فصل چهارم

خلاصه پروژه و پیشنهادات

خلاصه و پیشنهاداتی برای ساخت و تست بردهای XC9500 FPGA's:

- ساخت اتصالات محکم از برد به تغذیه و کابل های پورت موازی، همواره دشوار است. از سیمهای ۲ برای اتصال جک تغذیه به برد استفاده کرده سپس سیمها را به باسهای تغذیه برد ، متصل کنید.برای کانکتور DB25 ،25 سیم را به آن متصل کرده و سپس آنرا با یک جفت سیم فلزی به برد متصل کنید.
- ابتدا مدار تغذیه را ساخته، اطمینان حاصل کنید که ولتاژ صحیح را تولید می کند.
- سپس انطباق دهنده PLCC را روی برد قرار دهید ،فعلا FPGA را درون آن قرار ندهید . پینهای Vcc و GND را متصل کنید ، سپس تغذیه را وصل کرده و مطمئن شوید که ولتاژ روی پینهای Vcc و GND ظاهر شوند.سپس تغذیه را خاموش کنید.
- Interface پورت موازی را با قرار دادن چیپ 74LS14 روی برد و اتصال سیمها از کانکتور DB25 ، بسازید. فعلا FPGA یا CPLD را در انطبلق دهنده PLCC قرار ندهید . یک فایل با استفاده از نرم افزار Xilinx Foundation ساخته و سپس آنرا با استفاده از برنامه XSLOAD ، دانلود کنید. 7-segment را روی برد قرار داده و آنرا به منطبق کننده PLCC متصل کنید سپس منبع را روشن کرده و دوباره اطمینان حاصل کنید که ولتاژ صحیح روی پینهای Vcc و GND از منطبق کننده PLCC ظاهر می شوند.سپس تغذیه را خاموش کنید.
- اگر تست شما موفقیت آمیز نبود موارد زیر را چک کنید:
  - چک کنید که 74LS14 و چیپ XC9572 تغذیه را از چیپ های صحیح دریافت کرده و اتصالات زمین آنها در محل مناسب قرار دارند.
  - اطمینان حاصل کنید که مبدل 9VDC متصل شده است.
  - مطمئن شوید که کانکتور DB25 به درستی سیمبندی شده است.
  - ممکن است 74LS14 دارای مشکل باشد ، آنرا جایگزین کنید.

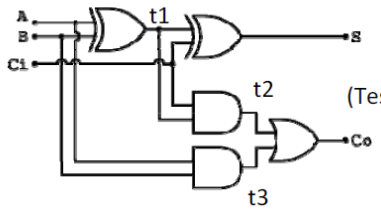
- بعد از اینکه ثابت شد می توانید با موفقیت در بردتان Program کنید، مدار تولید کلاک را روی برد قرار داده و چک کنید که سیگنال 12MHz تولید کند.
- در نهایت RAM استاتیک 32KB را روی بردتان قرار داده و آن را به منطبق کننده PLCC متصل کنید .مقاومت Pull-up را روی ورودی CS از RAM فراموش نکنید،سپس رشته بیتها را Program کرده و اجرای برنامه کارکرد صحیح حافظه را چک خواهد کرد.

منابع و ماخذ

- ✓ معماری کامپیوتر / هانی جوان همت ، نیما کریم پور / دانشگاه آزاد اسلامی – واحد لاهیجان
- ✓ خودآموز VHDL / افشین راجی
- ✓ مجموعه دستورات زبان توصیف سخت افزار / مرتضی شعبان زاده
- ✓ FPGA و تاریخچه آن / جواد مرادی
- ✓ آموزش FPGA / حامد سقایی
- ✓ سایت مربوط به دیتا شیت های المانها و قطعات

پیوست ۱

طراحی تمام‌جمع‌کننده



۱. برای یک مدار تمام‌جمع‌کننده کد VHDL بنویسید.
۱. برای مدار حاصل یک محیط تست با استفاده از کد VHDL بنویسید. (Test Bench)
۲. کد نهایی را کامپایل و در صورت وجود خطاهای احتمالی آن‌ها را تصحیح کنید.
۳. مدار حاصل را با استفاده از ModelSim شبیه‌سازی کنید.

یک کد VHDL بهمراه توضیحات بعنوان نمونه آورده شده است. (برای خوانایی بیشتر شماره خط‌ها به کد اصلی افزوده شده است)

```
00 - entity fulladder is
01 -    port (a, b, cin: IN BIT; s, cout: OUT BIT);
02 - end fulladder;
```

در اینجا یک `entity` برای تمام‌جمع‌کننده ساخته می‌شود. ورودی و خروجی‌ها با توجه به شکل مشخص شده است. (۰ تا ۲)

```
03 - architecture structural of fulladder is
04 - signal t1, t2, t3: BIT;
05 - begin
06 -    t1 <= a XOR b;
07 -    s  <= t1 XOR cin;
08 -    t2 <= t1 AND cin;
09 -    t3 <= a AND b;
10 -    cout <= t2 OR t3;
11 - end structural;
```

ساختار داخلی مدار با توجه به شکل بصورت ارتباط سیگنال‌ها و گیت‌ها مشخص شده است. (۳ تا ۱۱)

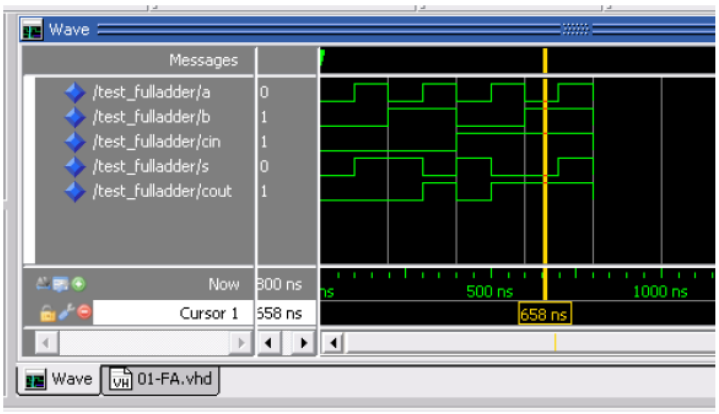
تا اینجا ورودی/خروجی‌ها و اتصالات داخلی مدار کامل شده است. برای بررسی صحت عملکرد مدار لازم است تا آن را در یک محیط مناسب تست کنیم. بهمین منظور یک `entity` خالی بعنوان محیط تست ایجاد می‌کنیم. (۱۲ تا ۱۳)

```
12 - entity test_fulladder is
13 - end test_fulladder;
```

