دانشکده مهندسی برق

گزارش پروژه درس VHDL

نام دانشجو:

مهرداد تنهایی ۴۰۱۶۱۱۰۶۷

استاد:

دکتر ستار میرزا کوچکی

بهمن ماه ۱۴۰۱


بسم الله الرحمن الرحيم

چکیده

پروژه تعریف شده، پیاده سازی پروتکل ارتباطی SPI مبدل آنالوگ به دیجتیالی است که شامل دو کانال با منبع تغذیه ۱.۸ ولت است. نرخ نمونه برداری این IC در بازه ۱۰۵ تا ۱۲۵ MSPS قرار دارد. این IC که به منظور تبدیل سیگنال‌های آنالوگ به دیجیتال است در سیستم‌های رادیویی متنوع، گیرنده دیجیتال چند حالته و ... کاربرد دارد.

پروتکل ارتباطی SPI استفاده شده، در این IC به صورت سه سیمِ است. این به این منظور است که برای ارسال و دریافت اطلاعات از یک خط سیم استفاده می‌کند و دو سیم دیگر برای کلاک و فعال کردن IC می‌باشد.

برای پیاده سازی پروتکل ارتباطی SPI به زبان vhdl، ابتدا timming Diagram این IC،تحلیل و بررسی شده است. پس از آن به کمک نرم افزار ISE، این پروتکل ارتباطی پیاده سازی می‌شود و برای تست گرفتن از کد نوشته شده، با نوشتن یک test bench برای این پروتکل، از صحت آن اطمینان حاصل می‌شود. test bench نوشته شده با توجه به جدول memory map این IC بوده است.

فهرست مطالب

فصل ۱:

AD9628

AD9628 یک مبدل آنالوگ به دیجیتال دو کانال با منبع تغذیه ۱.۸ ولت است که نرخ نمونه برداری آن در بازه ۱۰۵ تا ۱۲۵ MSPS و دارای یک مدار نمونه برداری و نگهداری با کارایی بالا و مرجع ولتاژ تراشه است. این IC در سیستم های دیجیتال به منظور تبدیل سیگنال‌های آنالوگ به دیجیتال است که در سیستم های رادیویی های متنوع، گیرنده دیجیتال چند حالته و... استفاده میشود. این محصول از معماری pipeline دیفرانسیل چند مرحله‌ای با منطق تصحیح خطای خروجی استفاده می‌کند تا دقت ۱۲ بیتی را با نرخ داده ۱۲۵ MSPS ارائه دهد و تضمین کند که کدهای از دست رفته در محدوده دمای عملیاتی کامل وجود ندارد. داده‌های خروجی دیجیتال در فرمت offset binary ، Gray Code یا مکمل دو ارائه می شود.

یکی دیگر از ویژگی های این IC این است که دارای low voltage differential signaling (LVDS) می‌باشد یعنی همانطور که از اسم آن بر می‌آید، گیرنده دیتا مقدار دیتا را از اختلاف در سیگنال متوجه می‌شود، جریان ثابت سیگنال‌های مقایسه شده باعث می شود نویز الکتریکی سیستم کاهش یابد همچنین جریان متضاد دو سیگنال ذکر شده باعث می گردد میدان‌های الکترومغناطیسی کاهش یافته و در کل باعث کاهش نویز محیط سیستم گردد.

فصل ۲:

پروتکل SPI در AD9628

آی سی AD9628 قابلیت ارتباط سریال از طریق پروتکل SPI را دارد. رابط پورت سریال (SPI) به کاربر این امکان را می دهد که مبدل را برای عملکردها یا عملیات های خاص از طریق یک فضای ثبت ساختار یافته در داخل ADC پیکربندی کند. SPI بسته به برنامه کاربردی، انعطاف و سفارشی سازی بیشتری را به کاربر می دهد. آدرس ها از طریق پورت سریال قابل دسترسی هستند و می توان از طریق پورت به آنها نوشت یا خواند. حافظه به بایت‌هایی سازماندهی شده است که می توان آنها را به فیلدهایی تقسیم کرد که در بخش نقشه حافظه مستند شده است.

ارتباط از طریق پروتکل SPI در سطح ۱.۸ ولت در این IC اتفاق می‌افتد و فرکانس آن تا ۲۰۰ مگاهرتز بالا می‌رود. پروتکل SPI در دو حالت چهار سیم و سه سیم قابل راه اندازی است که IC ذکر شده از حالت دوم یعنی سه سیم برای برقراری ارتباط عمل میکند.

پین SCLK پایه ورودی کلاک IC است که برای همگام سازی خواندن و نوشتن داده ها است. پین SDIO پایه دو منظوره ورودی و خروجی است که امکان ارسال و دریافت را به ADC memory map registers می‌دهد. پین CSB پایه فعال و غیر فعال کردن خواندن و نوشتن IC هست که با منطق low کار میکند.

برای شروع ارتباط ابتدا باید پایه CSB از حالت high به low برود و با یک شدن کلاک شروع stream خواهد بود. در حالت کلی می‌توان فریم داده بر روی SDIO را به دو قسمت تقسیم کرد.۱۶ بیت اول بیت های دستورالعمل است و مابقی بیت ها بیت های دیتا هستند.

۱۶ بیت دستور العمل شامل دستور خواندن و نوشتن داده، طول داده‌ها و آدرس ریجستر حافظه ADC می‌باشد. به این صورت که، پیشفرض باارزش ترین بیت دستورالعمل، برای دستور خواندن و نوشتن داده بر روی آدرس مورد نظر است و دو بیت پس از آن برای طول داده ها یا همان تعداد بایت داده است. که با حروف W_0 و W_1 نشان داده شده است. اگر W_0 و W_1 برابر 0b00 باشد یک ۸ بیت داده برای انتقال وجود دارد، اگر برابر 0b۰۱ باشد دو تا ۸ بیت داده، 0b۱۰ یعنی سه تا ۸ بیت داده و اگر برابر 0b۱۱ باشد یعنی چهار یا بیشتر از چهار تا ۸ بیت داده برای انتقال موجود است. البته باید به این نکته توجه کرد که تا اتمام اخرین بیت دیتا باید پایه CSB در وضعیت low باشد در غیر این صورت ارتباط از بین می رود

[W1:W0] Setting	Action	CSB Stalling
00	1 byte of data can be transferred.	Optional
01	2 bytes of data can be transferred.	Optional
10	3 bytes of data can be transferred.	Optional
11	4 or more bytes of data can be transferred. CSB must be held low for entire sequence; otherwise, the cycle is terminated, and an instruction cycle is anticipated when CSB returns low.	No

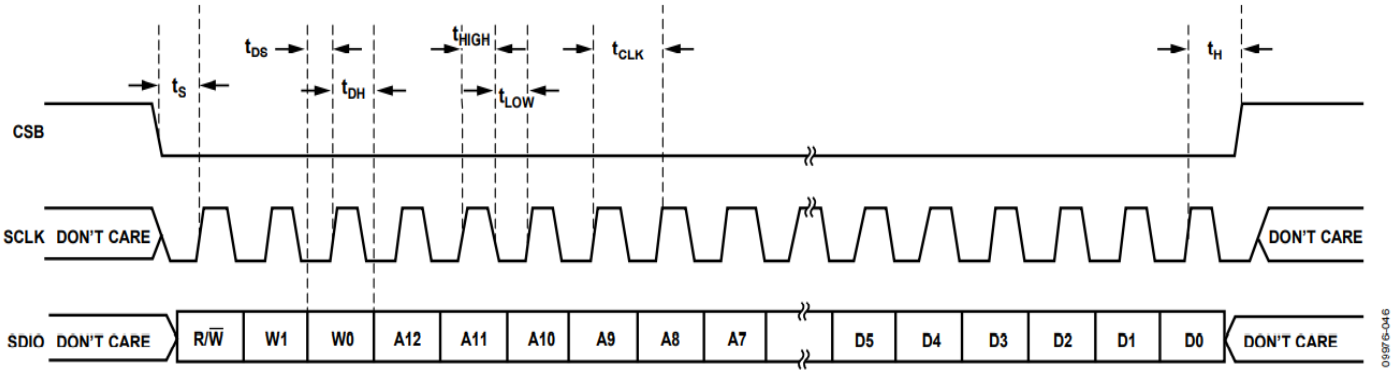
جدول۱-۲) Word Length Settings

این نکته هم حائز اهمیت است که به طور پیش فرض دیتا از MSB شروع میشود و با تغییر مقدار ادرس ریجستر 0x00 باعث میشود تا دیتا از LSB شروع به انتقال کند. در کل این IC شامل MEMORY MAP REGISTER TABLE است که با نگاه کردن به جدول دیتاشیت میتوان تغییراتی در روند کار این IC ایجاد کرد.

