به نام خدا

گزارش کار پروژه درس طراحی کامپیوتری سیستم های دیجیتالی

موضوع پروژه : پیاده سازی cache به روش 2 way associative

دی ماه ۱۴۰۲

اعضای گروه :

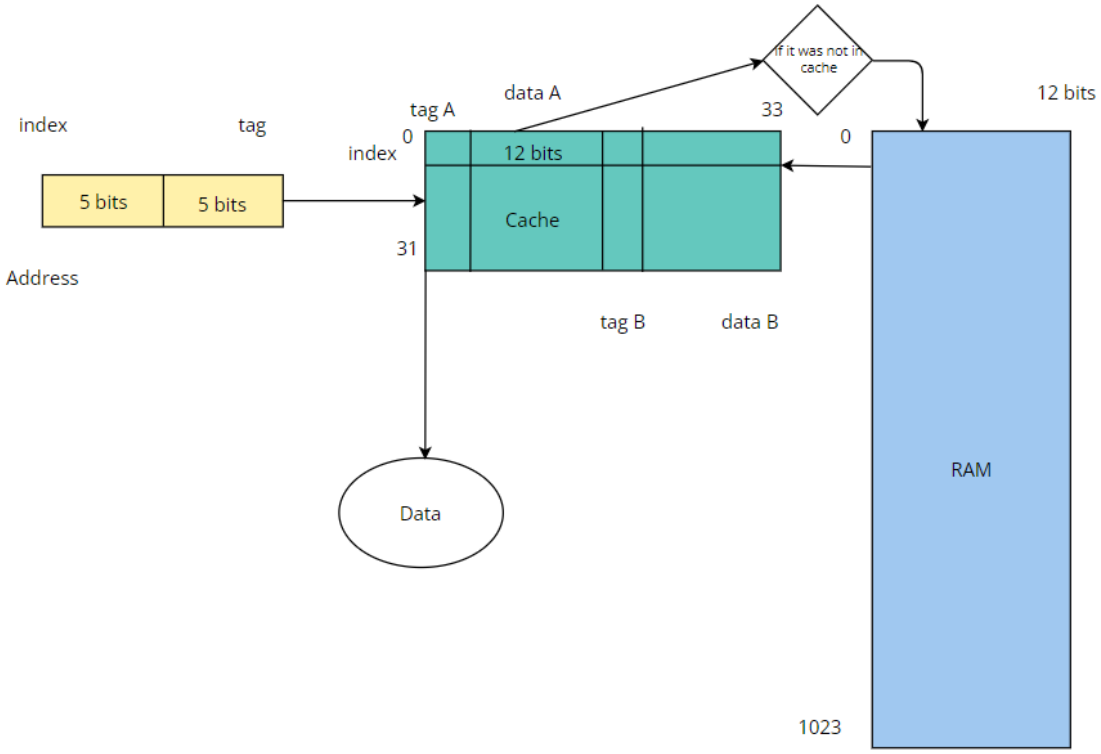
محمد امین طهماسبی نیا

مرتضی پوررمضان

آریان قهار پور

۱.آشنایی با پروژه

در این پروژه گروه از روش 2 way associative برای پیاده سازی cache استفاده کرده ایم. در پروژه ما سه ماژول اصلی به نام های ama_ram, ama_cache, ama_cache_ram ساخته شده است که هر کدام توضیح داده خواهد شد. همانطور که در شکل شماره یک مشاهده می شود آدرس های که به سمت حافظه cache ارسال می شوند ۱۰ بیتی بوده که ۵ بیت آن مربوط به index و ۵ بیت دیگر برای tag است. که توضیحات بیشتر آن در ادامه داده خواهد شد. در شکل ماژول های ram و cache نشان داده شده و منطق اصلی کد مشخص است. آدرس به حافظه موقت داده می شود ، اگر در آدرس مورد نظر دیتایی موجود بود بازگردانده می شود و اگر نبود از ram خوانده شده و در cache نوشته می شود و سپس دیتا به بازگردانده می شود. جزئیات آن در ادامه بحث خواهد شد.



