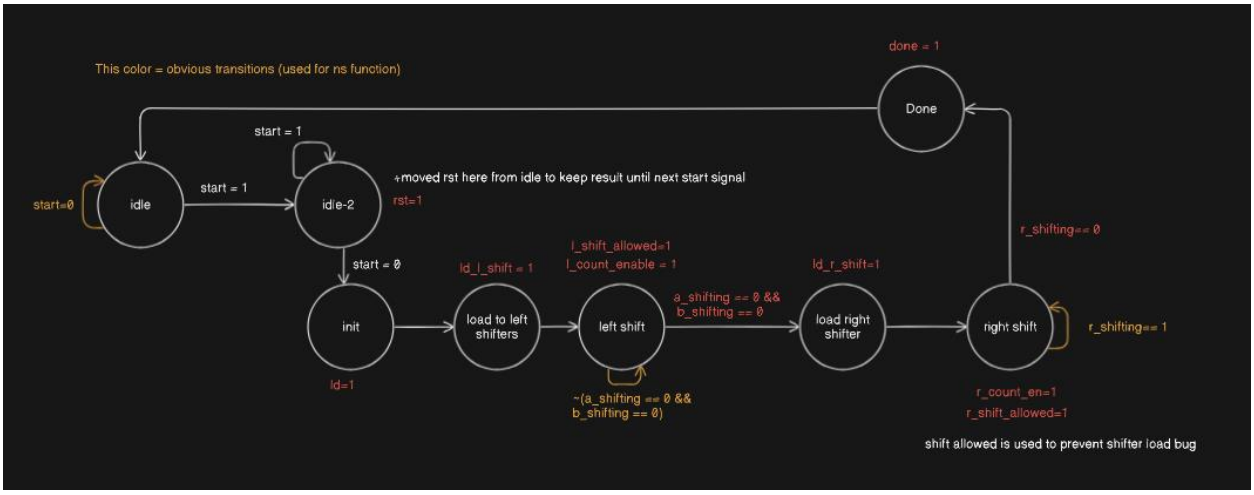
گزارش پروژه سوم درس طراحی کامپیوتری سیستم‌های دیجیتال

پریا پاسه‌ورز 810101393 آریان رجبی 810199421

طراحی پروژه سوم:

Controller:



تغییرات نسبت به پروژه 1:

Handshake بین ماژول ها پاک شده تا ورودی و خروجی و حالات کنترلر ساده تر شود. (قبلا هر ماژول هم به کلاک وابسته بود و عملا رجیستر بود، ولی الان فقط شیفتر ها و خود رحیستر به کلاک وابسته اند بقیه combinational)

