بسمه تعالی

VHDL

Final Report of Project

محمد میثم الهام بخش

۴۰۱۶۱۱۲۹۲

۱)توصیف کد :

توجه شود دیتاشیت آی سی ۴۰۲۰و۴۰۲۱و۴۰۲۲ یکسان میباشد و تفاوت جزئی دارند. بنابراین در این گزارش بیان به جای ۴۰۲۲ ، ۴۰۲۰ بیان می‌شود.

برای طراحی آی سی AD4020 کار طراحی در فایلی به نام AD4020 انجام شده است که از ماژول SPI_Abstract درSPI_Command و از ماژول SPI_Command در AD4020 استفاده شده است .

در طراحی این ماژول به کمک ماژول SPI_Command دستورات اولیه داده می شود و بعد از اعمال این دستورات آی سی شروع به کار میکند و دیتای ADC را بیت به بیت دریافت کرده به شکل ۲۰ بیتی نمایش میدهد .

برای توصیف کد، ابتدا به بررسی فایل اصلی یعنی AD4020 خواهیم پرداخت تا نحوه عملکرد کد به طور کامل شفاف شود در ادامه توضیحاتی درباره ماژول های زیر مجموعه این فایل که از آنها استفاده شده یعنی SPI_Command و SPI_Abstract داده خواهد شد.

```
1
2  library IEEE;
3  use IEEE.std_logic_1164.all;
4  use IEEE.numeric_std.all;
5
6  entity FE_AD4020 is
7    generic(
8      input_clk_freq        : natural range 1 to 1_000_000_000 := 100_000_000
9    );
10    port (
11     sys_clk               : in  std_logic;
12     sys_reset_n           : in  std_logic;
13     spi_clk               : in  std_logic;
14     AD4020_data_out       : out std_logic_vector(19 downto 0);
15     AD4020_valid_out      : out std_logic;
16     AD4020_MISO_in        : in  std_logic;
17     AD4020_MOSI_out       : out std_logic;
18     AD4020_SCLK_out       : out std_logic;
19     AD4020_CONV_out       : out std_logic
20    );
21 end entity FE_AD4020;
```

در ابتدای این فایل به معرفی متغیر ها و ثابت ها پرداخته شده است همانطور که مشاهده می‌کنید، چند ورودی و چند خروجی داریم که مهم ترین خروجی ما از نوع وکتور۲۰ بیتی است که همان دیتا خروجی ما می‌باشد.

همچنین در این پروژه فرکانس کاری این آی سی ۱۰۰ مگابیت در نظر گرفته شده است که به صورت ثابت تعریف شده است.

```
architecture Behavioral of FE_AD4020 is

  component spi_commands is
     generic(

     command_used_g          : std_logic  := '1';
     address_used_g          : std_logic  := '0';
     command_width_bits_g    : natural    := 8;
     address_width_bits_g    : natural    := 8;
     data_width_bits_g       : natural    := 8;
    output_bits_g            : natural    := 24;
     cpol_cpha               : std_logic_vector(1 downto 0) := "10"
     );
      port(
         clk              :in std_logic;
         rst_n            :in std_logic;

         command_in      : in  std_logic_vector(command_width_bits_g-1 downto 0);
         address_in      : in  std_logic_vector(address_width_bits_g-1 downto 0);
         master_slave_data_in      : in   std_logic_vector(data_width_bits_g-1 downto 0);
         master_slave_data_rdy_in  : in   std_logic;
         master_slave_data_ack_out : out  std_logic;
         command_busy_out          : out  std_logic;
         command_done              : out  std_logic;
         slave_master_data_out     : out std_logic_vector(output_bits_g-1 downto 0);
         slave_master_data_ack_out : out std_logic;
         miso           : in  std_logic;
         mosi           : out  std_logic;
         sclk           : out  std_logic;
         cs_n           : out  std_logic
      );
   end component;
```

اکنون معماری اصلی که از نوع رفتاری است ایجاد میشود در همان ابتدا ماژول SPI_Command تعریف می‌شود چون ما قبل از استفاده و دریافت خروجی از این ماژول نیاز داریم دستورات لازم را به آی سی بدهیم تا تعیین کنیم عملکرد آی سی باید چگونه باشد .

این بخش ابتدایی که دادن دستورات به آی سی هست به یک ماژول زیر مجموعه که SPI_Command سپرده شده است و به کمک این ماژول انجام می‌شود ما در فایل اصلی که AD4020 نام دارد فقط باید این ماژول را معرفی کرده و ورودی و خروجی های آن را طبق فایل اصلی که AD4020 است map انجام دهیم.

```

   signal   AD4020_data_r                : std_logic_vector(31 downto 0);
   signal   AD4020_spi_command           : std_logic_vector(1 downto 0);
   signal   AD4020_spi_register_address  : std_logic_vector(5 downto 0);
   signal   AD4020_spi_write_data        : std_logic_vector(7 downto 0);
   signal   AD4020_spi_write_data_rdy    : std_logic;
   signal   AD4020_spi_busy              : std_logic;
   signal   AD4020_spi_sclk              : std_logic;
   signal   AD4020_spi_sclk_delayed      : std_logic;
   signal   AD4020_conv                  : std_logic;
   signal   AD4020_valid                 : std_logic := '0';
   signal   AD4020_data                  : std_logic_vector(19 downto 0) := "00000000000000000000";

   constant control_reg                  : std_logic_vector(5 downto 0) := "010100";
   constant write_data                   : std_logic := '0';
   constant read_data                    : std_logic := '1';
   signal data_ready                     : std_logic := '0';

```

```
73   constant WEN                        : std_logic := '0';
74   constant RESERVED                   : std_logic := '0';
75   constant TURBO                      : std_logic := '0';
76   constant OV                         : std_logic := '0';
77   constant HIGHZ                      : std_logic := '0';
78   constant SPAN_COMPRESSION           : std_logic := '0';
79   constant ENABLE_STATUS_BITS         : std_logic := '0';
80   constant reg_config                 : std_logic_vector(7 downto 0)  := RESERVED & RESERVED & RESERVED & ENABLE_STATUS_BITS &
81                                                                          SPAN_COMPRESSION & HIGHZ & TURBO & OV;
82
```

در ادامه سیگنال ها و ثوابت مورد استفاده تعریف شده است برخی از این ثوابت داخل ماژول SPI_Command مورد استفاده قرار میگیرند چون نشان دهنده بیت های دستورات ما و یا نتیجه اعمال دستورات ما می‌باشند. به‌عنوان مثال OV,TURBO,write_data,read_data از این دست ثابت ها می‌باشند.

