دانشگاه صنعتی امیرکبیر
(پلی‌تکنیک تهران)
دانشکده برق

عنوان
گزارش پروژه معماری کامپیوتر و ریزپردازنده

نگارش
پارسا محمّدی – ۹۹۲۳۱۲۱
آتنا شیرچرندابی – ۹۹۲۳۰۴۳

استاد درس
دکتر شریعتمدار مرتضوی

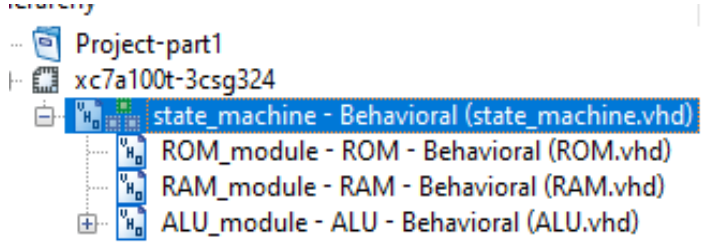
خرداد ۱۴۰۲

فهرست

مقدمه و توضیحات

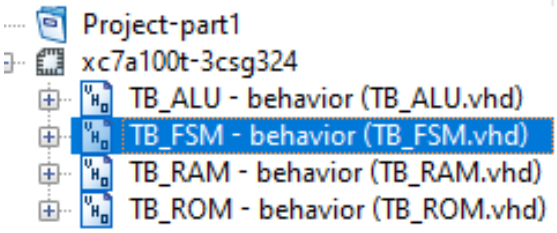
در این پروژه طبق خواست دستور پروژه یک ماژول RAM، یک ماژول ROM طراحی شده است و در FSM استفاده شده است. FSM اختصار Finite State Machine می‌باشد. طبق دستور پروژه ما از این روش برای شبیه سازی روند یک پردازنده ساده استفاده می‌کنیم. در صفحه کد های VHDL نرم افزار ISE چهار فایل به صورت کلی موجود می‌باشد.

- فایل RAM: در این فایل حافظه RAM شبیه سازی شده است.
- فایل ROM: در این فایل حافظه ROM شبیه سازی شده است.
- فایل ALU: در این فایل قطعه ALU شبیه سازی شده است که بعدا در FSM برای شبیه سازی عملیات های حسابی، منطقی و شیفت استفاده می‌شود.
- فایل STATE_MACHINE: در این فایل یک پردازنده ساده طبق خواست صورت پروژه طراحی شده شده و این فایل TOP MODULE پروژه می‌باشد.



تصویر ۱ – این تصویر فایل های موجود در پروژه را نمایش می‌دهد

همچنین در قسمت SIMULATION نرم افزار ISE نیز چهار فایل موجود می‌باشد. هر کدام از آنها برای تست و بررسی عملکرد فایل های طراحی شده می‌باشد.



تصویر ۲ – این تصویر فایل های موجود در قسمت شبیه سازی پروژه را نمایش می‌دهد

در ادامه توضیحات مربوط به هرکدام از فایل ها به تفصیل بیان خواهد شد. با توجه به خواست دستور پروژه نقش هر کدام از اعضای گروه در تهیه این پروژه به صورت زیر می‌باشد.

- RAM : پارسا محمّدی
- ROM : آتنا شیرچرندابی
- ALU : آتنا شیرچرندابی
- FSM : پارسا محمّدی
- فایل های تست بنچ و شبیه سازی : پارسا محمّدی

توجه شود تمام عملگرهای پیاده سازی شده تست شده و همه آنها به درستی کار می‌کنند این پروژه هیچ اروری ندارد ولی تعداد وارنینگ دارد که فقط در مورد اپیتمایز کردن کد است.

FINITE STATE MACHINE

در این بخش به بررسی کد های فایل state_machine می پردازیم.

```
1  ----------------------------------------------------
2  --              State Machine - Main file
3  ----------------------------------------------------
4  -- Parsa Mohammadi       9923121
5  -- Atena Shircharandabi  9923043
6  ----------------------------------------------------
7  library IEEE;
8  use IEEE.STD_LOGIC_1164.ALL;
9  use IEEE.NUMERIC_STD.ALL;
10 use IEEE.STD_LOGIC_UNSIGNED.ALL;
11
12 entity state_machine is
13     port (
14         clk : in std_logic
15     );
16 end state_machine;
17
```

ابتدا کتابخانه های مورد نیاز به این بخش اضافه شده اند.

تنها ورودی سیستم کلاک است که در محیط سیمولیشن به ماژول اعمال می‌شود. باید توجه داشت که این کامپوتر بر روی RAM کار انجام می‌دهد و آن را تغییر می‌دهد سیستم خروجی ندارد و تغییرات بر روی رم مشخص است و بعد از انجام عملیات های مختلف می‌توان مقادیر موجود در رم را تغییر داد. باید توجه داشت که رم یک حافظه قابل تغییر است و هم می‌توان بر روی آن نوشت و هم می‌توان از روی آن خواند.

استیت ها

```
18 architecture Behavioral of state_machine is
19
20     -- Define your states
21     type State is (receiving_command, decode_command, reading_value, prosses);
22     signal currentState, nextState : State;
23
24     -- Defining address size and data size
25     constant address_size : integer := 10;
26     constant data_size : integer := 16;
27
28     -- Program Counter
29     signal PC : std_logic_vector(address_size-1 downto 0) := (others => '0');
30
31     -- Instruction Register
32     signal IR : std_logic_vector(data_size-1 downto 0) := (others => '0');
33
34    -- Address Register
35    signal AR : std_logic_vector(address_size-1 downto 0) := std_logic_vector(to_unsigned(0, address_size));
36
37    -- Data register
38    signal DR : std_logic_vector(data_size-1 downto 0) := std_logic_vector(to_unsigned(0, data_size));
39
40    -- Acumalator
41    signal AC : std_logic_vector(data_size-1 downto 0) := std_logic_vector(to_unsigned(3, data_size));
42    signal E : std_logic := '0';
43
```

تصویر بالا ادامه کد را نمایش می دهد که در قسمت architecture کد موارد مورد نیاز تعریف شده اند. ابتدا برای ساخت state machine و ایجاد استیت های مختلف از سنتکس خط ۲۱ و ۲۲ استفاده شده است. طبق

دستور پروژه چهار استیت تعریف شده است که اولی استیت receiving_command می‌باشد؛ این استیت برای دریافت دستوارت از ROM می‌باشد. توجه شود که در این پروژه دستوارت از ROM خوانده شده و در محاسبات انجام می‌شود و در نهایت در اگر نیاز باشد می‌توان مقادیری در RAM ذخیره و یا دریافت شود.

استیت بعدی DECODE می‌باشد. این استیت برای فهم و درک این است که پردازنده باید چه عملی را در مراحل بعد انجام دهد.

استیت reading_value هم برای بعضی از دستورات است که نیاز به خواندن از حافظه دارند. در این استیت مقدار رم را به DR انتقال می‌دهد تا بعدا در استیت بعدی مورد استفاده قرار بگیرند.

استیت prosses هم برای اجرای دستورات است.

در ادامه دو ثابت address_size و data_size تعریف شده است در این دو ثابت اندازه بیت های رجیستر های حافظه و دیتا ذخیره شده اند. در ادامه برنامه بار ها نیاز هست که از این دو عدد استفاده شود. این گونه تعریف کردن باعث راحتی کار می‌شود.

در خط های ۲۹ تا ۴۲ رجیستر های مورد نیاز برای این پروژه تعریف شده اند. همه این این رجیستر های به صورت سیگنال تعریف شده اند. این سیگنال ها به صورت برداری از بیت ها تعریف شده اند که اندازه هر کدام با توجه به نوع داده‌ای که ذخیره می‌کنند متفاوت است.

این رجیستر ها دو صورت مقدار دهی اولیه شده اند. هردو این روش های یکسان اند و تفاوتی ندارند هردو این روش ها تمام بیت ها را صفر می‌کنند. روش اول استفاده از others است که تمام بیت های موجود در بردار را برابر صفر قرار می‌دهد. روش دوم استفاده از دستور

std_logic_vector(to_unsiged(integer_number , number_of_bits) می باشد. در این روش با وارد کردن یک عدد ده هی در قسمت integer_number و وارد کردن تعداد بیت هایی که می خواهیم آن عدد باید در قسمت number_of_bits یک بردارد از بیت ها به باینری به اندازه که خواسته ایم ایجاد می‌کند. برای مثال برای رجیستر IR یک صفر ۱۶ بیتی اختصاص داده می‌شود.

```
44      -- Additional Variables
45      signal RAM_on, ROM_on, ALU_on : std_logic := '1';
46      signal ram_read_on            : std_logic;         -- 1 is REAR and 0 is WRITE
47      signal RAM_IN                 : std_logic_vector(data_size-1 downto 0) := (others => '0');
48      signal ROM_OUT                : std_logic_vector(data_size-1 downto 0) := (others => '0');
49      signal ALU_out                : std_logic_vector(data_size-1 downto 0) := (others => '0');
50      signal ALU_carry_out          : std_logic := '0';
51
52  -- Inserting ROM
53  component ROM is
54  port(
55        address  :  in std_logic_vector(9 downto 0);
56        clk      :  in std_logic;
57        enb      :  in std_logic;
58        data_out :  out std_logic_vector(15 downto 0)
59  );
60  end component;
61
62  -- Inserting RAM
63  component RAM is
64  port(
65        address     : in std_logic_vector(9 downto 0);
66        data_in     : in std_logic_vector(15 downto 0);
67        read_write  : in std_logic; -- 0 for write and 1 for read
68        clk         : in std_logic;
69        enb         : in std_logic;
70        data_out    : out std_logic_vector(15 downto 0)
71  );
72  end component;
```

در بخش additional variables هم تعداد متغییر متناسب با نیاز پروژه تعریف شده است. متغییر های Ram_on, Rom_on,ALU_on برای مشخص کردن روشن و یا خاموش بودن این قطعات می‌باشد. اگه مقدار آن ۱ باشد قطعه روشن و اگر صفر باشد قطعه خاموش است. متغییر ram_read_on نیز برای مشخص کردن مود کاری رم می‌باشد. اگر مقدار آن ۱ باشد یعنی رم مود خواند می باشد و می‌توان اطلاعات از آن دریافت کرد و اگر در حالت ۰ باشد یعنی در مود نوشتن است و اطلاعات می‌تواند در آن بارگذاری شود. بقیه متغییر های نیز برای ارتباط با قطعه و دریافت ورودی و خروجی می‌باشند.