شکل ۱ : ساختار کلی پروژه cache

۲.آدرس

آدرس در پروژه ما از دو بخش index و tag تشکیل شده است. ۵ بیت برای index و ۵ بیت برای tag هر index نشان دهنده سطری در cache است از همین رو حافظه موقت ما ۳۲ سطر دارد.قسمت tag برای مشخص کردن این است که کدام قسمت از cache برای خواندن انتخاب شده در واقع در ساختار cache به روش 2-way associative به نوعی انگار دو حافظه موقت به هم چسبیده اند. که بیشتر در بخش خود توضیح داده خواهد شد. آدرس ها در حافظه ram هم به یک سطر خاص اشاره می کنند که به دلیل ۱۰ بیتی بودن کل آدرس ما ram از ۱۰۲۴ سطر تشکیل شده است.

۳.RAM

RAM یا Random Access Memory حافظه اصلی سیستم ما است که از یک آرایه دو بعدی ۱۲ در ۱۰۲۴ بیتی تشکیل شده است. ماژول از سه سیگنال ورودی و یک خروجی دارد. سه سیگنال ورودی ما clk, rd, address هستند که rd برای اجازه خواندن از رم است و آدرس نیز آدرس خانه مورد نظر است. پس از خواندن از خانه متناظر از data_array آن را به سیگنال خروجی data_out می دهیم.

```
20  library IEEE;
21  use IEEE.STD_LOGIC_1164.ALL;
22  use IEEE.numeric_std.ALL;
23
24  entity ama_ram is
25      Port ( rd : in STD_LOGIC;
26             clk : in  STD_LOGIC;
27             address : in  STD_LOGIC_VECTOR (9 downto 0);
28             data_out : out  STD_LOGIC_VECTOR (11 downto 0)
29             );
30  end ama_ram;
31
32  architecture Behavioral of ama_ram is
33
34  type data_array_data is array (1023 downto 0) of STD_LOGIC_VECTOR (11 downto 0);
35  signal data_array: data_array_data;
36
37  constant data_array_init : data_array_data := (
38          0  to 9  => "100010101000",
39          10 to 19 => "010101010101",
40          20 to 29 => "001100110011",
41          30 to 39 => "111000111000",
42          40 to 49 => "000111000111",
43          50 to 59 => "101010101010",
44          60 to 69 => "110011001100",
45          70 to 79 => "011001100110",
46          80 to 89 => "000000000001",
47          90 to 99 => "111111111110",
48          others   => (others => 'U')
49      );
50  begin
51
52  data_array <= data_array_init;
53
54  process (clk) begin
55     if rd = '1' then
56        data_out <= data_array(to_integer(unsigned(address)));
57     end if;
58  end process;
59
60  end Behavioral;
```

شکل ۲: کد ama_ram

Cache.۴

ماژول cache که در پروژه ما به نام ama_cache ثبت شده از چهار سیگنال ورودی و دو خروجی تشکیل شده است. سیگنال address یکی از سیگنال های ورودی است که آدرس مورد نظر برای خواندن از cache را نشان می دهد. سیگنال wr برای نوشتن در cache تعبیه شده در زمانی که آدرس مورد نظر در cache یافت نشد و از ram دیتا گرفته شده و در cache نوشته می شود. این نوشتن و گرفتن دیتای جدید از طریق data_in انجام می شود.