ما به طور دیفالت مقادیر OV,TURBO,HIGHZ,SPAN_COMPARISON را برابر صفر یعنی غیرفعال قرار می‌دهیم چون این مقادیر نشان دهنده ویژگی های فیزیکی می‌باشند و ما در اینجا فقط در محیط شبیه سازی در حال ساخت این آی سی هستیم بنابراین این بیت ها تعریف می‌شوند و در صورتی که قرار شود به صورت فیزیکی مورد تست قرار دهند این امکان قرار داده شده است تا تغییر این بیت ها در شرایط واقعی اعمال شوند.

```
85   type state_type is (
86     init_adc,
87     spi_init_load,
88     spi_data_load,
89     spi_read_data_busy,
90     spi_read_data_start,
91     spi_write_init_busy,
92     spi_write_init_start,
93     spi_read_data_finish
94   );
95
96   signal state : state_type;
97
98   signal sclk_en : boolean := true;
99   signal spi_clk_en : boolean := false;
100  signal read_complete : boolean := false;
101
102  signal one_cycle_delay : boolean := false;
103  signal SDI_is_low : boolean := true;
104  signal tquiet1_delay_complete : boolean := false;
105  signal tquiet2_delay_complete : boolean := false;
106  signal gpio_init_complete : boolean := false;
107
108  signal mosi_ctrl : boolean := true;
109  signal spi_mosi  : std_logic;
110  signal read_mosi : std_logic;
```

در ادامه ۸ استیت یا وضعیت تعریف می‌شود که به وسیله آن کار های لازم در هر وضعیت انجام میشود.

این استیت ها شامل :

init_adc , spi_init_load , spi_data_load , spi_read_data_busy , spi_read_data_start, spi_write_init_busy , spi_write_init_start , spi_read_data_finish

می‌باشد. در ادامه نیز تعدادی سیگنال که برای اعمال تغییرات لازم روی سیگنال خروجی مطابق با خواسته کاربر می‌باشد تعریف شده است.

```
112  begin
113
114    spi_AD4020: spi_commands
115    generic map (
116      command_used_g            => '1',
117      address_used_g            => '1',
118      command_width_bits_g      =>  2,
119      address_width_bits_g      =>  6,
120      data_width_bits_g         =>  8,
121      output_bits_g             =>  16,
122      cpol_cpha                 => "00"
123    )
124    port map (
125      clk                       => spi_clk ,
126      rst_n                     => sys_reset_n,
127      command_in                => AD4020_spi_command,
128      address_in                => AD4020_spi_register_address,
129      master_slave_data_in      => AD4020_spi_write_data,
130      master_slave_data_rdy_in  => AD4020_spi_write_data_rdy,
131      master_slave_data_ack_out => open,
132      command_busy_out          => AD4020_spi_busy,
133      command_done              => open,
134      slave_master_data_out     => open,
135      slave_master_data_ack_out => open,
136      miso                           => AD4020_MISO_in,
137      mosi                           => spi_mosi,
138      sclk                           => AD4020_spi_sclk,
139      cs_n                           => open
140    );
141
```

با شروع برنامه همانطور که مشاهده می‌کنید در همان ابتدا ماژول SPI_Command مقدار دهی می‌شود تا بتواند فرایند آغازین که همان دستورات اولیه است را انجام دهد.

در این پروژه از ۳ پروسس همزمان استفاده شده است تا بتوان به بهترین شکل ممکن به پیاده سازی این آی‌سی پرداخت. پروسس اول برای مدیریت استیت ها می‌باشد. پروسس دوم برای انجام کار ها در هر استیت و پروسس سوم هم برای اعمال تغییرات در سیگنال خروجی متناسب با کانفیگی که کاربر انجام می‌دهد، می‌باشد.

```
143    process (spi_clk, sys_reset_n)
144    begin
145      if sys_reset_n = '0' then
146        state     <= init_ADC;
147        init_done <= '0';
148      elsif (rising_edge(spi_clk)) then
149        case state is
150          when init_ADC =>
151            if init_done = '0' then
152              if gpio_init_complete then
153              state <= spi_init_load;
154              init_done <= '1';
155              end if;
156            elsif init_done = '1' then
157              state <= spi_read_data_start;
158            else
159              state <= init_ADC;
160            end if;
161          when spi_init_load =>
162            state <= spi_write_init_start;
163          when spi_write_init_start =>
164            if AD4020_spi_busy = '1' then
165              state <= spi_write_init_busy;
166            else
167              state <= spi_write_init_start;
168            end if;
169          when spi_write_init_busy =>
170            if AD4020_spi_busy = '1' then
171              state <= spi_write_init_busy;
```

```
          else
            state <= spi_read_data_start;
            mosi_ctrl <= false;
            sclk_en <= false;
          end if;
        when spi_read_data_start =>
          if one_cycle_delay then
            state <= spi_read_data_busy;
          else
            state <= spi_read_data_start;
          end if;
        when spi_read_data_busy =>
          if read_complete then
            state <= spi_read_data_finish;
          else
            state <= spi_read_data_busy;
          end if;
        when spi_read_data_finish =>
          if tquiet2_delay_complete then
            state <= spi_read_data_start;
          end if;
        when others =>
          state <= init_ADC;
      end case;
    end if; end process;
```

همانطور که در کد بالا مشاهده می‌کنید در پروسس اول متناسب با استیتی که قرار داریم و وضعیت سیگنال ها تصمیم گرفته می‌شود برنامه وارد کدام استیت در سیکل بعدی شود .

در واقع در این پروسس ماشین حالتی طراحی کردیم که به وسیله آن تمام حالات ممکن مدیریت شود و کد بتواند به بهترین شکل عمل کند. توجه شود این پروسس حساس به کلاک SPI و ریست می‌باشد

```
process (spi_clk)
  variable bits_left : integer := 20;
begin
  if (rising_edge(spi_clk)) then
    case state is
      when init_ADC =>
      when spi_init_load =>
        AD4020_conv                    <= '0';
        AD4020_spi_command             <= WEN & write_data;
        AD4020_spi_register_address    <= control_reg;
        AD4020_spi_write_data           <= reg_config;
      when spi_write_init_start =>
        AD4020_spi_write_data_rdy <= '1';
        AD4020_conv               <= '1';
      when spi_write_init_busy =>
        AD4020_spi_write_data_rdy <= '0';
      when spi_read_data_start =>
        if SDI_is_low then
          read_mosi <= '1';
        else
          AD4020_conv <= '0';
        end if;
        AD4020_valid <= '0';
        if tquiet1_delay_complete then
          read_mosi <= '0';
          one_cycle_delay <= true;
          if one_cycle_delay then
            spi_clk_en <= true;
          end if;
        end if;
      when spi_read_data_busy =>
        one_cycle_delay <= false;
        if not(bits_left > 0) then
          spi_clk_en <= false;
        end if;
      when spi_read_data_finish =>
        AD4020_valid <= '1';
        if tquiet2_delay_complete then
          AD4020_conv <= '1';
        end if;
      when others =>
    end case;
  end if;
```

```
243     if (falling_edge(spi_clk)) then
244       case state is
245         when spi_read_data_busy =>
246           if bits_left > 0 then
247             AD4020_data(bits_left - 1) <= AD4020_MISO_in;
248             bits_left := bits_left - 1;
249           else
250             read_complete <= true;
251           end if;
252         when spi_read_data_finish =>
253           read_complete <= false;
254           bits_left := 20;
255         when others =>
256       end case;
257     end if;
258   end process;
```

در این پروسس بر اساس بالارونده یا پایین رونده بودن کلاک SPI در هر استیت کار های لازم انجام می‌شود.