در داخل محیط تست ابتدا ماجول تمام جمع‌کننده را معرفی می‌کنیم (۱۵ تا ۱۷)، یک نمونه از تمام‌جمع‌کننده‌ای را که طراحی کرده‌ایم قرار می‌دهیم (۲۰) و سیگنال‌هایی را به ورودی و خروجی‌اش متصل می‌کنیم (۲۱ تا ۲۵). سپس با مقداردهی به سیگنال‌های ورودی در کد (۲۶ تا ۲۹) و بررسی مقادیر سیگنال‌های خروجی در محیط شبیه‌ساز، می‌توانیم صحت عملکرد مدار را ارزیابی کنیم.

```
14 - architecture tester of test_fulladder is

15 -   component fulladder
16 -     port (a, b, cin: IN BIT; s, cout: OUT BIT);
17 -   end component;

18 -   signal a, b, cin, s, cout: BIT := '0';

19 - begin

20 - dut : fulladder port map(
21 -    a => a,
22 -    b => b,
23 -    cin => cin,
24 -    s => s,
25 -    cout => cout);
26 - a <= '1' after 100 ns, '0' after 200 ns, '1' after 300 ns, '0' after 400 ns,
27 -      '1' after 500 ns, '0' after 600 ns, '1' after 700 ns, '0' after 800 ns;

28 - b <= '1' after 200 ns, '0' after 400 ns, '1' after 600 ns, '0' after 800 ns;

29 - cin <= '1' after 400 ns, '0' after 800 ns;

30 - end tester;
```

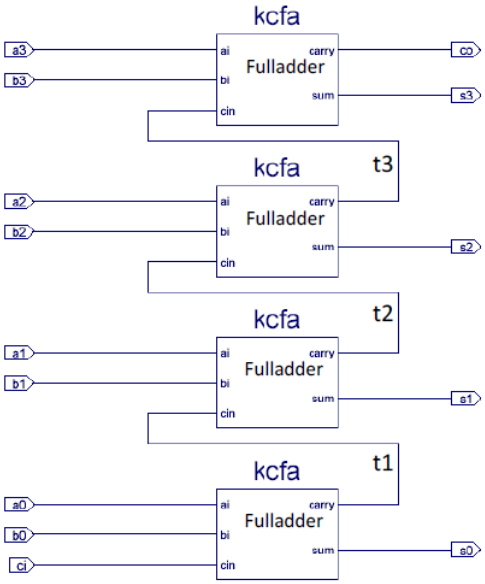
در پایان یک نمونه از خروجی شبیه‌سازی آورده شده است.



طراحی جمع‌کننده ۴ بیتی

دستور کار:

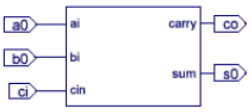
۱. برای یک مدار جمع‌کننده (با رقم نقلی موج‌گونه) ۴ بیتی کد VHDL بنویسید
(از کد قسمت قبل استفاده کنید).
۲. برای مدار حاصل یک محیط تست با استفاده از کد VHDL بنویسید. (Test Bench)
۳. کد نهایی را کامپایل و در صورت وجود خطاهای احتمالی آن‌ها را تصحیح کنید.
۴. مدار حاصل را با استفاده از ModelSim شبیه‌سازی کنید.

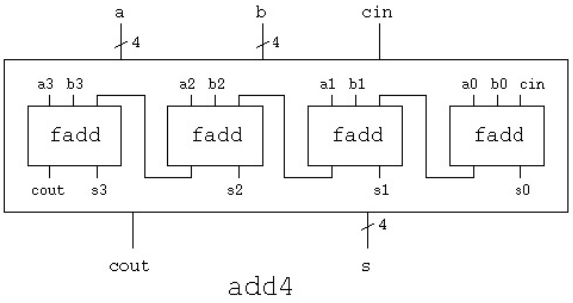


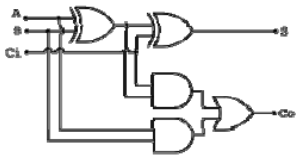
در این بخش با استفاده از گنجاندن کد قبلی در یک کد جدید، یک جمع‌کننده ۴ بیتی می‌سازیم.
(بحث استفاده مجدد از کدهای نوشته شده)

در ابتدا کد قسمت قبل (تمام‌جمع کننده) را مجدداً در شروع کد جدید قرار می‌دهیم:

```
entity fulladder is
 port (a, b, cin: IN BIT; s, cout: OUT BIT);
end fulladder;

architecture structural of fulladder is
 signal t1, t2, t3: BIT;
begin
   t1 <= a XOR b;
   s  <= t1 XOR cin;
   t2 <= t1 AND cin;
   cout <= t2 OR t3;
end structural;
```







حال باید یک entity جدید برای جمع‌کننده ۴ بیتی تعریف کنیم:

```
entity adder4bit is
   port (a, b: IN BIT_VECTOR(3 downto 0); cin: IN BIT;
         s :OUT BIT_VECTOR(3 downto 0); cout: OUT BIT);
end adder4bit;
```

ساختار داخلی این جمع‌کننده ۴ بیتی همانند شکل بالا از ۴ *تمام‌جمع‌کننده* که بطور سری به یکدیگر متصل شده‌اند، تشکیل شده است.

برای اینکار ابتدا *تمام‌جمع‌کننده* را بصورت یک component تعریف می‌کنیم و در داخل کد ۴ نمونه از آن را ایجاد می‌کنیم و ورودی/خروجی آن‌ها را به سیگنال‌های مناسب متصل می‌کنیم. در واقع یک جمع‌کننده ۴ بیتی بصورت *رقم نقلی موج‌گونه* می‌سازیم:

```
architecture structural of adder4bit is
 component fulladder port (a, b, cin: IN BIT; s, cout: OUT BIT); end component;
 signal t1, t2, t3: BIT;
begin
 FA0: fulladder port map (a=>a(0), b=>b(0), cin=>cin, s=>s(0), cout=>t1);
 FA1: fulladder port map (a=>a(1), b=>b(1), cin=>t1, s=>s(1), cout=>t2);
 FA2: fulladder port map (a=>a(2), b=>b(2), cin=>t2, s=>s(2), cout=>t3);
 FA3: fulladder port map (a=>a(3), b=>b(3), cin=>t3, s=>s(3), cout=>cout);
end structural;
```

اکنون طراحی جمع‌کننده ۴ بیتی به پایان رسیده است و برای بررسی نتیجه حاصل لازم است تا این جمع‌کننده را در یک محیط تست نماییم. برای اینکار یک entity بدون ورودی/خروجی ایجاد می‌کنیم.

```
entity test_adder4bit is
end test_adder4bit;
```