در حافظه cache نیز از یک آرایه دو بعدی ۳۲ در ۳۴ تحت عنوان data_array_data تعبیه شده که در واقع محل ذخیره سازی اطلاعات است هر index که از address استخراج می شود به یک سطر از این آرایه اشاره می کند و با توجه با ساختار cache (2 way associative بودن یعنی داشتن دو tag و data در هر سطر) بعد از یافتن index به سراغ چک کردن tag ها می رود و دیتا را می یابد. این دو tag و data که گفته شد برای چک شدن از هر row که با توجه به index انتخاب می شود، برداشته می شوند و درون tagA, tagB, dataA, dataB ریخته می شود.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.numeric_std.ALL;
4
5  -- Uncomment the following library declaration if using
6  -- arithmetic functions with Signed or Unsigned values
7  --use IEEE.NUMERIC_STD.ALL;
8
9  -- Uncomment the following library declaration if instantiating
10 -- any Xilinx primitives in this code.
11 --library UNISIM;
12 --use UNISIM.VComponents.all;
13
14 entity ama_cache is
15     Port ( address : in  STD_LOGIC_VECTOR(9 downto 0);
16            wr : in STD_LOGIC;
17            clk : in  STD_LOGIC;
18            data_in : in  STD_LOGIC_VECTOR(11 downto 0);
19            data_out : out  STD_LOGIC_VECTOR(11 downto 0);
20            hit_miss: out STD_LOGIC
21            );
22 end ama_cache;
23
24 architecture Behavioral of ama_cache is
25
26     type data_array_data is array (31 downto 0) of STD_LOGIC_VECTOR (33 downto 0);
27     signal data_array: data_array_data := (others => (others => 'U'));
28     signal index: STD_LOGIC_VECTOR(4 downto 0);
29     signal tag: STD_LOGIC_VECTOR(4 downto 0);
30
31     signal tagA: STD_LOGIC_VECTOR(4 downto 0);
32     signal dataA: STD_LOGIC_VECTOR(11 downto 0);
33     signal tagB: STD_LOGIC_VECTOR(4 downto 0);
34     signal dataB: STD_LOGIC_VECTOR(11 downto 0);
35     signal row: STD_LOGIC_VECTOR(33 downto 0);
36
```

شکل ۳: کد ama_cache

بعد از چک کردن سیگنال wr (صفر بودنش به این منظور است که هنگام خواندن از cache است) و استخراج tag و index از address به سراغ حلقه های پی در پی می رویم تا tag و data را از سطر مورد نظر استخراج کنیم. شماری که حلقه ها پیش میروند با توجه به طول tag و data مشخص شده است(۱۲ بیت برای data و ۵ بیت برای tag) برای مثال در حلقه ابتدایی که در قطعه کد زیر مشخص است از ۰ تا ۴ برای یافتن tag قسمت اول سطر مورد نظر است و از ۵ تا ۱۶ برای data پس از آن است. در ادامه پس از جدا کردن قطعه های tag و data به سراغ چک کردن تگ ها می رویم اگر tag آدرس با tagA یکی بود یعنی دیتای مورد نظر در data قرار دارد و آن را درون data_out خواهد ریخت و سیگنال hit_miss که نشان دهنده وضعیت cache هست را یک می کند. یک بودن hit_miss به این معنی است که cache دیتای مورد نظر را داشته و آن را مستقیم بازگردانده است.

