پروژه پایانی درس VHDL

حمیدرضا محمدی جوزانی ۴۰۱۶۱۲۱۳۵

علی مهدوی ۴۰۱۶۱۱۲۴۷

استاد : دکتر میرزا کوچکی

(پروتکل I2C برای DAC63202)

مشخصات دیتاشیت برای پروتکل I2C :

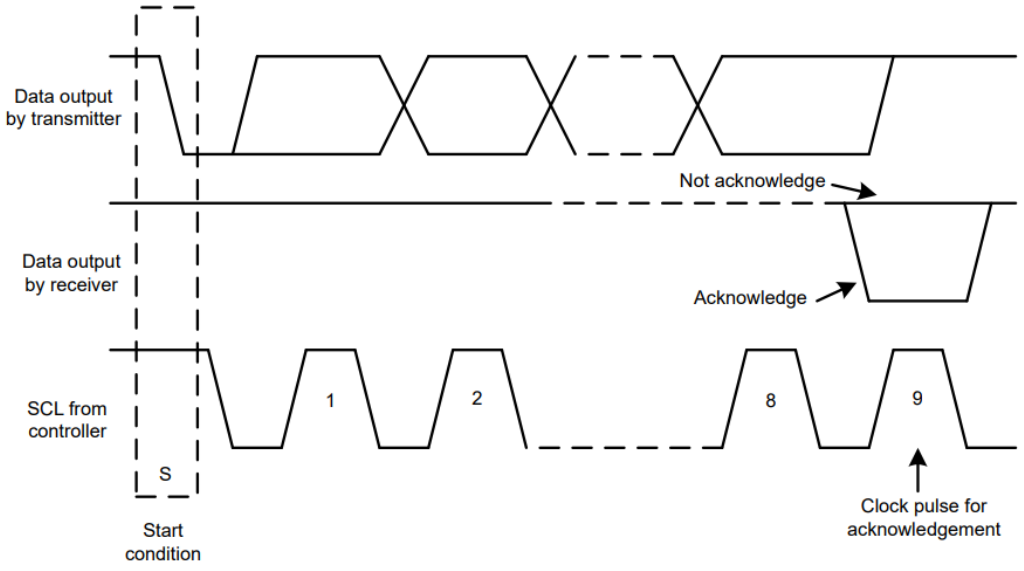
طبق مشخصات دیتاشیت ساختار باس های SDA و SCL در پروتکل I2C باید بصورت Pullup باشدیعنی در حالت idle این دو لاین یک هستند.

دراین پروژه کنترلر FPGA و تارگت DAC انتخابی ما میباشد. FPGA باید سیگنال SCL و همچنین تایمینگ های مورد نیاز پروتکل را (Start condition, stop condition) را تولید کند.

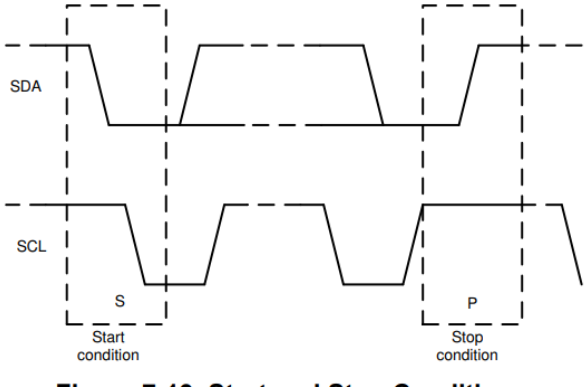
بطور کلی پروتکل i2c در این دیوایس بعنوان یک فرستنده عمل میکند اما اگر کنترلر نیاز به دسترسی به اطلاعات رجیستر های داخلی DAC داشته باشد میتواند بعنوان گیرنده اطلاعات نیز عمل کند.