در لبه های بالا رونده بعد از استیت init_ADC با ورود به استیت spi_init_load دستورات داده شده لود می‌شود و سیستم آماده شروع به کار می‌شود. در ادامه با ورود به استیت spi_write_init_start دستورات لود شده نوشته می‌شود. در این لحظه وارد استیت spi_write_init_busy شده و منتظر می‌شویم دستورات داده شده اعمال شوند. در استیت spi_read_data_start با توجه به دستورات اعمال شده به خواندن دیتای ۲۰ بیتی ADC خواهیم پرداخت و در این حین وارد استیت spi_read_data_busy در نهایت با اتمام خوانده شدن ۲۰ بیت وارد استیت آخر که spi_read_data_finish است، می‌شویم .

در لبه های پایین رونده نیز در استیت spi_read_data_start مشغول خواندن بیت ها شده و در استیت spi_read_data_finish کانتر را به عدد ۲۰ بر میگردانیم تا آماده خوانده شدن ۲۰ بیتی بعدی شویم.

```
261   delays : process(spi_clk, state)
262   constant tquiet1_ns : natural := 100;
263   constant tquiet2_ns : natural := 100;
264   constant gpio_init_ns : natural := 200;
265   constant period_ns : natural range 1 to 1000 := 1_000_000_000 / input_clk_freq;
266   constant tquiet1_cycles : natural := (((tquiet1_ns + period_ns - 1) / period_ns)-2);
267   constant tquiet2_cycles : natural := (((tquiet2_ns + period_ns - 1) / period_ns)-3);
268   constant gpio_init_cycles : natural := (((gpio_init_ns) / period_ns)-3);
269
270   variable counter : natural := 0;
271   begin
272     if rising_edge(spi_clk) then
273       case state is
274         when spi_read_data_start =>
275           if SDI_is_low then
276             SDI_is_low <= false;
277           elsif counter = tquiet1_cycles or tquiet1_delay_complete then
278             tquiet1_delay_complete <= true;
279             counter := 0;
280           else
281             counter := counter + 1;
282           end if;
283         when init_ADC =>
284           if counter = gpio_init_cycles or gpio_init_complete then
285             gpio_init_complete <= true;
286             counter := 0;
287           else
288             counter := counter + 1;
289           end if;
```

```
290         when spi_read_data_finish =>
291           if counter = tquiet2_cycles or tquiet2_delay_complete then
292             tquiet2_delay_complete <= true;
293             counter := 0;
294           else
295             counter := counter + 1;
296           end if;
297         when others =>
298           SDI_is_low <= true;
299           counter := 0;
300           tquiet1_delay_complete <= false;
301           tquiet2_delay_complete <= false;
302           gpio_init_complete <= false;
303       end case;
304     end if;
305   end process;
```

در پروسس آخر به اعمال دیلی های اولیه و انتهایی و دیلی ناشی از تنظیمات اولیه ورودی خروجی ها پرداخته می‌شود. با توجه به مقدار تاخیری که نیاز هست تا آی سی ما نتیجه را تولید کند میتوان تاخیر ایجاد کرد تا بتوان بدون از دست دادن حتی یک بیت دیتا نتیجه را بصورت تمام و کمال بدست آورد.

```
306
307   AD4020_SCLK_out <=AD4020_spi_sclk when sclk_en else spi_clk when spi_clk_en else '0';
308   AD4020_CONV_out <= AD4020_conv;
309   AD4020_data_out <= AD4020_data;
310   AD4020_MOSI_out <= spi_mosi when mosi_ctrl else read_mosi;
311   AD4020_valid_out <= AD4020_valid;
312
313 end architecture Behavioral ;
```

در انتهای این فایل نیز خروجی های بدست آمده از ۳ پروسس فوق داخل خروجی های اصلی این برنامه انتقال داده شده است.

اکنون به بررسی فایل SPI_Command پرداخته خواهد شد که وظیفه اعمال دستورات ابتدایی را دارد.

```
36  architecture Behavioral of spi_commands is
37
38  function to_natural( sl_in : std_logic ) return natural is
39  begin
40        if sl_in = '1' then
41        return 1;
42     else
43        return 0;
44     end if;
45  end function;
46
47  component spi_abstract is
48   generic (
49        cpol_cpha : std_logic_vector(1 downto 0) := "00";
50        data_width: natural := output_bits_g
51   );
52     port(
53        clk     :in std_logic;
54          rst_n      :in  std_logic;
55
56        mosi_data_i         : in std_logic_vector(data_width-1 downto 0);
57        miso_data_o         : out std_logic_vector(data_width-1 downto 0);
58        mosi_data_valid_i    :in   std_logic;
59        mosi_data_ack_o     :out   std_logic;
60        miso_data_valid_o    :out  std_logic;
61
62          miso            :in   std_logic;
63        mosi           :out  std_logic;
64        sclk           :out  std_logic;
65        cs_n           :out  std_logic
66
67        );
68  end component;
```

بخش Entity این فایل چون در فایل اصلی تعریف شد اینجا تکرار نمی‌شود و فورا سراغ معماری رفتاری آن خواهیم رفت. در این فایل از ماژول SPI_Abstract که پروتکل SPI خام ما می‌باشد استفاده می‌شود که بعد از این فایل به توصیف آن نیز خواهیم پرداخت.

```
    type SPI_STATE is   (
    SPI_STATE_WAIT,
    SPI_STATE_PAYLOAD,
    SPI_STATE_COMMAND_DONE,
    SPI_STATE_COMMAND_DONE_SIGNAL
    );

  signal cur_spi_state  : SPI_STATE;

  constant xfer_len : natural   :=  to_natural(command_used_g) * command_width_bits_g +
                                    to_natural(address_used_g) * address_width_bits_g +
                                    data_width_bits_g;

  signal command_signal : std_logic_vector(command_width_bits_g-1 downto 0);
  signal address_signal : std_logic_vector(address_width_bits_g-1 downto 0);
  signal data_signal :std_logic_vector(data_width_bits_g-1 downto 0);
  signal mosi_data_valid_spi : std_logic;
  signal mosi_data_ack_spi : std_logic;
  signal mosi_data_ack_spi_follower : std_logic;
  signal miso_data_valid_spi : std_logic;
  signal miso_data_spi : std_logic_vector(xfer_len-1 downto 0);
  signal mosi_data_spi : std_logic_vector(xfer_len-1 downto 0);
  signal miso_byte_ack_count : unsigned (7 downto 0);
  signal slave_master_data_ack_out_en : std_logic;
  signal cs_n_signal : std_logic;
  signal cs_n_signal_follower : std_logic;
  signal data_select : std_logic_vector(1 downto 0) := command_used_g & address_used_g;
```