```
37  begin
38
39      process (clk) begin
40
41          if (wr = '0') then
42              for i in 0 to 9 loop
43                  if (i < 5) then
44                      index(i) <= address(i);
45                  else
46                      tag(i-5) <= address(i);
47                  end if;
48              end loop;
49
50              row <= data_array(to_integer(unsigned(index)));
51
52              for i in 0 to 4 loop
53                  tagA(i) <= row(i);
54              end loop;
55
56              for i in 5 to 16 loop
57                  dataA(i-5) <= row(i);
58              end loop;
59
60              for i in 17 to 21 loop
61                  tagB(i-17) <= row(i);
62              end loop;
63
64              for i in 22 to 33 loop
65                  dataB(i-22) <= row(i);
66              end loop;
67
68              if (tag = tagA) then
69                 data_out <= dataA;
70                 hit_miss <= '1';
71              elsif (tag = tagB) then
72                 data_out <= dataB;
73                 hit_miss <= '1';
74              else
75                 hit_miss <= '0';
76
77              end if;
78
79
80          end if;
```

شکل ۴ : کد ama_cache

۱.۴. نوشتن درون cache در صورت نبودن دیتای مورد نظر

در صورت نبود دیتای مورد نظر در سطر اشاره شده اگر دیتای قسمت اول (dataA) خالی بود. آنگاه در آن مقدار data_in که از ram خوانده شده است را قرار می دهیم. در صورتی که قسمت اول نیز پر بود این کار را با قسمت دوم دیتا (dataB) انجام میدهیم. لازم به ذکر است که در این میان در صورتی که مقدار data_in درون هر یک از بخش های دیتای مورد نظر قرار گرفت tag آن بخش نیز به مقدار tag در آدرس دریافتی تغییر می کند. همان گونه که در قطعه کد زیر مشخص است ابتدا tag درون tagA ویا tagB سپس از آن درون پنج بیت اول هر کدام از بخش های انتخاب شده ریخته می شود.

```
82          if ( wr = '1') then
83
84              for i in 0 to 9 loop
85                  if (i < 5) then
86                      index(i) <= address(i);
87                  else
88                      tag(i-5) <= address(i);
89                  end if;
90              end loop;
91
92              row <= data_array(to_integer(unsigned(index)));
93
94              for i in 0 to 4 loop
95                  tagA(i) <= row(i);
96              end loop;
97
98              for i in 5 to 16 loop
99                  dataA(i-5) <= row(i);
100             end loop;
101
102             for i in 17 to 21 loop
103                 tagB(i-17) <= row(i);
104             end loop;
105
106             for i in 22 to 33 loop
107                 dataB(i-22) <= row(i);
108             end loop;
109
110
111             if dataA'length = 0 then
112
113                tagA <= tag ;
114                for j in 0 to 4 loop
115                   row(j) <= tagA(j);
116                end loop;
117                for j in 5 to 16 loop
118                   row(j) <= data_in(j-5);
119                end loop;
120
121
122             elsif dataB'length = 0 then
123
124                tagB <= tag;
125                for j in 17 to 21 loop
```

شکل ۵ : کد ama_cache

۲.۴. پر بودن هر دو خانه و نحوه انتخاب بین آنها

در صورت پر بودن هر دو خانه آنگاه از الگوریتمی که گروه برای آن طراحی کرده است استفاده می کنیم برای اینکار می شد از الگوریتم های آماده که برای مثال همواره قدیمی ترین مقدار را پاک می کردند و دیتای جدید را در آن می نوشتند استفاده کرد. اما گروه تصمیم بر انجام این بخش با خلاقیت خود و روش خود کرده است.

۳.۴. توضیح الگوریتم مورد نظر

در صورتی که هر دو خانه پر باشند در خانه مربوط بخش دوم B نوشته خواهد شد. ولی قبل از آن مقدار tag در بخش B و دیتای این بخش را به درون بخش A شیفت می دهیم (به نوعی شیفت چپ) با اینکار عملا هر بار که نیاز باشد دوبا ره در این خانه نوشته شود دیتایی که جدید تر نوشته شده باقی خواهد ماند. و دیتای قدیمی تر از درون cache بیرون خواهد رفت.