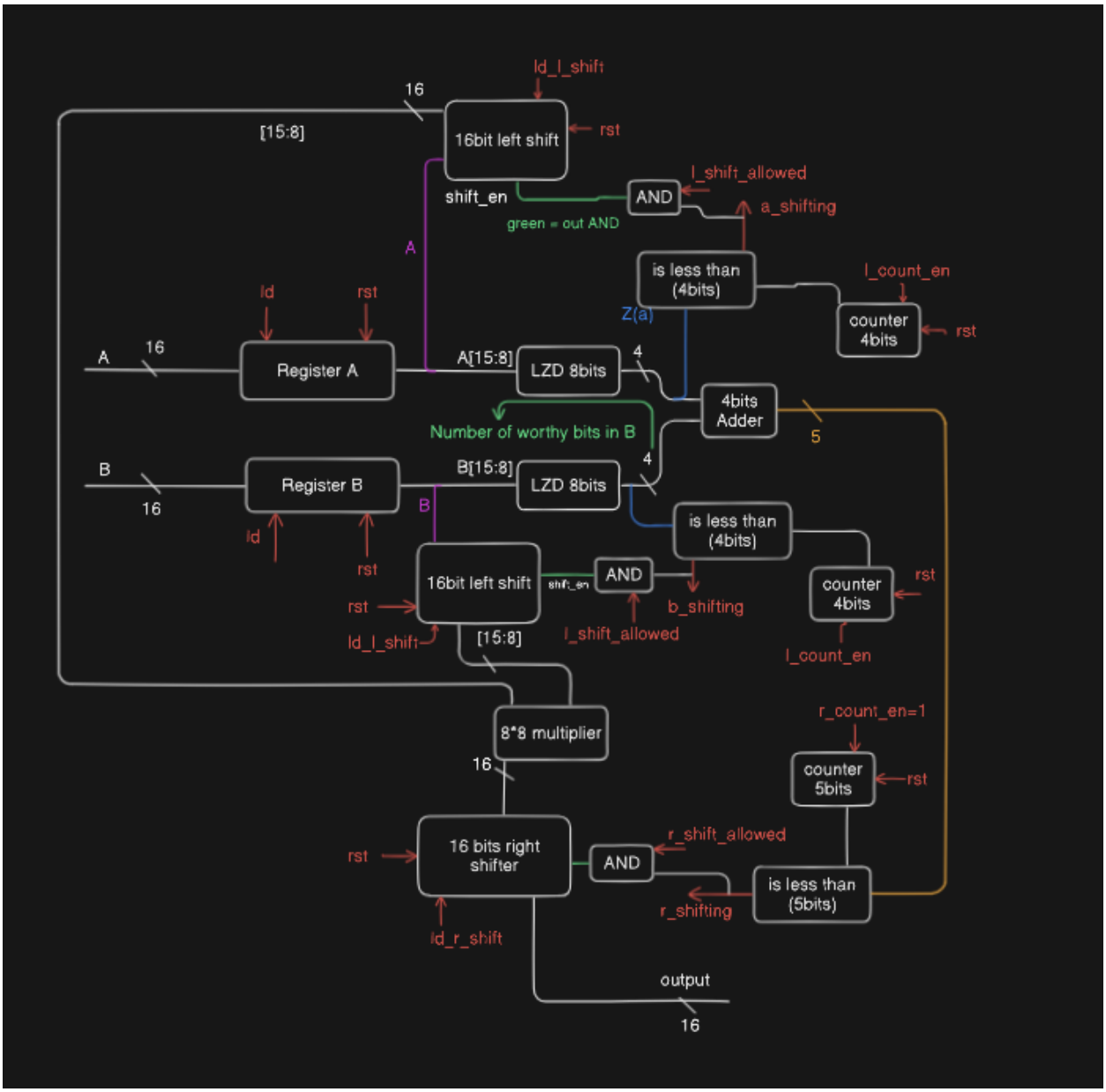
به حای مولتی شیفت، از counter, is less than و شیفت تکی (یک شیفت در کلاک) استفاده شده.

خواندن و نوشتن در حافظه حذف شده.

خروجی 16 بیتی شده (قبلا 32 بود)

توضیحات ککنترلر (تبدیل به SOP) در فایل controller.txt

Datapath:


ld_l_shift
16
[15:8]
16bit left shift
rst
l_shift_allowed
shift_en
AND
a_shifting
green = out AND
A
is less than (4bits)
l_count_en
ld
rst
Z(a)
counter 4bits
rst
A
16
Register A
A[15:8]
LZD 8bits
4
4bits Adder
Number of worthy bits in B
5
B
Register B
B[15:8]
LZD 8bits
4
16
B
is less than (4bits)
ld
rst
16bit left shift
shift_en
AND
counter 4bits
rst
rst
b_shifting
ld_l_shift
[15:8]
l_shift_allowed
l_count_en
r_count_en=1
8*8 multiplier
16
counter 5bits
rst
r_shift_allowed
rst
16 bits right shifter
AND
is less than (5bits)
r_shifting
ld_r_shift
output
16

