دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

بسم الله الرحمن الرحیم

گزارش پروژه درس معماری کامپیوتر

استاد: دکتر شریعتمدار

دانشکده مهندسی برق

اعضای گروه:

سمیرا سلجوقی 9823048

محمدمسیح شالچیان 9823051

فهرست مطالب

مقدمه

در این پروژه می خواهیم بخش های مختلف یک CPU ساده را پردازش کنیم. این پردازنده شامل حافظه های RAM و ROM، قسمت program counter ، بخش محاسباتی و منطقی ALU و قسمت اصلی State Machine می باشد که هر بخش به طور مفصل توضیح داده خواهند شد. سی پی یو در واقع مغز کامپیوتر است که تمام پردازش ها در این قطعه انجام می گیرد. اجرای برنامه ها و انجام محاسبات، همه و همه بر عهده سی پی یو است. نحوه کار کردن سی پی یو ها از ابتدای زمانی که مورد استفاده قرار گرفتند تا کنون تغییرات زیادی داشته است اما پایه عملکرد آن ها، تا کنون به مراحل واکشی، رمزگشایی و اجرا تقسیم پذیر بوده است.

- واکشی(Fetch)

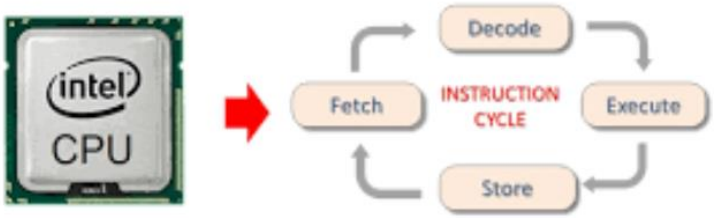
واکشی به معنی دریافت دستوراتی است که سی پی یو در نهایت باید اجرا کند. این دستور العمل ها در قالب صفر و یک از رم به سی پی یو می رسند و هر دستور یک بخش بسیار کوچک از یک عملیات است. نشانی دستور فعالی از طریق یک شمارشگر برنامه نگه داشته می شود و سپس این شمارش گر و دستور العمل ها در بخش ثبت دستور یا IR قرار می گیرند. پس از آن، طول شمارش گر افزایش پیدا می کند تا به نشانی دستورالعمل بعدی ارجاع دهد.

- رمزگشایی(Decode)

زمانی که یک دستور توسط سی پی یو واکشی و دریافت می شود و سپس در IR ذخیره می گردد، سی پی یو دستور را به مداری به اسم رمزگشا انتقال می دهد. این مدار دستور را به سیگنال هایی تبدیل می کند که برای فعالیت به سایر بخش های سی پی یو فرستاده می شوند.

- اجرا(Execute)

در نهایت، دستوراتی که توسط رمزگشا، کدگشایی شده اند می بایست اجرا شوند. نتایج این کد گشایی ها در قسمتی از سی پی یو ثبت می گردند تا در دستور العمل های بعدی نیز بتوان به آن ها بازگشت. مثل عملکرد حافظه ماشین حساب.[1]



بخش ROM

حافظه ROM نوعی از حافظه است که دستورات در آن بخش ذخیره می شوند و فقط عملیات خواندن در آن اجرا می شود. ROMمخفف Read-Only Memory است. حافظه رام برخلاف رم، پایدار است و حتی وقتی کامپیوتر خاموش شود، محتوای آن باقی می‌ماند. رام در واقع یک مدار کوچک کامپیوتری روی مادربورد است که اطلاعات آن را شرکت سازنده پر میکند. اطلاعات داخل این قطعه می‌تواند بارها اجرا شود؛ پس قطعه مهمی است و بر عکس رم، اطلاعات حافظه rom با خاموش شدن کامپیوتر از بین نمی‌رود.

در کد زیر که حافظه ROM طراحی شده است، در ابتدا پس از تعیین مقادیر ثابت اندازه حافظه -در اینجا 16*1024 است-، یک ورودی برای گرفتن آدرس و یک خروجی برای خواندن دستور تعریف می کنیم. سپس برای نوشتن دستورات یک rom_type با ابعاد گفته شده تعریف می کنیم و دستورات را به صورت زیر می نویسیم که شامل تمام دستورات خواسته شده است و برای دستوراتی که نیازمند آدرس هستند نیز آدرس یکی از خانه های حافظه RAM را قرار می دهیم. در آخر نیز با تبدیل آدرس ورودی به عدد صحیح از حافظه دستور مورد نظر را داخل خروجی قرار می دهیم. این بخش از کد با کمک سایت [2] نوشته شده است.

```
2  library IEEE;
3  use IEEE.STD_LOGIC_1164.ALL;
4  use IEEE.NUMERIC_STD.ALL;
5
6  entity ROM_Module is
7      generic(
8          addr_width : integer := 1024; --total number of elements to store (put exact number)
9          addr_bits  : integer := 10;  -- bits requires to store elements specified by addr_width
10         data_width : integer := 16   -- number of bits in each elements
11         );
12 port(
13     addr : in std_logic_vector(addr_bits-1 downto 0);  --input port for getting address
14     data : out std_logic_vector(data_width-1 downto 0) --ouput data at location 'addr'
15 );
16 end ROM_Module;
17 architecture Behavioral of ROM_Module is
18     type rom_type is array (0 to addr_width-1) of std_logic_vector(data_width-1 downto 0);
19     constant ROM_Memory : rom_type := (
20                           0 => "0000010000000100",  --AND
21                           1 => "0000100000001000",  --Store
22                           2 => "0000110000010000",  --Load
23                           3 => "0001000000000001",  --Add
24                           4 => "0001010000000000",  --Increment AC
25                           5 => "0001100000000000",  --Clear AC
26                           6 => "0001110000000000",  --Clear E
27                           7 => "0010000000000000",  --Circular Left Shift
28                           8 => "0010010000000000",  --Circular Right Shift
29                           9 => "0010100000000000",  --SPA
30                           10 => "0010110000000000", --SNA
31                           11 => "0011000000000000", --SZE
32                           12 => "0011010000000000", --SZA
33                           13 => "0011100000000000", --Linear Left Shift
34                           14 => "0011110000000000", --Linear Right Shift
35                           15 => "0100000000001100", --Multiply
36                           16 => "1000000000000011", -- SQR
37                           others => (others => '0'));
38 begin
39     data <= ROM_Memory(to_integer(unsigned(addr)));
40 end Behavioral;
```

بخش RAM

RAM که مخفف Random Access Memory است نوعی حافظه است که امکان خواندن و نوشتن را فراهم می کند. این حافظه فرار در نظر گرفته می شود، به این معنی که محتوای آن با قطع برق از بین می رود. RAM ذخیره‌سازی سریع و موقتی برای داده‌هایی فراهم می‌کند که پردازشگر یا سایر اجزا می‌توانند از آن‌ها خوانده و نوشته شوند.