ماژول RAM و ROM

```
75  -- Inserting ALU
76  component ALU is
77  port(
78      input1,input2  : in  STD_LOGIC_VECTOR(15 downto 0);  --AC & ADRESS
79      carry_in : in std_logic; -- E
80      input3  : in  STD_LOGIC_VECTOR(5 downto 0);  --UPCODE (for selecting function)
81      clk : in std_logic; --clk
82      enb : in std_logic; --ALU_on
83      output  : inout  STD_LOGIC_VECTOR(15 downto 0); --ALU_out
84      carry : out std_logic := '0' --ALU_carry_out
85  );
86  end component;
87
88  begin
89
90  ROM_module : ROM
91              port map(address => AR,
92                       clk => clk,
93                       enb => ROM_on,
94                       data_out => ROM_OUT
95                       );
96
97  RAM_module : RAM
98              port map(address => AR,
99                       data_in => RAM_IN,
100                      read_write => ram_read_on,
101                      clk => clk,
102                      enb => RAM_on,
103                      data_out => DR
104                      );
```

در ادامه ماژول های ROM و RAM هم اضافه شده اند. این ماژول ها با دستور component اضافه شده اند و عملکرد رم و رام را شبیه سازی می‌کنند. توضیحات مربوط به جزئیات هر کدام نیز در ادامه بیان خواهد شد. و بعد ماژول ALU اضافه شده است. ورودی های و خروجی های هر ماژول اضافه شده با دستور port map مشخص شده است.

ماژول رام ورودی AR را میگیرد که همان آدرس رجیستر است. این ورودی مشخص می‌کند که کدام خانه رام لود شود. ورودی دیگر آن کلاک است و ورودی بعدی انیبل است که روشن و خاموش بودن قطعه را مشخص می‌کند. خروجی در سیگنال rom_out ریخته می‌شود و بعدا در IR رجیستر ریخته خواهد شد. برای درک بهتر از نحویه کار کرد کد ما اول خروجی رام را در rom_out ریخه و سپس در IR می‌ریزیم.

رم هم مانند رام است با این تفاوت دو ورودی بیشتر دارد یک ورودی برای مشخص کردن مود فعالیت رم و دیگری داده ورودی که قرار است در رم ذخیره شود. خروجی رم میتواند مانند رام با واسته به داخل DR ریخته شود و بی واسته که تصمیم گروه بر این شد که بی واسته ریخته شود.

ماژول ALU

```
105  ALU_module : ALU
106              port map(input1 => AC,
107                       input2 => DR,
108                       input3 => IR(15 downto 10),
109                       carry_in => E,
110                       clk => clk,
111                       enb => ALU_on,
112                       carry => ALU_carry_out,
113                       output => ALU_out
114                     );
```

یکی از ورودی های این ماژول AC و دیگری DR می باشـــد. این ماژول با دریافت OP CODE عملیات حسابی منطقی و یا شیفت را انجام می دهد و خروجی می دهد. این ماژول مانند بقیه ماژول ها ورودی فعال کردن و کلاک نیز دارد. با توجه به اینکه بعضـــی از اعملیات های مانند جمع نیاز به بیت کری خروجی دارند خروجی ALU_carry_out برای این منظور در نظر گرفته شـــده اســـت. در ادامه برنامه این خروجی به فلیپ فلاپ E ریخته خواهد شد. همنچنین کری ورودی هم از E دریافت می‌شود.

پراسس ها

```
117      process (clk)
118      begin
119          if rising_edge(clk) then
120                  currentState <= nextState;
121          end if;
122      end process;
123
124      process (currentState)
125        begin
126
127          case currentState is
128
129              when receiving_command =>
130                    ROM_on <= '1';
131                    RAM_on <= '1';
132                    ALU_on <= '1';
133                    ram_read_on <= '1'; -- Read mood
134                    IR     <= ROM_OUT;
135                    nextState <= decode_command;
136
137              when decode_command =>
138
```

در این پروژه دو پراســس تعریف شـــده اســـت که یکی وابســـته به کلاک و دیگر وابســـته به تغییرات اســـتیت است. با ترکیب این دو پراسس ما می‌توانیم یک فرایند هماهنگ با کلاک و همچین حســاس به تغییر اســتیت را طراحی کنیم.

پراسس کلاک زمانی انجام می‌شود که در لبه بالا رونده کلاک قرار داشـــته باشـــد. زیرا یک شـــرط برای این امر قرار داده شده است. در صورت اجرای این شرط استیت تغییر می کند و با تغییر اســتیت پروســس دوم آغاز می شود. پروسس دو از دستور CASE تشکیل شده است که بررسی می‌کند برنامه در کدام استیت می‌باشد. ترکیب این دو پروسس این امکان را فراهم می‌کند در هر لبه بالا رونده کلاک فقط یکی از استیت ها اجرا شود.

جزئیات استیت ها:

نکته خیلی مهم در فرآیند های اســتیتی این است که هر اســتیت در یک کلاک انجام می‌شود. یعنی تمام فرآیند های موجود در هر اســتیت زمانی انجام می‌شود اســتیت تمام شود به اســتیت بعدی برویم. زمانی که اســتیت تغییر

می‌کند تمام فرایند ها به صورت موازی اجرا می‌شود و نمی‌توان فرآیند های ترتیبی در یک استیت داشت. این امر دلیل این است که مقدار دهی بعضی از رجیستر های مانند در استیت های دیکود و خواندن از حافظه انجام می‌شود.

اولین استیت دریافت دستورات بود. در این استیت ماژول های RAM, ROM, ALU فعال می‌شوند چرا که ما به این ماژول ها در ادامه کار برنامه نیاز داریم.

```
when receiving_command =>
      ROM_on <= '1';
      RAM_on <= '1';
      ALU_on <= '1';
      IR     <= ROM_OUT;
      nextState <= decode_command;
```

مقدار خروجی رام در IR ریخته می‌شود. این مقدار حاوی دستور و آدرس مورد نیاز برای ادامه برنامه است. در ادامه بعد انجام این عملیات ها به استیت بعدی یعنی دیکود کردن می‌رویم.

```
136         when decode_command =>
137
138               -- ADDRESS NEEDED COMMANDS
139               if (IR(15 downto 10) = "000100" OR IR(15 downto 10) = "000001" OR IR(15 downto 10) = "000011" OR
140                   IR(15 downto 10) = "010000" OR IR(15 downto 10) = "100000") then -- ADD & AND & LOAD & MLTP & SQR
141                     AR <= IR(9 downto 0);
142                     ALU_on <= '1';
143                     ram_read_on <= '1'; -- Read mood
144                     nextState <= reading_value;
145
146               -- Store also is address needed command but beacuse it requires different approch we have difined it separatly
147               elsif (IR(15 downto 10) = "000010") then -- Store
148                     AR <= IR(9 downto 0);
149                     ROM_on <= '1';
150                     RAM_on <= '1';
151                     PC     <= PC + 1 ;  -- For reading next line in rom
152                     ram_read_on <= '0'; -- write mood
153                     RAM_IN <= AC;
154
155                     nextState <= prosses;
```

```
157                 -- ADDRESS NOT NEEDED COMMANDS
158                 --                      INC                             CLEAR AC                        CLEAR E
159                 elsif(IR(15 downto 10) = "000101" OR IR(15 downto 10) = "000110" OR IR(15 downto 10) = "000111" OR
160                 --          Circular Left Shift         Circular Right Shift                SPA
161                       IR(15 downto 10) = "001000" OR IR(15 downto 10) = "001001" OR IR(15 downto 10) = "001010" OR
162                 --          SNA                         SZE                                 SZA
163                       IR(15 downto 10) = "001011" OR IR(15 downto 10) = "001100" OR IR(15 downto 10) = "001101" OR
164                 --          linear Left Shift           Linear Right Shift
165                       IR(15 downto 10) = "001110" OR IR(15 downto 10) = "001111") then
166                       ALU_on <= '1';
167                       PC     <= PC + 1 ;  -- For reading next line in rom
168                       nextState <= prosses;
169
170
171                 else
172                    nextState <= receiving_command;
173                 end if;
```

در دیکود کردن دستورات دو نوع هستند. یا به حافظه نیاز دارند یه ندارد. در جدولی که در دستور پروژه ارائه شده است مشخص شده است که کدام از دستورات به حافظه نیاز دارند و کدام ندارند. آنهای که به حافظه نیاز دارند و باید مقداری را از روی حافظه بخوانند باید به استیت سوم یعنی خواندن از حافظه بروند. ولی آنهایی که نیاز ندارند مستقیما به استیت پروسس می‌روند.