به عنوان مثال برای پیاده‌سازی جمع کننده، از ripple carry adder استفاده کردیم:

```
`timescale 1ps/1ps
module adder(
    input [3:0] inA,
    input [3:0] inB,
    output [4:0] out
);
    wire [4:0] carry;
    wire [3:0] sum;

    assign carry[0] = 1'b0;

    genvar i;
    generate
        for (i = 0; i < 4; i = i + 1) begin: full_adder_gen
            wire sum_temp;

            // Calculate propagate signal (P = A XOR B)
            Xor xor1(.a(inA[i]), .b(inB[i]), .out(sum_temp));

            // Calculate sum using propagate and carry
            Xor xor2(.a(sum_temp), .b(carry[i]), .out(sum[i]));

            // Calculate carry using generate and propagate
            // carry[i+1] = G + P·Ci = (A·B) + (A⊕B)·Ci
            Or or1(.a(inA[i] & inB[i]), .b(sum_temp & carry[i]), .out(carry[i+1]));
        end
    endgenerate

    // Combine the results
    assign out = {carry[4], sum};

endmodule
```

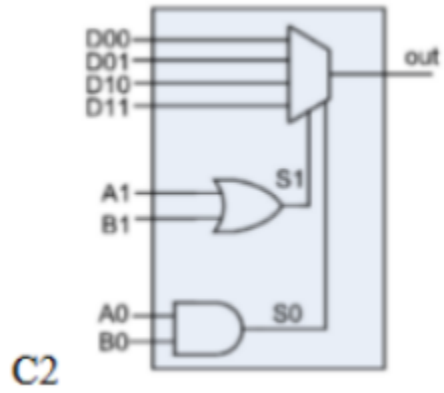
هر بیت از حاصل جمع (Sum) بر اساس دو ورودی (A و B) و بیت نقلی (Ci) از مرحله قبل محاسبه می‌شود.

Sum (S) = A ⊕ B ⊕ Ci (XOR gate logic)

Carry Out (Ci+1) = (A·B)+(A⊕B)·Ci

حال برای پیاده‌سازی Xor

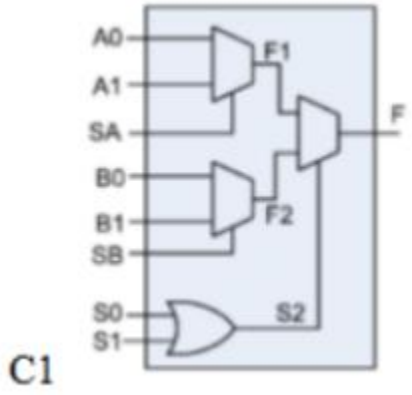
```
module Xor(
    input a,
    input b,
    output out
);

    c2 xor_impl(
        .D00(1'b0),   // when a=0,b=0
        .D01(1'b1),   // when a=0,b=1
        .D10(1'b1),   // when a=1,b=0
        .D11(1'b0),   // when a=1,b=1
        .A1(a),       // first input
        .B1(1'b0),    // tie to 0
        .A0(b),       // second input
        .B0(1'b1),    // tie to 1
        .out(out)
    );

endmodule
```



همانطور که می‌بینیم با کمک ماژول c2، از دو بیت ورودی به عنوان selector در Multiplexer استفاده کرده‌ایم و حالتهای مختلف ترکیب ورودی را به عنوان ورودی multiplexer کد کرده‌ایم تا در ترکیبهای مختلف فعال شوند و خروجی عملیات xor صادر شوند. دو ورودی دیگر به نحوی تعیین شده‌اند که تاثیری در عملیات نداشته باشند.

و برای پیاده‌سازی Or

```
module Or (input a ,b, output out);
    c1 C1_OR(.A0(a), .A1(1'b1), .SA(b), .B0(), .B1(), .SB(), .S0(1'b0), .S1(1'b0), .f(out));
endmodule
```



اگر مقادیر داده شده را در سیمها جایگذاری کنیم خواهیم داشت:

$F1=b\cdot A1+b\cdot A0$

$F1=b\cdot 1+b\cdot a=b+a$

و همانطور که می‌بینیم خروجی F1 برابر حاصل عملیات Or دو سیگنال ورودی خواهد بود. S0 و S1 طوری انتخاب شده‌اند که همیشه خروجی F1 به عنوان خروجی نهایی انتخاب شود.