ویژگی های کلیدی رم:

- عملیات خواندن و نوشتن: RAM اجازه می‌دهد داده‌ها هم از سلول‌های حافظه خوانده و هم نوشته شوند.
- حافظه فرار: RAM برای حفظ اطلاعات ذخیره شده خود به منبع تغذیه مداوم نیاز دارد.
- دسترسی تصادفی: سلول های حافظه در RAM به صورت تصادفی قابل دسترسی هستند، به این معنی که به هر مکانی می توان مستقیماً بدون نیاز به پیمایش متوالی کل حافظه دسترسی داشت.
- زمان دسترسی سریع: RAM دسترسی سریع به داده های ذخیره شده را فراهم می کند و آن را برای ذخیره سازی موقت و پردازش سریع داده ها مناسب می کند.

شرح کد:

در RAM_Module دو متغییر با مثدار ثابت adr_width و data_width تعریف شده که اولی نشان دهنده ی طول آدرس حافظه ی رم است و دومی برابر تعداد بیت های هر کلمه ی رم است.

ماژول رم ما دارای 5 ورودی به شرح زیر است:

- clk: سیگنال ساعت برای عملیات همزمان
- we: برای کنترل عملیات نوشتن، اگر برابر یک شود عملیات نوشتن انجام میشود
- addr: پورت ورودی برای آدرس محل حافظه برای خواندن یا نوشتن در آن.
- din: داده های ورودی برای ذخیره در ماژول RAM
- dout: داده های خروجی خوانده شده از ماژول RAM

ram_type به عنوان یک نوع آرایه که نشان دهنده حافظه RAM است تعریف می شود. اندازه آرایه با 2**addr_width تعیین می شود که برابر 2 به توان addr_width است. هر عنصر آرایه یک std_logic_vector از طول data_width است.

سیگنال RAM_Data با استفاده از ram_type تعریف می شود و با مقادیر از پیش تعریف شده مقداردهی اولیه می شود. با توجه به دستور کار برای حافظه RAM 64 مقدار دیتا در نظر می گیریم و مقدار دهی می کنیم. این سیگنال حافظه RAM را نشان می دهد و داده ها را ذخیره می کند.

```
--RAM Module

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.all;


entity RAM_Module is
  generic (
    addr_width : integer := 10;   --total number of elements to store (put exact number)
    data_width : integer := 16    --number of bits in each elements
  );

  port(
    clk: in std_logic;
    we : in std_logic;                                      --write enable
    addr : in std_logic_vector(addr_width-1 downto 0);  --input port for getting address
    din : in std_logic_vector(data_width-1 downto 0);   --input data to be stored in RAM
    dout : out std_logic_vector(data_width-1 downto 0)  --output data read from RAM
    );
end RAM_Module;

```

```
architecture Behavioral  of RAM_Module is
   type ram_type is array (0 to 2**addr_width-1) of std_logic_vector (data_width-1 downto 0);
   signal RAM_Data : ram_type := ( 0 => "0000000000000000",
                                   1 => "0000000000000001",
                                   2 => "0000000000000010",
                                   3 => "0000000000000011",
                                   4 => "0000000000000100",
                                   5 => "0000000000000101",
                                   6 => "0000000000000110",
                                   7 => "0000000000000111",
                                   8 => "0000000000001000",
                                   9 => "0000000000001001",
                                   10 => "0000000000001010",
                                   11 => "0000000000001011",
                                   12 => "0000000000001100",
                                   13 => "0000000000001101",
                                   14 => "0000000000001110",
                                   15 => "0000000000001111",
                                   16 => "0000000000010000",
                                   17 => "0000000000010001",
                                   18 => "0000000000010010",
                                   19 => "0000000000010011",
                                   20 => "0000000000010100",
                                   21 => "0000000000010101",
                                   22 => "0000000000010110",
                                   23 => "0000000000010111",
                                   24 => "0000000000011000",
                                   25 => "0000000000011001",
                                   26 => "0000000000011010",
                                   27 => "0000000000011011",
                                   28 => "0000000000011100",
                                   29 => "0000000000011101",
                                   30 => "0000000000011110",
                                   31 => "0000000000011111",
                                   32 => "0000000000100000",
                                   33 => "0000000000100001",
                                   34 => "0000000000100010",
```

در داخل فرآیند، یک بخش کلاک، حساس به سیگنال clk وجود دارد. هنگامی که یک لبه در حال افزایش تشخیص داده می شود(clk'event ='1') ، فرآیند اجرا میشود.

اگر سیگنال فعال کردن نوشتن به ما اعلام شود('1'=we) ، به این معنی است که یک عملیات نوشتن درخواست شده است. آدرس adr با استفاده از تبدیل to_integer(unsigned(addr)) از std_logic_vector به عدد صحیح تبدیل می شود. سپس داده ورودی (din) در آدرس مشخص شده در RAM ذخیره می شود.