```
125                 for j in 17 to 21 loop
126                    row(j) <= tagB(j);
127                 end loop;
128                 for j in 22 to 33 loop
129                    row(j) <= data_in(j-22);
130                 end loop;
131
132             else
133                 for j in 0 to 4 loop
134                     row(j) <= tagB(j);
135                 end loop;
136                 for j in 5 to 16 loop
137                     row(j) <= dataB(j-5);
138                 end loop;
139
140                 for j in 17 to 21 loop
141                     row(j) <= tag(j-17);
142                 end loop;
143                 for j in 22 to 33 loop
144                     row(j) <= data_in(j-22);
145                 end loop;
146             end if;
147              data_array(to_integer(unsigned(index))) <= row;
148         end if;
149
150     end process;
151
152 end Behavioral;
```

شکل ۶ : کد ama_cache

۵. ama_cache_ram

در این بخش ما به توضیح ماژول پایانی پروژه می پردازیم به نام ama_cache_ram در این ماژول ما از دو ماژول قبلی استفاده کرده ایم در واقع این ماژول برای اتصال بین ama_ram و ama_cache طراحی شده است. این ماژول از سه سیگنال ورودی و دو سیگنال خروجی تشکیل شده است. سیگنال های ورودی ما clk, rd, address هستند که در مورد آنها توضیحات لازم داده شد و اما سیگنال های خروجی که result و hit هستند که یکی دیتای خواسته شده در آدرس مورد نظر است و دیگری وضعیت cache را نشان می دهد.

باتوجه به شکل زیر ابتدا هر دو کامپوننت ama_ram و ama_cache تعریف شده اند و از آنها در ادامه بین begin و بلاک process یک نمونه از هر کدام ساخته شده است. سیگنال های میانی مورد نظر برای اتضال بین آنها نیز تعریف شده به نام های :

cache_data_out, ram_data_out, hit_miss_result, red, wrt, data

که کاربرد هر کدام در ادامه توضیح داده خواهد شد.

Cache_data_out : همان طور که از نام آن پیداست این سیگنال دیتای خروجی از cache را دار است.

Ram_data_out : همان طور که از نام آن پیداست این سیگنال دیتای خروجی از ram را دارا است.

Hit_miss_result : این سیگنال وضعیت hit و یا miss شدن درون کامپوننت cache را نشان می دهد. اگر صفر باشد یعنی cache در آدرس مورد نظر دیتای ما را نداشته و نیاز است از ram درون آن نوشته شود. در توضیحات بخش cache گفته شد که اگر دیتای مورد نظر یافت نشد یعنی زمانی که سیگنال wrt کامپیوننت cache که مربوط به نوشتن در آن است یک شد در خانه مورد نظر نوشته شود. اگر سیگنال hit_miss_result یک باشد یعنی دیتا درون cache بوده و آن را به result (خروجی نهایی ماژول ama_cache_ram) پاس می دهد.

Red : این سیگنال مربوط به سیگنال خواندن از ram است که درون این کامپوننت تعبیه شده و توضیحات آن داده شد.

Wrt : این سیگنال مربوط به سیگنال نوشتن درون cache است که اجازه نوشتن درون cache را چک می کند.

Data : این سیگنال برای مقدار دهی به سیگنال data_in کامپوننت cache است. سیگنالی که دیتای جدید را از رم درون cache می نویسد. مقدار ابتدایی آن ۱۲ بیت صفر است زیرا ممکن است نیازی به نوشتن درون cache نباشد و باید این سیگنال مقدار اولیه داشته باشد.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_ARITH.ALL;
4  use IEEE.STD_LOGIC_UNSIGNED.ALL;
5
6  entity ama_cache_ram is
7      Port ( clk : in STD_LOGIC;
8         rd : in STD_LOGIC;
9             address : in STD_LOGIC_VECTOR (9 downto 0);
10            result : out STD_LOGIC_VECTOR (11 downto 0);
11            hit : out STD_LOGIC
12            );
13 end ama_cache_ram;
14
15 architecture Behavioral of ama_cache_ram is
16
17 signal cache_data_out : std_logic_vector(11 downto 0);
18
19 component ama_cache is
20     port ( address : in STD_LOGIC_VECTOR(9 downto 0);
21            wr : in STD_LOGIC;
22            clk : in STD_LOGIC;
23            data_in : in STD_LOGIC_VECTOR(11 downto 0);
24            data_out : out STD_LOGIC_VECTOR(11 downto 0);
25             hit_miss: out STD_LOGIC
26            );
27 end component;
28
29 signal ram_data_out : std_logic_vector(11 downto 0);
30
31 component ama_ram is
32     port ( rd : in STD_LOGIC;
33        clk : in STD_LOGIC;
34           address : in STD_LOGIC_VECTOR(9 downto 0);
35            data_out : out STD_LOGIC_VECTOR(11 downto 0)
36            );
37 end component;
38
39 signal hit_miss_result: STD_LOGIC := '0';
40
41 signal red : STD_LOGIC := '0';
42 signal wrt : STD_LOGIC := '0';
43 signal data : STD_LOGIC_VECTOR(11 downto 0) := "000000000000";
44
45 begin
46
47 cache: ama_cache port map (
48         address => address,
```

