آز سیستم های دیجیتال

پیش گزارش آزمایش چهارم:

توصیف رفتاری پشته

امیرحسین علمدار 400105144

علیرضا سلیمیان 400105936

محمدپیام تائبی 400104867

بهار 1402

یک پشته با عمق ۸ و پهنای ٤ بیت طراحی کنید که دارای ورودی‌ها و خروجی‌های زیر باشد :

Inputs:	Clk	Clock signal
	RstN	Reset signal
	Data_In	4-bit data into the stack
	Push	Push Command
	Pop	Pop Command
Outputs:	Data_Out	4-bit output data from stack
	Full	Full=1 indicates that the stack is full
	Empty	Empty=0 indicates that the stack is empty

همانطور که می بینید سیگنال های ورودی و خروجی توسط دستور کار مشخص شده اند.(پهنای داده ها 4 بیتی است)

```
module stack_behavioural (
    input clk,
    input rstN,
    input [3:0] data_in,
    input push,
    input pop,
    output reg [3:0] data_out,
    output reg full,
    output reg empty
);
```

از آنجایی که طراحی رفتاری از ما خواسته شده است، می توانیم صرفا با متغیر ها کار کنیم و مقدارشان را متناسب با سیگنال های ورودی تغییر دهیم.

برای طراحی ساختار حافظه و پوینتر استک، از سریال رجیستر ها استفاده میکنیم:

```
reg [3:0] sp;
reg [3:0] stack_mem [7:0];
integer i;
```

(sp = stack pointer) همیشه به سر استک اشاره میکند. (در این خانه مقداری قرار داده نشده است. اگر بخواهیم آخرین مقدار پوش شده را بخوانیم باید خانه ی پایینی استک پوینتر را مشاهده کنیم)

برای استک پوینتر 4 بیت در نظر گرفتیم زیرا مقدار آن میتواند 8 شود که نشان دهنده پر شدن است.

همینطور در خط دوم، یک آرایه 8تایی سریال رجیستر 4 بیتی طراحی کردیم که با دادن ایندکسی که در واقع برابر با استک پوینتر است، 4 بیت مورد نظر را مشاهده می کنیم.

حال شروع به توصیف منطق حساس به لبه بالا رونده کلاک میکنیم:

```
always @(posedge clk) begin
```

ابتدا **rstN**(ریست اکتیو لو) را طراحی می کنیم. با صفر شدن ریست، باید تمام خانه های حافظه ما صفر شوند و پوینتر به خانه اول استک برود. همچنین empty باید صفر شود تا خالی بودن استک نشان داده شود:

```
if (rstN == 0) begin   // reset the whole memory //active low
        for (i = 0; i < 8; i = i + 1) begin
            stack_mem[i] = 0;
        end
        sp = 0;
        data_out = 0;
        empty = 0;
        full = 0;
end
```

rstN سنکرون است زیرا در always قرار دارد و وابسته به کلاک است.

همچنین اگر ریست فعال بود، هیچ بخش دیگری کار نمی کند زیرا تمام کار های مربوط به سیگنال های دیگر در else اند.

حال push و pop را پیاده میکنیم:

```
else begin
        if (push == 1 && pop == 0 && full == 0 ) begin   // push
            stack_mem[sp] = data_in;
            sp = sp + 1;
        end else if (pop == 1 && push == 0 && empty == 1) begin  // pop
            sp = sp - 1;
            data_out = stack_mem[sp];
        end
```

با یک شدن پوش و پر نبودن استک، می توانیم داده ورودی را در سر استک قرار دهیم و استک پوینتر را بالا ببریم. طبعا اگر استک پر بود نمی توانیم پوش کنیم.

همچنین با یک شدن پاپ و خالی نبودن استک، آخرین خانه پر استک را که همان خانه زیری استک پوینتر است، را در داده خروجی می ریزیم.

یادآوری میکنیم که خانه ای که استک پوینتر به آن اشاره می کند، اولین خانه خالی سر استک است.(این خانه می تواند خالی نباشد اما مطمئنا مقدار داخل آن معتبر نیست زیرا هنوز آپدیت نشده است و ممکن است از قبل مقداری درون آن باشد که میدانیم معتبر نیست. پس منظور از خالی بودن، عدم اعتبار است.)

در انتها نیز شرایط پر یا خالی بودن استک را آپدیت می کنیم:

```
    if (sp == 8) begin
            full = 1;
        end else begin
            full = 0;
        end
        // empty:
        if (sp == 0) begin
            empty = 0;
        end else begin
            empty = 1;
        end

    end
```

واضحا اگر استک پوینتر 8 باشد، استک پر شده است و اگر 0 باشد، استک خالیست.

