه ماژول پیاده‌سازی شده در بخش اول این سوال است؛ چرا که آن هم یک تغییردهنده فاز بر حسب ورودی بود.

۴. تشخیص دنباله با Shift Register

ماژول این بخش در فایل Q4/SequenceDetector.v پیاده‌سازی شده است. ماژول این بخش به شکل زیر است:

```

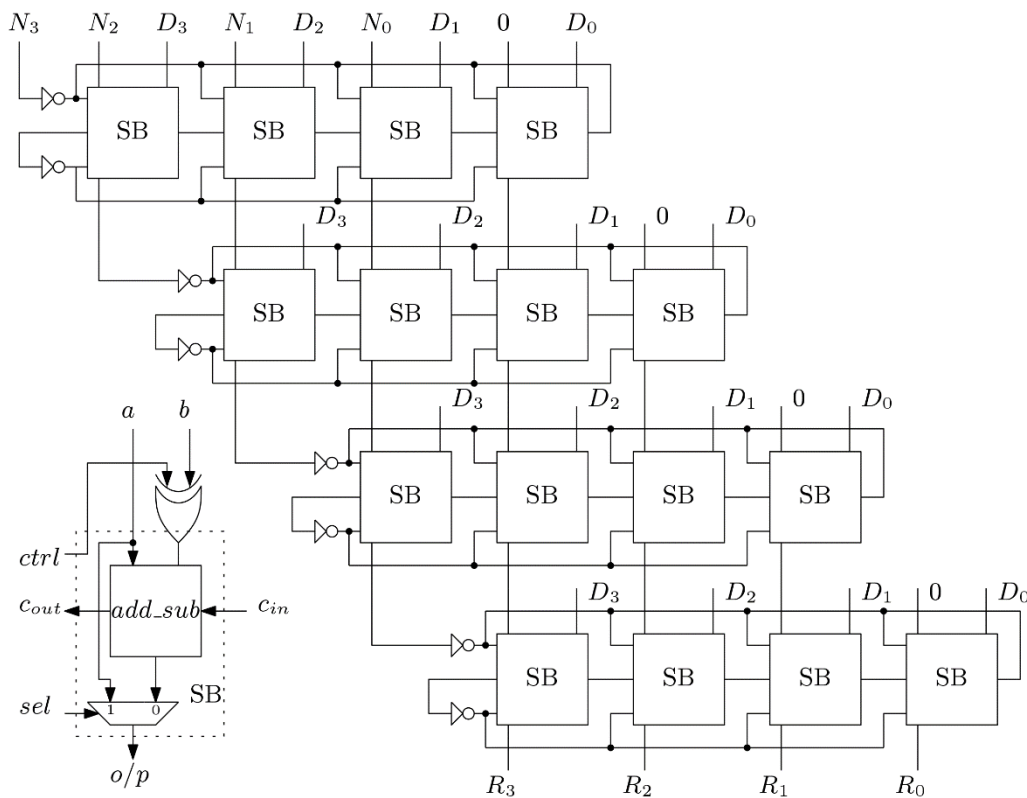
module SequenceDetector
#(
    parameter SEQUENCE_LENGTH = 11,
    parameter reg [SEQUENCE_LENGTH-1:0] SEQUENCE_PATTERN = SEQUENCE_LENGTH'b01011101000;
) (
    input wire clk,
    input wire in,
    output wire Sequence_detected

)
    reg [SEQUENCE_LENGTH-1:0] sequence = SEQUENCE_LENGTH'b0;
//----- Sequential Logic
    always @(posedge clk) begin
        sequence <= {in,sequence[SEQUENCE_LENGTH-1:1]}; //Shift Register
    end
//----- Continuous Assignment
    assign Sequence_detected = SEQUENCE_PATTERN == sequence;
endmodule

```

۵. پیاده‌سازی ALU

برای پیاده‌سازی تقسیم از روش Restoring division استفاده میکنیم که بلوک‌دیگرام عملکرد آن به شکل زیر است:



فایل alu در Q5/ALU.sv قرار دارد و به شکل زیر است:

```

module ALU
(
    input wire [3:0]A,
    input wire [3:0]B,
    input wire [1:0]sel,
    output wire [7:0] alu_out
)
//----- Reg Declaration
    reg [0:7]internal_signal[0:6];
//----- Combinational Logic
    always @(*) begin
        case (sel)
            00: alu_out = A + B;
            01: alu_out = A - B;
            10: alu_out = A * B;
            11:begin
                alu_out[7] = A >= {B,3'b000};
                internal_signal[0] = alu_out[7]?{A,4'h0}-{B,3'h0}:{A,4'h0};
                alu_out[6] = internal_signal[0] >= {B,2'b00};
                internal_signal[1] = alu_out[6]?internal_signal[0]-
{B,2'b0}:internal_signal[0];
                alu_out[5] = internal_signal[1] >= {B,1'b0};
                internal_signal[2] = alu_out[5]?internal_signal[1]-
{B,1'b0}:internal_signal[1];
                alu_out[4] = internal_signal[2] >= B;
                internal_signal[3] = alu_out[4]?internal_signal[2]-
B:internal_signal[2];
                alu_out[3] = (internal_signal[3]<<1) >= B;
                internal_signal[4] = alu_out[3]?internal_signal[3]<<1)-
B:(internal_signal[3]<<1);
                alu_out[2] = (internal_signal[4]<<1) >= B;
                internal_signal[5] = alu_out[2]?internal_signal[4]<<1)-
B:(internal_signal[4]<<1);
                alu_out[1] = (internal_signal[5]<<1) >= B;
                internal_signal[6] = alu_out[1]?internal_signal[5]<<1)-
B:(internal_signal[5]<<1);
                alu_out[0] = (internal_signal[6]<<1) >= B;
                internal_signal[6] = alu_out[1]?internal_signal[6]<<1)-
B:(internal_signal[6]<<1);
            end
            default: alu_out = 0;
        endcase
    end
endmodule

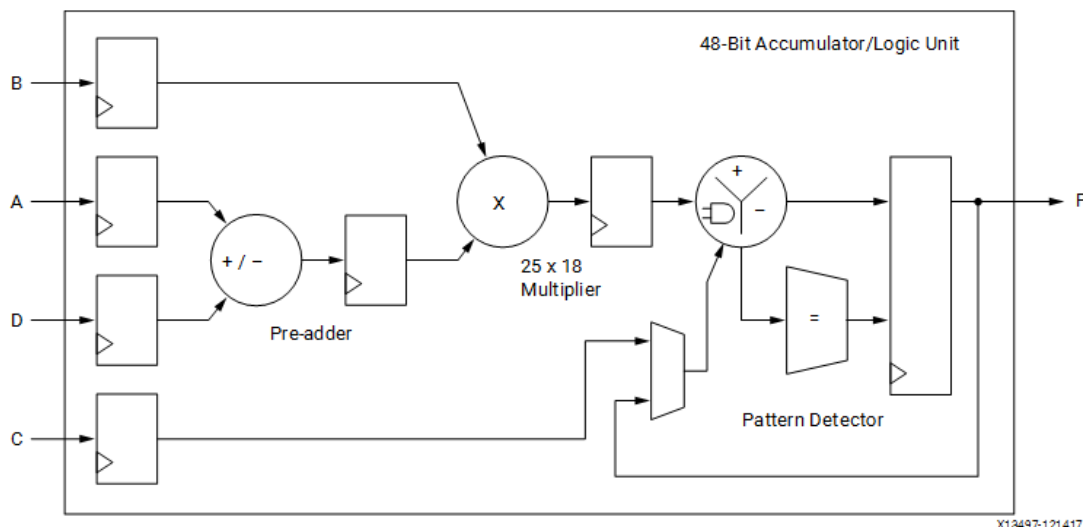
```

خروجی‌ها همگی unsigned در نظر گرفته شده‌اند؛ خروجی جمع و تفریق طبیعتاً همان ۴ بیتی خواهد ماند و خروجی حاصل ضرب هم فرمت fixed-point را خواهد داشت که همه بیت‌ها نشان‌دهنده اعداد صحیح هستند. برای عملیات تقسیم خروجی (0,8,4) fixed-point خواهد بود و ۴ بیت کم‌ارزش‌تر نشان‌دهنده ارقام اعشاری حاصل تقسیم‌اند. در نتیجه بالاترین دقت تقسیم برابر است با $2^{-4} = 0.0001$