اگر به شکل Timing Diagram آی سی نگاه کنیم، زمان های مختلفی میبینیم. به عنوان مثال t_s برای setup time بین CSB و کلاک است یعنی فاصله بین دو زمان low شدن CSB تا high شدن کلاک، t_h hold time از زمان اخرین high شدن کلاک برای اخرین بیت داده تا زمان high شدن CSB است. t_{ds} setup time بین زمانی است که یک بیت شروع میشود تا زمان high شدن کلاک است و t_{dh}، hold time بین کلاک زده شده تا انتهای یک بیت و شروع یک بیت جدید است. t_{clk} یک تناوب کلاک را نشان میدهد. در جدول شماره (۲-۲) مابقی زمان های مربوط را میتوان مشاهده نمود.

Parameter	Description	Limit	Unit
SYNC TIMING REQUIREMENTS			
t_{SSYNC}	SYNC to rising edge of CLK+ setup time	0.24	ns typ
t_{HSYNC}	SYNC to rising edge of CLK+ hold time	0.40	ns typ
SPI TIMING REQUIREMENTS			
t_{DS}	Setup time between the data and the rising edge of SCLK	2	ns min
t_{DH}	Hold time between the data and the rising edge of SCLK	2	ns min
t_{CLK}	Period of the SCLK	40	ns min
t_S	Setup time between CSB and SCLK	2	ns min
t_H	Hold time between CSB and SCLK	2	ns min
t_{HIGH}	SCLK pulse width high	10	ns min
t_{LOW}	SCLK pulse width low	10	ns min
t_{EN_SDIO}	Time required for the SDIO pin to switch from an input to an output relative to the SCLK falling edge	10	ns min
t_{DIS_SDIO}	Time required for the SDIO pin to switch from an output to an input relative to the SCLK rising edge	10	ns min

جدول۲-۲) مشخصات زمانی



شکل۳-۲) Serial Port Interface Timing Diagram

پس از اتمام فریم دستور العمل، فریم مربوط به دیتا شروع می‌شود که با توجه به W_0 و W_1 شروع به انتقال میکند و باید تا ارسال اخرین بیت CSB در وضعیت low باشد.

فصل ۳:
پیاده سازی پروتکل SPI

3-1- توضیح کد پروژه

کدهای VHDL از دو بخش entity و Architecture تشکیل شده‌اند. در بخش entity باید پورت‌های مورد نیاز را مشخص کرد و در بخش Architecture با استفاده از نوشتن کد، نحوه اتصالات این المان‌ها را مشخص می‌کند. Architecture را در ۳ سطح رفتاری، ساختاری، RTL می‌توان نوشت. برای نوشتن این پروژه از سطح رفتاری استفاده شده است. در ادامه به توضیح کد میپردازیم.

مانند تمام زبان‌های برنامه نویسی، ابتدا باید کتابخانه های مورد نیاز را به پروژه اضافه کنیم. در شکل(۱-۳) نحوه اضافه شدن کتابخانه‌های مورد نیاز مشاهده می‌شود.

```
--import library

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
-----------------------------------------end import library
```

شکل۱-۳) اضافه کردن کتابخانه ها

پس از اضافه کردن کتابخانه‌های مورد نیاز، بایستی entity را بنویسیم. برای سیستم یک کلاک و همچنین برای خود ماژول هم یک کلاک سنکرون ساز با سایر ماژول ها تعریف می‌کنیم. برای این که ارتباط سریال شروع شود یک ورودی دیگر باید تعریف کرد. از طرف دیگر برای دریافت و ارسال اطلاعات نیاز به یک ورودی دیگر داریم که به صورت یک vector که حداکثر۴۸ بیت دریافت می‌کند، تعریف می‌کنیم. در ادامه سه پایه ورودی دیگر مورد نیاز است. یک پایه که به کمک آن و مدار ترکیبی که در ادامه کدی که می‌بینیم، به آن متصل است، دریافت و ارسال داده را مشخص میکنیم. برای این که می‌خواهیم شبیه سازی انجام دهیم باید دو پایه را به صورت خروجی تعریف کنیم، که یکی از آن ها برای کنترل نوشتن و خواندن دیتا است و پایه دیگر برای انتخاب تراشه است که به صورت active low است.

```
---------------------------------------
--entity of spi module
entity spi_3wire is
    port (
        --input
        clk_sys         :in     std_logic;                          --system clock
        sclk            :in     std_logic;                          --module clock
        start           :in     std_logic;
        data_in         :in     std_logic_vector(47 downto 0);

        --in/out
        sdio            :inout  std_logic;

        --out
        cs_n            :out    std_logic;
        rw_ctrl         :out    std_logic
    );
end spi_3wire;
-----------------------------------end entity
```

شکل۲-۳) نوشتن entity

پس از نوشتن entity باید بخش Architecture برنامه را بنویسیم. Architecture از دو بخش declaration و کد برنامه است، که این دو با کلمه begin از هم متمایز می‌شوند. در برنامه نویسی اصولی و استاندارد برای هر پایه یک رجیستر تعریف می‌شود تا از اشتباهات احتمالی جلوگیری شود. پس از آن، به سبب دیتایی که ارسال یا دریافت می‌شود، از دو بخش دستورالعمل و داده را شامل می‌شود، یک record تعریف می‌کنیم که شامل این دو بخش می‌شود. record در vhdl مانند ساختارها در برنامه نویسی C است. پس از آن از record که تعریف کردیم یک سیگنال معرفی می‌کنیم. پس از آن برای این که تعداد بایت داده‌ها را که میخواهیم ارسال یا دریافت کنیم را مشخص کنیم از یک سیگنال به عنوان counter استفاده می‌کنیم.

سیگنال tx به ما کمک می‌کند تا مدار ترکیبی ذکر شده را برای مشخص کردن نوشتن یا خواندن پیاده سازی کنیم. به دلیل آن که، ۱۶ بیت دستور العمل وجود دارد، و متوجه شویم که چه زمانی بیت‌های داده شروع می‌شود، سیگنال bit_cnt را می‌نویسیم، که چه زمانی تمام بیت‌های داده ارسال یا دریافت شده است. برای پیاده سازی AD9628 از ماشین حالت استفاده کرده‌ایم. به همین علت باید حالت‌های مختلف را بیان کنیم. سپس یک سیگنال از type که تعریف کردیم، می‌نویسیم.