قسمت آخر کد با خواندن حافظه رم از آدرسی که توسط adr مشخص شده است، داده های خروجی را در dout قرار میدهد. این بخش از کد با کمک سایت [3] نوشته شده است.

```
                                43 => "0000000000101011",
                                44 => "0000000000101100",
                                45 => "0000000000101101",
                                46 => "0000000000101110",
                                47 => "0000000000101111",
                                48 => "0000000000110000",
                                49 => "0000000000110001",
                                50 => "0000000000110010",
                                51 => "0000000000110011",
                                52 => "0000000000110100",
                                53 => "0000000000110101",
                                54 => "0000000000110110",
                                55 => "0000000000110111",
                                56 => "0000000000111000",
                                57 => "0000000000111001",
                                58 => "0000000000111010",
                                59 => "0000000000111011",
                                60 => "0000000000111100",
                                61 => "0000000000111101",
                                62 => "0000000000111110",
                                63 => "0000000000111111");

begin


   process(clk)
   begin
      if (clk'event and clk='1') then
         if (we='1') then                    -- write data to address 'addr'
            --convert 'addr' type to integer from std_logic_vector
             RAM_Data(to_integer(unsigned(addr))) <= din;
         end if;
      end if;
   end process;

  -- read data from address 'addr'
  -- convert 'addr' type to integer from std_logic_vector
  dout <=  RAM_Data(to_integer(unsigned(addr)));
end Behavioral ;
```

بخش ALU

یک واحد منطقی حسابی (ALU) یک مدار دیجیتالی است که عملیات حسابی و منطقی را روی داده های باینری انجام می دهد. این یک جزء اساسی از واحد پردازش مرکزی (CPU) در یک کامپیوتر است. ALU مسئول اجرای عملیات حسابی است مانند جمع کردن ، شیفت دادن و عملیات منطقی AND کردن. این نکته قابل تامل هست که عملیات قابل انجام توسط هر ALU در پردازنده های متفاوت فرق میکند.

شرح کد:

در ابتدا به ورودی و خروجی های ماژول ALU میپردازیم.

پورت های ورودی:

- AC_IN: ورودی 16 بیتی برای AC اولیه
- X_IN: ورودی 16 بیتی دیتا از حافظه
- ALU_Sel: ورودی 6 بیتی اپکد
- E_IN: مقدار ابتدایی فلیپ فلاپ E
- Enable: فعال کردن سیگنال برای عملیات ALU.

پورت های خروجی:

- AC_OUT: خروجی 16 بیتی برای AC بدست آمده بعد انجام عملیات
- E_OUT: خروجی مقدار E

```
1  --ALU_Module
2
3  library IEEE;
4  use IEEE.STD_LOGIC_1164.ALL;
5  use IEEE.NUMERIC_STD.ALL;
6  use IEEE.STD_LOGIC_UNSIGNED.ALL;
7
8  entity ALU_Module is
9     Port (
10     AC_IN :  in  STD_LOGIC_VECTOR(15 downto 0);     --input AC
11     X_IN   : in  STD_LOGIC_VECTOR(15 downto 0);     --input of memory
12     ALU_Sel  : in  STD_LOGIC_VECTOR(5 downto 0);    --input 6-bit for selecting function
13     E_IN : in STD_LOGIC;                            --input E
14     Enable : in STD_LOGIC;                          --input enable
15     AC_OUT   : out  STD_LOGIC_VECTOR(15 downto 0); --output AC
16     E_OUT : out STD_LOGIC                           --output E
17     );
18 end ALU_Module;
```

سیگنال ها:

- ALU_Result: سیگنال 17 بیتی (با در نظر گرفتن E) بدون علامت برای محاسبات میانی در ALU استفاده می شود.
- MUL_Result: سیگنال 12 بیتی برای ذخیره نتیجه عملیات ضرب.
- SQRT_Result: سیگنال 8 بیتی برای ذخیره نتیجه عملیات ریشه مربع.

در ادامه نیز ماژول های ضرب کننده و جذرگیر را اضافه کرده و سیگنال های مربوطه را متصل می کنیم.

```
20  architecture Behavioral of ALU_Module is
21
22      -- Temporary
23      signal ALU_Result : unsigned (16 downto 0);
24      signal MUL_Result : STD_LOGIC_VECTOR (11 downto 0);
25      signal SQRT_Result : STD_LOGIC_VECTOR (7 downto 0);
26
27      -- Component of Multiplier Module
28      component simple_multi
29           port (
30               x : in  STD_LOGIC_VECTOR (5 downto 0);
31               y : in  STD_LOGIC_VECTOR (5 downto 0);
32               p : out  STD_LOGIC_VECTOR (11 downto 0));
33      end component;
34
35      -- Component of Square root Module
36      component main
37            port ( A : in  STD_LOGIC_VECTOR (16 downto 1);
38                   q : out  STD_LOGIC_VECTOR (8 downto 1));
39      end component;
40
41
42
43  begin
44
45      Multiplier_Module: simple_multi
46         port map ( x  => AC_IN(5 downto 0),
47                    y => X_IN(5 downto 0),
48                    p => MUL_Result);
49                    |
50      Square_root_Module: main
51         port map( A => X_IN(15 downto 0),
52                   q =>  SQRT_Result);
```

وقتی Enable فعال یا برابر 1 باشد، مقدار ALU_Sel با دستور case حالت بندی میشود و برای آپکد دستورات و محاسبات مربوط به آن آپکد اجرا می‌کند.

اپکد ها:

- AND

وقتی ALU_Sel =000001 است، ALU یک عملیات AND منطقی را بین نمایش های بدون علامت AC_IN و X_IN انجام می دهد.نتیجه در ALU_Result (بیت های 15 تا 0) ذخیره می شود.خروجی 16 بیتی AC_OUT با نتیجه به روز می شود. سیگنال خروجی E_OUT برابر 0 قرار داده میشود.

- ADD

وقتی ALU_Sel= 000100است ALU نمایش های بدون علامت AC_IN و X_IN را اضافه می کند و نتیجه را در ALU_Result ذخیره می کند. خروجی 16 بیتی AC_OUT با مجموع AC_IN و X_IN به روز میشود. سیگنال خروجی E_OUT با توجه به مقدار سرریز به روز می شود (ALU_Result[16]).
ALU_Result با چسباندن 0 به مجموع AC_IN و X_IN تشکیل می شود.

```
56  stim_proc: process
57  begin
58     wait for 1 ns;
59     if Enable='1' then
60     case(ALU_Sel) is
61        when "000001" => -- AND
62           ALU_Result(15 downto 0) <= unsigned(AC_IN) and unsigned(X_IN);
63           AC_OUT <= std_logic_vector(ALU_Result(15 downto 0));
64           E_OUT <= '0';
65
66        when "000100" => -- ADD
67           ALU_Result <= '0' & unsigned(AC_IN) + unsigned(X_IN);
68           AC_OUT <= std_logic_vector(ALU_Result(15 downto 0));
69           E_OUT <= ALU_Result(16);
```

- INC

وقتی ALU_Sel= 000101 است

ALU نمایش بدون علامت AC_IN را 1 افزایش می دهد.