شکل ۷ : کد ama_cache_ram

در ادامه درون بلاک process با چک کردن سیگنال hit_miss_result متوجه وضعیت کامپوننت cache و تصمیم بر نوشتن درون آن یا دادن خروجی آن به خروجی اصلی می شود.

در صورت صفر بودن مقدار red و wrt برای خواندن و نوشتن درون رم و cache به یک تبدیل می شوند و دیتای خروجی ram درون سیگنال data که به data_in درون cache اتصال دارد ریخته می شود و خروجی مورد نظر به result داده می شود. در غیر این صورت خروجی cache به result داده می شود یعنی که دیتا درون cache بوده است.

```
49          wr => wrt,
50          clk => clk,
51          data_in => data,
52          data_out => cache_data_out,
53          hit_miss => hit_miss_result
54  );
55
56  ram : ama_ram port map(
57          rd => red,
58          clk => clk,
59          address => address,
60          data_out => ram_data_out
61  );
62
63  process (clk) begin
64     if (hit_miss_result = '0') then
65        red <= '1';
66        data <= ram_data_out;
67        wrt <= '1';
68        result <= ram_data_out;
69     else
70        result <= cache_data_out;
71
72     end if;
73
74  end process;
75
76  end Behavioral;
```

شکل ۸ : کد ama_cache_ram

۶. تست بنچ

در فایل تست بنچ خود ماژول ama_cache_ram را استفاده کردیم و از آن یک نمونه ساختیم.

در ابتدا آدرس خانه ی که می دانیم درون آن نوشته نشده است را به ماژول مورد نظر می دهیم و تا درون cache نوشته شود و سپس آدرس همان خانه را به آن می دهیم تا این دفعه از cache دیتای خواسته شده را به ما بدهد.

```
-- TestBench Template

  LIBRARY ieee;
  USE ieee.std_logic_1164.ALL;
  USE ieee.numeric_std.ALL;

  ENTITY testbench IS
  END testbench;

  ARCHITECTURE behavior OF testbench IS

  -- Component Declaration
          COMPONENT ama_cache_ram is
          Port ( clk : in STD_LOGIC;
           rd : in STD_LOGIC;
           address : in STD_LOGIC_VECTOR (9 downto 0);
           result : out STD_LOGIC_VECTOR (11 downto 0)
           );
          END COMPONENT;


--Inputs
signal clk : std_logic;
signal rd : std_logic;
signal address :  std_logic_vector(9 downto 0);


--Outputs
signal result : std_logic_vector(11 downto 0) := "000000000000" ;

-- Clock period definitions
constant clk_period : time := 10 ns;


BEGIN

-- Component Instantiation
cache_ram: ama_cache_ram PORT MAP(
      clk => clk,
      rd => rd,
      address => address,
      result => result
 );
```