- Idel : حالت اولیه، کاری صورت نمی‌گیرد
- Instruction : حالتی که بیت‌های دستورالعمل ارسال یا دریافت می‌شوند.
- Write_s : حالت مربوط به نوشتن داده‌ها
- Read_s : حالت مربوط به خواندن داده‌ها
- Delay_instruction : حالتی است که برای جلوگیری از دست رفتن بیت‌های دستورالعمل ایجاد شده است.
- Delay_read : حالتی است که برای جلوگیری از دست رفتن بیت‌های داده ایجاد شده است.

```
--architecture of spi module
architecture Behavioral of spi_3wire is

    --spi input/output register
    signal cs_n_int                 :std_logic                          :='1';
    signal rw_ctrl_int              :std_logic                          :='0';
    signal start_int                :std_logic                          :='0';
    signal data_in_int              :std_logic_vector(31 downto 0)      :=(others=>'0');
    -----------------

    --spi internal signal
    type   instruction_data_path  is record
      inst_byte                     :std_logic_vector(7 downto 0);              --signal for instruction section
      rx_data                       :std_logic_vector(31 downto 0);             --signal for data section
    end record;

    signal data_path                :instruction_data_path              :=(inst_byte=>(others=>'0'),rx_data=>(others=>'0'));
    signal cnt_limit                :unsigned (4 downto 0)              :=(others=>'0');
    signal tx                       :std_logic                          :='Z';
    -----------------


    --spi counter
    signal bit_cnt                  :unsigned (4 downto 0)              :="00111";
    -----------------

    --spi state
    type fsm is (idle , instruction , write_s ,read_s , delay_instruction , delay_read);
    signal state                    :fsm                                 :=idle;
    -----------------
```

شکل ۳-۳) تعاریف سیگنال‌های لازم

از این بخش به بعد قسمت declaration تمام شده است و وارد قسمت کد برنامه می‌شویم. به علت این که کد برنامه به صورت موازی اجرا می‌شود، باید خروجی ها را، خارج از process بنویسیم تا به صورت موازی با process اجرا شوند. در نتیجه با استفاده از سیگنال‌های تعریف شده، مقدار آن‌ها را داخل خروجی‌ها assign می‌کنیم. با توجه به توضیحات داده شده برای آن که مشخص کنیم که دیتا ارسال کند یا دریافت باید به کمک یک مدار ترکیبی که کد آن را در شکل (۴-۳) مشاهده می‌شود، پیاده سازی کردیم.

از بخش process به بعد وارد قسمتی از کد می‌شویم که به صورت سریال اجرا می‌شود، ولی در کل، تمام قسمت process با سایر قسمت‌های کد به صورت موازی اجرا می‌شوند. به دلیل این که تمام eventهایی که می‌خواهیم زمانی اتفاق بیفتد که کلاک زده می‌شود، بنابراین شرط آن را داخل شرط process قرار می‌دهیم. و در ادامه به دلیل این که می‌خواهیم تمام eventها در لبه بالا رونده اتفاق بیفتد، شرط آن را داخل شرط if قرار می‌دهیم.

استاندارد دیگری که وجود دارد در برنامه نوسی vhdl، این است که خروجی ها را خارج از process و ورودی ها در داخل و ابتدا آن assign می‌کنیم. به همین سبب بخش‌های دیتا، دستورالعمل و داده، را هر کدام در بخش خودش assign کرده و سیگنال start هم assign کرده تا روند ارسال و دریافت دیتا، آماده شود.

```
begin

    cs_n        <=cs_n_int;
    rw_ctrl     <=rw_ctrl_int;
    sdio        <=tx when rw_ctrl_int ='0' else 'Z';                    --combinational circiut for choose read or write

    process(clk_sys)
    begin
        if(rising_edge (clk_sys)) then
            data_in_int                 <=data_in(31 downto 0);         --data
            data_path.inst_byte         <=data_in(39 downto 32);        --instruction
            start_int                   <= start;                       -- start operation

            ----
```

شکل۴-۳) نسبیت دادن سیگنال‌ها به ورودی و خورجی

به دلیل این که از ماشین حالت می‌خواهیم استفاده کنیم بنابراین از ساختار case استفاده می‌کنیم. اولین حالتی که تعریف می‌کنیم، مربوط به حالت Idel است که اولین حالت و هیچ اتفاق خاصی در آن نمیفتد. در این حالت اگر بیت start_int یک باشد، سبب می‌شود تا به حالت Delay_instruction برویم و با صفر دادن به cs_n_init سبب می‌شود تا تراشه انتخاب شود و آماده به کار شود. اما اگر start_int برابر با یک نباشد، در همان حالت Idel باقی می‌ماند و مقدار cs_n_init را هم یک می‌دهیم تا تراشه فعال نشود و توان مصرف نکند.

```
when idle =>                                                    -- idle state ( no happen occure)
  rw_ctrl_int               <='0';
  tx                        <='Z';
  bit_cnt                   <="01111";                          -- seven bit instruction
  data_path.rx_data         <=(others=>'0');


  if(start_int = '1') then                                      -- ready next state to assign to the current state

     state                  <=delay_instruction;
     cs_n_int               <='0';


  else
     state                  <= idle;
     cs_n_int               <='1';
     cnt_limit              <= (others=>'0');
  end if;
```

شکل۵-۳) حالت idle

در ادامه به علت این که بیت‌های دستور العمل از بین نرود، باید قسمت Delay_instruction را بنویسیم. در این بخش، حالت بعدی را مشخص می‌کند و سیگنال‌های یعنی حالت .instruction rw_ctrl_int و cs_n_int را صفر می‌دهیم تا تراشه هم انتخاب شود و هم آمادگی خواندن و نوشتن داده را داشته باشد. سپس اولین بیت یا همان با ارزش ترین بیت دستور العمل را به سیگنال tx ، assign می‌کنیم و در ادامه به دلیل این که یک بیت انتقال داده شده است باید از مقدار bit_cnt یک واحد کم شود.

یکی دیگر از ترفندهای برنامه نویسی استاندارد به این صورت است که باید در هر stateکه رفتیم به سیگنال یک مقدار assign کنیم. به همین علت به سیگنال cnt_limit در داخل خودش قرار می‌دهیم و بیت‌های data_path.rx_data نیز برابر صفر قرار می‌دهیم.

```
when delay_instruction =>                                       -- for dont lose bit struction
  state                     <=instruction;                      -- assign next state
  rw_ctrl_int               <='0';
  cs_n_int                  <='0';                              --must be 0 so that chip gets active
  tx                        <=data_path.inst_byte(to_integer(bit_cnt));
  bit_cnt                   <=bit_cnt - 1;
  cnt_limit                 <=cnt_limit;
  data_path.rx_data         <=(others=>'0');
```

شکل۶-۳) حالت delay_instruction

State بعدی که می‌نویسیم، متعلق به instruction است. در این حالت نیز برای ارسال یا دریافت اطلاعات باید cs_n_init صفر باشد و چون می‌خواهیم داده ارسال شود باید مقدار rw_ctrl_init نیز صفر باشد. سپس با توجه به این که مقدار bit_cnt یک واحد از آن کم گردید است، از بیت بعدی متعلق به دستورالعمل به

سیگنال tx، assign می‌شود. در ادامه، همانطور که پیش از آن ذکر گردید، برای استانداردسازی کد دو سیگنال دیگر هم به ترتیب مقادیر صفر و خودش را assign می‌کنیم.

پس از آن باید بررسی کنیم که آیا تمام بیت‌های دستور العمل انتقال یافته است یا خیر، به همین علت شرط مساوی نبودن bit_cnt با مقدار صفر را بررسی می‌کنیم. اگر bit_cnt برابر با صفر نباشد، در همان State، instruction باقی می‌ماند و یک واحد دیگر از مقدارbit_cnt کم خواهد شد. این روند تا زمانی که مقدار bit_cnt برابر صفر باشد تکرار می‌شود، و زمانی که برابر با صفر شود، باید به State بعدی برویم، به همین دلیل، اخرین بیت دستور العمل یعنی بیت شانزدهم را بررسی می‌کنیم، اگر برابر با صفر باشد، به این معنا است که داده باید نوشته شود و باید به حالت write_s رفت. در غیر این صورت، اگر بیت ۱۶ برابر با یک باشد، داده خواندنی است و باید به State، delay_read رفت.

عمل دیگری که باید در این State انجام داد، این است که باید بررسی کنیم بیت‌های پانزدهم و چهاردهم، دستورالعمل برابر با چه مقداری هستند. همان طور که در فصل قبل ذکر شد، مقادیر این دو بیت، تعیین کننده تعداد بایت داده‌ای است، که میخواهد ارسال شود. به عنوان مثال اگر مقدار هر دو بیت برابر با صفر باشد، یک بایت داده ارسال می‌شود یا اگر بخواهیم چهار بایت یا بیشتر ارسال کنیم باید این دو بیت را یک بدهیم.