در داخل این محیط تست پس از تعریف سیگنال‌های ورودی تست را برای جمع‌کننده ایجاد کنیم و خروجی‌های متناظر را بررسی نماییم.

```
architecture tester of test_adder4bit is

 component adder4bit port (
    a, b: IN BIT_VECTOR(3 downto 0);
    cin: IN BIT;
    s: OUT BIT_VECTOR(3 downto 0); cout: OUT BIT);
end component;
```

در کد بالا جمع‌کننده ۴ بیتی را بعنوان یک component تعریف کردیم. (در ادامه یک نمونه از آن را ایجاد می‌کنیم تا سیگنال‌های تست را به آن اعمال کنیم و نتایج و صحت عملکرد آن را در محیط شبیه‌ساز بررسی کنیم)

```
 signal A_input, B_input, S_output : BIT_VECTOR(3 downto 0);
 signal C_input, C_output : BIT;


 begin
```

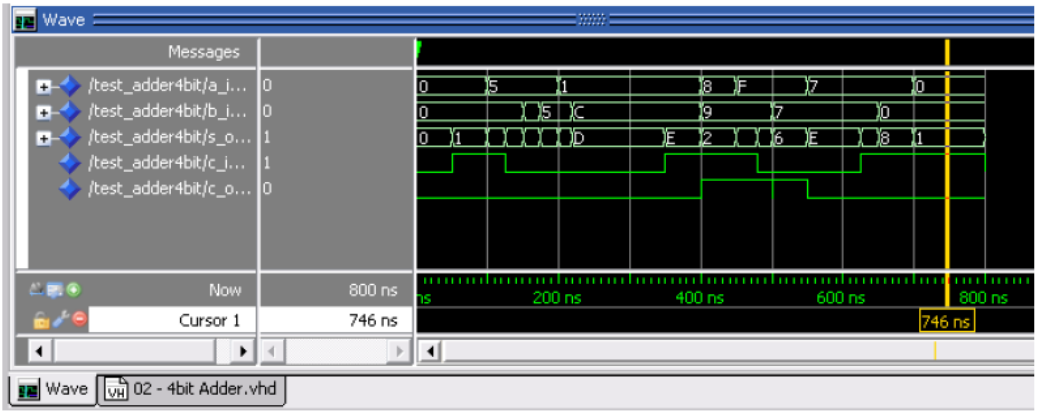
ایجاد یک نمونه از جمع‌کننده ۴ بیتی:

```
  DUT : adder4bit port map (
    a => A_input,
    b => B_input,
   cin => C_input,
    s => S_output,
   cout => C_output  );
```

اعمال سیگنال‌های تست ورودی:

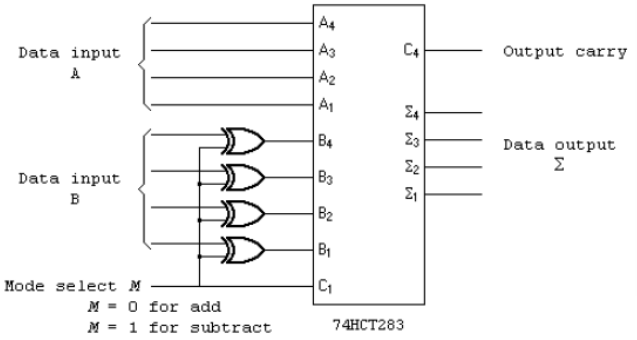
```
  A_input <= "0101" after 100 ns, "0001" after 200 ns, "1000" after 400 ns,
             "1111" after 450 ns, "0111" after 550 ns, "0000" after 700 ns;

  B_input <= "0011" after 150 ns, "0101" after 175 ns, "1100" after 220 ns,
             "1001" after 400 ns, "0111" after 500 ns, "0000" after 650 ns;

  C_input <= '1' after 50 ns, '0' after 125 ns, '1' after 350 ns,
             '0' after 480 ns, '1' after 625 ns, '0' after 800 ns;

 end tester;
```

مشاهده نتایج شبیه‌سازی در محیط ModelSim:



طراحی واحد محاسبات ریاضی

واحد محاسبات ریاضی در اینجا دو عمل جمع و تفریق را انجام می‌دهد. برای پیاده‌سازی آن از جمع‌کننده قسمت قبل بصورت زیر استفاده می‌کنیم.



دستور کار:

۱. برای یک مدار جمع/تفریق‌کننده ۴ بیتی کد VHDL بنویسید (از کد قسمت قبل استفاده کنید).
۲. برای مدار حاصل یک محیط تست با استفاده از کد VHDL بنویسید. (Test Bench)
۳. کد نهایی را کامپایل و در صورت وجود خطاهای احتمالی آن‌ها را تصحیح کنید.
۴. مدار حاصل را با استفاده از ModelSim شبیه‌سازی کنید.

در این بخش با افزودن گیت‌های XOR به جمع‌کننده قسمت قبل یک جمع/تفریق‌کننده ۴ بیتی می‌سازیم. (بحث استفاده مجدد از کدهای نوشته شده)

در ابتدا کد قسمت قبل (تمام‌جمع کننده) را مجدداً در شروع کد جدید قرار می‌دهیم:

اضافه کردن تمام‌جمع‌کننده یک بیتی:

```
entity fulladder is
 port (a, b, cin: IN BIT; s, cout: OUT BIT);
end fulladder;

architecture structural of fulladder is
   ...
end structural;
```

اضافه کردن جمع‌کننده ۴ بیتی:

```
entity adder4bit is
 port (a, b: IN BIT_VECTOR(3 downto 0); cin: IN BIT;
   s: OUT BIT_VECTOR(3 downto 0); cout: OUT BIT);
end adder4bit;

architecture structural of adder4bit is
   ...
end structural;
```

ساختن جمع/تفریق‌کننده ۴ بیتی:

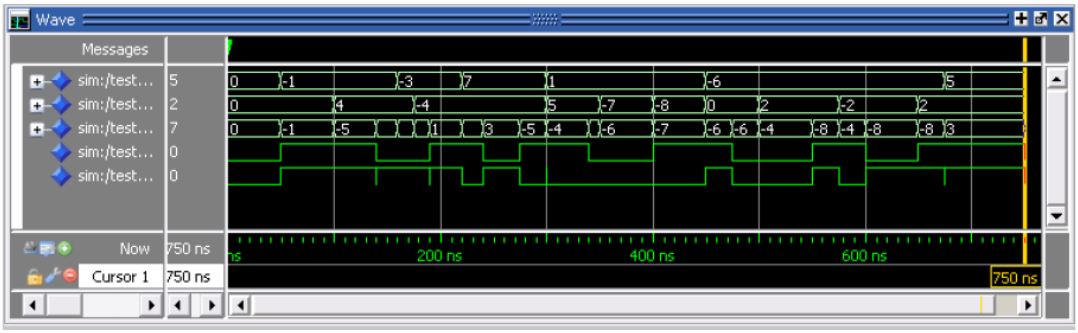
```
entity adder_substractor4bit is
 port (a, b: IN BIT_VECTOR(3 downto 0); mode: IN BIT;
```

```
   s: OUT BIT_VECTOR(3 downto 0); cout: OUT BIT);
end adder_substractor4bit;


architecture structural of adder_substractor4bit is

 component adder4bit port (a, b: IN BIT_VECTOR(3 downto 0); cin: IN BIT;
            s: OUT BIT_VECTOR(3 downto 0); cout: OUT BIT); end component;

   signal temp : BIT_VECTOR(3 downto 0);

begin

    temp(0) <= b(0) XOR mode;
    temp(1) <= b(1) XOR mode;
    temp(2) <= b(2) XOR mode;
    temp(3) <= b(3) XOR mode;

    adder : adder4bit port map (
     a => a,
     b => temp,
     cin => mode,
     s => s,
     cout => cout );

end structural;
```

در اینجا کار طراحی جمع/تفریق‌کننده به پایان رسیده است، برای ارزیابی طرح آن را شبیه‌سازی می‌کنیم و سیگنال‌های ورودی تست را به آن اعمال می‌کنیم، صحت عملکرد طرح را می‌توانیم در محیط شبیه‌سازی با بررسی خروجی‌ها مشخص کنیم.

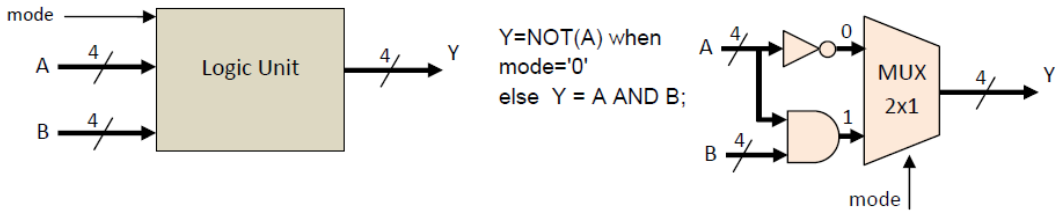
```
entity test_adder_substractor4bit is
end test_adder_substractor4bit;

 architecture tester of test_adder_substractor4bit is

    component adder_substractor4bit port (a, b: IN BIT_VECTOR(3 downto 0); mode: IN BIT;
              s: OUT BIT_VECTOR(3 downto 0); cout: OUT BIT); end component;

    signal A_input, B_input, S_output : BIT_VECTOR(3 downto 0);
    signal Mode_input, C_output : BIT;

 begin

    DUT : adder_substractor4bit port map (
       a => A_input,
       b => B_input,
       mode => Mode_input,
       s => S_output,
       cout => C_output );

     A_input <= "1111" after 50 ns, "1101" after 160 ns, "0111" after 220 ns,
                "0001" after 300 ns, "1010" after 450 ns, "0101" after 675 ns;

     B_input <= "0100" after 100 ns, "1100" after 175 ns, "0101" after 300 ns,
                "0011" after 350 ns, "1000" after 400 ns, "0000" after 450 ns,
                "0010" after 500 ns, "1110" after 575 ns, "0010" after 650 ns;

     Mode_input <= '1' after 50 ns, '0' after 133 ns, '1' after 207 ns,
                   '0' after 240 ns, '1' after 275 ns, '0' after 340 ns,
                   '1' after 400 ns, '0' after 475 ns, '1' after 550 ns,
                   '0' after 600 ns, '1' after 650 ns, '0' after 750 ns;
 end tester;
```

مشاهده نتایج شبیه‌سازی در محیط ModelSim:



طراحی واحد منطقی

آنچه بعنوان واحد منطقی در نظر گرفته شده است، فقط دو عمل AND منطقی و Complement را انجام می‌دهد. در این قسمت به یک مالتی‌پلکسر برای ایجاد خروجی لازم نیاز است. البته برای پیاده‌سازی مالتی‌پلکسر از ساختار زیر (یا مشابه آن) استفاده می‌کنیم.



دستور کار:

۱. برای مدار بالا کد VHDL بنویسید.
۲. برای مدار حاصل یک محیط تست با استفاده از کد VHDL بنویسید. (Test Bench)
۳. کد نهایی را کامپایل و در صورت وجود خطاهای احتمالی آن‌ها را تصحیح کنید.
۴. مدار حاصل را با استفاده از ModelSim شبیه‌سازی کنید.

در ادامه به واحد منطقی می‌پردازیم. ابتدا entity را به همراه پورت‌های ورودی و خروجی (طبق شکل) مشخص می‌کنیم:

```
entity logic_unit is
 port (a, b: IN BIT_VECTOR(3 downto 0); mode: IN BIT;
       y: OUT BIT_VECTOR(3 downto 0);
end logic_unit;
```

برای تعریف ساختار داخلی آن بنوعی از مالتی‌پلکسر نهفته استفاده می‌کنیم (بصورت زیر یا کد بالا هر دو نتیجه یکسانی خواهد داشت):

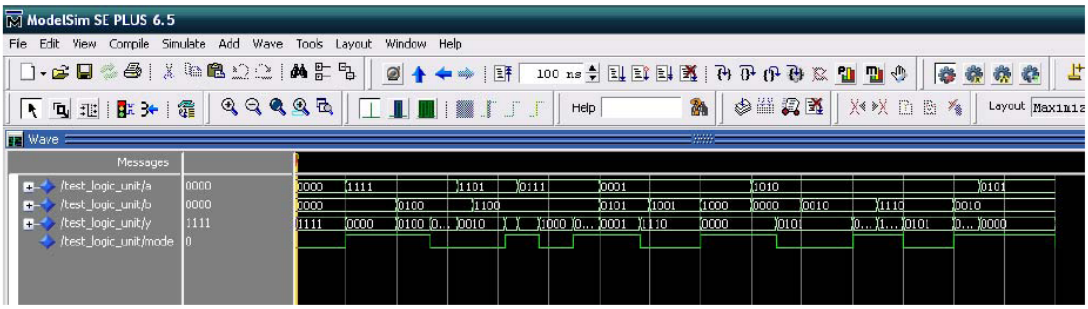
```
architecture arch of logic_unit is
begin

 y <= a AND b when mode='1' else NOT a;

end arch;
```