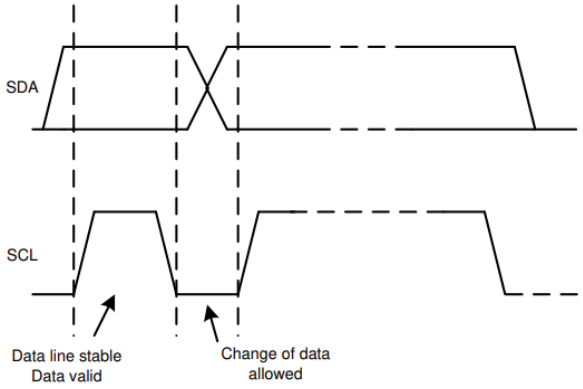
مود های کاری مختلفی برای ارسال اطلاعات وجود دارد که در اینجا ما از مود STANDARD با سرعت 100Kbps استفاده میکنیم.

در ارتباط i2c پس از دریافت هر بایت دیتا در کلاک نهم از SCL باید یک بیت acknowledge توسط تارگت ارسال شده و در صورت صفر بود این بیت میتوانیم باید بعدی را برای تارگت ارسال کنیم در غیر این صورت سیگنال ارور فعال شده و ارسال اطلاعات متوقف میشود تا زمانی که start condition دوباره اتفاق بیوفتد و ارسال اطلاعات مجددا اغاز شود. شکل زیر Start condition و acknowledge را نشان میدهد.



همانطور که در شکل زیر پیداست برای بوقوع پیوستن حالت های start و stop باید به ترتیب در زمان یک بودن SCL لاین SDA از یک به صفر و از صفر به یک تغییر پیدا کند.





هر سیکل در ارسال دیتا شامل مراحل زیر میباشد:

MSB	....	LSB	ACK	MSB	...	LSB	ACK	MSB	...	LSB	ACK	MSB	...	LSB	ACK
Address (A) byte Section 7.5.2.2.1				Command byte Section 7.5.2.2.2				Data byte - MSDB				Data byte - LSDB			
DB [31:24]				DB [23:16]				DB [15:8]				DB [7:0]			

در هشت بیت اول ادرس رجیستر مورد نظر به همراه بیت RW که نوشتن یا خواندن از DAC را مشخص میکند ارسال میشود و پس از ان در کلاک نهم ack ارسال میشود. در هشت بیت دوم به اصطلاح command byte ارسال میشود. در دوبایت اخر هم دیتا های اصلی فرستاده میشوند. پس در کل ۳۲ بیت اصلی بعلاوه ۴ بیت acknowledge در یک سیکل ارسال میشوند بنابراین نیاز به ۳۶ کلاک برای بوقوع پیوستن یک سیکل داریم.

چهار بیت اول ادرس ثابت بود و مقدار ان ۱۰۰۱ میباشد. اما سه بیت مابقی را یکی از پین های DAC به اسم A0 تعیین میکند که بصورت زیر میباشد:

TARGET ADDRESS	A0 PIN
000	AGND
001	VDD
010	SDA
011	SCL

در اینجا برای راحتی کار فرض میکنیم در کل شبیه سازی پایه A0 به زمین متصل است بنابراین سه بیت اخر ادرس برابر صفر میشود.

بایت Command هم در حقیقت مود کاری DAC را تعریف میکند.

بررسی کد VHDL پروتکل I2C

چرخه نوشتن اطلاعات :