در دستوارتی که به رم نیاز دارند آدرس مورد نیاز به آدرس رجیستر ریخته می‌شود تا بعدا از رم فراخوانی شوند. و رم در حالت خواندن قرار می‌گیرد و تا در استیت بعدی اطلاعات از رم خوانده شود. اما دستور STORE استثناء است زیرا این دستور با اینکه آدرس حافظه را می‌خواهد ولی می‌خواهد در آن بنویسد به همین دلیل رم را در حالت نوشتن قرار می‌دهد. با رسیدن لبه بالا رونده و رسیدن به استیت بعدی رم آماده نوشتن در آدرس موجود در AR می باشد.

زمانی دستور خوانده شده جزو هیچ گروهی نیست به قسمت else می ورود تا دستور بعدی خوانده شود. توجه شود دستوراتی مستقیما به پروسس می‌روند در این مرحله است که یکی به پروگرم کانتر اضافه می‌کنند.

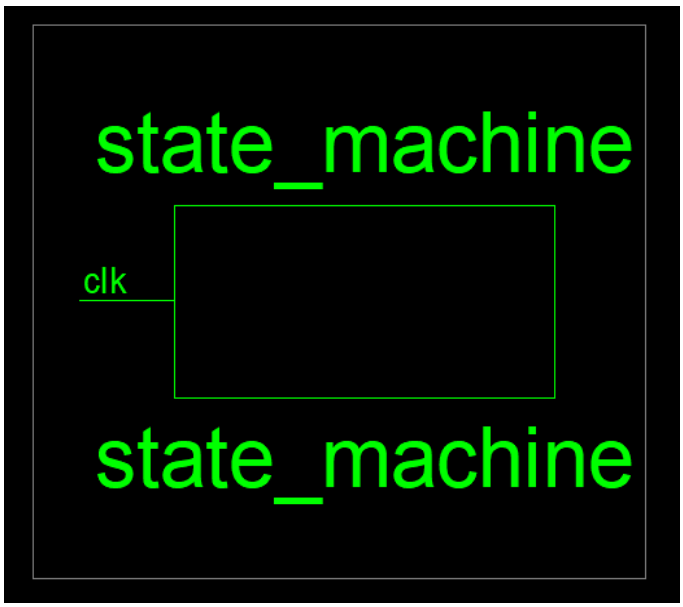
استیت لود کردن از حافظه

در این استیت اطاعات با توجه به اینکه AR مقدار دهی شده است در DR با گذاری می‌شود.

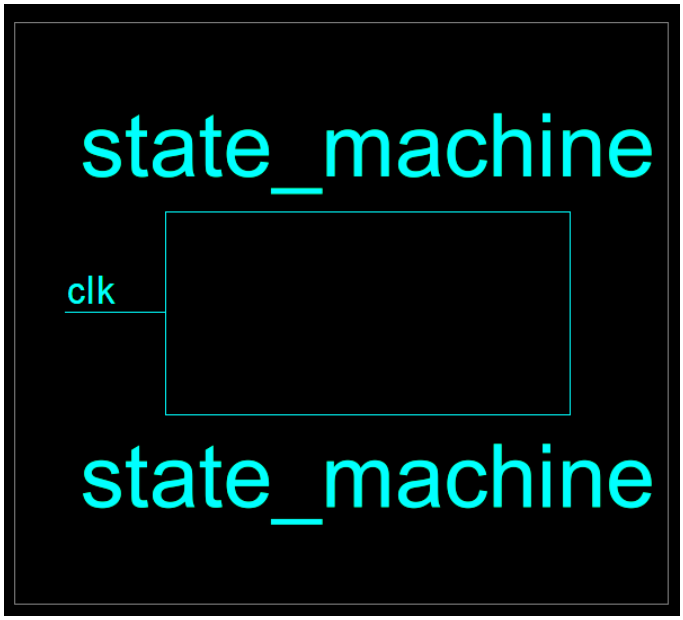
```
when reading_value =>
      ALU_on <= '1';
      RAM_on <= '1';       -- RAM is on
      ram_read_on <= '1'; -- saving to DR in reed mood
      PC     <= PC + 1 ;  -- For reading next line in rom

      nextState <= prosses;
```

```
183             when prosses =>
184                   if (IR(15 downto 10) = "000100" OR IR(15 downto 10) = "000001" OR IR(15 downto 10) = "000101" OR
185                       IR(15 downto 10) = "001000" OR IR(15 downto 10) = "001001" OR IR(15 downto 10) = "001110" OR
186                       IR(15 downto 10) = "001111" OR IR(15 downto 10) = "010000" OR IR(15 downto 10) = "100000") then -- ALU Commands
187                         AC <= ALU_out;
188                         E  <= ALU_carry_out;
189                         nextState <= receiving_command;
190
191                   elsif (IR(15 downto 10) = "000010") then -- store
192                      ram_read_on <= '1'; -- trun to read mood. write of data has been done when states changes decode to process.
193                      nextState <= receiving_command;
194
195                   elsif (IR(15 downto 10) = "000011") then -- LOAD
196                      AC <= DR;           -- LOADING DATA FROM DATA REGISTER
197                      nextState <= receiving_command;
198
199                   elsif (IR(15 downto 10) = "000110") then -- CLEAR AC
200                      AC <= std_logic_vector(to_unsigned(0,data_size));
201                      nextState <= receiving_command;
202
203                   elsif (IR(15 downto 10) = "000111") then -- CLEAR E
204                      E <= '0';
205                      nextState <= receiving_command;
206
207                   elsif (IR(15 downto 10) = "001010") then -- SKIP IF AC IS POSITIVE
208                      if (AC(data_size-1) = '0') then  -- AC is positive
209                         PC <= PC + 1; --
210                      end if;
211                      nextState <= receiving_command;
212
212
213                     elsif (IR(15 downto 10) = "001011") then -- SKIP IF AC IS Negetive
214                        if (AC(data_size-1) = '1') then  -- AC is Negetive
215                           PC <= PC + 1; --
216                        end if;
217                        nextState <= receiving_command;
218
219                     elsif (IR(15 downto 10) = "001100") then -- SKIP IF E IS zero
220                        if (E = '0') then  -- E is zero
221                           PC <= PC + 1; --
222                        end if;
223                        nextState <= receiving_command;
224
225                     elsif (IR(15 downto 10) = "001101") then -- SKIP IF AC IS zero
226                        if (AC = std_logic_vector(to_unsigned(0,data_size))) then  -- AC is zero
227                           PC <= PC + 1; --
228                        end if;
229                        nextState <= receiving_command;
230
231                     else
232                        nextState <= receiving_command;
233                     end if;
234                     AR     <= PC; -- Giving address to AR resgister for loading from ROM
235
236
237             end case;
238         end process;
239
240     end Behavioral;
```

دستورات محاسباتی، منطقی، شیفت و یکی اضافه کردن همگی توسط ماژول ALU انجام می‌شود به همین منظور تمام این دستورات در یک IF قرار گرفته‌اند و خروجی تمام آنها به AC و E وارد می‌شود. دستورات رجیستری هم همانگونه که در صورت پروژه توضیح داده شد پیاده سازی شده اند. در نهایت تمام این دستورات باید به استیت اول بازگردند به همین دلیل در پایان هر دستور این امر نوشته شده است.