مانند فایل قبل در این فایل نیز استیت هایی تعریف شده است در ادامه سیگنال های مورد نیاز برای اعمال دستورات که وظیفه اصلی این ماژول می‌باشد تعریف شده اند.

```
begin
  slave_master_data_out <= miso_data_spi;
  cs_n <= cs_n_signal;

  spi_slave: spi_abstract
  generic map(
      cpol_cpha => cpol_cpha,
      data_width => output_bits_g
  )
   port map(
      clk       => clk,
        rst_n         => rst_n,

      mosi_data_i     =>  mosi_data_spi,
      miso_data_o     => miso_data_spi,
      mosi_data_valid_i  => mosi_data_valid_spi,
      mosi_data_ack_o     => mosi_data_ack_spi,
      miso_data_valid_o  => miso_data_valid_spi,

        miso            => miso,
      mosi            => mosi,
      sclk            => sclk,
      cs_n            => cs_n_signal

      );


```

با شروع برنامه در همان ابتدا فایل مربوط به SPI یعنی SPI_Abstract با توجه به ورودی خروجی های این فایل map شده است تا مورد استفاده قرار گیرد.

```
126 spi_state_state_machine: process(clk, rst_n)
127 begin
128   if rst_n = '0' then
129     cur_spi_state <= SPI_STATE_WAIT;
130
131   elsif rising_edge(clk) then
132     case cur_spi_state is
133       when SPI_STATE_WAIT =>
134       if (master_slave_data_rdy_in = '1') then
135           cur_spi_state <= SPI_STATE_PAYLOAD;
136       else
137         cur_spi_state <= SPI_STATE_WAIT;
138       end if;
139
140       when SPI_STATE_PAYLOAD =>
141         cur_spi_state <= SPI_STATE_COMMAND_DONE;
142
143       when SPI_STATE_COMMAND_DONE   =>
144         if (cs_n_signal_follower /= cs_n_signal) and (cs_n_signal = '1') then
145             cur_spi_state <= SPI_STATE_COMMAND_DONE_SIGNAL;
146         else
147           cur_spi_state <= SPI_STATE_COMMAND_DONE;
148         end if;
149
150       when SPI_STATE_COMMAND_DONE_SIGNAL =>
151         cur_spi_state <= SPI_STATE_WAIT;
152       when others =>
153     end case;
154   end if;
155 end process spi_state_state_machine;
```

پروسس اول مربوط به مدیریت استیت ها یا وضعیت های کار این ماژول می‌باشد. و در واقع یک ماشین حالت می‌باشد که تمام حالات ممکن را مدیریت می‌کند تا برنامه ما به بهترین صورت اجرا شود.

```
157 spi_command_state_machine:  process (clk, rst_n)
158 begin
159   if rst_n = '0' then
160
161     command_signal <= (others => '0');
162     address_signal <= (others => '0');
163     data_signal <= (others => '0');
164     mosi_data_valid_spi <= '0';
165     mosi_data_spi <= (others => '0');
166     mosi_data_ack_spi_follower <= '0';
167     master_slave_data_ack_out <= '0';
168     cs_n_signal_follower <= '1';
169
170   elsif rising_edge(clk) then
171
172     mosi_data_valid_spi <= '0';
173     master_slave_data_ack_out <= '0';
174
175     case cur_spi_state is
176
177     when SPI_STATE_WAIT =>
178       mosi_data_ack_spi_follower <= '0';
179       if (master_slave_data_rdy_in = '1') then
180         command_signal <= command_in;
181         address_signal <= address_in;
182         data_signal <= master_slave_data_in;
183       end if;
184
185     when SPI_STATE_PAYLOAD   =>
186       if (mosi_data_ack_spi_follower /= mosi_data_ack_spi) then
187         mosi_data_ack_spi_follower <= mosi_data_ack_spi;
188
189         case data_select is
190           when "00" =>
191             mosi_data_spi <= data_signal;
192
193           when "01" =>
194             mosi_data_spi <= address_signal & data_signal;
195
196           when "11" =>
197             mosi_data_spi <= command_signal & address_signal & data_signal;
198
199           when "10" =>
200             mosi_data_spi <= command_signal & data_signal;
201           when others =>
202         end case;
```

```
203
204           mosi_data_valid_spi <= '1';
205         else
206             mosi_data_valid_spi <= '0';
207         end if;
208
209         when SPI_STATE_COMMAND_DONE    =>
210           if (cs_n_signal_follower /= cs_n_signal) then
211             cs_n_signal_follower <= cs_n_signal;
212           end if;
213
214         when SPI_STATE_COMMAND_DONE_SIGNAL    =>
215
216         end case ;
217     end if ;
218   end process spi_command_state_machine ;
219
```

پروسس فوق که پروسس اصلی این ماژول می‌باشد مربوط به اعمال دستوراتی است که در ابتدای برنامه داده می‌شود در این پروسس با توجه به دستورات داده شده تغییرات لازم اعمال شده است. به عنوان مثال دستورات می‌تواند به فرمت ۸ ، ۱۶ و یا ۲۴ بیتی داده شوند در این پروسس شکل دستورات ارسالی و خروجی حاصله از آن ها اعمال می‌شود.

```
221   spi_state_output:  process (cur_spi_state)
222   begin
223
224   command_busy_out <= '1';
225   command_done <= '0';
226
227   case cur_spi_state is
228
229     when SPI_STATE_WAIT =>
230       command_busy_out <= '0';
231     when SPI_STATE_PAYLOAD =>
232     when SPI_STATE_COMMAND_DONE =>
233     when SPI_STATE_COMMAND_DONE_SIGNAL =>
234       command_done <= '1';
235
236
237   end case;
238
239   end process spi_state_output ;
```