نتیجه در ALU_Result ذخیره می‌شود و یک 0 اضافی به سمت چپ اضافه می‌شود.

خروجی 16 بیتی AC_OUT با مقدار افزایش یافته AC_IN به روز می شود

سیگنال خروجی E_OUT با توجه به مقدار سرریز به روز می شود.

- CLA

مقدار AC_OUT را برابر 0 قرار میدهد.

- CLE

مقدار E_OUT را 0 قرار میدهد.

```
70        when "000101" =>   --Increment AC
71           ALU_Result <= unsigned('0' & unsigned(AC_IN) + 1);
72           AC_OUT <= std_logic_vector(ALU_Result(15 downto 0));
73           E_OUT <= ALU_Result(16);
74
75        when "000110" =>    --clear AC
76           AC_OUT <= (others => '0');
77
78        when "000111" =>    -- clear E
79           E_OUT <= '0';
80
```

- Circular Left Shift

وقتی ALU_Sel= 001000 باشد، ALU یک جابجایی دایره ای به چپ در ورودی 16 بیتی AC_IN انجام می دهد. بیت های AC_IN[0:14] یک موقعیت به چپ منتقل می شوند و در ALU_Result[1:15] ذخیره می شوند. مهم ترین بیت (AC_IN[15]) به کم اهمیت ترین بیت (ALU_Result[0]) برای تغییر دایره ای اختصاص داده می شود. خروجی 16 بیتی AC_OUT با نتیجه شیفت دایره ای به چپ (ALU_Result[15:0]) به روز می شود. سیگنال خروجی E_OUT با مقدار سمت چپ ترین بیت AC_IN به روز می شود.

- Circular Right Shift

مشابه چرخش به چپ است.

```
81          when "001000" =>     -- circular left shift
82              ALU_Result(15 downto 1) <= unsigned(AC_IN(14 downto 0));
83              ALU_Result(0) <= E_IN;
84              AC_OUT <= std_logic_vector(ALU_Result(15 downto 0));
85              E_OUT <= AC_IN(15);
86
87          when "001001" =>     -- circular right shift
88              ALU_Result(14 downto 0) <= unsigned(AC_IN(15 downto 1));
89              ALU_Result(15) <= E_IN;
90              AC_OUT <= std_logic_vector(ALU_Result(15 downto 0));
91              E_OUT <= AC_IN(0);
```

- Linear Left Shift

وقتی ALU_Sel = 001110، ALU یک شیفت خطی به چپ در ورودی 16 بیتی AC_IN انجام می دهد. بیت‌های AC_IN[0:14] یک موقعیت به چپ منتقل می‌شوند و در ALU_Result[1:15] ذخیره می‌شوند. کم اهمیت ترین بیت (ALU_Result[0]) برابر 0 قرار داده میشود. خروجی 16 بیتی AC_OUT با نتیجه شیفت خطی به چپ (ALU_Result[15:0]) به روز می شود. سیگنال خروجی E_OUT با سمت چپ ترین بیت (AC_IN[15]) به روز می شود.

- Linear Right Shift

مشابه چرخش به چپ با این فرق که E_OUT برابر صفر میشود.

```
94          when "001110" =>     -- linear left shift
95              ALU_Result(15 downto 1) <= unsigned(AC_IN(14 downto 0));
96              ALU_Result(0) <= '0';
97              AC_OUT <= std_logic_vector(ALU_Result(15 downto 0));
98              E_OUT <= AC_IN(15);
99
100
101         when "001111" =>     -- linear right shift
102             ALU_Result(14 downto 0) <= unsigned(AC_IN(15 downto 1));
103             ALU_Result(15) <= E_IN;
104             AC_OUT <= std_logic_vector(ALU_Result(15 downto 0));
105             E_OUT <= '0';
106
```

- Multiply

وقتی ALU_Sel = 010000 ورودی‌های AC_IN(0:5) و X_IN(0:5) به ترتیب به پورت‌های x و y کامپوننت simple_multi متصل می‌شوند.خروجی 12 بیتی ماژول ضرب، MUL_Result، به 12 بیت سمت راست AC_OUT اختصاص داده می شود. 4 بیت سمت چپ AC_OUT برابر 0000 قرار داده میشود. سیگنال خروجی E_OUT روی 0 تنظیم شده است که نشان دهنده از عدم سرریز است.

- SQR

وقتی ALU_Sel = 100000 (نشان دهنده عملیات ریشه مربع است)

ورودی X_IN (15 تا 0) به پورت A ماژول ریشه مربع متصل می شود. خروجی SQRT_Result ماژول ، به 8 بیت سمت راست AC_OUT اختصاص داده می شود. 8 بیت سمت چپ AC_OUT برابر 00000000 قرار داده میشود. سیگنال خروجی E_OUT برابر 0 قرار داده میشود است که عدم سرریز از عملیات ریشه مربع را نشان دهد.

```
107
108         when "010000" =>     -- Multiply
109            AC_OUT(15 downto 12) <= "0000";
110            AC_OUT(11 downto 0) <= std_logic_vector(MUL_Result);
111            E_OUT <= '0';
112
113
114         when "100000" =>     --SQR
115             AC_OUT(7 downto 0) <= SQRT_Result;
116             AC_OUT(15 downto 8) <= "00000000";
117             E_OUT <= '0';
118
119
120         when others =>       --Nothing
121
122
123       end case;
124       end if;
125    end process;
126
127 end Behavioral;
```

بخش Program Counter

رجیستر شمارنده برنامه (PC) جزء اساسی یک CPU است که آدرس حافظه دستورالعمل بعدی را که قرار است fetch و execute شود را نگهداری می کند و نقش مهمی در اجرای برنامه ها و کنترل جریان پردازنده دارد.