تصویر ۳ – Technology مربوط به STATE MACHINE

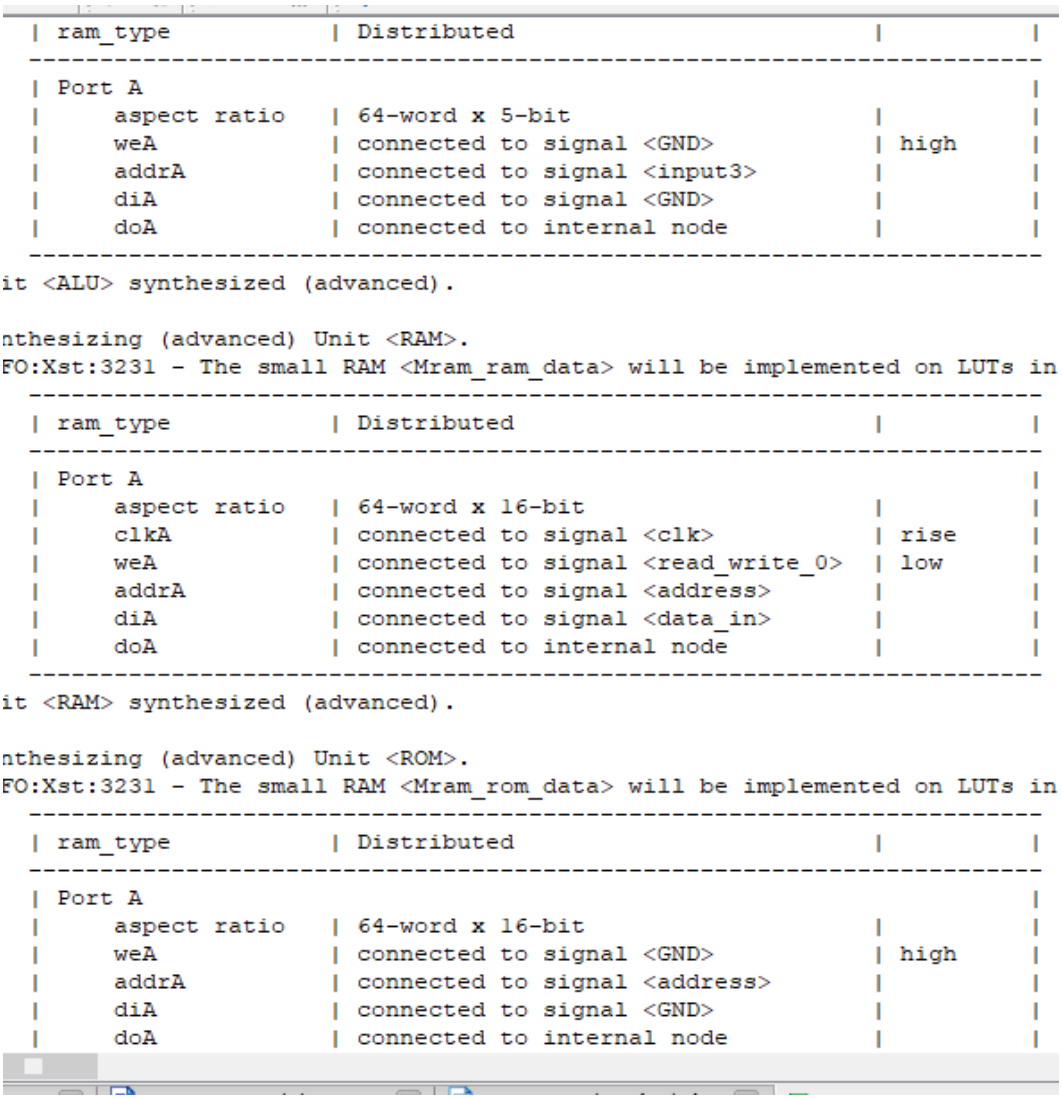


تصویر 4 – RTL مربوط به STATE MACHINE

اطلاعات مصرف:

```
*                            Design Summary
================================================================================

Top Level Output File Name         : state_machine.ngc

Primitive and Black Box Usage:
------------------------------

Device utilization summary:
---------------------------

Selected Device : 7a100tcsg324-3


Slice Logic Utilization:

Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:      0
   Number with an unused Flip Flop:       0  out of      0
   Number with an unused LUT:             0  out of      0
   Number of fully used LUT-FF pairs:     0  out of      0
   Number of unique control sets:         0

IO Utilization:
 Number of IOs:                           1
 Number of bonded IOBs:                   0  out of    210     0%

Specific Feature Utilization:
```

اطلاعات مربوط به رم:



```
| ram_type           | Distributed                          |           |
-----------------------------------------------------------------------
| Port A                                                    |           |
|     aspect ratio   | 64-word x 5-bit                      |           |
|     weA            | connected to signal <GND>            | high      |
|     addrA          | connected to signal <input3>         |           |
|     diA            | connected to signal <GND>            |           |
|     doA            | connected to internal node           |           |
-----------------------------------------------------------------------
it <ALU> synthesized (advanced).

nthesizing (advanced) Unit <RAM>.
FO:Xst:3231 - The small RAM <Mram_ram_data> will be implemented on LUTs in
-----------------------------------------------------------------------
| ram_type           | Distributed                          |           |
-----------------------------------------------------------------------
| Port A                                                    |           |
|     aspect ratio   | 64-word x 16-bit                     |           |
|     clkA           | connected to signal <clk>            | rise      |
|     weA            | connected to signal <read_write_0>   | low       |
|     addrA          | connected to signal <address>        |           |
|     diA            | connected to signal <data_in>        |           |
|     doA            | connected to internal node           |           |
-----------------------------------------------------------------------
it <RAM> synthesized (advanced).

nthesizing (advanced) Unit <ROM>.
FO:Xst:3231 - The small RAM <Mram_rom_data> will be implemented on LUTs in
-----------------------------------------------------------------------
| ram_type           | Distributed                          |           |
-----------------------------------------------------------------------
| Port A                                                    |           |
|     aspect ratio   | 64-word x 16-bit                     |           |
|     weA            | connected to signal <GND>            | high      |
|     addrA          | connected to signal <address>        |           |
|     diA            | connected to signal <GND>            |           |
|     doA            | connected to internal node           |           |
```

کد ماژول ROM

این ماژول دارای سه ورودی clk،address و enb است که در ورودی address ورودی AR ریخته میشود که همان آدرس رجیستر میباشد. این ورودی مشخص میکند که کدام خانه ی رام لود میشود. ورودی enb مربوط به روشن یا خاموش بودن قطعه میباشد که اگر enb = ‘1’ باشد قطعه ی ما روشن است و خانه ی مورد نظر رام لود میشود و در غیر این صورت قطعه خاموش است. ورودی CLK نیز مربوط به کلاک ورودی میباشد.

```
12  entity ROM is
13     port(
14     address  :  in std_logic_vector(9 downto 0);
15     clk      :  in std_logic;
16     enb    :  in std_logic;
17     data_out :  out std_logic_vector(15 downto 0)
18  );
19  end ROM;
```

که ورودی های CLK و enb تک بیتی و ورودی آدرس ۱۰ بیتی می‌باشد. در این ماژول همچنین یک خروجی ۱۶ بیتی تحت عنوان data_out داریم که این خروجی بعد ها از طریق یک سیگنال در IR رجیستر ریخته خواهد شد (درکد استیت ماشین).

```
23  type rom is array (0 to 63) of std_logic_vector (15 downto 0);
24
25  signal rom_data: rom :=( -- Initialize ROM
26  --  AND AC WITH RAM(0)  STORE AC IN RAM(2) ADD AC WITH RAM (0) STORE AC IN RAM(3)  LOAD RAM(1) TO AC   STORE AC IN RAM(4)   INCREAMENT AC      STORE AC IN RAM(5)
27     b"0000010000000000",b"0000100000000010",b"0001000000000000",b"0000100000000011",b"0000110000000001",b"0000100000000100",b"0001010000000000",b"0000100000000101",
28  --  CLEAR AC           CLEAR E
29     b"0001100000000000",b"0001110000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",
30     b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",
31     b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",
32     b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",
33     b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",
34     b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",
35     b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000"
36     );
```

در خط ۲۳ آرایه دو بعدی تعریف کردیم که طول هر کلمه ۱۶ بیت و تعداد کلمات ۶۴ تا است و هر کلمه از وکتور های ۱۶ بیتی تشکیل شده اند.

در خط ۲۵ تا ۳۶ rom مقدار دهی اولیه شده است. رام به این صورت مقدار دهی شده است که مقدار همه خانه های حافظه مشخص باشند وگرنه روش های دیگری نیز برای مقدار دهی اولیه وجود دارد. چون دستوارت در رام

نوشته می‌شوند و بعد یکی یکی اجرا می‌شوند برای بررسی درستی عملکرد کد تعدادی از دستورات در رام نوشته شده اند که در بخش اجرای تست بنچ به آنها اشاره خواهد شد.

```
39  begin
40
41     process(clk)
42     begin
43        if(rising_edge(clk)) then
44           if ( enb = '1') then -- if ROM is on
45              data_out <= rom_data(conv_integer(address));
46           else
47              data_out <= std_logic_vector(to_unsigned(0,data_out'length));
48           end if;
49        end if;
50     end process;
51
52  end Behavioral;
53
```

همانطور که انتظار میرود این ماژول حساس به لبه ی بالا رونده ی کلاک می‌باشد دستور rising_edge(clk) باعث می‌شود فقط زمانی که کلاک از صفر به یک می‌رود کد اجرا شود. همچنین در ادامه تحت تعریف دستور if بیان میکنیم که در هر لبه ی بالا رونده ی کلاک در صورت ’1‘ بودن ورودی enb (روشن بودن قطعه)، مقدار ذخیره شده در سیگنال rom_data، در خروجی data_out ریخته شود تا به IR رجیستر منتقل شود. اما در غیر این صورت، یعنی یک نبودن ورودی enb، خروجی صفر در درون data_out ریخته میشود.

توضیح دستورات:

دستور conv_integer(binry_number) این دستور یک عدد باینری را به یک عدد اینتیجر تبدیل می‌کند. باتوجه به اینکه ما رام را به صورت یک آرایه دو بعدی تعریف کردیم پس باید برای دست رسی به المان های درون آرایه و indexing ما باید به آرایه شماره المانی که می‌خواهیم بدهیم تا به ما آن المان را خروجی دهد. به همین دلیل ما عدد باینری آدرس را به اینتیجر تبدیل کردیم تا بتوانیم به المان های درون آریه دست رسی داشته باشیم.