در ادامه یک محیط تست برای اعمال سیگنال‌های ورودی و بررسی نتایج خروجی ایجاد می‌کنیم:

```
  entity test_logic_unit is
  end test_logic_unit;

  architecture tester of test_logic_unit is

     component logic_unit port (a, b: IN BIT_VECTOR(3 downto 0));
        mode: IN BIT; y: OUT BIT_VECTOR(3 downto 0);
     end component;

     signal a, b, y: BIT_VECTOR(3 downto 0);
     signal mode: BIT;

  begin

     DUT : logic_unit port map (
        a => a,
        b => b,
```

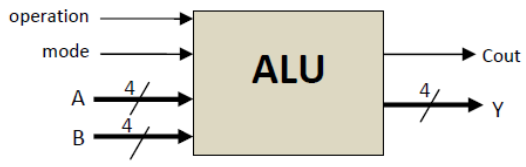
```
      mode => mode,
      y => y );

   A_input <= "1111" after 50 ns, "1101" after 160 ns, "0111" after 220 ns,
              "0001" after 300 ns, "1010" after 450 ns, "0101" after 675 ns;

   B_input <= "0100" after 100 ns, "1100" after 175 ns, "0101" after 300 ns,
              "0011" after 350 ns, "1000" after 400 ns, "0000" after 450 ns,
              "0010" after 500 ns, "1110" after 575 ns, "0010" after 650 ns;

   Mode_input <= '1' after 50 ns, '0' after 133 ns, '1' after 207 ns,
                 '0' after 240 ns, '1' after 275 ns, '0' after 340 ns,
                 '1' after 400 ns, '0' after 475 ns, '1' after 550 ns,
                 '0' after 600 ns, '1' after 650 ns, '0' after 750 ns;
end tester;
```

یک نمونه از شبیه‌سازی مانند شکل زیر می‌باشد:



طراحی واحد محاسبه و منطق (ALU)

از ترکیب دو واحد محاسبات ریاضی و عملیات منطقی، ALU را می‌سازیم.



عملکرد	operation	mode	خروجی ALU
ریاضی	0	0	Y = A + B
		1	Y = A – B
منطقی	1	0	Y = NOT A
		1	Y = A and B

دستور کار:

۲. برای مدار ALU کد VHDL بنویسید.
۳. برای مدار حاصل یک محیط تست با استفاده از کد VHDL بنویسید. (Test Bench)
۴. کد نهایی را کامپایل و در صورت وجود خطاهای احتمالی آن‌ها را تصحیح کنید.
۵. مدار حاصل را با استفاده از ModelSim شبیه‌سازی کنید.

مانند قسمت‌های قبل ALU را به همراه پورت‌های ورودی و خروجی (طبق شکل) طراحی می‌کنیم:

ابتدا ماجول‌های لازم برای ساختن ALU را اضافه می‌کنیم (قسمت‌های حذف شده کد را از جلسات قبل اضافه کنید):

```
entity fulladder is
   port (a, b, cin: IN BIT; s, cout: OUT BIT);
end fulladder;

architecture structural of fulladder is

   ...
end structural;

entity adder4bit is
   port (a, b: IN BIT_VECTOR(3 downto 0); cin: IN BIT;
         s: OUT BIT_VECTOR(3 downto 0); cout: OUT BIT);
end adder4bit;

architecture structural of adder4bit is
   ...
end structural;

entity adder_substractor4bit is
   port (a, b: IN BIT_VECTOR(3 downto 0); mode: IN BIT;
         s: OUT BIT_VECTOR(3 downto 0); cout: OUT BIT);
end adder_substractor4bit;

architecture structural of adder_substractor4bit is
   ...
end structural;

entity logic_unit is
   port (a, b: IN BIT_VECTOR(3 downto 0); mode: IN BIT;
         y: OUT BIT_VECTOR(3 downto 0));
end logic_unit;

architecture arch of logic_unit is
   ...
end arch;
```

در اینجا با اتصال واحدهای منطقی و ریاضی، ALU را می‌سازیم. ماجول ALU را مانند شکل بالا طراحی می‌کنیم:

```
entity ALU_4bit is
   port (A, B: IN BIT_VECTOR(3 downto 0); mode, operation: IN BIT;
         Y: OUT BIT_VECTOR(3 downto 0); cout: OUT BIT);
end ALU_4bit;
```

در ساختار داخلی ALU یک نمونه از واحد منطقی و یک نمونه از واحد ریاضی قرار می‌دهیم. خروجی ALU با توجه به ورودی operation مشخص می‌شود:

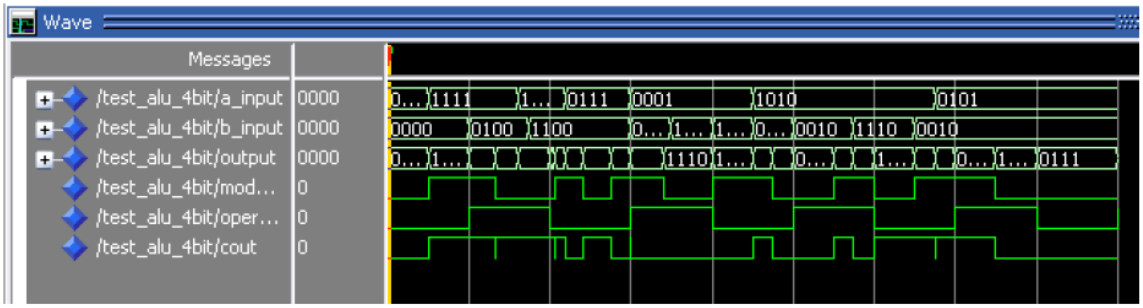
```
architecture arch of ALU_4bit is

   component logic_unit
      port (a, b: IN BIT_VECTOR(3 downto 0); mode: IN BIT;
            y: OUT BIT_VECTOR(3 downto 0));
   end component;

   component adder_substractor4bit
      port (a, b: IN BIT_VECTOR(3 downto 0); mode: IN BIT;
            s: OUT BIT_VECTOR(3 downto 0); cout: OUT BIT);
   end component;

   signal Y_logic , Y_arith: BIT_VECTOR(3 downto 0);

begin

   Logic_part : logic_unit port map (
      a => a,
      b => b,
      mode => mode,
      y => Y_logic );

   Arith_part : adder_substractor4bit port map (
      a => a,
      b => b,
      mode => mode,
      cout => cout,
      s => Y_arith );

   y <= Y_arith when operation='0' else Y_logic;

end arch;
```