در این حالت باید تمامی بیت‌های bit_cnt را برابر با یک قرار داد. هدف از این کار این است که، زمانی که به حالت بعدی برویم، برای این که شمارش تعداد بیت‌های ارسال یا دریافت شده را بدانیم و با توجه به مقادیر بیت‌های پانزدهم و چهاردهم دستورالعمل، تعداد بیت‌های قابل ارسال یا دریافت را محدود کنیم.

```
when instruction =>                                                    -- instruction state
  rw_ctrl_int                 <='0';
  cs_n_int                    <='0';
  tx                          <=data_path.inst_byte(to_integer(bit_cnt));
  data_path.rx_data           <=(others=>'0');
  cnt_limit                   <=cnt_limit;

    if(bit_cnt /= 0) then
        state                 <=instruction;
        bit_cnt               <=bit_cnt - 1;
    else
        bit_cnt               <=(others =>'1');
                case to_integer(unsigned(data_path.inst_byte(14 downto 13))) is   -- choose number of data byte

                   when 0             =>  cnt_limit <= to_unsigned(24,5);
                   when 1             =>  cnt_limit <= to_unsigned(16,5);
                   when 2             =>  cnt_limit <= to_unsigned(8,5);
                   when others        =>  cnt_limit <= to_unsigned(0,5);
                end case;

        if(data_path.inst_byte(15) ='0')then                                -- choose read or write
            state             <=write_s;
        else
            state             <=delay_read;
        end if;
    end if;
```

شکل۷-۳) حالت instruction

بافرض این که باارزش ترین بیت دستور العمل برابر با صفر باشد، به state، write می‌رویم. در این حالت مانند حالت قبل مقادیر rw_ctrl_init و cs_n_init را صفر می‌دهیم. به دلیل این که از سیگنال

data_path.rx_data استفاده نمی‌شود، مقادیر تمام بیت‌های آن را برابر با صفر قرار می‌دهیم و مقادیری که بر روی سیگنال data_in_int قرار دارند را باید بیت به بیت به صورت سریال به سیگنال tx ارسال کرد. این کار با بررسی شرط مساوی یا غیرمساوی بودن مقدار bit_cnt و cnt_limit تحقق می‌یابد. اگر مقدار bit_cnt برابر با cnt_limit نباشد، به این معنی است که تمام بیت‌ها انتقال نیافته است، پس باید در همین حالت write باقی ماند و از مقدار bit_cnt یک واحد کم شود تا زمانی که مقادیر دو سیگنال ذکر شده برابر شود. زمانی که دو سیگنال برابر شوند، حالت بعدی مشخص می‌شود(idle) و مقدار bit_cnt را برابر با ۰۱۱۱۱ قرار می‌دهیم، به این دلیل که زمانی که به حالت idle باز میگردیم، حالت بعدی idle، instruction است و به سبب این که ۱۶ بیت دستور العمل داریم، مقدار bit_cnt را برابر با ۰۱۱۱۱ قرار می‌دهیم.در اخر مقدار cnt_limit نیز برابر صفر قرار می‌دهیم.

```
when write_s =>                                                    -- write state
  rw_ctrl_int               <='0';
  cs_n_int                  <='0';
  tx                        <=data_in_int(to_integer(bit_cnt));
  data_path.rx_data         <=(others=>'0');

  if(bit_cnt /= cnt_limit)then                                     -- to check all data's bit transfer
      state                 <=write_s;
      bit_cnt               <=bit_cnt - 1;
      cnt_limit             <=cnt_limit;

  else
      state                 <=idle;
      bit_cnt               <="01111";
      cnt_limit             <=(others=>'0');

  end if;
```

شکل۸-۳) حالت write

با فرض آن که باارزش ترین بیت دستورالعمل برابر با یک باشد، State بعدی delay_read است، این State به این دلیل گذاشته شده است که از درست رفتن اخرین بیت داده جلوگیری شود. در این حالت، State بعدی خود را حالت read معرفی می‌کند و مقدار cs_n_init را هم برای فعال بودن تراشه برابر صفر قرار می‌دهد. به دلیل این که می‌خواهیم به حالت read برویم، پس باید مقدار rw_ctrl_init برابر با یک قرار دهیم. و به سبب این که دیتایی ارسال نمی‌شود و باید دیتا دریافت کنیم باید مقدار tx را برابر با z قرار دهیم. زیرا اگه صفر یا یک بدهیم، به این معنی می‌شود که داده‌ای دارد.

در ادامه این حالت، برای این که داده‌ای به صورت سریال بخوانیم، یک واحد از bit_cnt کم می‌کنیم و برای رعایت استانداردهای موجود، مقدار cnt_limit را در خودش می‌ریزیم. در اخر این حالت نیز باید مقادیر که بر روی پایه sdio قرار داده می‌شود را بخوانیم

```
when delay_read =>                                                    --for dont lose bit
  state                          <=read_s;
  rw_ctrl_int                    <='1';
  cs_n_int                       <='0';
  tx                             <='Z';
  bit_cnt                        <=bit_cnt - 1;
  cnt_limit                      <=cnt_limit;
  data_path.rx_data (to_integer(bit_cnt))  <=sdio;
```

شکل۹-۳) حالت delay_read

اخرین State که باید بررسی کنیم، حالت read است. در این جا به دلیل این که حالت دیگری باقی نمانده است، می‌توان به جای استفاده از کلمه read، از کلمه others استفاده کرد. در این حالت به سبب این که می‌خواهیم داده دریافت کنیم یا به عبارتی داده را بخوانیم پس باید سیگنال rw_ctrl_init یک باقی بماند و به دلیل این که باید تراشه فعال باشد، مقدار cs_n_init نیز صفر باقی می‌ماند. در این State نیز مانند State، delay_read باید مقدار سیگنال tx برابر با z باقی بماند. در ادامه نیز برای دریافت داده‌ها از پایه sdio، آن‌ها را داخل سیگنال مورد نظر assign می‌کنیم. در ادامه باید بررسی کنیم که آیا تمام بیت‌ها دریافت شده است یا خیر، اگر تمام بیت‌ها دریافت نشده باشد در همین حالت باقی مانده تا تمام بیت‌ها به صورت سریال، بیت به بیت دریافت شوند. در غیر این صورت به State، idle برمی‌گردیم و مقدار bit_cnt را برابر با ۱۵ قرار می‌دهیم تا ۱۶ بیت دستورالعمل را بتوانیم در مرحله بعدی دریافت کنیم.

```
            when others =>                                                -- read_s state
              rw_ctrl_int                <='1';
              cs_n_int                   <='0';
              tx                         <='Z';
              cnt_limit                  <=cnt_limit;
              data_path.rx_data (to_integer(bit_cnt))  <=sdio;

                if(bit_cnt /= cnt_limit)then                              -- to check all data's bit recive
                   state                 <=read_s;
                   bit_cnt               <=bit_cnt - 1;

                else
                   state                 <=idle;
                   bit_cnt               <="01111";

                end if;

          end case;
       end if;
    end process;
end Behavioral;
-----------------------------------end architecture
```

شکل۱۰-۳) حالت read

به این ترتیب کد برنامه به پایان می‌رسد. قبل از شرح دادن شبیه سازی انجام شده، باید با رجیسترهای این تراشه آشنا بشویم تا بتوانیم با مقدار دهی به رجیسترهای آن، از تراشه استفاده کنیم.

memory mapping -3-2

پیش از آن که شبیه سازی انجام شده، مشاهده شود، بهتر است با بعضی از آدرس‌های رجیستر این IC که برای تنظیمات اولیه IC است، آشنا شویم. آدرس اول حافظه رجیستر یعنی 0x00 برای پیکربندی SPI این IC است. مقدار پیش فرضی که در این خانه قرار دارد برابر است با 0x18. این مقدار سبب می‌شود تا IC ابتدا از باارزش ترین بیت دیتا خود برای ارسال اطلاعات استفاده کند. اگر بخواهیم با کم ارزش ترین بیت دیتا برای ارسال اطلاعات شروع کنیم، باید مقادیر خانه‌های دوم و هفتم، ادرس حافظه 0x00 را برابر با یک بدهیم. در همین آدرس از این رجیستر، خانه‌های سوم و ششم، سبب برگشتن IC به تنظیمات اولیه خود میشود.