دستور std_logic_vector(to_unsigned(0, data_out’length))

این درستور از چند بخش تشکیل شده است. ابتدا data_out’length را بیان می‌کنیم. هر وکتوری که قبل از کوتیشن قرار بگیرد طول محاسبه می‌شود. این دستور طول داده ورودی را محاسبه می‌کند و به صورت یک عدد دسیمال خروجی می‌دهد.

دستور to_unsigend(number_in_decimal, number_of_bits) این دستور دو ورودی دریافت می‌کند و خروجی آبجکت عدد باینری می‌دهد. ورودی اول عدد دسیمالی از که می‌خواهیم ایجاد شود و دومی تعداد بیت هایی است که می‌خواهیم آن عدد داشته باشد.

در نهایت دستور std_logic_vector است که آبجکت ایجاد شده را به یک وکتوری از اعداد باینری تبدیل می‌کند.

کد ماژول RAM

رم نیز همانند رام ورودی های address، clk و enb را داراست اما علاوه بر این ها دو ورودی دیگر نیز دارد. یک ورودی تک بیتی برای مشخص کردن مود فعالیت رم (read_write). که اگر مقدار این ورودی ۱ باشد قطعه در حالت read و در صورت صفر بودن در حالت write قرار میگیرد. (میدانیم که رم یک حافظه ی تغییر پذیر است و علاوه بر خواندن اطلاعات از روی آن میتوان داده های جدید را در آن قرار داد). و دیگری داده ورودی ۱۶ بیتی که قرار است در رم ذخیره شود (data_in) .

```
12  entity RAM is
13     port(
14     address     : in std_logic_vector(9 downto 0);
15     data_in     : in std_logic_vector(15 downto 0);
16     read_write  : in std_logic; -- 0 for write and 1 for read
17     clk         : in std_logic;
18     enb         : in std_logic := '1';
19     data_out    : out std_logic_vector(15 downto 0)
20  );
21  end RAM;
```

در این قطعه نیز همانند رام تنها یک خروجی ۱۶ بیتی تحت عنوان (data_out) داریم که بعد ها درون DR ریخته میشود.

```
23  architecture Behavioral of RAM is
24
25  type ram is array (0 to 63) of std_logic_vector (15 downto 0);
26
27  signal ram_data: ram :=( -- Initialize RAM
28     b"0000000000000001",b"0000000000000100",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",
29     b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",
30     b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",
31     b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",
32     b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",
33     b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",
34     b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",
35     b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000",b"0000000000000000"
36     );
```

این بخش هم مانند قسمت رام ساخته و مقدار دهی اولیه شده است. اما تمام خانه های حافظه به غیر از اولی و دومی صفراند. مقادیر موجود در دو خانه اول برای این است که ما آنها در بخش تست استفاده می‌کنیم و با اجرای دستورات مقدار بقیه خانه های رم تغییر می‌کند.

همانند ماژول قبلی این ماژول نیز حساس به لبه ی بالا رونده ی کلاک میباشد که این موضوع مجددا به عنوان process در کد تعریف شده است و در ادامه تحت تعریف دستور if مشخص می‌کنیم که در هر لبه بالا رونده کلاک در صورت '1' بودن ورودی enb (روشن بودن قطعه) میتوان عملیات خواندن و نوشتن را روی این قطعه پیاده کرد. که این خود نیازمند حلقه ی if دیگریست پس مجددا با تعریف یک حلقه جدید مشخص می‌کنیم که در صورت صفر بودن ورودی read_write همانطور که قبلا گفته شد قطعه در حالت write خود قرار دارد پس مقدار ورودی data_in در سیگنال ram_data ریخته و ذخیره میشود.

واضح است که در غیر این صورت نیز (read_write = '1') حالت read قطعه فعال میشود و آدرس ذخیره شده

```
38  begin
39
40     process(clk)
41     begin
42        if(rising_edge(clk)) then
43           if (enb = '1') then -- if ram is on
44              if ( read_write = '0') then -- write mode
45                 ram_data(conv_integer(address)) <= data_in ;
46              else -- read mode
47                 data_out <= ram_data(conv_integer(address));
48              end if;
```

در سیگنال ram_data در خروجی data_out ریخته میشود. در این بخش به اتمام شرط تعریف شده ی دوم میرسیم ولی هنوز کار if اول ناتمام مانده است پس با تعریف یک else جدید به برنامه دستور میدهیم که در صورت یک نبودین ورودی enb و روشن نشدن قطعه چه خروجی باید داشته باشد.

```
49           else -- if ram is off
50              data_out <= std_logic_vector(to_unsigned(0,data_in'length)); -- This line will puts zero as vector length of data_in
51           end if;
52        end if;
53     end process;
54
55  end Behavioral;
```

که همانطور که در تصویر مشاهده می‌شود در این صورت مقدار ۰ را درون data_out میریزیم اما با توجه به ۱۶ بیتی بودن خروجی باید ۱۶ بیت ازین صفر ها تولید کنیم که این کار نیز به وسیله ی دستورات نوشته شده امکان پذیر میشود.

دستورات در استفاده شده در این بخش مانند دستورات استفاده شده در رام است دوباره آنها را بیان می‌کنیم.

دستور conv_integer(binry_number) این دستور یک عدد باینری را به یک عدد اینتیجر تبدیل می‌کند. باتوجه به اینکه ما رام را به صورت یک آرایه دو بعدی تعریف کردیم پس باید برای دست رسی به المان های درون آرایه و indexing ما باید به آرایه شماره المانی که می‌خواهیم بدهیم تا به ما آن المان را خروجی دهد. به همین دلیل ما عدد باینری آدرس را به اینتیجر تبدیل کردیم تا بتوانیم به المان های درون آریه دست رسی داشته باشیم.

دستور std_logic_vector(to_unsigned(0, data_out'length))

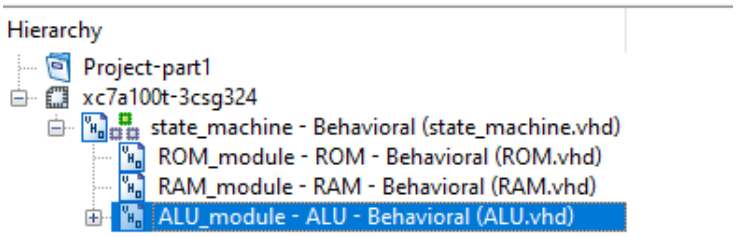
این درستور از چند بخش تشکیل شده است. ابتدا data_out'length را بیان می‌کنیم. هر وکتوری که قبل از کوتیشن قرار بگیرد طول محاسبه می‌شود. این دستور طول داده ورودی را محاسبه می‌کند و به صورت یک عدد دسیمال خروجی می‌دهد.

دستور to_unsigend(number_in_decimal, number_of_bits) این دستور دو ورودی دریافت می‌کند و خروجی آبجکت عدد باینری می‌دهد. ورودی اول عدد دسیمالی از که می‌خواهیم ایجاد شود و دومی تعداد بیت هایی است که می‌خواهیم آن عدد داشته باشد.

در نهایت دستور std_logic_vector است که آبجکت ایجاد شده را به یک وکتوری از اعداد باینری تبدیل می‌کند.

کد ماژول ALU

قطعه ی ALU در فایلی با همین نام در صفحه کد های VHDL نرم افزار ISE شبیه سازی شده است. این



قطعه در استیت ماشین برای شبیه سازی عملیات های حسابی، منطقی و شیفت استفاده میشود.

این ماژول همانند ماژور های قبلی دو ورودی تخت عنوان clk و enb دارد که به ترتیب مربوط به کلاک و فعال کردن قطعه میباشند. یعنی همانطور که از قبل مشاهده کردیم در صورت ۱ بودن ورودی enb قطعه روشن و در غیر این صورت خاموش است.

این قطعه همچنین دو ورودی ۱۶ بیتی تحت عنوان input1 و input2 دارد که به ترتیب برای دریافت مقادیر آکومالاتور (AC) و IR مورد استفاده قرار میگیرند. ورودی سوم این قطعه (input3) که یک ورودی ۶ بیتی است برای دریافت opcode میباشد که وظیفه ی مشخص کردن دستورات را دارد. که این دستورات و opcode مربوط به هرکدام در روی سوال به ما داده شده است.

```
12  entity ALU is
13
14      Port (
15      input1,input2   : in  STD_LOGIC_VECTOR(15 downto 0);  --AC & ADRESS
16      carry_in : in std_logic; -- E
17      input3   : in  STD_LOGIC_VECTOR(5 downto 0);  --UPCODE (for selecting function)
18      clk : in std_logic; --clk
19      enb : in std_logic; --ALU_on
20      output   : inout  STD_LOGIC_VECTOR(15 downto 0); --ALU_out
21      carry : out std_logic := '0' --ALU_carry
22      );
23
24  end ALU;
```

ALU همچنین یک carry ورودی دریافت میکند که ما در کد خود با carry_in مشخص کرده ایم و یک کری هم در خروجی خود دارد که مقدار آن در این کد صفر قرار داده شده است.