در یک CPU، شمارنده برنامه به صورت متوالی افزایش می یابد تا دستور بعدی را در حافظه دریافت کند، و به پردازنده اجازه می دهد دستورالعمل ها را به ترتیب صحیح اجرا کند. با این حال، شمارنده برنامه را می‌توان از طریق دستورالعمل‌های جریان کنترلی مانند پرش‌ها، شاخه‌ها و فراخوانی‌های ساب روتین تغییر داد و اجرای دستورات غیر متوالی را امکان‌پذیر کرد.

شمارنده برنامه از یک ریجستر باینری تشکیل شده است که آدرس حافظه را ذخیره می کند. بر اساس آپکد فعلی و سیگنال های کنترلی از دیکدر دستورالعمل به روز می شود. آپکد عملیاتی را که باید روی شمارنده برنامه انجام شود، مشخص می کند که کدام یک از عملیات که شامل افزایش، تخصیص مقدار جدید، ریست کردن یا بدون تغییر نگه داشتن مقدار فعلی است انجام شود.

شرح کد: Program Counter را با 4 ورودی که به شرح زیر است تعریف میکنیم.

- clk: یک سیگنال ساعت ورودی
- PC_in: یک سیگنال ورودی که نشان دهنده مقداری است که باید به شمارنده برنامه نسبت داده شود زمانی که کد عملیات روی PC_ASSIGN تنظیم شود.
- PC_opcode: یک سیگنال ورودی که کد opcode را برای عملیات شمارنده برنامه نشان می دهد
- PC_out: یک سیگنال خروجی نشان دهنده مقدار فعلی شمارنده برنامه

معماری Behavioral رفتار شمارنده برنامه را توصیف می کند. یک سیگنال current_pc از نوع STD_LOGIC_VECTOR را برای ذخیره مقدار فعلی شمارنده برنامه اعلام می کند. در ابتدا تماما روی صفر تنظیم شده است.

در داخل عبارت process، شمارنده برنامه بر اساس کد opcode و سیگنال ساعت به روز می شود. هنگامی که لبه افزایشی ساعت رخ می دهد (clk'event و clk='1')، فرآیند با استفاده از یک دستور case، کد عملیاتی را ارزیابی می کند.

- اگر کد عملیاتی PC_HLT باشد، شمارنده برنامه بدون تغییر باقی می ماند (هیچ عملیاتی انجام نمی شود).

- اگر کد عملیاتی PC_INC باشد، شمارنده برنامه با استفاده از تبدیل بدون علامت یک عدد افزایش می یابد و سپس به current_pc اختصاص داده می شود.
- اگر کد عملیاتی PC_TWO_INC باشد، شمارنده برنامه با استفاده از تبدیل بدون علامت دو عدد افزایش می یابد و سپس به current_pc اختصاص داده می شود.
- اگر کد عملیات PC_RESET باشد، شمارنده برنامه به صفرها بازنشانی می شود.
- اگر کد عملیاتی مقدار دیگری باشد، هیچ عملیاتی انجام نمی شود.

در نهایت، مقدار current_pc به PC_out اختصاص داده می شود تا خروجی ماژول شمارنده برنامه را فراهم کند. این بخش از کد با کمک سایت [4] نوشته شده است.

```
--Program Counter
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;


entity Program_Counter is
   generic (
       PC_width : integer := 10   --total number of program counter
     );

   Port ( clk : in  STD_LOGIC;
          PC_in : in  STD_LOGIC_VECTOR (PC_width-1 downto 0);
          PC_opcode : in  STD_LOGIC_VECTOR (1 downto 0);
          PC_out : out  STD_LOGIC_VECTOR (PC_width-1 downto 0)
          );
end Program_Counter;


architecture Behavioral of Program_Counter is
   signal current_pc: STD_LOGIC_VECTOR(PC_width-1 downto 0) := "0000000000";

   -- PC unit opcodes
   constant PC_HLT: STD_LOGIC_VECTOR(1 downto 0):= "00";    --Halt (keep PC the same)
   constant PC_INC: STD_LOGIC_VECTOR(1 downto 0):= "01";    --one increment
   constant PC_TWO_INC: STD_LOGIC_VECTOR(1 downto 0):= "10"; --two increment
   constant PC_RESET: STD_LOGIC_VECTOR(1 downto 0):= "11";  --Reset
```

```
begin


  process (clk)
  begin
    if (clk'event and clk='1') then
      case PC_opcode is
        when PC_HLT =>

        when PC_INC =>
          current_pc <= STD_LOGIC_VECTOR(unsigned(current_pc) + 1);

        when PC_TWO_INC =>
          current_pc <= STD_LOGIC_VECTOR(unsigned(current_pc) + 2);

        when PC_RESET =>
          current_pc <= "0000000000";

        when others =>
      end case;
    end if;
  end process;

  PC_out <= current_pc;

end Behavioral;
```

بخش اصلی برنامه

این قسمت بخش اصلی برنامه و CPU است که در آن استیت های مختلف اجرا می شوند. کلاک و ریست را به عنوان ورودی در نظر می گیریم و در ادامه ماژول هایی که در بخش های قبل توضیح داده شدند را به صورت component به کد اضافه می کنیم.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.all;

entity State_Machine is
   port( reset : in STD_LOGIC;
         clk : in  STD_LOGIC);
end State_Machine;

architecture Behavioral of State_Machine is
--------------------------------------------------------------------------------ALU
component ALU_Module
   Port (
    AC_IN :  in  STD_LOGIC_VECTOR(15 downto 0);     --input AC
    X_IN   : in  STD_LOGIC_VECTOR(15 downto 0);     --input of memory
    ALU_Sel  : in  STD_LOGIC_VECTOR(5 downto 0);    --input 6-bit for selecting function
    E_IN : in STD_LOGIC;                            --input E
    Enable : in STD_LOGIC;
    AC_OUT   : out  STD_LOGIC_VECTOR(15 downto 0);  --output AC
    E_OUT : out STD_LOGIC                           --output E
    );
end component;
--------------------------------------------------------------------------------RAM
component RAM_Module
  generic (
    addr_width : integer := 10;   --total number of elements to store (put exact number)
    data_width : integer := 16    --number of bits in each elements
  );

  port(
    clk: in std_logic;
    we : in std_logic;                                      --write enable
    addr : in std_logic_vector(addr_width-1 downto 0);  --input port for getting address
    din : in std_logic_vector(data_width-1 downto 0);   --input data to be stored in RAM
    dout : out std_logic_vector(data_width-1 downto 0)  --output data read from RAM
    );
end component;
```