پروسس فوق به این منظور است که در هر یک از استیت ها کدام سیگنال ها نیاز به فعال یا غیرفعالسازی دارند.

```
242   slave_master_data_out_handler : process(clk,rst_n)
243   begin
244   if rst_n = '0' then
245     miso_byte_ack_count <= to_unsigned(0,miso_byte_ack_count'length);
246     slave_master_data_ack_out_en <= '0';
247     slave_master_data_ack_out <= '0';
248   elsif rising_edge(clk) then
249
250       if (cur_spi_state = SPI_STATE_WAIT) then
251       miso_byte_ack_count <= to_unsigned(0,miso_byte_ack_count'length);
252       slave_master_data_ack_out_en <= '0';
253       elsif ( miso_data_valid_spi = '1') then
254       miso_byte_ack_count <= miso_byte_ack_count + 1;
255       end if;
256
257
258     if(command_used_g = '1' and address_used_g = '1') then
259       if ( miso_byte_ack_count = to_unsigned(command_width_bits_g + address_width_bits_g,miso_byte_ack_count'length)) then
260         slave_master_data_ack_out_en <= '1';
261       end if;
262     elsif(command_used_g = '1' and address_used_g = '0') then
263      if ( miso_byte_ack_count = to_unsigned(command_width_bits_g,miso_byte_ack_count'length)) then
264         slave_master_data_ack_out_en <= '1';
265       end if;
266     end if;
```

```
267
268   if(command_used_g = '0' and address_used_g = '0') then
269     slave_master_data_ack_out_en <= '1';
270   end if;
271
272   if (slave_master_data_ack_out_en = '1') then
273   slave_master_data_ack_out <= miso_data_valid_spi;
274   end if;
275
276 end if;
277 end process slave_master_data_out_handler;
278
279
280 end Behavioral;
```

در پروسس پایانی با توجه به دستورات اعمال شده پاسخی که این آی سی به دستورات اعمال شده می‌دهد ساخته می‌شود.

به این ترتیب این فایل که وظیفه انجام دستورات ابتدایی را داشت، و در فایل اصلی در مرحله اول فعال می‌شد با شروع کار برنامه دستورات را ارسال می‌کند پاسخ آن را دریافت می‌کند و تغییرات لازم اعمال می‌شود. در ادامه هم برنامه در فایل اصلی به مراحل بعدی که منجر به بدست آمدن خروجی ۲۰ بیتی ADC می‌شود، می‌پردازد.

در نهایت به ماژول آخر که همان SPI_Abstract است، پرداخته خواهد شد. این ماژول در واقع SPI را برای ما پیاده سازی می‌کند و ما به کمک این ماژول یک پروتکل SPI را خواهیم داشت . همانطور که در ابتدای توضیحات بیان شد، این پروژه یک فایل اصلی به نام AD4020 دارد این فایل یک ماژول زیر شاخه به نام SPI_Command دارد و این ماژول نیز یک ماژول زیر شاخه به نام SPI_Abstract دارد که ما ابتدا فایل اصلی سپس ماژول مربوط به اعمال دستورات را تشریح کردیم اکنون این فایل که پیاده سازی شده پروتکل SPI می‌باشد را تشریح خواهیم کرد.

```
27  architecture Behavioral of spi_abstract is
28
29    type     spi_state   is
30   (
31     SPI_WAIT,
32     SPI_CS,
33     SPI_SHIFT,
34     SPI_CSN_DELAY,
35     SPI_CSN
36     );
37
38   signal cur_spi_state  : spi_state;
39   signal  read_shift    : std_logic_vector(data_width-1 downto 0);
40   signal sclk_counter   : std_logic := '0';
41   signal sclk_cpol_cpha_oop : std_logic := '0';
42   signal sclk_cpol_cpha_ip : std_logic := '0';
43   signal  send_shift          :std_logic_vector(data_width-1 downto 0);
44   signal  send_shift_next     :std_logic_vector(data_width-1 downto 0);
45   signal  mosi_data_valid_i_follower  : std_logic;
46   signal  data_read  : std_logic;
47   signal  data_read_follower  : std_logic;
48   signal  new_data  : std_logic;
49   signal cs_n_signal       :std_logic;
50   signal mosi_signal       :std_logic;
51   signal  rd_en : std_logic;
52   signal  wr_en : std_logic;
53   signal  clk_off  : std_logic;
54   signal  sclk_en : std_logic;
55   signal  reload : std_logic;
56   signal  bits_sent : unsigned(data_width-1 downto 0);
57
```

با توجه به این که Entity این فایل را در SPI_Command دیدیم یک راست به سراغ معماری رفتاری این فایل خواهیم رفت.

همانطور که مشاهده می‌کنید مانند ۲ ماژول قبل در این ماژول نیز از تعدادی استیت تعریف شده و در ادامه سیگنال هایی که به منظور اعمال تغییرات لازم در استیت های مختلف می‌باشند بیان شده اند.

```
58  begin
59
60     mosi <= mosi_signal;
61     cs_n <= cs_n_signal;
62
63  next_state : process(clk_off,rst_n)
64  begin
65    if rst_n = '0' then
66      cur_spi_state <= SPI_WAIT;
67      miso_data_valid_o <= '0';
68      data_read <= '0';
69      reload <= '0';
70
71      miso_data_o <= (others => '0');
72      bits_sent <= (others => '0');
73
74    elsif rising_edge(clk_off) then
75
76      miso_data_valid_o <= '0';
77
78      if (data_read_follower = '1' and  data_read ='1') then
79        data_read <= '0';
80      end if;
81
82      case cur_spi_state is
83
84        when SPI_WAIT =>
85          if (new_data = '1') then
86            cur_spi_state <= SPI_CS;
87            data_read <= '1';
88            reload <= '1';
89          end if;
90
91        when SPI_CS =>
92          reload <= '0';
93
94        if (reload = '0') then
95              cur_spi_state <= SPI_SHIFT;
96        end if;
97
98        when SPI_SHIFT =>
99        reload <= '0';
100       bits_sent <= bits_sent + 1;
101
102         if (bits_sent = data_width-2 and new_data = '1') then
103           reload <= '1';
104
105         elsif (bits_sent = data_width-1 and new_data = '1') then
106           cur_spi_state <= SPI_SHIFT;
107           data_read <= '1';
108
109           miso_data_valid_o <= '1';
110           miso_data_o <= read_shift;
111           bits_sent <= to_unsigned(0,bits_sent'length);
112
113         elsif (bits_sent = data_width-1 and new_data = '0') then
114           miso_data_valid_o <= '1';
115           miso_data_o <= read_shift;
116           cur_spi_state <= SPI_CSN_DELAY;
117           bits_sent <= to_unsigned(0,bits_sent'length);
118         else
119           cur_spi_state <= SPI_SHIFT;
120         end if;
121
122       when SPI_CSN_DELAY =>
123         cur_spi_state <= SPI_CSN;
```

```
124
125
126         when SPI_CSN =>
127           reload <= '0';
128           cur_spi_state <= SPI_WAIT;
129
130         end case;
131       end if;
132
133     end process;
```