شکل ۹: کد تست بنچ

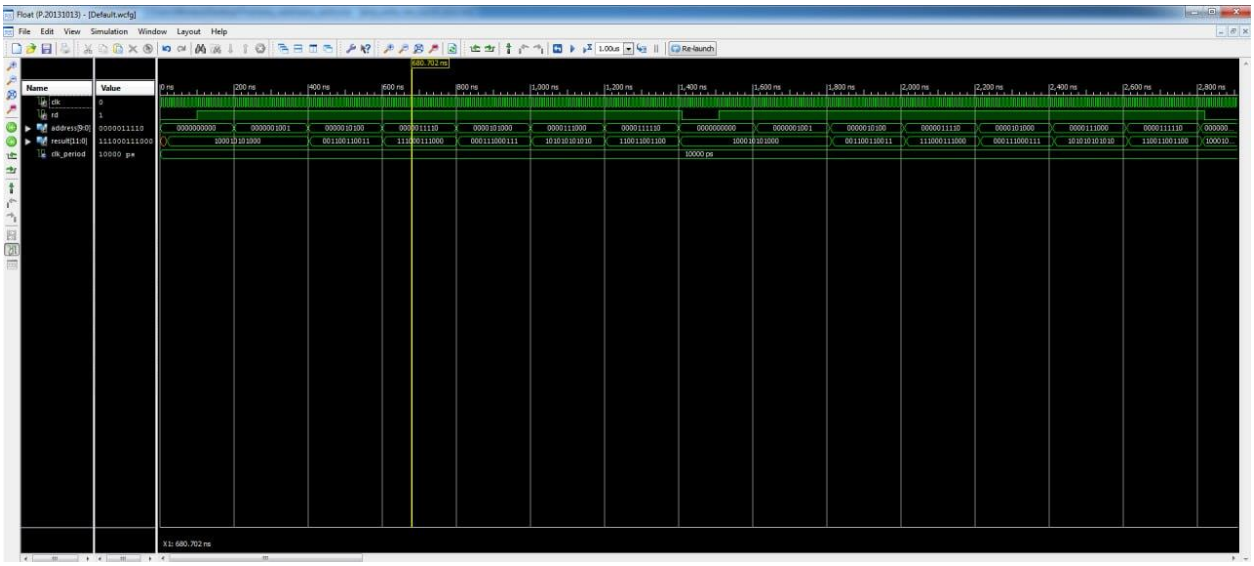
```
45    clk_process :process
46    begin
47       clk <= '0';
48       wait for clk_period/2;
49       clk <= '1';
50       wait for clk_period/2;
51    end process;
52
53   --  Test Bench Statements
54      tb : PROCESS
55      BEGIN
56
57         rd <= '0';
58         address <= "0000000000";
59
60         wait for 100 ns;
61
62         rd <= '1';
63         address <= "0000000000";
64
65         wait for 100 ns;
66
67         rd <= '1';
68         address <= "0000001001";
69
70         wait for 100 ns;
71
72         rd <= '1';
73         address <= "0000001001";
74
75         wait for 100 ns;
76
77         rd <= '1';
78         address <= "0000010100";
79
80         wait for 100 ns;
81
82         rd <= '1';
83         address <= "0000010100";
84
85         wait for 100 ns;
86
87         rd <= '1';
88         address <= "0000011110";
```

شکل ۱۰: کد تست بنچ

```
        wait for 100 ns;

        rd <= '1';
        address <= "0000011110";

        wait for 100 ns;

        rd <= '1';
        address <= "0000101000";

         wait for 100 ns;

        rd <= '1';
        address <= "0000101000";

        wait for 100 ns;

        rd <= '1';
        address <= "0000111000";

        wait for 100 ns;

        rd <= '1';
        address <= "0000111000";

        wait for 100 ns;

        rd <= '1';
        address <= "0000111110";

        wait for 100 ns;

        rd <= '1';
        address <= "0000111110";

        wait for 100 ns;

        rd <= '1';
        address <= "0000000000";


        wait for clk_period; -- wait until global set/reset completes


    END PROCESS tb;
--  End Test Bench
```

شکل ۱۱: تست بنچ

در شکل موج مشخص است که با توجه به آدرس داده شده دیتا بازگردانده می شود



شکل ۱۲ : شکل موج

پایان