```
--------------------------------------------------------------------------------ROM
component ROM_Module
    generic(
        addr_width : integer := 1024; --total number of elements to store (put exact number)
        addr_bits  : integer := 10;  -- bits requires to store elements specified by addr_width
        data_width : integer := 16   -- number of bits in each elements
        );
port(
    addr : in std_logic_vector(addr_bits-1 downto 0);  --input port for getting address
    data : out std_logic_vector(data_width-1 downto 0) --ouput data at location 'addr'
);
end component;
--------------------------------------------------------------------------------PC
component Program_Counter
   generic (
       PC_width : integer := 10   --total number of program counter
     );

   Port ( clk : in  STD_LOGIC;
          PC_in : in  STD_LOGIC_VECTOR (PC_width-1 downto 0);
          PC_opcode : in  STD_LOGIC_VECTOR (1 downto 0);
          PC_out : out  STD_LOGIC_VECTOR (PC_width-1 downto 0)
          );
end component;
```

در قسمت تعریف کردن سیگنال های مورد نیاز با استفاده از دستور type برای 6 استیت مورد نظر که شامل fetch,decode,read_ram,execute,ac_update,hlt است را مشخص می کنیم که استیت Ac_update برای گرفتن خروجی های alu استفاده می شود و hlt پایان اجرای دستورات است. مقدار دهی اولیه Fetch قرار می دهیم. در ادامه نیز برای اتصال ماژول ها به برنامه اصلی نیاز داریم تا تعدادی سیگنال تعریف و مقداردهی کنیم که مطابق کد زیر تعریف شده اند:

```
63  ---------------------------------------------------------------------------useful signals
64  attribute FSM_ENCODING : string;
65  type State is (Fetch, Decode, Read_RAM, Execute , AC_UPDATE , HLT);
66  signal curr_state : State := Fetch;
67  signal nxt_state  : State := Fetch;
68
69  signal PC_state: std_logic_vector(1 downto 0):= "00";
70  signal PC_out: std_logic_vector(9 downto 0):= "0000000000";
71  signal PC_input: std_logic_vector(9 downto 0):= "0000000000";
72  signal IR: std_logic_vector(15 downto 0);
73  signal WriteEnable: std_logic:='0';
74
75  signal RAM_Address : std_logic_vector (9 downto 0):="0000000000";
76  signal RAM_Input: std_logic_vector(15 downto 0);
77  signal DR: std_logic_vector(15 downto 0);
78
79  signal E: std_logic;
80  signal AC: std_logic_vector(15 downto 0):="0000000000000000";
81  signal Operand: std_logic_vector(15 downto 0);
82  signal Opcode : std_logic_vector(5 downto 0);
83  signal Address : std_logic_vector(9 downto 0);
84
85  signal AC_signal: std_logic_vector(15 downto 0):="0000000000000000";
86  signal E_signal: std_logic:='0';
87  signal Enable_ALU: std_logic:= '0';
```

در قسمت بعدی برای راحت تر و واضح تر شدن استفاده از opcode های دستورات در برنامه، آن ها را به صورت زیر نام گذاری کردیم:

```
93  ---------------------------------------------------------------------------constant opcodes
94
95  constant op_and :std_logic_vector:= "000001";
96  constant op_sta :std_logic_vector:= "000010";
97  constant op_lda :std_logic_vector:= "000011";
98  constant op_add :std_logic_vector:= "000100";
99  constant op_inc :std_logic_vector:= "000101";
100 constant op_cla :std_logic_vector:= "000110";
101 constant op_cle :std_logic_vector:= "000111";
102 constant op_cil :std_logic_vector:= "001000";
103 constant op_cir :std_logic_vector:= "001001";
104 constant op_spa :std_logic_vector:= "001010";
105 constant op_sna :std_logic_vector:= "001011";
106 constant op_sze :std_logic_vector:= "001100";
107 constant op_sza :std_logic_vector:= "001101";
108 constant op_lls :std_logic_vector:= "001110";
109 constant op_lrs :std_logic_vector:= "001111";
110 constant op_mul :std_logic_vector:= "010000";
111 constant op_sqr :std_logic_vector:= "100000";
112
```

در بخش بعدی نیز پورت های ورودی و خروجی ماژول های ROM، RAM، Program Counter و ALU را به سیگنال هایی که تعریف شدند، متصل کردیم:

```
121  ------------------------------------------------------------PORTMAP Modules
122  PC_Module: Program_Counter port map(
123                              clk => clk,
124                              PC_in => PC_input,
125                              PC_opcode => PC_state,
126                              PC_out => PC_out);
127
128  ROM: ROM_Module port map(
129                  addr => PC_out,
130                  data => IR
131              );
132
133  RAM: RAM_Module port map(
134                     clk => clk,
135                     we => WriteEnable,
136                     addr => RAM_Address,
137                     din => RAM_Input,
138                     dout => DR
139                  );
140
141  ALU: ALU_Module port map (
142                  AC_IN => AC,
143                  X_IN => Operand,
144                  ALU_Sel => Opcode,
145                  E_IN => E,
146                  Enable => Enable_ALU,
147                  AC_OUT => AC_signal,
148                  E_OUT => E_signal
149                  );
150
```

در بخش process که قسمت اصلی برنامه است و به clock و reset حساس است، در ابتدا حالت pc را بر روی HLT مقدار دهی می کنیم و سپس بررسی می کند که اگر ریست فعال شده است، استیت کنونی را به حالت Fetch در بیاورد و در غیر این صورت با آمدن لبه بالارونده کلاک به استیت بعدی رفته و تغییر استیت بدهد که در ادامه هر کدام را توضیح می دهیم.