پورت های ورودی خروجی در کنترلر (FPGA) بصورت زیر میباشد:

```
port(clk            :       in      STD_LOGIC                       := '0';  -- standard mode (fc = 50 Mhz)
     reset_n        :       in      STD_LOGIC                       := '1';
     enable         :       in      STD_LOGIC                       := '0';
     busy           :       out     STD_LOGIC                       := '0';
     ack_error      :       inout   STD_LOGIC                       := '0';
     sda            :       inout   STD_LOGIC                       := 'Z';
     scl            :       out     STD_LOGIC                       := 'Z';
     rw             :       in      STD_LOGIC                       := '0';
     rw_ctrl        :       out     STD_LOGIC                       := '0';
     read_mode      :       out     STD_LOGIC                       := '0';
     command_byte   :       in      STD_LOGIC_vector(7 downto 0)    := (others => '0');
     addres         :       in      STD_LOGIC_vector(6 downto 0)    := "1001000"; --(A0 is connected to GND)
     data_wr        :       in      STD_LOGIC_vector(7 downto 0)    := (others => '0');
     data_rd        :       out     STD_LOGIC_vector(7 downto 0)    := (others => '0');
     rd_cnt         :       buffer  integer         range 0 to 7    := 0
```

سیگنال RESET_N بیت ورودی به کنترلر برای ریست کردن پروتکل است که اینجا ان را بصورت اسنکرون طراحی کردیم.

سیگنال BUSY خروجی است کنترلر است که یک بودن ان به معنی این است که در حین انجام عملیات انتقال دیتا هستیم.

سیگنال ACKNWOLEDGE_ERROR نیز یک پورت خروجی است که یک بودن ان حاجی از این است که تارگت ما پیس از دریافت هشت بیت دیتا، بیت acknowledge را برای ما ارسال نکرده است.

پورت های SDA و SCL هم ارتباط اصلی بین کنترلر و تارگت هستند که در اینجا SCL را فقط به عنوان خروجی تعریف کرده ایم زیرا طبق دیتاشیت DAC63202 این دیوایس نمیتواند سیگنال دیتایی ارسال کند و فقط ان را دریافت میکند. اما SDA بصورت ورودی خروجی تعریف شده چرا که باید بتواند بصورت دوطرفه اطلاعات را بین کنترلر و تارگت ردوبدل کند.

پورت RW ورودی کنکترلر است که مشخص کننده این است که در یک چرخه ارسال اطلاعات میخواهیم از DAC بخوانیم یا در ان بنویسیم.

پورت RW_CTRL یک خروجی است که به تارگت میگوید که در حال حاضر در حال انجام خواندن است یا نوشتن. در شبیه سازی نقش این سیگنال بیشتر مشخص میشود.

پورت READ_MODE یک خروجی است که مقدار ان در زمانی که بخواهیم اطلاعات را از DAC بخوانیم یک میشود.

پورت COMMAND_BYTE شامل هشت بیت دیتای ورودی به کنترلر میباشد که مود کاری DAC را در ارتباط I2C مشخص میکند. این مود در رجیستر مپ دیتاشیت این دیوایس موجود میباشند.

دو پورت DATA_IN و DATA_OUT هم بترتیب ورودی و خروجی به کنترلر میباشند که دیتا هایی که میخواهیم در DAC بنویسیم یا دیتایی که از ان خوانده شده است را نشان میدهند.

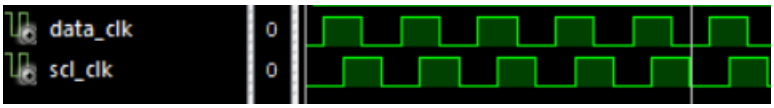
سیگنال های داخلی این که برای ارتباط I2C تعریف کردیم بصورت زیر میباشند:

```
type      machine is (ready ,start_con ,addr_seq ,slave_ack1 , command_seq , slave_ack2 , wr , rd ,slave_ack3 , mstr_ack , stop_con);
signal    state            :          machine                        := ready;
signal    data_clk         :          STD_LOGIC                      := '0';
signal    data_clk_prev    :          STD_LOGIC                      := '0';
signal    scl_clk          :          STD_LOGIC                      := '0';
signal    scl_en           :          STD_LOGIC                      := '0';
signal    sda_int          :          STD_LOGIC                      := '1';
signal    tx               :          STD_LOGIC                      := '0';
signal    sda_enable       :          STD_LOGIC                      := '0';
signal    addr_rw          :          STD_LOGIC_vector(7 downto 0)   :=(others => '0');
signal    command_byte_r   :          STD_LOGIC_vector(7 downto 0)   :=(others => '0');
signal    data_tx          :          STD_LOGIC_vector(7 downto 0)   :=(others => '0');
signal    data_rx          :          STD_LOGIC_vector(7 downto 0)   :=(others => '0');
signal    bit_cnt          :          integer range 0 to 7           := 7;
signal    read_mode_int    :          STD_LOGIC                      := '0';
signal    rw_ctrl_int      :          STD_LOGIC                      := '0';
```