در این پروسس به مدیریت استیت هایی که در ابتدا تعریف شده است پرداخته می‌شود. در واقع این پروسس مانند ماشین حالت عمل می‌کند و با توجه به سیگنالینگ های انجام شده و وضعیت فعلی وضعیت بعدی را مشخص می‌کند در این ماژول از آن جا که در حال پیاده سازی SPI هستیم استیت ها متناسب با وضعیت ورود و خروج دیتا به وسیله این پروتکل تعریف شده اند.

```
118 output_logic : process(rst_n,cur_spi_state,send_shift)
119 begin
120 if rst_n = '0' then
121   rd_en <= '0';
122   wr_en <= '0';
123   cs_n_signal <= '1';
124   sclk_en <= '0';
125   mosi_signal <= '0';
126   else
127     case cur_spi_state is
128       when SPI_WAIT =>
129         cs_n_signal <= '1';
130         sclk_en <= '0';
131         rd_en <= '0';
132         wr_en <= '0';
133         mosi_signal <= '0';
134       when SPI_CS =>
135         sclk_en <= '0';
136         cs_n_signal <= '0';
137         rd_en <= '0';
138         wr_en <= '0';
139         mosi_signal <= '0';
140       when SPI_SHIFT =>
141         mosi_signal <= send_shift(data_width-1);
142         rd_en <= '1';
143         wr_en <= '1';
144         cs_n_signal <= '0';
145         sclk_en <= '1';
146       when SPI_CSN_DELAY =>
147         mosi_signal <= '0';
148         rd_en <= '0';
149         wr_en <= '0';
150         cs_n_signal <= '0';
151         sclk_en <= '0';
152       when SPI_CSN =>
153         mosi_signal <= '0';
154         rd_en <= '0';
155         wr_en <= '0';
156         cs_n_signal <= '1';
157         sclk_en <= '0';
158       end case;
159   end if;
160 end process;
```

در این پروسس با توجه به استیتی که در آن قرار داریم سیگنال های مربوط به خواند و نوشتن و کلاک و چیپ سلکت و دیتای ارسالی سیگنالینگ میشوند که در روند پیاده سازی پروتکل SPI استفاده می‌شوند.

```
rcv_MISO:   process(clk_off,rst_n)
begin
  if (rst_n = '0') then
    read_shift <=  (others => '0');
  elsif rising_edge(clk_off) then
       if(rd_en = '1') then
        read_shift(data_width-1 downto 0) <= read_shift(data_width-2 downto 0) & miso;
       end if;
  end if;
end process;


send_MOSI:  process(clk_off,rst_n)
begin
  if (rst_n = '0') then
     send_shift <= (others => '0');
  elsif rising_edge(clk_off) then
    if (reload = '1') then
      send_shift <= send_shift_next;
    else
       if (wr_en = '1') then
             send_shift(data_width-1 downto 0) <= send_shift(data_width-2 downto 0) & '0';
        end if;
    end if;
  end if;
end process;

```

دو پروسس بعدی مربوط به دریافت و ارسال داده به وسیله پروتکل SPI می‌باشد همانطور که می‌دانیم دیتا چه برای ارسال و چه برای دریافت به وسیله این پروتکل بیت به بیت ارسال و دریافت می‌شود و در این دو پروسس نیز برای ارسال و دریافت آن از شیفت دادن بیت به بیت استفاده شده است.

```
mosi_data_valid_i_process: process(clk_off,rst_n)
begin
  if (rst_n = '0') then


  data_read_follower <= '0';
  mosi_data_valid_i_follower  <= '0';
  new_data  <= '0';
  mosi_data_ack_o <= '1';
  send_shift_next <= (others => '0');

  elsif clk'event and clk = '1' then

    if (mosi_data_valid_i_follower /= mosi_data_valid_i) then
        mosi_data_valid_i_follower <= mosi_data_valid_i;
      if (mosi_data_valid_i = '1') then
        send_shift_next <= mosi_data_i;
        new_data <= '1';
        mosi_data_ack_o <= '0';
      end if;
    elsif (data_read_follower /= data_read) then
      data_read_follower <= data_read;
      if (data_read = '1') then
      new_data <= '0';
      mosi_data_ack_o <= '1';
      end if;

    end if;
  end if;
end process;

```

پروسس فوق برای دادن سیگنالینگ ولید بودن یا معتبر بودن داده دریافتی توسط این پروتکل می‌باشد.

با توجه به دیتا استراکچری که این آی سی دارد به کمک این پروسس در زمانی که ۲۰ بیت به طور کامل و به طور صحیح دریافت شود سیگنالی داده می‌شود که نشان دهنده معتبر بودن این داده می‌باشد.

```
222
223  clk_off <= sclk_counter;
224
225  clock_process: process(clk)
226  begin
227    if rising_edge(clk) then
228        sclk_counter <= not sclk_counter;
229    end if;
230  end process;
231
232  sclk_ip_process: process(clk)
233  begin
234    if rising_edge(clk) and sclk_en = '1' then
235      sclk_cpol_cpha_ip <= not sclk_cpol_cpha_ip;
236    end if;
237  end process;
238
239  sclk_oop_process: process(clk)
240  begin
241    if falling_edge(clk) and sclk_en = '1' then
242        sclk_cpol_cpha_oop <= not sclk_cpol_cpha_oop;
243    end if;
244  end process;
245
246
247  nioop_clk: if (cpol_cpha = "00") generate
248    sclk <= sclk_cpol_cpha_oop and sclk_en;
249  end generate;
250
251  niip_clk: if (cpol_cpha = "01") generate
252    sclk <= not sclk_cpol_cpha_ip and sclk_en;
253  end generate;
254
255  iip_clk: if (cpol_cpha = "11") generate
256    sclk <= sclk_cpol_cpha_ip when sclk_en = '1' else '1';
257  end generate;
258
259  ioop_clk: if (cpol_cpha = "10") generate
260    sclk <= not sclk_cpol_cpha_oop when sclk_en = '1' else '1';
261  end generate;
262
263
264
265  end Behavioral;
```