همانطور که ذکر شده AD9628، یک IC دو کانالِ است. با مقدار دهی به آدرس حافظه 0x05 این رجیستر می‌توان مشخص کرد که برای ارسال دستور بعدی، از کدام کانال می‌توان استفاده کرد. مقدار پیش فرض این آدرس برابر است با 0x03 است، که سبب بکار گیری دو کانال موجود می‌شود.

آدرس 0x01 از این رجیستر، یک آدرس فقط خواندنی است که ID تراشه در آن قرار دارد که ID این تراشه برابر است با 0x89

آدرس 0x08 این رجیستر، مربوط به حالت‌های عملکرد تراشه است. به طور پیش فرض مقدار این آدرس 0x00 است. در خانه اول و دوم این آدرس اگر مقدار 0b00 قرار دهیم سبب می‌شود تا تراشه عملکرد عادی داشته باشد، اگر مقدار 0b01 قرار بدهیم، سبب می‌شود تراشه به طورکامل کار نکند، اگر مقدار 0b10 بدهیم سبب می‌شود تا به حالت standby برود و اگر مقدار 0b11 بدهیم، سبب reset شدن تراشه می‌شود.

آدرس 0x09، برای تثبیت کردن duty cycle تراشه است. به طور پیش فرض مقدار این آدرس 0x01 می‌باشد. اگر بخواهیم تثبیت کننده غیر فعال شود باید خانه اول این آدرس را صفر بدهیم.

آدرس 0x0B، مربوط به مقسم کلاک است. مقدار پیش فرض این آدرس برابر است با 0x00. اگر سه بیت کم ارزش را برابر با 0b000 قرار بدهیم، با همان کلاک که دریافت می‌کند، کار میکند.اگر مقدار آن سه بیت برابر با 0b001 بدهیم، سبب می‌شود سرعت کار این تراشه دو برابر شود. در تصویر زیر تغییر کلاک تراشه را با توجه به سه بیت دریافتی ذکر شده را شرح میدهد.

Clock divide ratio
000 = divide by 1
001 = divide by 2
010 = divide by 3
011 = divide by 4
100 = divide by 5
101 = divide by 6
110 = divide by 7
111 = divide by 8

شکل۱۱-۳) خانه‌های آدرس مربوط به مقسم کلاک

آدرس 0x16 این رجیستر، مربوط به تنظیم اختلاف تاخیر در تعداد کلاک های، مقسم کلاک و clock که برای رمزگذاری در کار ADC به کار برده می‌شود، است. مقدار پیش فرض این آدرس برابر است با 0x00 که سبب میشود که DCO[1] به صورت معکوس کار نکند و دو کلاک ذکر شده، نسبت به هم تاخیر نداشته باشند. اگر سه بیت کم ارزش، این آدرس را 0b001 قرار بدهیم، سبب می‌شود تا دو کلاک ذکر شده نسبت بهم به اندازه یک کلاک ورودی تاخیر داشته باشند. در تصویر زیر تاخیر این دو کلاک را نسبت بهم با توجه به سه بیت کم ارزش مشاهده می‌کنید.

000 = no delay
001 = one input clock cycle
010 = two input clock cycles
011 = three input clock cycles
100 = four input clock cycles
101 = five input clock cycles
110 = six input clock cycles
111 = seven input clock cycles

شکل۱۲-۳)خانه های آدرس مربوط به Clock phase control

آدرس 0x17، مربوط به تاخیر کلاک خروجی است. مقدار پیش فرض این آدرس برابر است با 0x00 که سبب می‌شود، تاخیر کلاک DCO نداشته باشیم و تاخیری در داده خروجی نداشته باشیم. برای این که تاخیر کلاک خروجی را عوض کنیم، ابتدا باید خانه ششم از این آدرس را برابر با یک قرار بدهیم، و با توجه به تاخیر مورد نظر ، سه بیت کم ارزش این آدرس را با توجه به شکل زیر تغییر بدهیم.

Delay selection
000 = 0.56 ns
001 = 1.12 ns
010 = 1.68 ns
011 = 2.24 ns
100 = 2.80 ns
101 = 3.36 ns
110 = 3.92 ns
111 = 4.48 ns

شکل۱۳-۳) خانه های آدرس مربوط به تاخیر خروجی

آدرس 0x18 این رجیستر، مربوط به تعین ولتاژ مرجع برای ADC، در صورت استفاده از ولتاژ مرجع داخلی است. مقدار پیش فرض این آدرس برابر است با 0x04 . که سبب می‌شود ولتاژ پیک تا پیک مرجع ۲ ولت باشد. برای در نظر گرفتن ولتاژهای مرجع دیگر، باید مقادیر سه بیت کم ارزش این آدرس را تغییر بدهیم. با توجه به شکل زیر می‌توان ولتاژ مرجع دلخواه را مشخص کرد.

[1]digitally controlled oscillator

Internal V_{REF} digital adjustmer
000 = 1.0 V p-p
001 = 1.14 V p-p
010 = 1.33 V p-p
011 = 1.6 V p-p
100 = 2.0 V p-p

شکل۱۴-۳) خانه‌های آدرس مربوط به تنظیم ولتاژ مرجع

آدرس 0x100 این رجیستر، مربوط به نرخ نمونه برداری ADC است. خانه های ششم، پنجم و چهارم از این آدرس resolution، ADC را مشخص میکند، به این معنی که ولتاژ انالوگی که دریافت میکند به 2^{10} یا 2^{12} پله تقسیم می‌کند. اگر مقدار این سه خانه را برابر با 0b100 قرار بدهیم، خروجی ADC، ۱۲ بیتی می‌شود و اگر برابر با 0b110 قرار دهیم، خروجی ۱۰ بیتی می‌شود. برای تعیین نرخ نمونه برداری ADC نیز باید سه بیت کم ارزش این آدرس را مقدار بدهیم. با توجه به شکل زیر می‌توان نرخ نمونه برداری را مشخص کنیم.

Sample rate
011 = 80 MSPS
100 = 105 MSPS
101 = 125 MSPS

شکل۱۵-۳) خانه‌های ادرس مربوط به نرخ نمونه برداری

آدرس 0x3A نیز، برای همگام سازی بین مدارات داخلی این تراشه است، که برای این کار باید مقدار 0x00 به این آدرس داده شود.

آدرس 0x2E، برای قرار دادن خروجی بر روی یکی از دو کانال موجود در تراشه است. اگر کم ارزش‌ترین بیت این آدرس برابر با صفر باشد، خروجی در کانال A و اگر برابر با یک باشد در کانال B قرار میگیرد.

Addr (Hex)	Register name	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	Default value(Hex)
0x00	SPI port config	open	LSB	Soft reset	1	1	Soft reset	LSB	open	0x18
0x01	Chip ID	8-bit chip ID [7:0]					AD9628=0x89			Read only
0x08	Power mode	open	opne	External power down pin function 0 = PDWN 1 = standby	open	open	open	Intenal power-down mode 00=normal 01=full power 10=standby 11=digital reset		0x00
0x09	Global clock	open	open	open	open	open	open	open	Duty cycle stabilizer 0=disable 1=enable	0x01
0x0B	Clock divide	open	open	open	open	open	Clock divide ratio 000=divde by 1 001=divde by 2 010=divde by 3			0x00