بیشتر این سیگنال ها رجیستر هایی هستند که در بدنه اصلی کد از ان ها استفاده میکنیم که در ادامه بخش گزارش به هر یک از انها پرداخته خواهد شد. همانطور که در توضیحات دیتاشیت گفته شد نمیتوانیم از فرکانس سیگنال کلاک ورودی به FPGA بعنوان SCL استفاده کنیم و باید از فرکانس پایین تری برای این منظور استفاده کرد که ما اینجا از STANDARD MODE که فرکانسی برابر ۱۰۰ کیلو هرتز دارد استفاده میکنیم. از انجایی که فرکانس کلاک متصل به FPGA را ۵۰ مگاهرتز درنظر گرفتیم پس باید فرکانس کلاک SCL را با استفاده از یک مقسم فرکانسی بر ۱۵ تقسیم کنیم. کد نوشته شده برای این منظور به صورت زیر است :

```
if(reset_n = '0')then
    count := 0 ;
elsif(clk 'event and clk ='1')then
    data_clk_prev <= data_clk ;
    if(count = 499)then -- i2c clock = 50000000/500 = 100000 hz
        count := 0 ;
    else
        count := count + 1;
    end if;

    case count is

        when 0 to 124 =>
            scl_clk <= '0';
            data_clk <= '0';

        when 125 to 249  =>
            scl_clk <= '0';
            data_clk <= '1';

        when 250 to 374 =>
            scl_clk <= '1';
            data_clk <= '1';

        when 375 to 499 =>
            scl_clk <= '1';
            data_clk <= '0';
        when others =>
            null;
    end case ;
```

همین طور که در شکل ملاحظه میکنید دو سیگنال کلاک تعریف شده است که در شبیه سازی این دو سیگنال را ملاحظه میکنید:



همانطور که ملاحظه میکنید سیگنال کلاک SCL_CLK نسبت به DATA_CLK به اندازه ۱۸۰ درجه جلو تر است. دلیل اینکه سیگنال DATA_CLK را تعریف کردیم این است که در ارتباط I2C فقط در زمانی میتوان سیگنال SDA را تغییر داد که مقدار SCL صفر باشد و برای اینکه هنگام تغییر این مقدار این باس از صفر بودن SCL مطمئن باشیم یک سیگنال کلاک دیگر به اسم DATA_CLK ساختیم تا اگر خواستیم مقدار SDA را عوض کنیم ان را در لبه بالا رونده سیگنال DATA_CLK تغییر دهیم تا از صفر بودن کلاک SCL مطمئن باشیم.

در بدنهی اصلی کد VHDL ارتباط I2C درابتدا یک ریست اسنکرون قرار میدهیم که بصورت Active low میباشد.

```
process(clk ,reset_n)
begin

    if(reset_n = '0')then
        state <= ready;
        busy <= '1';     --in reset condition we cannot transport data
        scl_en <= '0';   -- scl is high impedance
        sda_int <= '1';  --sda is high impedance
        ack_error <= '0';
        rw_ctrl_int <= 'Z';
        bit_cnt <= 7 ; -- meaning that we reseted the bit counter
        data_rd <= (others => '0');
```