در دستور کار حالات empty و full اینگونه خواسته شده بود:

Full	Full=1 indicates that the stack is full
Empty	Empty=0 indicates that the stack is empty

حال به سراغ ساخت ماژول تست بنچ می رویم:

```
module testbench();

reg clk = 0;
reg rstN;

reg [3:0] data_in;
reg push = 0;
reg pop = 0;

wire [3:0] data_out;
wire full, empty;
```

ابتدا سیگنال ها ورودی و خروجی را مشخص می کنیم.

```
localparam CLK_PERIOD = 10;
always #(CLK_PERIOD/2) clk=~clk;
```

سپس کلاک را تعیین می کنیم. همانطور که مشاهده می کنید بازه نات شدن کلاک، 5 واحد زمانیست، که در این پروژه ثانیه است.

```
stack_behavioural sb (clk, rstN, data_in, push, pop, data_out, full, empty);
```

از ماژول stack_behavioutal اینستنس می سازیم و اسم آن را sb می گذاریم.

مقدار دهی و تعیین بازه های زمانی شبیه سازی را شروع می کنیم:

```
initial begin
    rstN = 0; // reset the stack memory
    #(CLK_PERIOD*2) // now after 2 clks we start processing
    rstN = 1;
    pop = 1; // popping from empty stack
    #(CLK_PERIOD*3)
```

ابتدا ریست را صفر می کنیم و برای 2 کلاک صبر می کنیم، در موج های شبیه سازی خواهیم دید که با بالا رفتن لبه کلاک، ریست کار خواهد کرد.

سپس از حالت ریست در میاوریم و پاپ را فعال می کنیم و تغییری در مقدار خروجی مشاهده نخواهیم کرد زیرا استک خالی بوده و فعال شدن پاپ، بی اثر است.

سپس در چند کلاک، مقادیری را پوش میکنیم:

```
    pop = 0;
    push = 1;
    data_in = 12;

    #(CLK_PERIOD*1)
    data_in = 11;

    #(CLK_PERIOD*1)
    data_in = 10;

    #(CLK_PERIOD*1)
    data_in = 9;

    #(CLK_PERIOD*1)
    data_in = 8;
    #(CLK_PERIOD*1)
```

آخرین مقدار را پاپ میکنیم: (در wave trace تغییر خروجی به 8 را مشاهده خواهیم کرد)

```
    push = 0;
    pop = 1;

    #(CLK_PERIOD*1)
```

```
 #(CLK_PERIOD*1)
    push = 1;
    pop = 0;
    data_in = 15;

    #(CLK_PERIOD*1)
    data_in = 14;

    #(CLK_PERIOD*1)
    data_in = 13;

    #(CLK_PERIOD*1)
    data_in = 9;
    #(CLK_PERIOD*1)
```

در اینجا یک شدن سیگنال full را مشاهده می کنیم که به درستی نشان دهنده پر بودن استک است. (در مجموع 9 مقدار پوش کردیم که یکی را آن وسط پاپ کردیم)

در نهایت نیز 8 بار کلاک میزنیم و پاپ شدن تمامی مقادیر ذخیره شده در استک را مشاهده می کنیم. سپس می بینم که سیگنال empty صفر شده است که به معنای خالی بودن استک است.

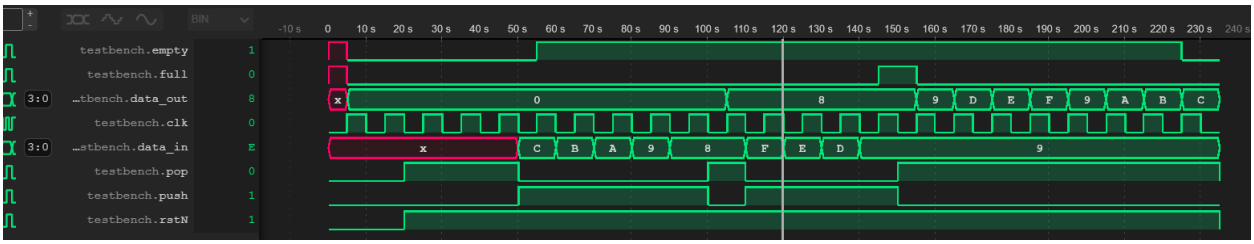
```
    push = 0;

    pop = 1;

    #(CLK_PERIOD*8.5);
    $finish;

end
endmodule
```

تصویر خروجی شبیه سازی:



برای مشاهده بهتر، فایل تصویر ضمیمه شد.