							011=divde by 4 100=divde by 5 101=divde by 6 110=divde by 7 111=divde by 8			
0x16	Clock phase control	Invert DCO clock 0 = not inverted 1 = inverted	Open	Open	Open	Input clock divider phase adjust relative to the encode clock 000 = no delay 001 = one input clock cycle 010 = two input clock cycles 011 = three input clock cycles 100 = four input clock cycles 101 = five input clock cycles 110 = six input clock cycles 111 = seven input clock cycles				**0x00**
0x17	Output delay	DCO Clock delay 0 = disabled 1 = enabled	Open	Data delay 0 = disabled 1 = enabled	Open	Delay selection 000 = 0.56 ns 001 = 1.12 ns 010 = 1.68 ns 011 = 2.24 ns 100 = 2.80 ns 101 = 3.36 ns 110 = 3.92 ns 111 = 4.48 ns				0x00
0x18	VREF select	Open	Open	Open	Open	Internal VREF digital adjustment 000 = 1.0 V p-p 001 = 1.14 V p-p 010 = 1.33 V p-p 011 = 1.6 V p-p 100 = 2.0 V p-p				0x04
0x2E	Output assign	Open	Open	Open	Open	Open	Open	Open	0 = ADC A 1 = ADC B	ADC A = 0x00 ADC B = 0x01
0x3A	Sync control	Open	Open	Open	Open	Open	Clock divider next sync only	Clock divider sync enable	Open	**0x00**
0x100	Sample rate override	Open	Sample rate override enable	Resolution 100 = 12 bits 110 = 10 bits			Sample rate 011 = 80 MSPS 100 = 105 MSPS 101 = 125 MSPS			0x00

جدول۱-۳) memory map

3-3- شبیه سازی پروتکل

پس ازسنتز کردن، کد نوشته شده و رفع خطاهای موجود باید وارد بخش شبیه سازی پروتکل شویم.برای شبیه سازی پروتکل نوشته شده، باید یک فایل test bench ایجاد کنیم. پس از ایجاد فایل test bench خود برنامه، کدهایی را تولید می‌کند. برای شبیه سازی باید در کدهای ایجاد شده یک سری تغییراتی ایجاد کنیم.

```
--------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

------------------------

ENTITY spi_3wire_tb IS
END spi_3wire_tb;

-------------------------

ARCHITECTURE behavior OF spi_3wire_tb IS

    ---Inputs
   signal CLK_SYS : std_logic := '0';
   signal SCLK   : std_logic := '0';
   signal Start      : std_logic := '0';
   signal Data_In : std_logic_vector(47 downto 0) := (others => '0');

    ---BiDirs
   signal SDIO   : std_logic;

    ---Outputs
   signal CS_n   : std_logic;
   signal RW_CTRL    : std_logic;

    --- Internal Signal
    signal SCLK_Start      : std_logic := '0';
```

شکل۱۶-۳) تعریف‌های اولیه test bench

از جمله تغییراتی که باید بدهیم، تنظیم کردن کلاک تراشه است.همانطور که در فصل دوم هم توضیح داده شد، یک دوره کلاک برای SPI برابر است با 40ns است. پس باید این مورد را در test bench تصحیح کرد.

```
---Clock period definitions
constant CLK_SYS_period : time := 40 ns;
constant SCLK_period    : time := 40 ns;

 BEGIN

 --- Instantiate the Unit Under Test (UUT)
uut: entity work.spi_3wire PORT MAP (
        CLK_SYS => CLK_SYS,
        SCLK    => SCLK,
        Start   => Start,
        Data_In => Data_In,
        CS_n    => CS_n,
        RW_CTRL  => RW_CTRL,
        SDIO    => SDIO
      );

 --- CLK_SYS generator
 CLK_SYS_Pro :process
begin
    CLK_SYS <= '0';
    wait for CLK_SYS_period/2;
    CLK_SYS <= '1';
    wait for CLK_SYS_period/2;
end process CLK_SYS_Pro;
```

شکل۱۷-۳) تنظیم کلاک سیستم

تمامی سیگنال‌ها همان سیگنال‌های موجود در کد هستند به جز سیگنال SCLK_Start که شروع کلاک زدن SCLK را مشخص می‌کند و با استفاده از سیگنال SCLK_Start، سیگنال SCLK تولید شده است. به علت این که بیتی که تو هر لبه داریم مشاهده می‌کنیم، درواقع مربوط به تغییرات لبه‌ی قبلی است، پس بنابراین بایدSCLK را طوری تنظیم میکنیم که در یک لبه باهم تاخیر دارند. برای این که توان مصرفی زیادی مصرف نکنیم، SCLK را از صفر شروع نکردیم.

```
--- SCLK_Start generator
  SCLK_Start_Pro :process
  begin
      SCLK_Start <= '0','1' after     300ns;
 wait;
  end process SCLK_Start_Pro;
  ---

  --- SCLK Generate
  SCLK_Pro : process
  begin
      ---
      if(SCLK_Start = '1') then
          ---
          SCLK <= '0';
          wait for SCLK_period/2;
          SCLK <= '1';
          wait for SCLK_period/2;
          ---
      else
          ---
          SCLK <= '0';
          wait until SCLK_Start = '1';
          ---
      end if;
      ---
  end process SCLK_Pro;
  ---
```

شکل۱۸-۳) تنظیم SCLK_Start

پس از تنظیم کلاک‌ها، باید سیگنال start را تنظیم کنیم. به دلیل این که SCLK از صفر شروع نشده، در نتیجه start را بعد از شروع شدن کلاک SCLK تغییر مقدار می‌دهیم. اولین زمانی که مقدار start را یک می‌کنیم چند نانوثانیه قبل از لبه بالا رونده clk_sys است، به این دلیل که در لبه بعدی مقداری که تغییر کرده را مشاهده کنیم. و برای این که مطمئن باشیم که تغییر را مشاهده کردیم، به اندازه یک دوره، مقدار start را یک نگه داشته و سپس صفر می‌کنیم. در ادامه باید ببنییم که چه زمانی تمام بیت‌های انتقال می‌یابد و پروتکل دوباره به حالت idle برگشته است. سپس اگر خواستیم پروتکل SPI را دوباره با دادن

پالس به سیگنال start راه‌اندازی کنیم. به همین علت زمان‌هایی که به سیگنال start نسبت داده شده است، ابتدا شبیه سازی شده و زمان بازگشت به حالت idle، مشاهده شده است.

```
--- Start generator
Start_Pro :process
begin
    ---
        Start  <= '0', '1' after    335ns, '0' after   375ns, '1' after  1455ns, '0' after  1495ns,
                  '1' after  3215ns, '0' after  3255ns, '1' after  4775ns, '0' after  4815ns;
wait;

end process start_Pro;
```

شکل۱۹-۳) تنظیم start

پس از آن باید داده‌هایی را که می‌خواهیم ارسال یا دریافت کنیم را به سیگنال Data_In نسبت بدهیم. این نکته قابل ذکر است که اگر بخواهیم داده‌ای دریافت کنیم، ۱۶ بیت باارزش Data_In که مربوط به دستورالعمل است، فقط ارسال می‌شود.