حال وقت آن رسیده است که برای بررسی صحت عملکرد ALU طراحی شده یک محیط تست فراهم کنیم و در آن سیگنال‌های لازم را اعمال کنیم و به ارزیابی خروجی‌ها در محیط شبیه‌سازی بپردازیم:

```
entity test_ALU_4bit is
end test_ALU_4bit;

architecture tester of test_ALU_4bit is

   component ALU_4bit
      port (A, B: IN BIT_VECTOR(3 downto 0); mode, operation: IN BIT;
            Y: OUT BIT_VECTOR(3 downto 0); cout: OUT BIT);
   end component;

   signal a_input, b_input, output: BIT_VECTOR(3 downto 0);

   signal mode_input, operation_input, cout: BIT;
```

```
begin

   DUT : ALU_4bit port map (
      A => a_input,
      B => b_input,
      Y => output,
      cout => cout,
      mode => mode_input,
      operation => operation_input );

   a_input <= "1111" after 50 ns, "1101" after 160 ns, "0111" after 220 ns,
              "0001" after 300 ns, "1010" after 450 ns, "0101" after 675 ns;

   b_input <= "0100" after 100 ns, "1100" after 175 ns, "0101" after 300 ns,
              "1001" after 350 ns, "1000" after 400 ns, "0000" after 450 ns,
              "0010" after 500 ns, "1110" after 575 ns, "0010" after 650 ns;

   mode_input <= '1' after 50 ns, '0' after 133 ns, '1' after 207 ns,
                 '0' after 240 ns, '1' after 275 ns, '0' after 340 ns,
                 '1' after 400 ns, '0' after 475 ns, '1' after 550 ns,
                 '0' after 600 ns, '1' after 650 ns, '0' after 750 ns;

   operation_input <= '1' after 100 ns, '0' after 200 ns, '1' after 300 ns,
                      '0' after 400 ns, '1' after 500 ns, '0' after 600 ns,
                      '1' after 700 ns, '0' after 800 ns, '1' after 900 ns;
end tester;
```

یک نمونه از شبیه‌سازی خروجی بصورت زیر می‌باشد:



طراحی فلیپ‌فلاپ و رجیستر (ثبات)

در این قسمت برای طراحی فلیپ‌فلاپ از امکانات پیشرفته‌تر زبان و طراحی رفتاری Behavioral استفاده می‌کنیم. یعنی به جای ساختن ساختار داخلی فلیپ‌فلاپ، رفتار آن را مدلسازی می‌کنیم. برای این‌کار در واقع یک پروسه (در اینجا رفتار فلیپ‌فلاپ) تعریف می‌کنیم که به محرک‌ها (ورودی‌ها) پاسخ مناسب می‌دهد.

فلیپ‌فلاپ مورد نظر در اینجا از نوع D-FF حساس به *لبه بالارونده* می‌باشد. سپس کنر هم قرار دادن فلیپ‌فلاپ‌ها یک ثبات ۴ بیتی می‌سازیم.

دستور کار:

---

۱. برای مدار فلیپ‌فلاپ (D-FF) و ثبات ۴ بیتی (Register 4-bit) بطور مجزا کد VHDL بنویسید. (البته در ثبات از کد D-FF استفاده می‌شود)
۲. برای مدار حاصل یک محیط تست با استفاده از کد VHDL بنویسید. (Test Bench)
۳. کد نهایی را کامپایل و در صورت وجود خطاهای احتمالی آن‌ها را تصحیح کنید.
۴. مدار حاصل را با استفاده از ModelSim شبیه‌سازی کنید.

ساختن مدل رفتاری فلیپ‌فلاپ از نوع D (دقت شود که خروجی Q را از جنس بافر (buffer) تعریف می‌کنیم):

```
entity D_FF is
   port (D, clk, reset: IN BIT; Q: BUFFER BIT :='0');
end D_FF;

architecture bhv of D_FF is
begin
```

رفتار فلیپ‌فلاپ را در قالب یک process یا پروسه[1] به این صورت تعریف می‌کنیم:

اولاً فلیپ‌فلاپ به تغییرات پالس ساعت (clk) حساس می‌باشد. اگر رخدادی (تغییری) در مورد پالس ساعت اتفاق بیفتد و پس از این رخداد پالس ساعت مقدارش '1' شده باشد، یعنی یک لبه بالارونده اتفاق افتاده است پس فلیپ‌فلاپ باید واکنش نشان دهد. اگر پایه reset فعال باشد باید ری‌ست شود (ری‌ست سنکرون) و اگر پایه ری‌ست فعال نباشد فلیپ‌فلاپ کار عادی خود را انجام می‌دهد یعنی ورودی D را به خروجی Q منتقل می‌کند. این مراحل دقیقاً در پروسه زیر انجام می‌گیرد:

```
  process (clk)
  begin
     if (clk'event AND clk='1') then
        if (reset='1') then
           Q <= '0';
        else
           Q <= D;
        end if;
     end if;
  end process;

end bhv;
```

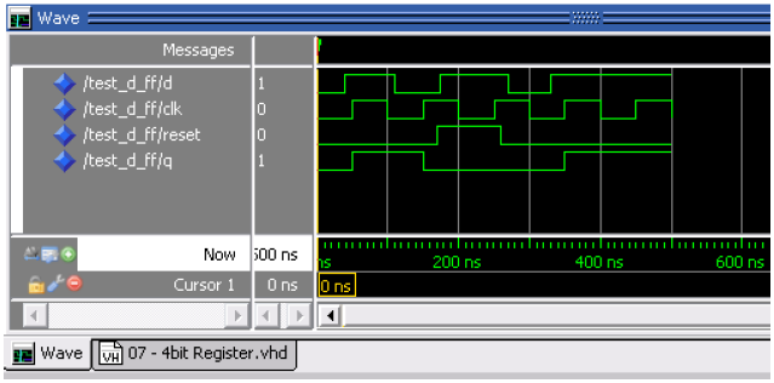
پس از طراحی فلیپ‌فلاپ مانند قبل باید یک محیط تست برای ارزیابی عملکرد آن فراهم کنیم:

```
entity test_D_FF is
end test_D_FF;

architecture tester of test_D_FF is

   component D_FF port (D, clk, reset: IN BIT; Q: BUFFER BIT); end component;

   signal d, clk, reset, q : BIT;

begin

   DUT : D_FF port map (
      D => d,
      clk => clk,
      reset => reset,
      Q => q );

   d <= '1' after 40 ns, '0' after 110 ns, '1' after 175 ns,
        '0' after 270 ns, '1' after 330 ns;

   reset <= '1' after 170 ns, '0' after 260 ns;
```