خروجی اصلی این قطعه نیز که با output مشخص شده است یک خروجی ۱۶ بیتی میباشد.

این ماژول در opcode های مشخصی باید عملیات ضرب و جذرگیری را برروی ورودی های خود پیاده کند که برای این کار نیز از کد های ضـــرب کننده و جذر گیری که به ترتیب در تمرین های ســـری اول و دوم عملی به انجام رسانده ایم استفاده کرده ایم.

برای این کار این دو ماژول ضـــرب کننده (MLP) و جذرگیر(SQR) را به صـــورت component در قســـمت Behavior کد خود افزوده ایم.

```
26  architecture Behavioral of ALU is
27
28  component MLP is
29      Port ( a,b : in  STD_LOGIC_VECTOR (5 downto 0);
30             p : out  STD_LOGIC_VECTOR (11 downto 0));
31
32  end component;
33
34  component SQR is
35      Port ( A : in  STD_LOGIC_VECTOR (16 downto 1);
36             q : out  STD_LOGIC_VECTOR (8 downto 1));
37  end component;
```

که طبق تصویر بالا مشاهده میگنیم که ماژول ضرب کننده دو ورودی ۶ بیتی a و b را گرفته و حاصل ضرب آن هارا در قالب یک خروجی ۱۲ بیتی p تحویل میدهد و ماژول جذر گیر، برای هر ورودی ۱۶ بیتی A یک خروجی ۸ بیتی که حاصل جذر آن میباشد را به ما تحویل میدهد (q).

```
39  signal AC : std_logic_vector(15 downto 0) := std_logic_vector(to_unsigned(0, 16));
40  signal ml : std_logic_vector(11 downto 0) := (others => '0');
41  signal sq : std_logic_vector(7 downto 0)  := (others => '0');
42  signal temp : std_logic_vector(16 downto 0) := std_logic_vector(to_unsigned(0, 17));
43  signal E : std_logic := '0';
44
45
46  begin
```

طبق آن چیزی که در تصویر مشاهده میکنید قبل از شروع دستورات این ماژول تعدادی سیگنال نیز تعریف شده که به نقش هرکدام هنگام استفاده مفصل پرداخته شده است.

در ابتدای شروع دستورات مجددا با ماژول های ضرب کننده و جذر گیر مواجه میشویم که این به این دلیل است که تصـمیم گرفتیم برای هر ورودی که میگیریم در ابتدای کار مقادیر حاصـل از این ماژول ها تحت تاثیر ورودی

```
46  begin
47
48  MLP1 : MLP port map (a => input1(5 downto 0), b => input2(5 downto 0), p => ml);
49  SQR1 : SQR port map (A => input1, q => sq);
50
```

های مفروض را بدســـت آوریم و در صـــورت گرفتن opcode مربوط به آن ها به یکباره خروجی این ماژول هارا مورد استفاده قرار دهیم و از انجام عملیات مجدد و پیچیده تر خودداری کنیم.

باتوجه به اینکه ضـرب کننده ی ما ۶ بیتی اسـت پس تنها ۶ بیت کم ارزش ورودی های اول و دوم خود را به آن میدهیم و از آن میخواهیم که خروجی خود یعنی حاصلضرب ورودی هارا در سیگنال از قبل تعریف شده ی m1 بریزد که باتوجه به ۱۲ بیتی بودن خروجی ضرب کننده این سیگنال یک سیگنال ۱۲ بیتی تعریف شده است.

برای جذر گیر SQR نیز با دادن ورودی اول خود یعنی AC به آن از آن میخواهیم که خروجی را در سـیگنال ۸ بیتی sq ذخیره کند که بعد ها ازین سیگنال ها در دستورات ALU استفاده خواهیم کرد.

برای انجام هردوی این عملیات این ماژول هارا به صورت portmap تعریف کرده و خواسته های خود را به همین شکل در آن پیاده میکنیم.

حال به قسـمت اصـلی این ماژول میرسـیم و دسـتورات را به گونه ای در آن تعریف میکنیم تا تشـخیص دهد با گرفتن هر opcode چه عملیاتی بر روی داده ها انجام خواهد داد. این ماژول حسـاس به تغییرات هر سـه ورودی اصلی خود و همچنین تغییرات کلاک میباشد که این ها به صورت process در آن تعریف میشوند.

```
52  process(input1,input2,input3,clk)
53  begin
54
55    if (enb = '1') then
56       case(input3) is
57
58          when "000001" => --AND
59           output <= input1 and input2;
60
61          when "000100" => --ADD
62
63           temp <= std_logic_vector(to_unsigned(to_integer(unsigned(input1)), 17))
64           output<= temp( 15 downto 0);
65           carry <= temp (16);
66
67          when "000101" => --Increment AC
68           output <= input1 + '1';
69
70          when "001000" => --circular left shift
71           AC(15 downto 1) <= input1(14 downto 0);
72           AC(0) <= carry_in;
73           output <= AC;
74           E <= input1(15);
75           carry <= E;
76
77          when "001001" =>  --circular right shift
78           AC(14 downto 0) <= input1(15 downto 1);
79           E <= input1(0);
```

برای این قطعه نیز همانند قطعات قبلی یک حلقه if تعریف میکنیم تا در صـــورت ۱ بودن ورودی enb که به معنای روشن بودن قطعه است تمام دستورات خواسته شده را انجام دهد و در غیر این صورت مقدار صفر را در

```
104          end case;
105        else
106
107          output <= std_logic_vector(to_unsigned(0, 16));
108          carry <= '0';
109
110       end if;
111
112  end process;
113
114
115
116  end Behavioral;
```

هردو خروجی output و carry خود بریزد البته باید توجه کنیم که خروجی output ۱۶ بیتی اســـت که به همین دلیل این قسمت به شکل زیر تعریف شده است.

حال به بررســـی دســـتورات مربوط به این قطعه میپردازیم و بار دیگر یادآور میشـــویم که این قطعه دســـتورات محاسباتی، منطقی، شیفت و یکی اضافه کردن را بر عهده دارد.

طبق آنچه که طراح از ما خواسته است این ماژول باید در صورت دریافت opcode، ۰۰۰۰۰۱ عملیات AND را برروی ورودودی های خود انجام دهد و در صورت داشتن opcode، ۰۰۰۱۰۰ ورودی های خود را باهم ADD

```
58   when "000001" => --AND
59    output <= input1 and input2;
60
61   when "000100" => --ADD
62
63    temp <= std_logic_vector(to_unsigned(to_integer(unsigned(input1)), 17))  + std_logic_vector(to_unsigned(to_integer(unsigned(input2)), 17));
64    output<= temp( 15 downto 0);
65    carry <= temp (16);
```

کنـد کـه برای این کـار از عملیـات ریـاضـــی "+" اســـتفـاده کرده ایم و برای اســـتفـاده از آن کتـاب خـانـه ی (ieee.NUMERIC_STD) را افزوده ایم.

چون این حاصـل جمع ممکن اسـت دارای overflow باشـد و خروجی ما میتواند تنها ۱۶ بیت را در خود جای دهد نیازمند تعریف یک سـیگنال ۱۷ بیتی هسـتیم تا بیت کری خود را نیز در صـورت وجود دریافت کنیم. که سیگنال ۱۷ بیتی temp دقیقا برای همین منظور تعریف شده است. پس حاصل جمع خود را درون این سیگنال

```
67      when "000101" => --Increment AC
68       output <= input1 + '1';
```

میرزیم و همانطور که در تصـویر مشـاهده میکنید ۱۶ بیت کم ارزش آن را در خروجی output و با ارزش ترین بیت آن را که همان کری ما خواهد بود را در خروجی carry میریزیم.

Opcode بعدی ما که معادل ۰۰۰۱۰۱ می باشـد برای increment کردن آکومالاتور اسـت که به همین منظورورودی خود را با تک بیت ’1‘ جمع کرد و در خروجی میریزیم.

دسـتور بعدی ما شـیفت چپ حلقه ای اسـت که مربوط که opcode، ۰۰۱۰۰۰ میباشـد. برای این منظور یک سـیگنال ۱۶ بیتی تحت عنوان AC تعریف کرده ایم تا عملیات شـیفت را درون آن انجام دهیم. به گونه ای که

```
70          when "001000" => --circular left shift
71           AC(15 downto 1) <= input1(14 downto 0);
72           AC(0) <= carry_in;
73           output <= AC;
74           E <= input1(15);
75           carry <= E;
```

۱۵ بیت اول ورودی را در بیت دوم تا ۱۶ ام سـیگنال AC ریخته و بیت carry ورودی را در اولین بیت این سیگنال میریزیم. حال کل سیگنال را در خروجی output میرزیم.