```
145  ------------------------------------------------------------process
146  process(clk,reset)
147     begin
148     PC_state <= "00";
149     if reset = '1' then
150        curr_state <= Fetch;
151     elsif rising_edge(clk) then
152        curr_state <= nxt_state;
153
154     end if;
155
```

در این قسمت حالت های مختلف استیت کنونی را بررسی می کنیم. اگر در حالت Fetch باشد در ابتدا با استفاده از بیت های 10 تا 15 رجیستر IR که از ROM خوانده می شود، OPCODE را مشخص می کند و به استیت بعدی یعنی دیکد کردن می رود.

```
163         case curr_state is
164            when Fetch =>
165                Opcode <= IR(15 downto 10);
166                nxt_state <= Decode;
```

در بخش دیکد دوباره مقدار OPCODE را به دست آورده و اگر یکی از 6 دستور and,store,load,add,multiply,sqr باشد که به آدرس نیاز دارند، به استیت خواندن از رم می رود و در غیر این صورت چون نیازی به خواندن از حافظه ندارد مستقیما به مرحله اجرا می رود و اگر اپکد از دستورات مشخص شده نباشد HLT می کند.

```
169         when Decode =>
170               case Opcode is
171                 when op_and | op_sta | op_lda | op_add | op_mul | op_sqr =>
172                   nxt_state <= Read_RAM;
173                 when op_inc | op_cla | op_cle | op_cil | op_cir | op_spa | op_sna | op_sza | op_sze | op_lls | op_lrs =>
174                   nxt_state <= Execute;
175                 when others =>
176                   nxt_state <= HLT;
177               end case;
```

در بخش Read_RAM در ابتدا مقدار آدرس مربوطه را از IR جدا کرده و از طریق RAM مقدار دیتا را به دستور می آوریم و به استیت بعدی که اجرا است می رویم.

```
184         when Read_RAM =>
185            RAM_Address <= IR(9 downto 0);
186            Operand <= DR ;
187            nxt_state <= Execute;
188
```

در بخش اجرا opcode گرفته شده را بررسی می کنیم که اگر جزو دستوراتی باشد که در ALU قابل محاسبه است مقدار operand را از حافظه گرفته و پایه enable ماژول ALU را فعال کند تا در این ماژول مقدار Ac و E محاسبه گردد و در نهایت به استیت AC_UPDATE می رود. همچنین سایر دستورات نیز به صورت زیر اجرا می شوند.

اگر دستور Store باشد، پایه نوشتن RAM فعال شده، مقدار AC ذخیره شده و به FETCH رفته و یکی به مقدار PC اضافه می کند.

اگر دستور Load باشد، پایه نوشتن RAM غیر فعال شده و مقدار DR در AC ریخته می شود. در نهایت به FETCH رفته و یکی به مقدار PC اضافه می کند.

اگر 4 دستور SPA، SNA،SZE و SZA باشد، پس از بررسی شرط های لازم AC و E، اگر شروط برقرار نبود فقط یک مقدار به PC اضافه می کند، اما اگر شروط موردنظر برقرار بودند، دو عدد به PC اضافه می شود.

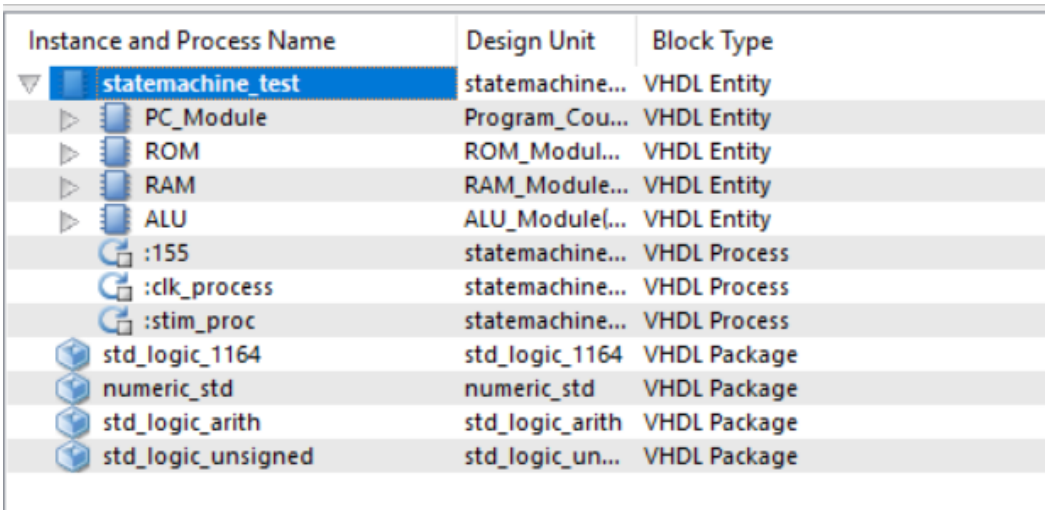
در استیت AC_UPDATE نیز مقادیر به دست آمده از ALU را ذخیره کرده و پایه ENABLE آن را صفر می کند و به PC یک مقدار اضافه کرده و به مرحله FETCH می رود.