برای ساختن پالس ساعت نیز از یک پروسه همیشه فعال استفاده می‌کنیم و مقدار کلاک را پس از ۵۰ نانوثانیه عوض می‌کنیم، با این کار یک پالس ساعت با پریود زمانی ۱۰۰ نانوثانیه ایجاد می‌گردد:

```
   clock_generator : process
   begin
      wait for 50 ns; clk  <= not clk;
    end process clock_generator;

end tester;
```

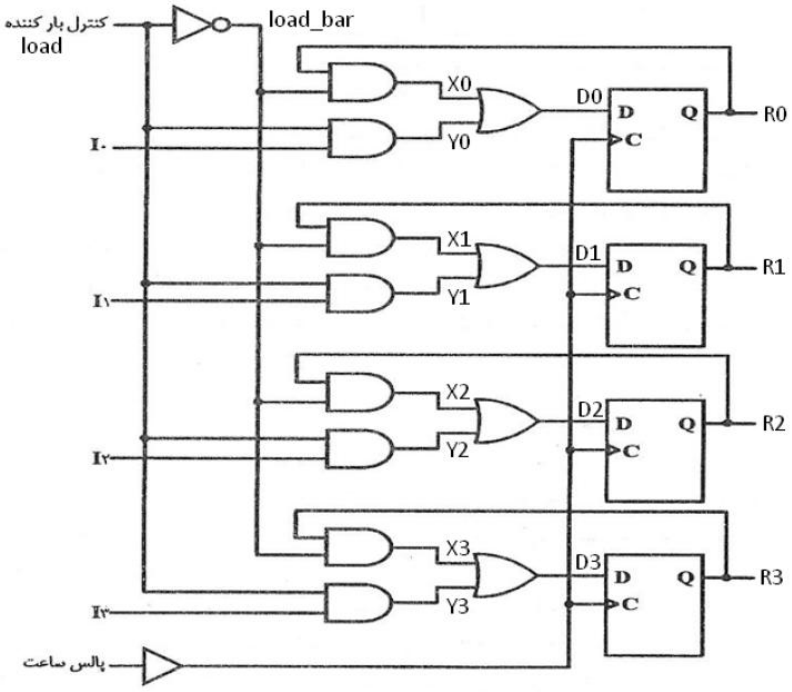
یک نمونه از شبیه‌سازی خروجی بصورت زیر می‌باشد:



برای طراحی رجیستر ۴ بیتی، ۴ فلیپ‌فلاپ را در کنار هم در یک قالب بسته‌بندی می‌کنیم (کد زیر را به قسمت بالا می‌افزاییم):

```
entity Reg_4bit is
   port (I: IN BIT_VECTOR(3 downto 0); clk, reset, load: IN BIT;
         Rout: OUT BIT_VECTOR(3 downto 0));
end Reg_4bit;
```

رجیستر (ثبات) ۴ بیتی از ۴ فلیپ‌فلاپ موازی تشکیل شده است که البته قابلیت بارگزاری به آنها اضافه شده است:



ثبات چهاربیتی با بارکننده Load موازی

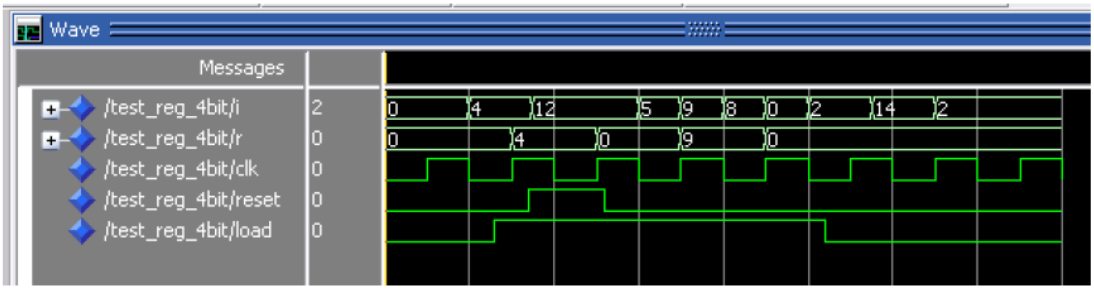
```
architecture arch of Reg_4bit is

   component D_FF port (D, clk, reset: IN BIT; Q: OUT BIT); end component;

   signal load_bar: BIT;
   signal R, D: BIT_VECTOR(3 downto 0);

begin

   load_bar <= NOT load;

   D(0) <= (load_bar AND R(0)) OR (load AND I(0));
   D(1) <= (load_bar AND R(1)) OR (load AND I(1));
   D(2) <= (load_bar AND R(2)) OR (load AND I(2));
   D(3) <= (load_bar AND R(3)) OR (load AND I(3));
```

```
   B0 : D_FF port map ( D=>D(0), clk=>clk, reset=>reset, Q=>R(0) );
   B1 : D_FF port map ( D=>D(1), clk=>clk, reset=>reset, Q=>R(1) );
   B2 : D_FF port map ( D=>D(2), clk=>clk, reset=>reset, Q=>R(2) );
   B3 : D_FF port map ( D=>D(3), clk=>clk, reset=>reset, Q=>R(3) );

   Rout(0) <= R(0);
   Rout(1) <= R(1);
   Rout(2) <= R(2);
   Rout(3) <= R(3);

end arch;
```

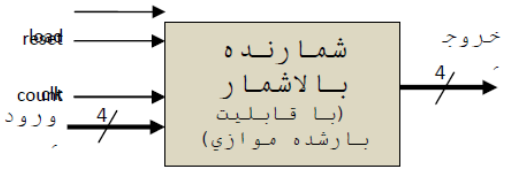
در ادامه برای ارزیابی عملکرد مدار، یک محیط تست برای اعمال سیگنال ورودی و مشاهده و بررسی خروجی فراهم می‌کنیم:

```
entity test_Reg_4bit is
end test_Reg_4bit;

architecture tester of test_Reg_4bit is

   component Reg_4bit
      port(I:IN BIT_VECTOR(3 downto 0); clk, reset:IN BIT;
      R:OUT BIT_VECTOR(3 downto 0));
   end component;

   signal i, r: BIT_VECTOR(3 downto 0);

   signal clk, reset: BIT :='0';

begin

   DUT : Reg_4bit port map
      ( I => i,
        clk => clk,
        reset => reset,
        load => load,
        R => r );

   reset <= '1' after 170 ns, '0' after 260 ns;

   load <= '1' after 130 ns, '0' after 520 ns;

   i <= "0100" after 100 ns, "1100" after 175 ns, "0101" after 300 ns,
        "1001" after 350 ns, "1000" after 400 ns, "0000" after 450 ns,
        "0010" after 500 ns, "1110" after 575 ns, "0010" after 650 ns;

clock_generator : process
   begin
      wait for 50 ns; clk  <= not clk;
   end process clock_generator;

end tester;
```