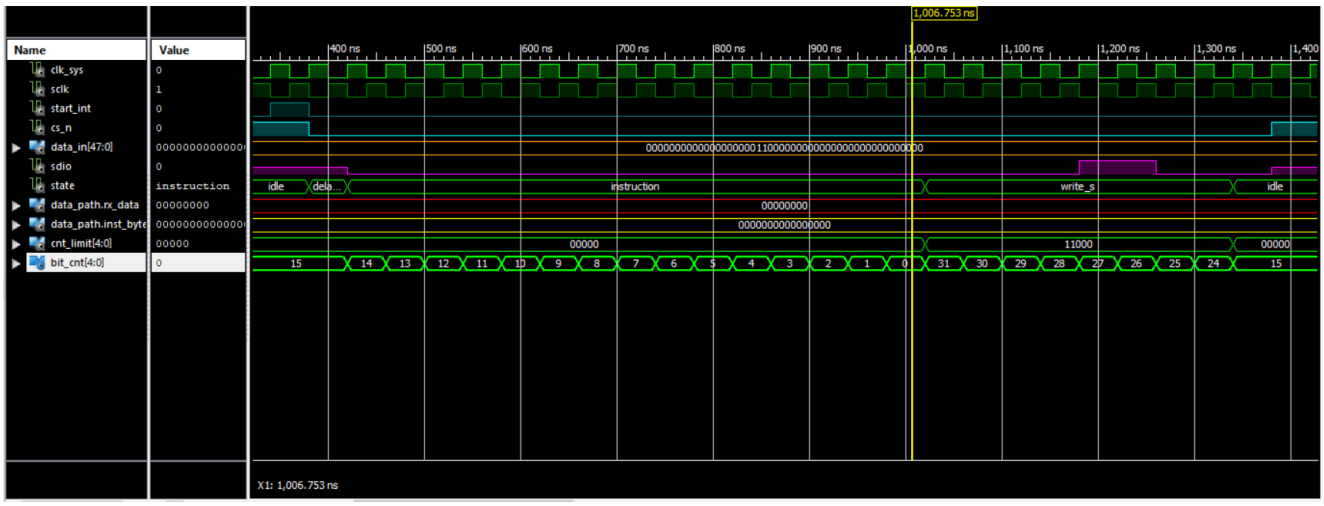
مقادیر داده شده به سیگنال Data_In، با توجه به رجیستری است که در بخش قبلی توضیح داده شده بود. اولین ارتباطی که می‌خواهیم برقرار کنیم، config کردن پروتکل SPI است، که در آدرس 0x00 قرار دارد و مقدار پیش فرضی که در دیتاشیت ذکر شده است ارسال می‌کنیم(0x18). به دلیل این که میخواهیم داده بنویسیم پس باید باارزش ترین بیت صفر باشد، سپس چون یک بایت می‌خواهیم ارسال کنیم باید دو بیت بعدی صفر باشد و چون آدرسی که میخواهیم داخل آن مقداری بریزیم 0x00 است در نتیجه سیزده تا صفر دیگر باید بگذاریم تا بیت‌های دستورالعمل تکمیل شود. در ادامه در هشت بیت باید مقدار 0x18 قرار دهیم و مابقی بیت‌ها اعتباری ندارد، زیرا تنها یک بایت ارسال می‌شود بنابراین با مابقی بیت‌ها را صفر می‌کنیم.

```
--- data_In_generator
Data_In_Pro:process
begin
    ---
    Data_In <= "00000000000000000011000000000000000000000000000",
               "00100000000010000000000000000001000000000000000" after 1470ns,
               "10000000000000010000000000000000000000000000000" after 3230ns,
               "01000000000101100000000000000000000010000000000" after 4790ns;

    wait;
    ---
end process Data_In_Pro;
```

شکل۲۰-۳) مقدار دهی به Data_in

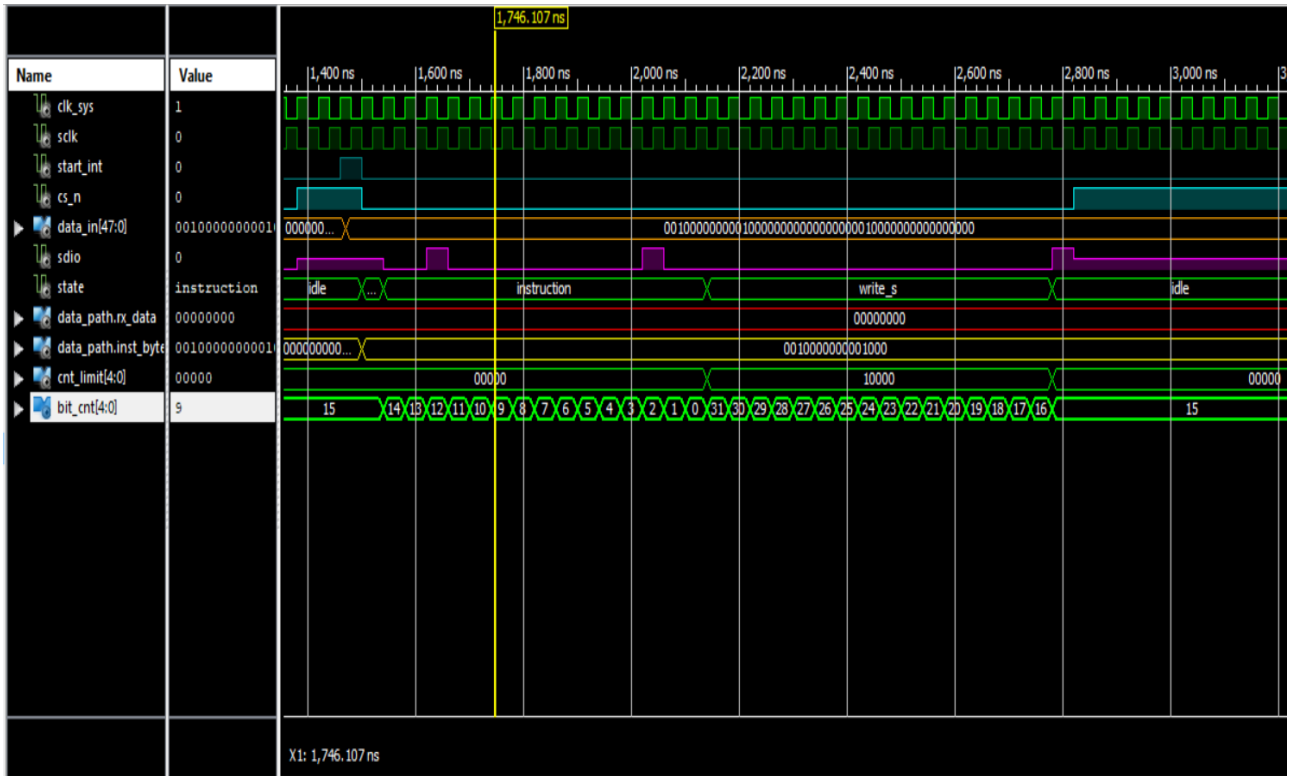
همانطور که در شکل زیر ملاحظه میشود، پس از خوردن start، هم زمان cs_n صفر می‌شود و تراشه شروع به کار میکند و از حالت idle خارج می‌شود وارد بخش delay_instruction می‌شود. پس از یک کلاک به بخش instruction انتقال می‌یابیم. پس از دریافت ۱۶ بیت دستور العمل به state، write می‌رویم و یک بایت دریافت می‌کنیم.



شکل۲۱-۳) شبیه سازی اولین دیتا ارسالی

در ادامه برای شبیه سازی بیشتر، با چند آدرس دیگر از این رجیستر کار می‌کنیم. دیتا بعدی که میخواهیم ارسال کنیم، دو بایت داده 0x00 و 0x01 در آدرس‌های 0x08 و 0x09 می‌باشد. در نتیجه با ارزش ترین بیت را بار دیگر برابر با صفر قرار می‌دهیم و به دلیل این که می‌خواهیم دو بایت ارسال کنیم باید دو بیت بعدی را برابر با 0b01 قرار دهیم. اکنون باید آدرس حافظه‌ای که می‌خواهیم بنویسم را انتخاب کنیم. زمانی که آدرس یک حافظه را مشخص می‌کنیم، ولی اگر دو بایت داده برای ارسال داشته باشیم، بایت اول در آدرس مشخص شده، ارسال میشود و بایت بعدی در آدرس حافظه بعدی نوشته می‌شود. به همین سبب در بیت‌های دستورالعمل، بیت چهارم دستور العمل را برابر با یک قرار میدهیم تا آدرس 0x08 انتخاب شود. سپس در بیت‌های داده، مقادیر ذکر شده را قرار می‌دهیم. بایت اول داده را برابر با 0x00 و بایت دوم را برابر با 0x01 قرار می‌دهیم.

همانطور که در شکل زیر می‌بینید مقدار bit_cnt از ۳۱ تا ۱۶ شمارش میکند که نشان دهنده آن است که دو بایت داده ارسال کرده است. و با توجه به شکل می‌توان دید پیش از آن که cs_n یک شود، اخرین مقداری که sdio دارد برابر با یک است، که با توجه به داده ای که ارسال کردیم، صحیح می‌باشد.