حال ۱۶ امین بیت ورودی خود که با ارزش ترین بیت آن میشـود را در سـیگنال از قبل تعریف شـده ی E ریخته و بار دیگر مقدار ذخیره شـده در E را در carry خروجی خود میریزیم. در پایان این دسـتورات کل ورودی ما

```
77          when "001001" =>  --circular right shift
78           AC(14 downto 0) <= input1(15 downto 1);
79           E <= input1(0);
80           AC(15) <= carry_in;
81           carry <= E;
82           output <= AC;
```

یکبیت به سمت چپ شیفت داده شده و در طی این حرکت چپ ترین بیت که بیت کری بود اینبار در کم ارزش ترین بیت ما و پر ارزش ترین بیتمان در carry ریخته میشود.

opcode، ۰۰۱۰۰۱ هم مربوط به شـیفت راسـت حلقه ای اسـت که دقیقا با همان منطق شـیفت چپ انجام میپذیرد با این تفاوت که اینبار بیت دوم تا ۱۶ ام ورودی در ۱۵ بیت کم ارزش سـیگنال AC ریخه شـده و کم ارزش ترین بیت ورودی در سیگنال E ریخته میشود تا در carry خروجی ریخته شود. همچنین کری ورودی در پر ارزش ترین بیت سیگنال AC ریخته میشود و در آخر کل این سیگنال درون خروجی output ریخته میود.

حال نوبت شیفت های خطی میرسد که دقیقا با همان منطق شیفت های حلقوی صورت میگیرد و تنها با این تفاوت که مثلا در شیفت خطی چپ که مربوط به opcode، ۰۰۱۱۱۰ است، این بار به جای ریختن carry در

```
84          when "001110" => --linear left shift
85           AC(15 downto 1) <= input1(14 downto 0);
86           carry <= input1(15);
87           AC(0) <= '0';
88           output <= AC;
```

AC(0) مقدار ’0‘ درون آن ریخته میشود و input(15) به صورت مستقیم در کری خروجی ریخته میشد.

در شیفت خطی راست نیز که مربوط به opcode، ۰۰۱۱۱۱ میباشد به جای ریخته شدن carry_in در AC(15) ورودی ’0‘ را درون آن میریزیم. و input(0)مستقیما و بدون واسطه ی سیگنال درون carry خروجی ریخته میشود.

```
90          when "001111" => --linear right shift
91           AC(14 downto 0) <= input1(15 downto 1);
92           carry <= input1(0);
93           AC(15) <= '0';
94            output <= AC;
```

در آخر دو حلقه ی دیگر when نیز برای ضرب کننده و جذرگیر تعریف میکنیم که به ترتیب مربوط به opcode های ۰۱۰۰۰۰و۱۰۰۰۰۰ هستند. همانطور که بالاتر توضیح داده ایم این عملیات از قبل انجام شده اند و

```
95
96          when "010000" => --multiply
97           output <= std_logic_vector(to_unsigned(to_integer(unsigned(m1)), 16));
98
99          when "100000" => --SQR
100          output <= std_logic_vector(to_unsigned(to_integer(unsigned(sq)), 16));
101
```

خروجی های شان در سیگنال های m1 و sq ریخته شده است پس حال کافیست مقدار همین سیگنال هارا در output بریزیم.

میدانیم که در پایان استفاده از حلقه ی when حتما باید از عبارت "when others" نیز استفاده کنیم که به همین جهت در پایان کد خود این قسمت را آورده ایم ولی پون در این حالت انتظار انجام عملیات خاصی را از کد نداریم دستوری درون آن تعریف نکرده ایم.

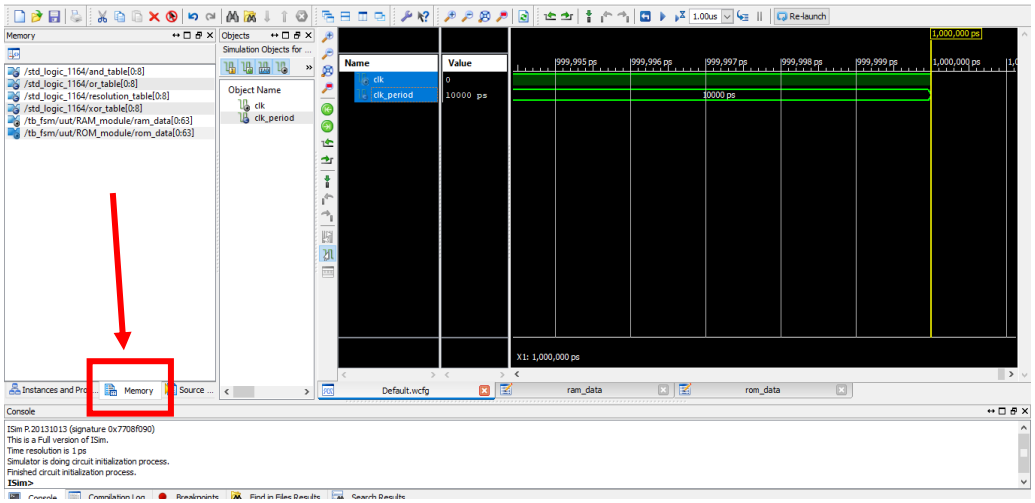
```
102          when others => -- Do nothing
103
104          end case;
105        else
106
```

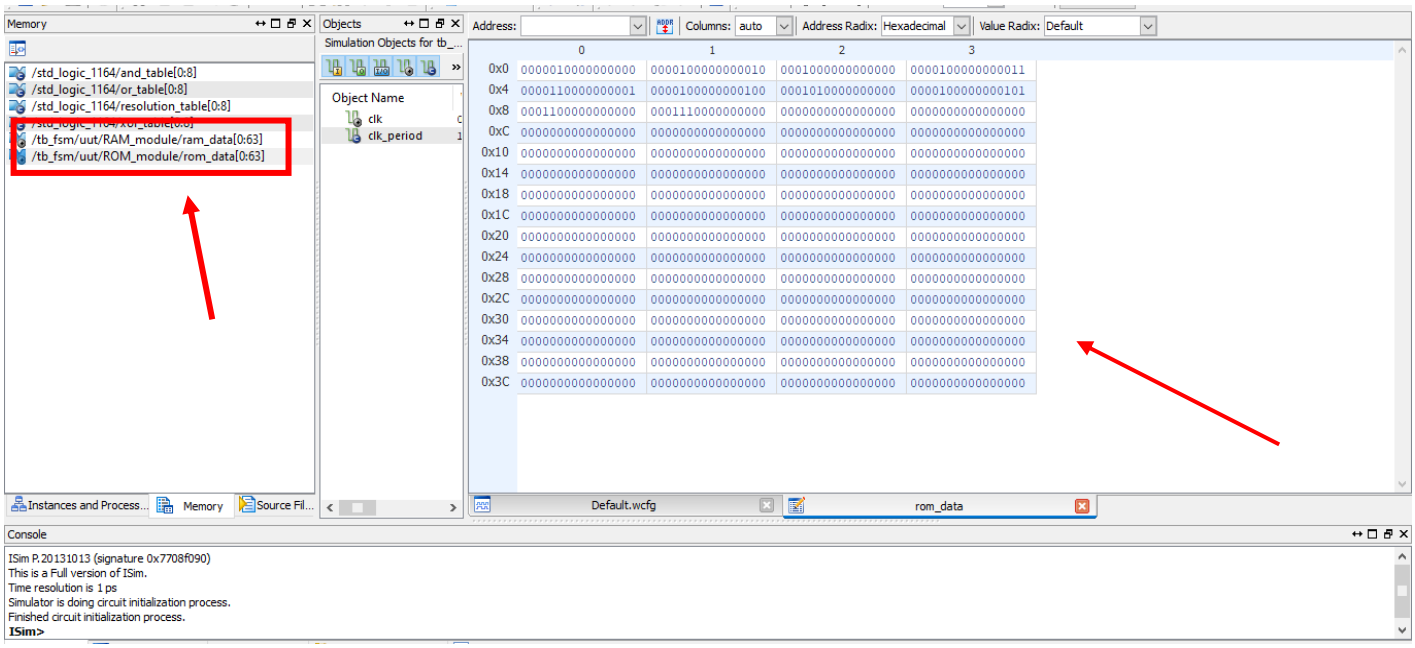
تست برنامه

در صورت پروژه بیان شده است که نیاز به پیاده سازی تست بنچ نمی‌باشد. اما در این پروژه ما باتوجه به پیچیدگی کد اطمینان حاصل کردن از درستی کد نوشته شده یک تست تعریف شده است. باتوجه به اینکه سیستم و ورودی و خروجی خارجی ندارد و دستورات را از ROM دریافت کرده پرادازش کرده و در RAM می‌ریزد. سیگنالی برای نمایش وجود ندارد فقط مقدار اولیه و نهایی رم مقایسه می‌شود. مقدار اولیه رم در قسمت کد رم نوشته شده است بعد اجرا مقدار اولیه آن تغییر خواهد کرد.

فایل تست این برنامه فایل TB_FSM می‌باشد که در قسمت سیمولیشن می‌باشد. این فایل کار خاصی را نجام نمی‌دهد فقط بعد از اجرای شبیه سازی می‌توانیم مقدار نهایی رم را ببینیم.

نحویه دسترسی به مقدار نهایی رم





نکته مهم:

در فایل state_machine مقدار اولیه AC صفر گذاشته نشده است برای اینکه نتیجه تست ها بهتر مشخص شود مقدار اولیه آن معادل باینری ۳ در نظر گرفته شده است.