۶. آشنایی با DSP48:

A. ساختار کلی DSP48:

این بلوک شامل چهار ورودی است که توانایی انجام جمع و ضرب توامان $(B*(A+B))$ و همچنین جمع و تفریق با پاسخ این عملیات از طریق ورودی C را در یک سیکل کلاک دارد که باعث میشود یکی از عواملی که به شدت روی critical path مدار در FPGA تاثیر میگذارد را به شدت سرعت ببخشد. بلوک دیاگرام این IP به شکل زیر است:



همچنین این IP از توانایی انجام متوالی عملیات بر روی خروجی خودش نیز بهره می‌برد (accumulation) که باعث می‌شود در پیاده‌سازی نهایی نیاز به تعداد کمتری از DSP48 داشته باشیم. نکته قابل توجه در استفاده از این core این است که بهتر از ورودی‌های از جنس Signed استفاده کنیم تا به شکل بهینه از DSP48 استفاده شود.

B. پیاده‌سازی ضرب‌کننده مختلط پایپلاین شده ۱۸ بیتی

ماژول در ۴ مرحله (پس از ۴ سیکل) به شکل پایپلاین خروجی هر متناسب با هر دو عدد مختلط ورودی را محاسبه میکند.

فایل ضرب‌کننده در Q5/ComplexMult18bit.sv قرار دارد و به شکل زیر است:

```

module ComplexMult18bit(
    input clk,
    input signed [17:0] ar, ai,
    input signed [17:0] br, bi,
    output signed [36:0] pr, pi
);
    reg signed [17:0] ai_d, ai_dd, ai_ddd, ai_dddd ;
    reg signed [17:0] ar_d, ar_dd, ar_ddd, ar_dddd ;
    reg signed [17:0] bi_d, bi_dd, bi_ddd, br_d, br_dd, br_ddd ;
    reg signed [18:0] addcommon ;
    reg signed [18:0] addr, addi ;
    reg signed [36:0] mult0, multr, multi, pr_int, pi_int ;
    reg signed [36:0] common, commonr1, commonr2 ;
    always @(posedge clk)begin
        ar_d    <= ar;
        ar_dd   <= ar_d;
        ai_d    <= ai;
        ai_dd   <= ai_d;
        br_d    <= br;
        br_dd   <= br_d;
        br_ddd  <= br_dd;
        bi_d    <= bi;
        bi_dd   <= bi_d;
        bi_ddd  <= bi_dd;
    end
    always @(posedge clk)begin
        addcommon <= ar_d - ai_d;
        mult0     <= addcommon * bi_dd;
        common    <= mult0;
    end
    always @(posedge clk) begin
        ar_ddd    <= ar_dd;
        ar_dddd   <= ar_ddd;
        addr      <= br_ddd - bi_ddd;
        multr     <= addr * ar_dddd;
        commonr1  <= common;
        pr_int    <= multr + commonr1;
    end
    always @(posedge clk)begin
        ai_ddd    <= ai_dd;
        ai_dddd   <= ai_ddd;
        addi      <= br_ddd + bi_ddd;
        multi     <= addi * ai_dddd;
        commonr2  <= common;
        pi_int    <= multi + commonr2;
    end
    assign pr = pr_int;
    assign pi = pi_int;
endmodule

```

شیفت رجیستر زیر پیاده‌سازی شده است که در فایل Q7/ShiftRegister.sv قرار دارد.

```
module ShiftRegister
#(
    parameter OUTPUT_LENGTH = 10
) (
    input wire clk,
    input wire serial_in,
    input wire [1:0] control_signal,
    output wire [OUTPUT_LENGTH-1:0] parallel_out
)
//----- Combinational Logic
    always @(*) begin
        case (control_signal)
            2'b00: parallel_out <= parallel_out;
//latch
            2'b01: parallel_out <= {parallel_out[COUNTER_LENGTH-
2:0],serial_in}; //shift left
            2'b10: parallel_out <= {serial_in, parallel_out[COUNTER_LENGTH-
1:1]}; //shift right
            2'b11: parallel_out <= {COUNTER_LENGTH{serial_in}};
//load
            default:
                parallel_out <= parallel_out;
        endcase
    end
endmodule
```