شکل۲۲-۳) شبیه سازی دومین دیتا ارسالی

در ادامه یک بایت داده را از تراشه دریافت می‌کنیم. به دلیل این که SDIO هم مانند کد اصلی دارای یک مدار ترکیبی است که به rw_ctrl وابسته است. به صورتی که اگر rw_ctrl برابر با صفر باشد مقادیر را از بدنه کد میگیرد و در صورتی که یک باشد مقادیر را از test_bench دریافت می‌کند.

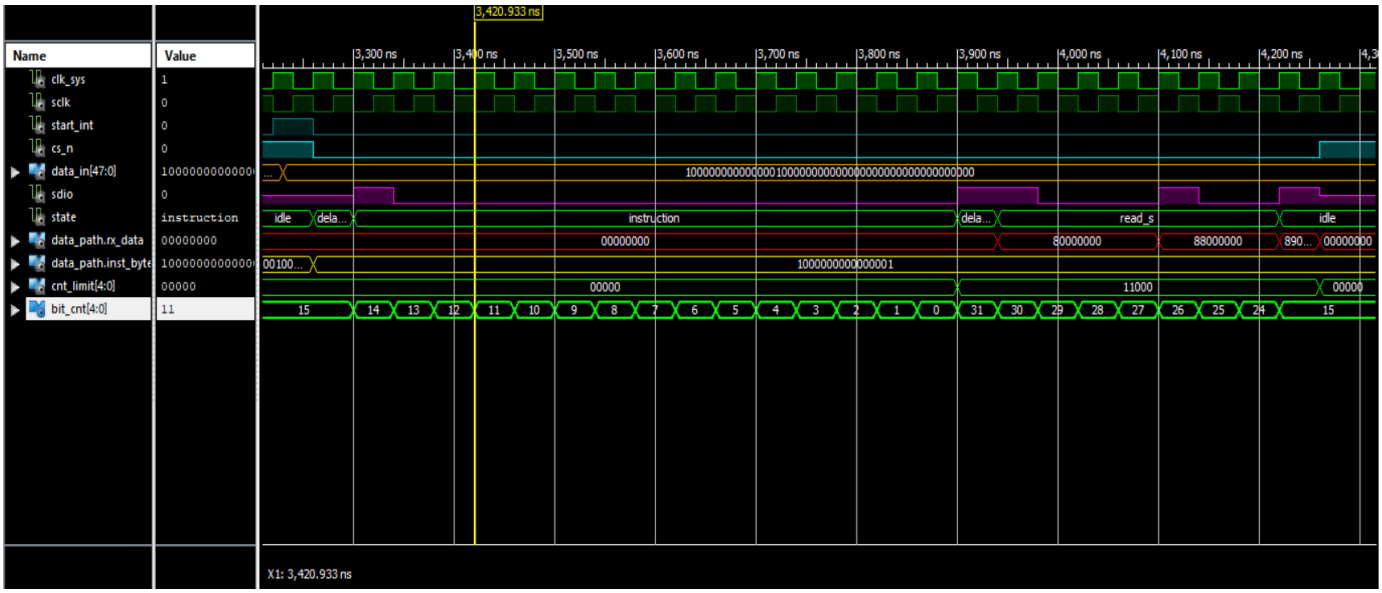
```
--- SDIO genarator
SDIO <= 'Z' when RW_CTRL = '0' else '1'                                    , '0' after SCLK_period*1,
                                        '1' after SCLK_period*4 , '0' after SCLK_period*5,
                                        '1' after SCLK_period*7 ;

END;
```

شکل۲۳-۳) داده‌های مربوط به دریافت از تراشه

برای شبیه سازی بخش مربوط به دریافت داده از تراشه، با آدرس 0x01 که یک آدرس فقط خواندنی است استفاده می‌کنیم و مقداری که باید دریافت کنیم برابر است با 0x89. به همین علت زمانی که rw_ctrl برابر یک باشد، مقدار 0x89 را با استفاده از کلاک تنظیم می‌کنیم که در داخل SDIO قرار بگیرد. همانطور که در سیگنال data_path.rx_data مشاهده می‌شود، مقداری که دریافت کرده است برابر است با 0x89 . این نکته هم حائز اهمیت است، به سبب این که می‌خواهیم داده دریافت کنیم، باارزش ترین بیت دستور العمل باید

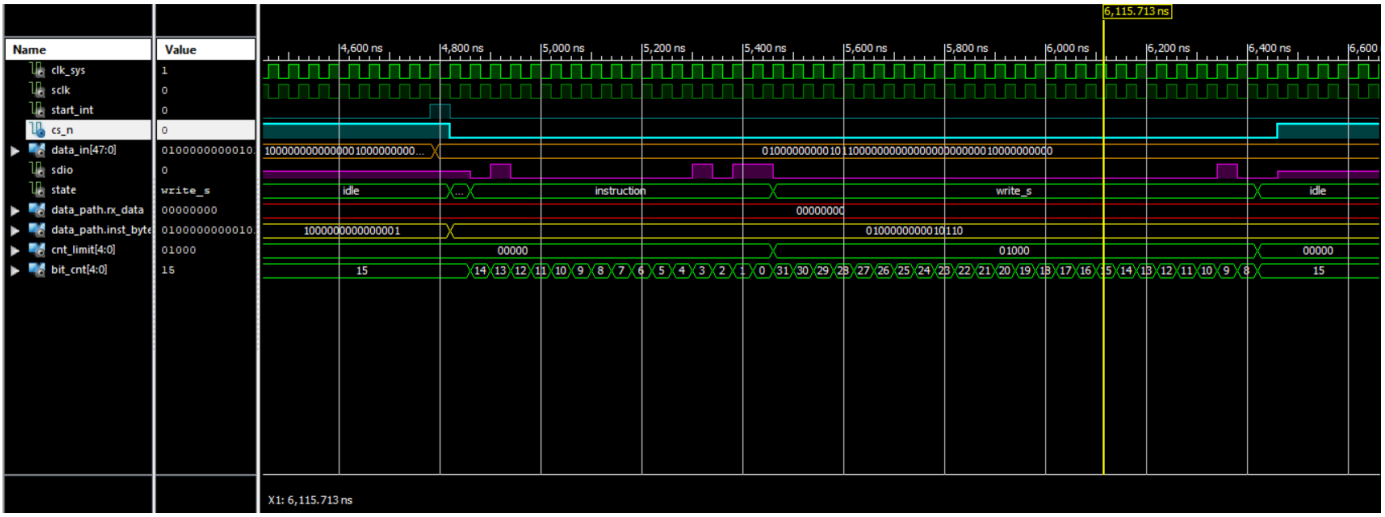
برابر با یک باشد و چون می‌خواهیم یک بایت داده دریافت کنیم دو بیت بعدی باید صفر باشند.



شکل۲۴-۳) شبیه سازی دریافت دیتا

شبیه سازی بعدی را برای آدرس‌های 0x16، 0x17 و 0x18 انجام می‌دهیم. با توجه به دیتاشیت، مقادیری که ارسال می‌کنیم به ترتیب برابر با 0x00، 0x00 و 0x04 می‌باشد. به همین علت پس از این که state، idle را مشاهده کردیم، به سگنال start یک پالس میدهیم تا تراشه آماده ارسال دیتا شود. سپس به دلیل این که میخواهیم داده ارسال کنیم باید باارزش ترین بیت دستورالعمل صفر باشد و به علت این که سه بایت داده میخواهیم ارسال کنیم دو بیت بعدی باید 0b10 باشند. پس از آن در ۱۳ بیت آدرسی که در بیت‌های دستورالعمل قرار دارد مقدار 0x16 را قرار میدهیم که پس از آن دو بایت دیگر را در آدرس‌های بعدی قرار دهد. پس از ۱۶ بیت دستورالعمل باید سه بایت داده را با توجه به مقادیر ذکر شده بنویسیم. به همین علت دو بایت اول داده برابر با صفر قرار داده و بایت سوم را برابر با 0x04 قرار میدهیم.

همانطور که در شکل زیر دیده می‌شود، bit_cnt از ۳۱ تا ۸ شمارش میکند که نشان دهنده آن است که سه بایت داده ارسال شده است.



شکل۲۵-۳) شبیه سازی سومین دیتا ارسالی