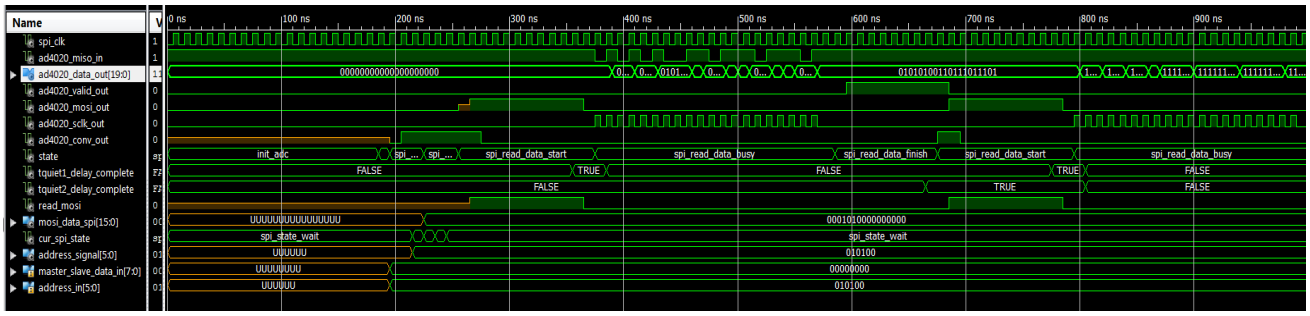
در پایان نیز تعدادی پروسس که برای اعمال فاز های مختلف کلاکینگ در پروتکل SPI می‌باشد قرار داده شده است. در این آی سی از این قابلیت پروتکل SPI استفاده نشده بنابراین از بین این پروسس ها فقط پروسسی که کلاک عادی با فاز عادی تحویل می‌دهد استفاده می‌شود.

به این ترتیب هر ۳ فایل AD4020 و SPI_Command و SPI_Abstract به طور کامل تشریح شدند.

اکنون به سراغ نتایج شبیه سازی و تطابق با خواسته های دیتا شیت خواهیم رفت.

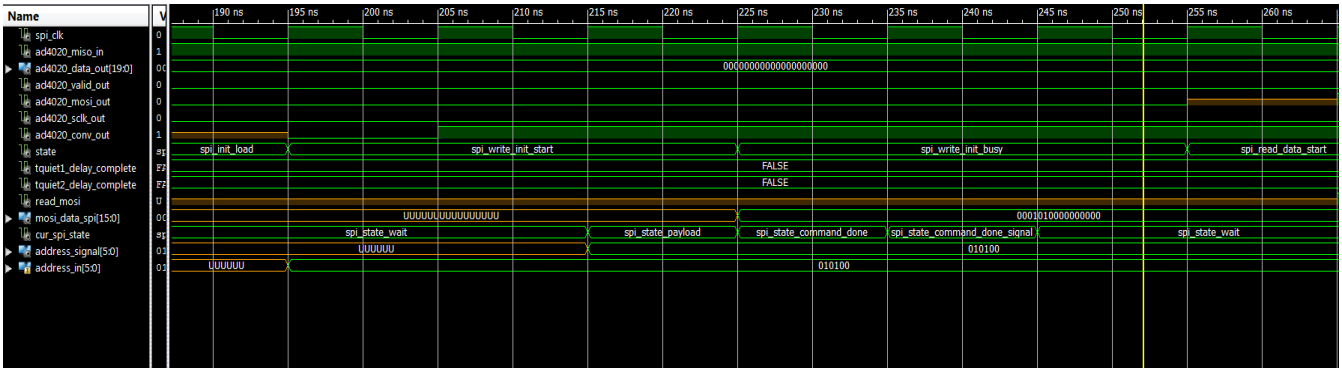
۲)خروجی شبیه سازی :

ابتدا نمایی کلی از خروجی بدست آمده را مشاهده کرده سپس به بررسی دقیق تر بازه های زمانی آن پرداخته خواهد شد.



با شبیه سازی انجام شده نتیجه فوق بدست آمد که نشان دهنده پیاده سازی آی سی AD4020 می باشد.

اکنون به بررسی بازه های زمانی به صورت دقیق تر پرداخته می‌شود.



همانطور که مشاهده می‌کنید در ابتدای برنامه ماژول SPI_Command فعال شده و دستور داده شده را در ردیف cur_spi_state طی استیت های spi_state_wait و سپس spi_data_payload و spi_state_command_done و در نهایت spi_command_done_signal اعمال کرده است و خروجی متناسب با آن را در ردیف mosi_data_spi در لحظه ۲۲۵ نانو ثانیه دریافت کرده است. اگر ردیف مربوطه به state را مشاهده بفرمایید حدود ۲۰۰ نانو ثانیه کل برنامه در استیت spi_init_load بوده سپس از ۲۰۰ تا ۲۲۵ نانوثانیه برنامه در استیت spi_write_init_start قرار گرفته که در این مدت کامند های لازم اعمال شده و در لحظه ۲۲۵ نانو ثانیه با اعمال شدن دستورات برنامه وارد استیت spi_write_busy شده است در نهایت در لحظه ۲۵۵ نانو ثانیه دستورات با اعمال نهایی دستورات برنامه وارد مرحله spi_read_data_start شده تا منتظر خواندن داده های adc شود.

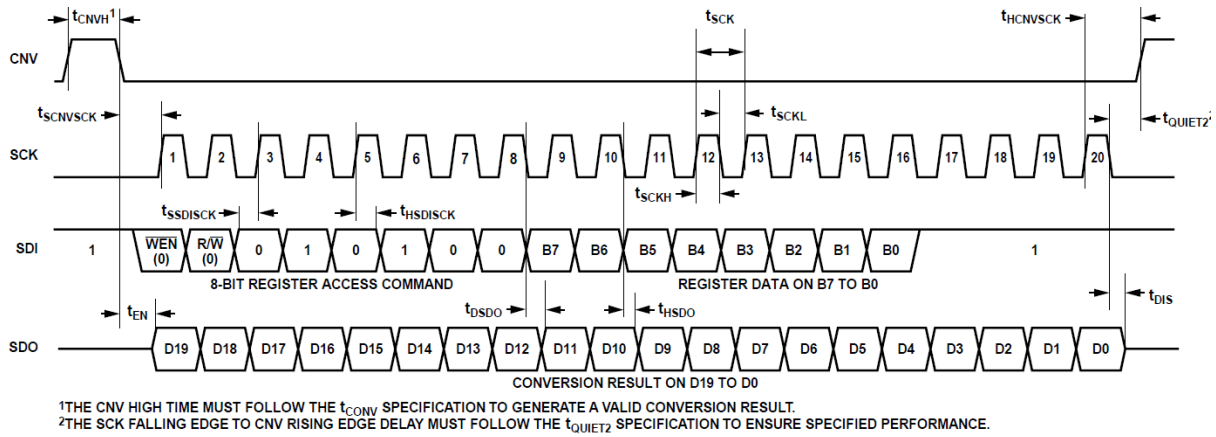
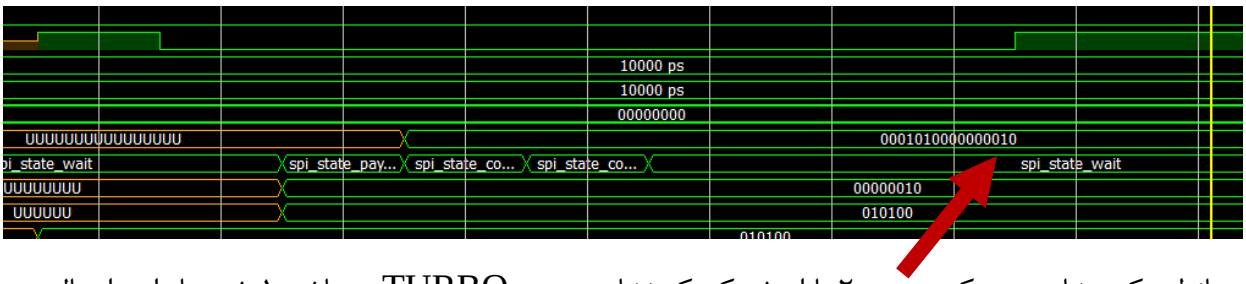