قسمتی از شبیه‌سازی خروجی بصورت زیر می‌باشد:



طراحی شمارنده ۴ بیتی بالاشمار (کنتور)



در این قسمت برای طراحی شمارنده (مانند بخش قبلی) از روش مدلسازی رفتاری (Behavioral) برای طراحی شمارنده استفاده می‌کنیم. خواهیم دید که این روش به مراتب طراحی شمارنده را راحت‌تر می‌کند. شمارنده‌ای که در اینجا طراحی خواهیم کرد قرار است بصورت زیر رفتار کند:

- دارای یک سیگنال ورودی load سنکرون می‌باشد که این امکان را فراهم می‌کند تا بتواند داده‌های لازم را همگام با پالس ساعت بارگزاری کند.
- دارای یک سیگنال ورودی reset آسنکرون می‌باشد که این امکان را فراهم می‌کند تا در هر لحظه (بدون نیاز به همگامی با پالس ساعت) بتوان شمارنده را صفر کرد.
- و نیز دارای یک سیگنال ورودی count سنکرون می‌باشد که در واقع مثل فرمان شمارش عمل می‌کند یعنی اگر این سیگنال فعال شود شمارنده همگام با پالس ساعت بصورت بالاشمار (صعودی) می‌شمارد و در حالت غیر فعال شدن این سیگنال مقدار شمارنده تغییر نخواهد کرد.

دستور کار:

۱. با استفاده از کد VHDL مدار شمارنده بالا را بصورت رفتاری مدلسازی کنید.
۲. برای مدار حاصل یک محیط تست با استفاده از کد VHDL بنویسید. (Test Bench)
۳. کد نهایی را کامپایل و در صورت وجود خطاهای احتمالی آن‌ها را تصحیح کنید.
۴. مدار حاصل را با استفاده از ModelSim شبیه‌سازی کنید.

مدل رفتاری شمارنده بصورت زیر خواهد بود:

برای مدلسازی رفتاری شمارنده از یک سری توابع از پیش ساخته شده در قالب کتابخانه IEEE استفاده می‌کنیم. لذا کتابخانه مذکور و قسمت‌های مورد نیاز را اضافه می‌کنیم:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
```

بسته std_logic در واقع یک منطق استاندارد تعریف می‌کند (به جای منطق BIT) و نیز std_logic_unsigned شامل توابع لازم برای کارکردن با این نوع داده‌ها می‌باشد. در ادامه به طراحی شمارنده می‌پردازیم (دقت شود تمامی ورودی/خروجی‌ها با استاندار جدید تعریف شده‌اند که در اینجا عملاً با نوع قبلی فرقی نمی‌کند.):

```
entity counter_4bit is

   port (input: IN std_logic_vector(3 downto 0);
         load, count, reset, clk: IN std_logic;
         output: OUT std_logic_vector(3 downto 0));

end counter_4bit;
```

قسمت مهم طراحی در این قسمت می‌باشد که دقیقاً طبق تعریف مساله از شمارنده (قسمت بالا) به مدلسازی رفتار آن می‌پردازد: برای راحتی کار ابتدا یک متغیر temp تعریف می‌کنیم و تمامی مقداردهی‌ها را روی آن انجام می‌دهیم و در نهایت مقدار آن به خروجی منتقل می‌شود.

از آنجا که شمارنده همگام با پالس ساعت کار می‌کند، لذا در لیست محرک‌های پروسه سیگنال clk گذاشته شده است. بقیه سیگنال‌های ورودی (بغیر از reset) با پالس ساعت سنکرون (هماهنگ) هستند. و نیز چون reset بصورت آسنکرون عمل می‌کنند پس باید در لیست محرک‌های پروسه قرار گیرد.

```
architecture bhv of counter_4bit is
begin

   counter : process (clk, reset)

      variable temp: std_logic_vector(3 downto 0):= "0000";

   begin

      if (reset='1') then
         temp := "0000";

      elsif (clk'event AND clk='1') then

         if (load='1') then
            temp := input;

         elsif (count='1') then
            temp := temp + '1';

         end if;

     end if;

     output <= temp;

  end process counter;

end bhv;
```

همانطور که از کد بالا مشخص است، ابتدا سیگنال reset بررسی می‌شود و اگر فعال باشد مقدار شمارنده صفر می‌گردد و در غیر اینصورت (غیرفعال بودن reset) بروز لبه بالارونده پالس ساعت چک می‌شود که در لبه بالارونده ابتدا بار شدن ورودی در شمارنده در صورت فعال بودن سیگنال load انجام می‌گیرد و در غیر اینصورت با فعال بودن سیگنال count شمارنده بصورت صعودی می‌شمارد که در واقع یک واحد به مقدارش افزوده می‌گردد.

در ادامه یک محیط تست برای شمارنده ساخته می‌شود:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity test_counter_4bit is
end test_counter_4bit;

architecture tester of test_counter_4bit is

component counter_4bit
   port (input: IN std_logic_vector(3 downto 0);
         load, count, reset, clk: IN std_logic;
         output: OUT std_logic_vector(3 downto 0));
end component;
```

```
signal i, y : std_logic_vector(3 downto 0):= "0000";

signal clk, count, reset, load : std_logic:='0';


begin

   DUT : counter_4bit port map (
      input => i,
      output => y,
      clk => clk,
      reset => reset,
      load => load,
      count => count);

   reset <= '1' after 75 ns, '0' after 170 ns, '1' after 920 ns, '0' after 1010 ns;

   load <= '1' after 280 ns, '0' after 390 ns, '1' after 910 ns, '0' after 980 ns;

   count <= '1' after 180 ns, '0' after 710 ns, '1' after 840 ns;

   i <= "1001" after 110 ns, "1101" after 260 ns, "1111" after 525 ns;

   clock : process
   begin
      wait for 50 ns; clk <= not clk;
   end process clock;

end tester;
```

قسمتی از شبیه‌سازی خروجی بصورت زیر می‌باشد: