گزارش آزمایش پنجم

اعضای گروه:

صادق محمدیان:۴۰۱۱۰۹۴۷۷

متین محمدی:۴۰۱۱۱۰۳۲۹

امیرحسین ملک محمدی:۴۰۱۱۰۶۵۷۷

هدف آزمایش:

در این آزمایش می خواهیم با استفاده از الگوریتم بوث حاصل ضرب دو عدد را حساب کنیم.

ورودی ها:

ورودی شامل سیگنال کلاک و سیگنال ریست و دو عدد که multiplier و multiplicand می باشد.

خروجی ها:

خروجی نیز شامل سیگنال done و حاصل ضرب دو عدد ورودی داده شده می باشد . زمانی که حاصل ضرب دوعدد پس از اجرای الگوریتم آماده باشد این سیگنال فعال می شود و عدد که در خروجی هست نشان دهنده حاصل ضرب می باشد.

شرح آزمایش:

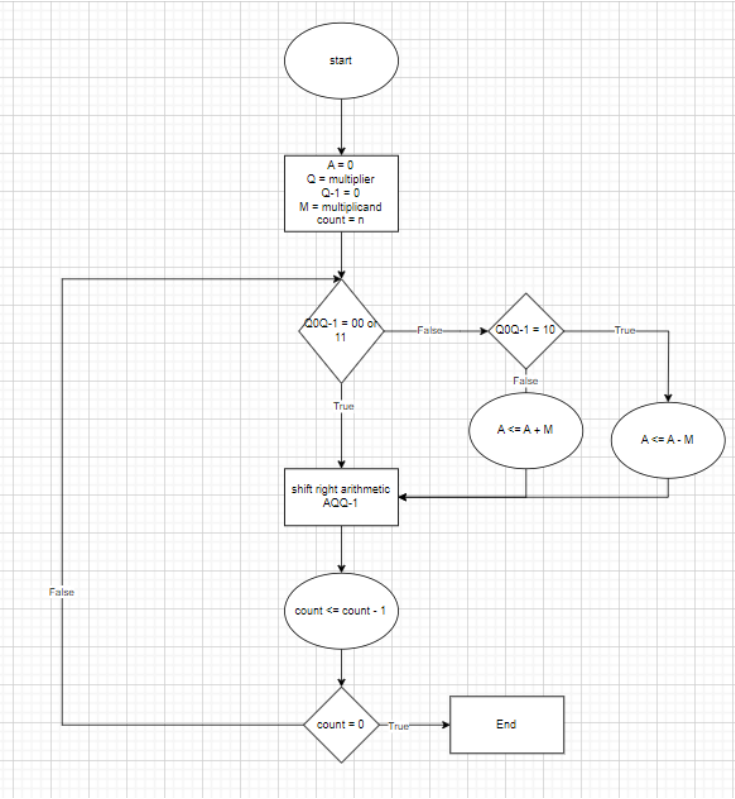
هدف از این آزمایش طراحی یک واحد ضرب کننده است که برای انجام عمل ضرب از روش بوث (booth) استفاده می‌کند. برای انجام این آزمایش، مسیر داده (data_path) و واحد کنترل (control_unit) جداگانه طراحی کردیم و سپس با اتصال آن ها به یکدیگر ضرب کننده را می‌سازیم.

دقت داریم که واحد شیفت دهنده مان قابلیت شیفت دادن بیش از یک بیت در یک پالس ساعت را داشته باشد تا به سرعت عملکرد ضرب کننده کمک کند و آن را از shift and add سریعتر گرداند.

این روش به صورت زیر می‌باشد:

1. $A = 0, Q = multiplier, Q_{-1} = 0, M = multiplicand, count = n\ (number\ of\ bits)$
2.
    a. $Q_0 Q_{-1} = 10 => A = A - M$
    b. $Q_0 Q_{-1} = 01 => A = A + M$
    c. Otherwise do nothing
3. Shift arithmetic right AQQ_{-1} and decrement the counter
4. If the counter is not zero then go back to step 2, otherwise finish

در اینجا، استیت دیاگرام مربوط به این الگوریتم را نیز مشاهده می‌کنید:



حال به بررسی ماژول های طراحی شده می‌پردازیم.

ابتدا خود ماژول booth را داریم که وظیفه تلفیق واحد کنترل و مسیر داده می‌باشد:

```
module booth (input[3:0] multiplicand,input[3:0] multiplier,input rst,input
clk,output[7:0] result,output ended);
wire [2:0] A_shift;
wire [2:0] B_shift;
wire [3:0] B;
control_unit CU (B, rst, clk, A_shift, B_shift, sub_add, ended);
data_path DP (multiplicand, multiplier, rst, clk, A_shift, B_shift, sub_add,
ended, result, B);
endmodule
```

در اینجا، ورودی ها، سیگنال های ریست و کلاک، multiplier و multlipicand هستند و خروجی نیز سیگنال ended و عدد خروجی درون result می‌باشد. همچنین از سیگنال های A_shift و B_shift و B برای ارتباط بین CU و DP استفاده شده است.

حال به بررسی مسیر داده می‌پردازیم که در واقع ماژول data path می‌باشد:

```
module data_path (input[3:0] multiplicand, input[3:0] multiplier, input rst,
   input clk, input[2:0] A_shift, input[2:0] B_shift, input sub_add,
   input ended, output reg[7:0] result, output reg[3:0] B);
reg [7:0] A;
always @(posedge clk or negedge rst) begin
   if (~rst) begin
        A <= {{4{multiplicand[3]}}, multiplicand};
        B <= {{4{multiplier[3]}}, multiplier};
        result <= 0;
   end else if (~ended) begin
        B <= B >> B_shift;
        if (sub_add == 1) begin
            result <= result + (A << A_shift);
        end
        if (sub_add == 0) begin
            result <= result - (A << A_shift);
        end
   end
end
endmodule
```

در اینجا با ورودی گرفتن multiplier و multiplicand و سیگنال های ریست و کلاک و همچنین A_shift و B_shift و sub_add و ended، عملیات های لازم (جمع یا تفریق یا هیچکدام و شیفت دادن) انجام می‌شود و سپس خروجی حاصل در result نگهداری می‌شود. همچنین B تغییر یافته نیز بعنوان خروجی در نظر گرفته می‌شود.

بلاک always با لبه بالارونده کلاک یا لبه پایین رونده ریست فعال می‌شود.

در حالت ریست multiplier و multiplicand، ساین اکستند می‌شوند و در B و A ذخیره می‌شوند. همچنین result نیز صفر می‌گردد.

در حالت پایان نیافته، B را به اندازه B_shift، شیفت راست می‌دهیم و سپس با توجه به عملیات (sub_add)، جمع یا تفریق را انجام می‌دهیم. (شیفت A به سمت چپ و جمع یا تفریق آن از result در نهایت معادل می‌شود با همان جمع یا تفریق و سپس شیفت به راست دادن)

حال به بررسی واحد کنترل می‌پردازیم:

```
module control_unit (input [3:0] B, input rst, input clk, output[2:0] A_shift,
output[2:0] B_shift, output sub_add, output done);
reg [2:0]   shifted;
reg state;
wire [1:0] one_index;
wire [2:0] zero_index;
```

```
localparam load = 0;
localparam calculate = 1;
find_one first_one (B, one_index);
find_zero first_zero (B, zero_index);
assign sub_add = B [0] & (~state);
assign B_shift = sub_add ? zero_index : {1'b0, one_index};
assign A_shift = shifted + B_shift;
assign done = shifted + B_shift >= 4;
always @ (posedge clk or negedge rst) begin
   if (~rst) begin
       shifted <= 0;
       state <= calculate;
   end else begin
       state <= load;
       shifted <= shifted + B_shift;
   end
end
endmodule
```

در اینجا، ورودی های B و ریست و کلاک را داریم و با توجه به این مقادیر، مقدار لازم برای شیفت (A_shift و B_shift) بدست می‌آیند و همچنین عملیات (sub_add) و اتمام یا عدم اتمام (ended) نیز تعیین می‌شود.