Figure 50. Register Write Timing Diagram

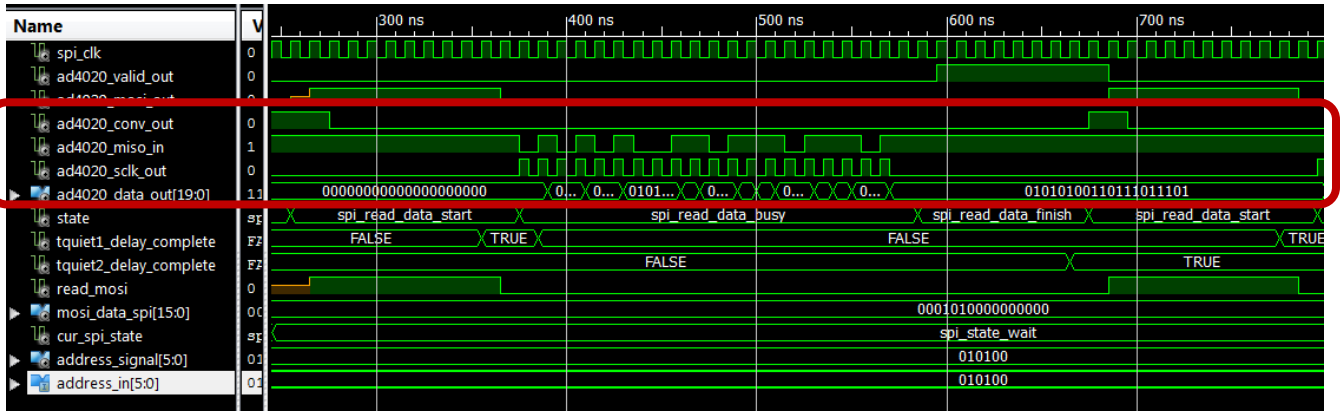
مطابق دیتا شیت نیز دقیقا در پاسخ ۱۶ بیتی که ۸ بیت ابتدایی آن ۰۰۰۱۰۱۰۰ است خروجی شامل این ۸ بیت و بیت های پاسخ به این ۱۶ بیت را دریافت کردیم. بیت های با ارزش کم همان , SPAN , OV HIGHZ , TURBO می باشند با تغییر مقدار TURBO یک بار دیگر برنامه را اجرا می‌کنیم خروجی مطابق شکل زیر می‌شود



همانطور که مشاهده می‌کنید بیت ۲ با ارزش کم که نشان دهنده TURBO می‌باشد ۱ شد بنابراین اعمال دستورات ابتدایی برنامه به درستی پیاده شده است.

در ادامه به بررسی خروجی ADC خواهیم پرداخت برای مقایسه بهتر با خروجی نشان داده شده در دیتاشیت در صفحه بعد به این مقایسه خواهیم پرداخت.

برای بهتر شدن امکان مقایسه خروجی بدست آمده با دیتاشیت جای ردیف ها تغییر کرد به این ترتیب ردیف های ad4020_conv_out و ad4020_miso_in و ad4020_sclk_out و ad4030_data_out که در زیر هم قرار دارند به ترتیب همان ردیف هایی است که در ادامه در دیتاشیت مقایسه می‌کنید.



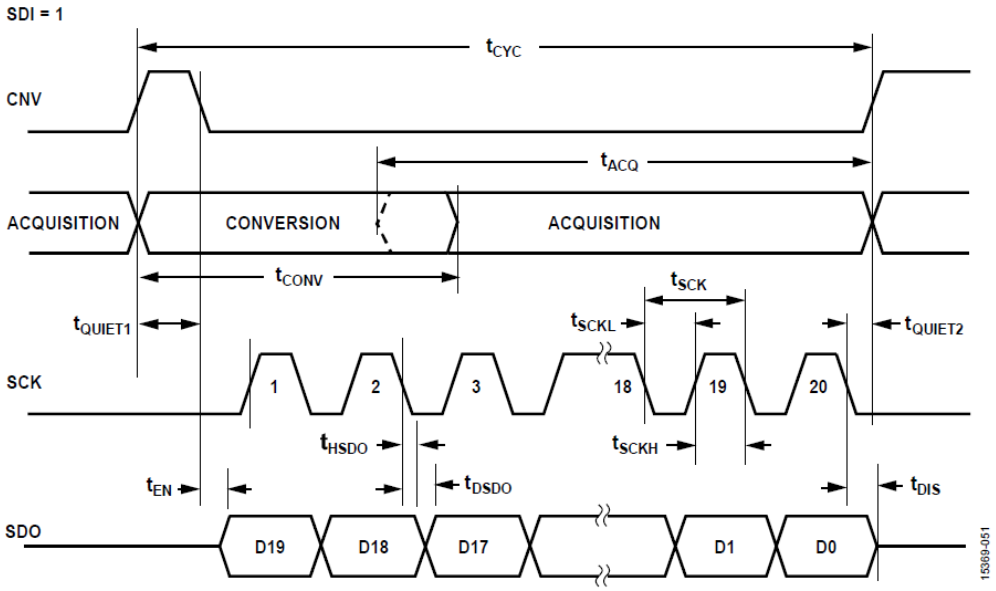


Figure 54. $\overline{CS}$ Mode, 3-Wire Turbo Mode Serial Interface Timing Diagram (Status Bits Not Shown)

همانطور که مشاهده می‌کنید دور سیگنال هایی از شبیه سازی انجام شده که در دیتاشیت آمده کادر کشیده شده و این شبیه سازی توانسته کاملا دقیق شکل موج این دیتاشیت را پیاده سازی کند.

همچنین دیلی های tquiet1 , 2 نیز که هر کدام ۱۰۰ نانوثانیه در نظر گرفته شده بود، کاملا دقیق بدست آمده اند.

به این ترتیب این کد و شبیه سازی انجام شده توانست، به آنچه در دیتاشیت این آی سی تعریف شده بود برسد.