توضیح تست های انجام شده و گزارش نتایج

دستورات در رام گذاشته شده‌اند و توسط برنامه یکی یکی اجرا می‌شوند. زیرا pc بعد از اجرای هر دستور یکی اضافه شده و باعث می‌شود دستور بعدی در حافظه لود شود.

	0	1	2	3
0x0	0000010000000000	0000100000000010	0001000000000000	0000100000000011
0x4	0000110000000001	0000100000000100	0001010000000000	0000100000000101
0x8	0001100000000000	0001110000000000	0000000000000000	0000000000000000
0xC	0000000000000000	0000000000000000	0000000000000000	0000000000000000
0x10	0000000000000000	0000000000000000	0000000000000000	0000000000000000
0x14	0000000000000000	0000000000000000	0000000000000000	0000000000000000
0x18	0000000000000000	0000000000000000	0000000000000000	0000000000000000
0x1C	0000000000000000	0000000000000000	0000000000000000	0000000000000000
0x20	0000000000000000	0000000000000000	0000000000000000	0000000000000000
0x24	0000000000000000	0000000000000000	0000000000000000	0000000000000000
0x28	0000000000000000	0000000000000000	0000000000000000	0000000000000000
0x2C	0000000000000000	0000000000000000	0000000000000000	0000000000000000
0x30	0000000000000000	0000000000000000	0000000000000000	0000000000000000
0x34	0000000000000000	0000000000000000	0000000000000000	0000000000000000
0x38	0000000000000000	0000000000000000	0000000000000000	0000000000000000
0x3C	0000000000000000	0000000000000000	0000000000000000	0000000000000000

دستور خانه ۰ و۰ عدد ۰۰۰۰۰۱۰۰۰۰۰۰۰۰ می باشد که این دستور and می‌باشد و چون این دستور مراجه به حافظه است داریم:

$$RAM[0] = \ 0000000000000001$$

$$ROM[0] = IR = \ 0000010000000000 \rightarrow and\ AC\ with\ 0\ elemet\ of\ RAM$$

$$AC = \ 0000000000000011$$

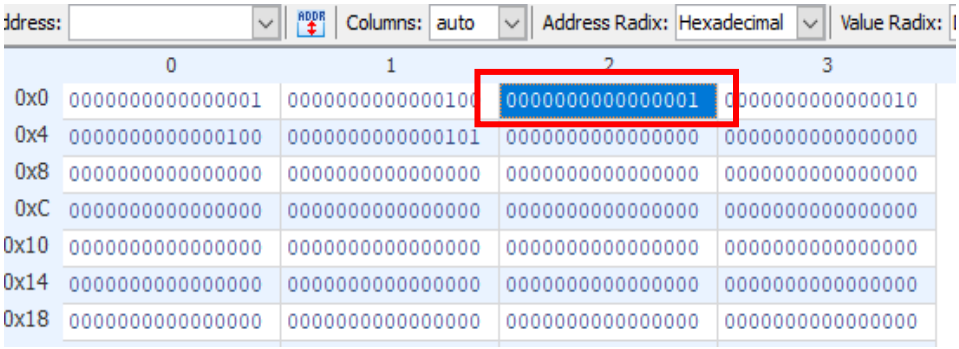
بعد اجرای دستور خواهیم داشت

$$AC = \ 0000000000000001$$

دستور بعدی تست AND:

$$ROM[1] = IR = \ 0000100000000010 \rightarrow \ STORE\ AC\ IN\ RAM[2]$$

$$RAM[2] \leftarrow AC = \ 0000000000000001$$



دستور بعدی تست ADD:

$$ROM[2] = IR = 0001000000000000 \rightarrow \ ADD\ AC\ WITH\ RAM\ [0]$$

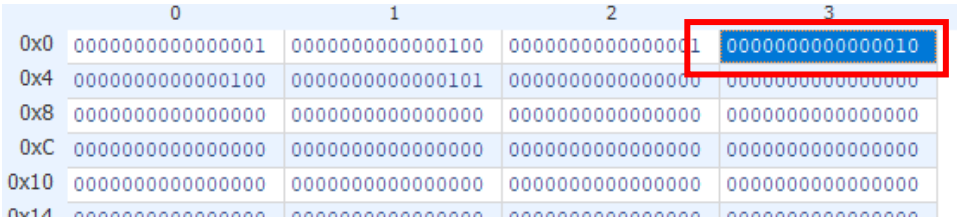
$$AC \leftarrow \ AC + RAM[0]$$

$$AC = \ 0000000000000010$$

دستور بعدی:

$$ROM[3] = IR = \ 0000100000000011 \rightarrow \ STORE\ AC\ IN\ RAM[3]$$

$$RAM[3] \leftarrow AC$$



دستور بعدی تست دستور STA , LDA:

$$ROM[4] = IR = 0000110000000001 \rightarrow LOAD\ RAM[1]\ TO\ AC$$

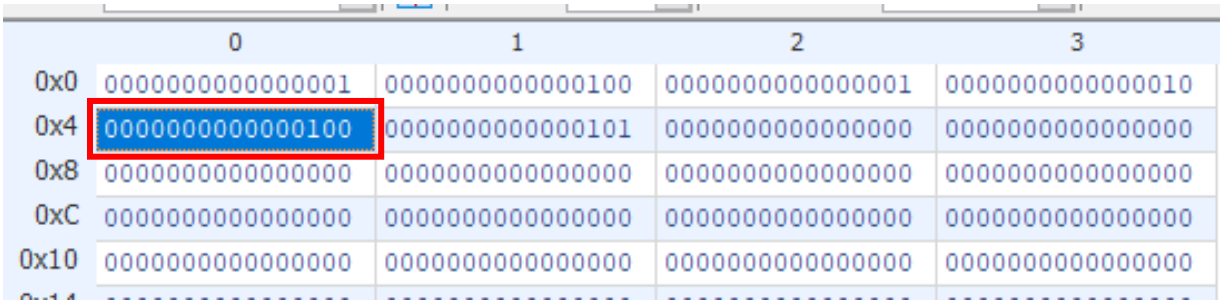
$$AC \leftarrow RAM[1]$$

$$AC = 0000000000000100 = 4$$

دستور بعدی:

$$ROM[5] = IR = \ 0000100000000100 \rightarrow STORE\ AC\ IN\ RAM[4]$$

$$RAM[4] \leftarrow AC$$



	0	1	2	3
0x0	0000000000000001	0000000000000100	0000000000000001	0000000000000010
0x4	0000000000000100	0000000000000101	0000000000000000	0000000000000000
0x8	0000000000000000	0000000000000000	0000000000000000	0000000000000000
0xC	0000000000000000	0000000000000000	0000000000000000	0000000000000000
0x10	0000000000000000	0000000000000000	0000000000000000	0000000000000000

تست دستور INC:

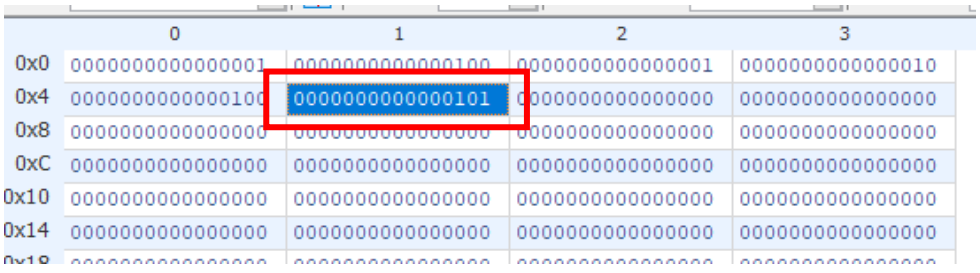
$$ROM[6] = IR = 0001010000000000 \rightarrow INCREAMENT\ AC$$

$$AC \leftarrow AC + 1 \equiv 4 + 1$$

دستور بعدی برای ذخیره کردن آن:

$$ROM[7] = IR = 0000100000000101 \rightarrow STORE\ AC\ IN\ RAM[5]$$

$$RAM[5] \leftarrow AC$$



	0	1	2	3
0x0	0000000000000001	0000000000000100	0000000000000001	0000000000000010
0x4	0000000000000100	0000000000000101	0000000000000000	0000000000000000
0x8	0000000000000000	0000000000000000	0000000000000000	0000000000000000
0xC	0000000000000000	0000000000000000	0000000000000000	0000000000000000
0x10	0000000000000000	0000000000000000	0000000000000000	0000000000000000
0x14	0000000000000000	0000000000000000	0000000000000000	0000000000000000

همین طور مشخص است در خانه پنجم رم مقدار ۵ ذخیره شده است.

دستور بعدی CLA:

$$ROM[8] = IR = 0001100000000000 \rightarrow \ Clear\ AC$$

$$AC \leftarrow 0$$

دستور بعدی CLE:

$$ROM[9] = IR = 0001110000000000 \rightarrow \ CLEAR\ E$$

$$E \leftarrow 0$$

این پایان دستورات موجود در رام بود. این بخش تستی از برنامه بود و عملکرد درست تمام بخش های برنامه را نمایش می‌دهد.

برای تمام دیگر کامپوننت های برنامه (RAM, ROM, ALU) و تست بنچ طراحی شده است و درستی عملکرد آنها بررسی شده است.