```
187         when Execute =>
188           case Opcode is
189               when op_and | op_add | op_inc | op_cla | op_cle | op_cil | op_cir | op_lls | op_lrs | op_mul | op_sqr =>
190                   Operand <= DR ;
191                   Enable_ALU <= '1';
192                   nxt_state <= AC_UPDATE;
193   |
194               when op_sta =>
195                    WriteEnable <= '1';
196                    RAM_Input <= AC;
197                    nxt_state <= Fetch;
198                    PC_state <= "01";
199
200               when op_lda =>
201                    WriteEnable <= '0';
202                    AC <= DR;
203                    nxt_state <= Fetch;
204                    PC_state <= "01";
205
206               when op_spa =>
207                   if AC(15) = '0' then
208                       PC_state <= "10";
209                   else
210                       PC_state <= "01";
211                   end if;
212                   nxt_state <= Fetch;
213
214               when op_sna =>
215                   if AC(15) = '1' then
216                       PC_state <= "10";
217                   else
218                       PC_state <= "01";
219                   end if;
220                   nxt_state <= Fetch;
```

```
221                          -
222                when op_sze =>
223                     if E = '0' then
224                         PC_state <= "10";
225                     else
226                         PC_state <= "01";
227                     end if;
228                     nxt_state <= Fetch;
229
230                when op_sza =>
231                     if AC = "0000000000000000" then
232                        PC_state <= "10";
233                     else
234                        PC_state <= "01";
235                     end if;
236                     nxt_state <= Fetch;
237
238                when others =>
239               end case;
240
241           when AC_UPDATE =>
242
243               AC <= AC_signal;
244               E <= E_signal;
245               Enable_ALU <= '0';
246               PC_state <= "01";
247               nxt_state <= Fetch;
248
249           when others  =>
250           |
251           end case;
252
253   end process;
```

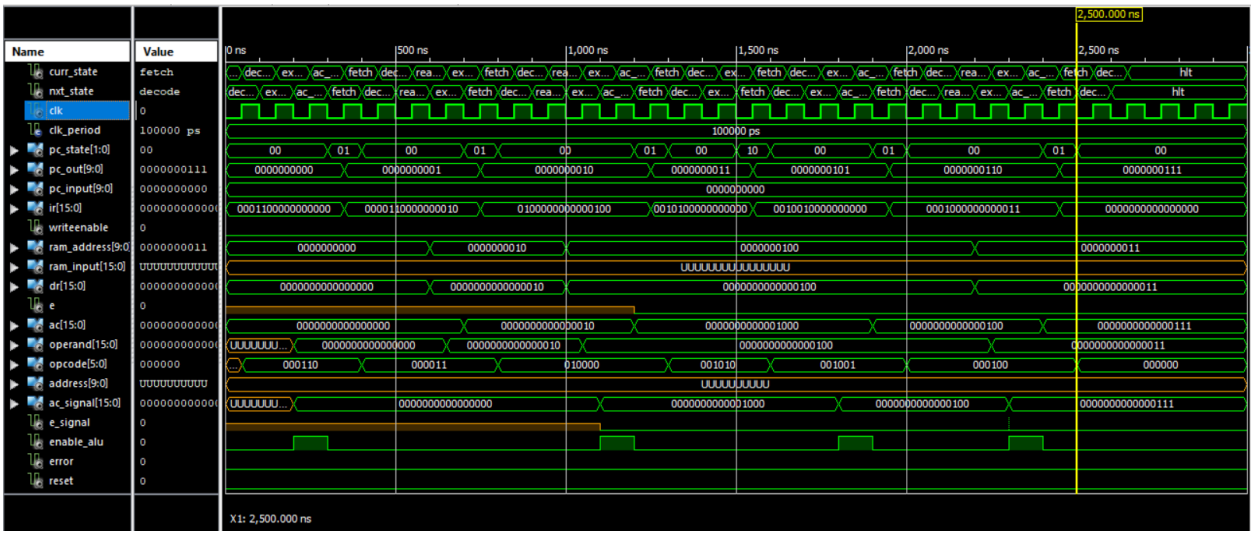
تست پردازنده

در این قسمت یک برنامه test bench به نام StateMachine_test ایجاد کرده که در آن کلاک ساخته می شود. پس از اجرای آن می توان از ستون سمت چپ برنامه تمام سیگنال های مورد نظر و رجیستر های لازم را اضافه کرده و مقادیر آن ها را به صورت زیر بررسی کنیم.



به عنوان نمونه دستورات زیر را به پردازنده اعمال می کنیم:

```
17  architecture Behavioral of ROM_Module is
18      type rom_type is array (0 to addr_width-1) of std_logic_vector(data_width-1 downto 0);
19      constant ROM_Memory : rom_type := (
20                            0 => "0001100000000000",  --Clear AC
21                            1 => "0000110000000010",  --Load
22                            2 => "0100000000000100", --Multiply
23                            3 => "0010100000000000",  --SPA
24                            4 => "0000010000000001",  --AND
25                            5 => "0010010000000000",  --Circular Right Shift
26                            6 => "0001000000000011",  --Add
27                            others => (others => '0'));
28  begin
29      data <= ROM_Memory(to_integer(unsigned(addr)));
30  end Behavioral;
```



همانطور که مشخص است مقدار ac در ابتدا صفر شده و سپس مقدار باینری عدد 2 در آن load می شود. در ادامه مقدار باینری عدد 2 با عدد 4 ضرب شده و مقدار 8 در ac ذخیره می شود. سپس بررسی می کند که اگر مقدار ac مثبت است، به دو خط بعدی برود ه مقدار ac را شیفت چرخشی به سمت راست می دهد و در نهایت با عدد 3 جمع می کند که مقدار نهایی برابر 7 خواهد شد.

به همین ترتیب می توان دستورات متنوع دیگری را نیز با این پردازنده بررسی و تست کرد.

منابع و مراجع

[1] cpu چیست و چه کاربردی دارد | انواع سی پی یو و اجزای آن | تاریخچه cpu (aryatehran.com)

[2] 11. Design examples — FPGA designs with VHDL documentation (vhdlguide.readthedocs.io)

[3] 11. Design examples — FPGA designs with VHDL documentation (vhdlguide.readthedocs.io)

[4] Designing a CPU in VHDL, Part 6: Program Counter, Instruction Fetch, Branching - Domipheus Labs

[5]https://web.engr.oregonstate.edu/~traylor/ece474/vhdl_lectures/essential_vhdl_pdfs/essential_vhdl107-127.pdf

قابل ذکر است در ابتدا کلیت بخش های RAM و ROM توسط محمدمسیح شالچیان و بخش های Program Counter و ALU توسط سمیرا سلجوقی نوشته شده و سایر موارد اعم از بخش اصلی کد و گزارشکار و بررسی کلی کد ها توسط هردو نفر همزمان انجام شده است.