در بلاک always، وقتی سیگنال ریست صفر شود، مقدار shifted برابر ۰ و state بعدی برابر ۱ می‌شود و در غیر اینصورت، state بعدی ۰ و مقدار shifted نیز با مقدار B_shift جمع می‌گردد.

بیت sub_add به کمک بیت B[0] و بیت first_clock حاصل می‌شود.

مقدار B_shift بسته به sub_add به کمک جایگاه اولین صفر یا اولین ۱ بدست می‌آید.

مقدار A_shift به کمک مقدار shifted و B_shift حاصل می‌گردد و ended نیز برای اطلاع رسانی وضعیت عملیات به کمک B_shift و shifted بدست می‌آید.

در این ماژول، از دو ماژول find_one و find_zero، instance گرفته شده است که به شرح زیر اند:

ماژول find_one:

```
module find_one (input [3:0] A,output[1:0] out);
assign out = {~ (A [1] | A [0]), ~A [0] & (A [1] | ~A [2])};
endmodule
```

که جایگاه اولین یک را باز می‌گرداند. اگر اولین یک در ۰ یا ۱ بود، به وضوح بیت سمت چپ باید ۰ باشد و در غیر اینصورت ۱ باشد که این معادل است با:

~ (A [1] | A [0])

همچنین اگر اولین بیت ۱ در ۰ بود باید بیت سمت راست ۰ باشد و همچنین اگر در ۰ و ۱، هیچ یکی وجود نداشت و در ۲ وجود داشت بیت راست باید ۰ باشد و در غیر اینصورت باید ۱ باشد که معادل است با:

~A [0] & (A [1] | ~A [2])

و ماژول find_zero که مشابه ماژول بالا است با این تفاوت که بررسی می‌کند که اگر A کاملا شامل یک بود، آن را مشخص کند (به همین علت خروجی ۳ بیتی است)

```
module find_zero (input[3:0] A, output[2:0] out);
assign out[2] = A[3] & A[2] & A[1] & A[0];
assign out[1] = A[1] & A[0] & (~(A[3] & A[2]));
assign out[0] = A[0] & (~A[1] | A[2]);
endmodule
```

اگر خروجی کاملا شامل یک باشد، out[2] آن را مشخص می‌کند. خروجی out[0] و out[1] نیز مشابه با منطق بالا بدست می‌آیند.

حال با بررسی چند حالت در تست بنچ، از صحت کد خود اطمینان حاصل میکنیم:

```
module booth_TB ();
reg signed [3:0]   A;
reg signed [3:0]   B;
reg  rstN = 1, clk = 1;
wire signed [7:0]  res;
wire done;
booth MUL (A, B, rstN, clk, res, done);
always #10 clk = ~clk;
initial begin
    #20;
    A = 3;
    B = 4;
    rstN = 0;
    #20 rstN = 1;
    wait (done);
    $display("%d * %d = %d", A, B, res);
    #20;
    A = -3;
    B = 7;
    rstN = 0;
    #20 rstN = 1;
    wait (done);
```

```
    $display("%d * %d = %d", A, B, res);
    #20;
    A = -8;
    B = -6;
    rstN = 0;
    #20 rstN = 1;
    wait (done);
    $display("%d * %d = %d", A, B, res);
    #20;
    A = 0;
    B = -6;
    rstN = 0;
    #20 rstN = 1;
    wait (done);
    $display("%d * %d = %d", A, B, res);
    #20;
    A = 4;
    B = 12;
    rstN = 0;
    #20 rstN = 1;
    wait (done);
    $display("%d * %d = %d", A, B, res);
    #20;
    $stop();
end
endmodule
```

و خروجی تست بنچ، به صورت زیر در می‌آید که مشاهده می‌شود دقیقا همان چیزی است که انتظار داشتیم:

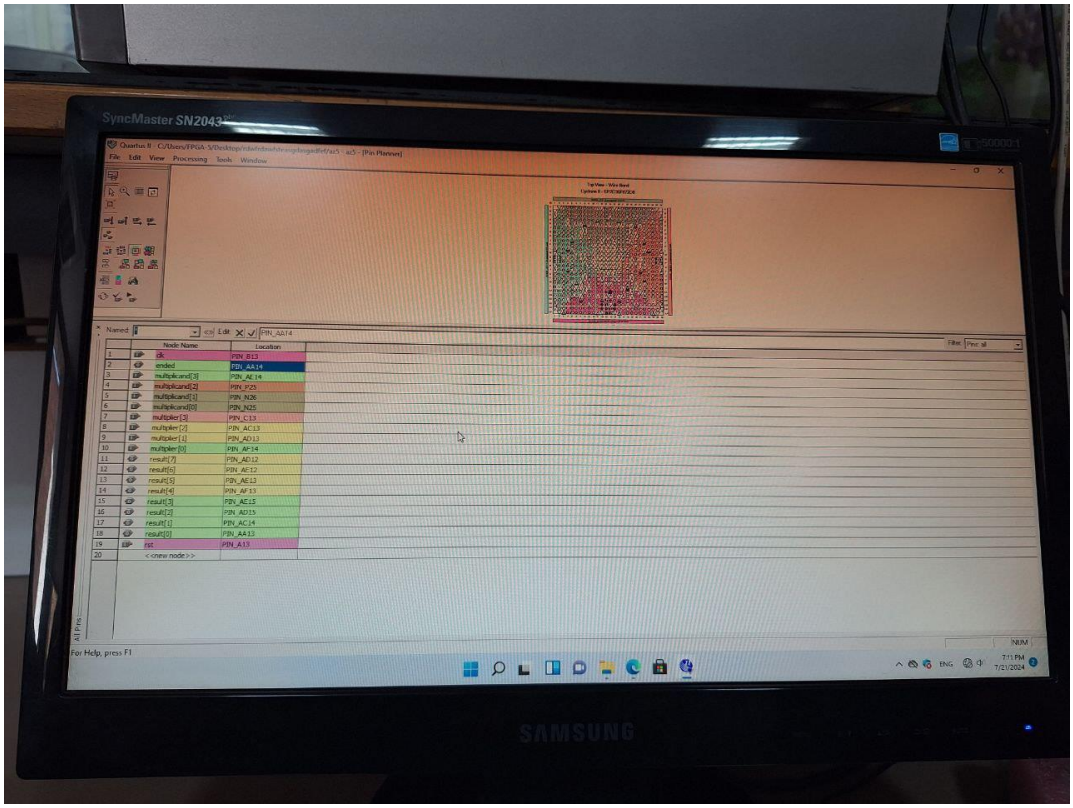
```
VSIM 6> restart
# Loading work.booth_TB
# Loading work.booth
# Loading work.cu
# Loading work.find_one
# Loading work.find_zero
# Loading work.dp
VSIM 7> run -all
#    3 *    4 =   12
#   -3 *    7 =  -21
#   -8 *   -6 =   48
#    0 *   -6 =    0
#    4 *   -4 =  -16
# Break in Module booth_TB at D:/matin/University/summer03/DSDLab/az5/booth_t.v line 63

VSIM 8>
```

حال پس از توضیح کلیات کد (که در پیش گزارش نیز به همین صورت قرار گرفته است) به سر اغ مراحلی می رویم که در آزمایشگاه سپری کردیم.

ابتدا بورد را روی حالت programقرار دادیم و سپس کدهای بالا را درون یک پروژه جدید در نرم افزار کوارتوس وارد کردیم و ماژول اصلی خود را انتخاب کردیم سپس پروژه را compile کردیم تا کد وریلاگ را سنتز کنیم.

سپس به بخش pin planner رفتیم و مشخص کردیم که هرکدام از ورودی ها به کدام سوییچ متصل باشند و خروجی های آزمایش یعنی عدد حاصل ضرب و سیگنال done به کدام LED متصل باشند. در این مرحله از فایل پی دی اف راهنما آدرس پین های مورد نیاز را برداشتیم.



پس از این مراحل به کمک programmerفایل main.scf را انتخاب کرده و پس از تغییر مورد از program به run برنامه را استارت زدیم.

تمامی فایل های مربوط به مراحل کامپایل و سنتز در نرم افزار کوآرتوس در پوشه ی quartus قابل مشاهده است و مدار توسط TA تست گردید و صحت مدار مورد تایی قرار گرفت.