پس از ان (وقتی در حالت نوشتن هستیم) حالت های ماشین حالت را با هر لبه ی بالا رونده ی DATA_CLK اپدیت میکنیم. در شکل زیر حالت اول یعنی READY را توصیف میکنیم:

```
-- writing cycle
if(data_clk = '1' and data_clk_prev = '0')then -- rising e
    case state is

        when ready =>
            busy <= '0';
            rw_ctrl_int <= '0';
            if(enable = '1')then
               busy <= '1';
               addr_rw <= addres & rw ;
               data_tx <= data_wr;
               state <= start_con ;
            else
                scl_en <= '0';
                state <= ready ;
            end if ;
```

در این استیت کنترلر هنوز دستوری مبنی بر شروع انتقال اطلاعات دریافت نکرده است بنابراین پورت BUSY برابر با صفر و رجیستر RW_CTRL_INT که بصورت CUNCURRENT مقدارش در پورت RW_CTRL ذخیره میشود هم در مقدار اولیه خود یعنی صفر میباشد.

پس از اینکه پورت ENABLE برابر با یک شد انتقال اطلاعات شروع میشود. کد این قسمت بدین شکل است:

```
when start_con =>

    rw_ctrl_int <= '0';
    busy <= '1';
    sda_int <= addr_rw(bit_cnt);
    if(rw = '1')then
        addr_rw <= addres & rw ;
        state <= addr_seq;
    else
        state <= addr_seq ;
    end if;
```

همانطور که میبینم سیگنال های BUSY, RW_CTRL_INT مقدار دیفالت خود را در این قسمت دارند. همچنین در این استیت بیت MSB ادرس که در استیت قبلی در ADDR_RW رجیستر شده است هم درون سیگنال داخلی SDA_INT ریخته میشود. قسمت شرطی بعدی در این قسمت مربوط به بخش خواندن اطلاعات است که در بخش مربط به ان گفته میشود. در اینجا فرض میکنیم که پس از ریختن MSB در SDA_INT مستقیم به استیت ADDR_SEQ میرویم.

```
when addr_seq =>

    rw_ctrl_int <= '0';
    if(bit_cnt = 0)then
        bit_cnt <= 7 ;
        sda_int <= '1'; -- sda is high impedance so slave can send the acknowledge data
        if(addr_rw(0) = '0')then
            state <= slave_ack1 ;
            rw_ctrl_int <= '1';
        else
            state <= slave_ack2 ;
            rw_ctrl_int <= '1';
        end if;
    else
        bit_cnt <= bit_cnt - 1 ;
        sda_int <= addr_rw(bit_cnt - 1);
        state <= addr_seq ;
    end if;
```

در این استیت هم بطور عادی سیگنال RW_CTRL_INT برابر صفر است مگر اینکه پس از ارسال هر هشت بیت ادرس متوجه شویم بیت LSB ادرس که همان RW است برابر یک باشد و در ان صورت سیگنال

RW_CTRL_INT برابر با یک میشود. در حالت نوشتن پس از این استیت به استیت SLAVE_ACK1 و در حالت خواندن (RW = 1) پس از این استیت به استیت SLAVE_ACK2 میرویم.

در استیت SLAVE_ACK1 در حقیقت در کلاک نهم هستیم و در اینجا DAC باید یک بیت ACK برای ما بفرستد تا از درست بودن اطلاعات اطمینان حاصل شود :

```
when slave_ack1 =>

    rw_ctrl_int <= '0';
    state <= command_seq ;
    command_byte_r <= command_byte ;
    busy <= '1' ;
```

نکته ای که در اینجا وجود دارد صفر بودن RW_CTRL_INT میباشد که بدلیل اینجا باید کنترلر بیت را دریافت کند طبیعتا باید یک میبود. اما این سیگنال را ما در اخرین کلاک استیت قبلی یک کردیم که تایمینگ ما برای یک بودن این سیگنال در استیت SLAVE_ACK بهم نخورد. همچنین در این استیت مقدار پورت ورودی COMMAND_BYTE را رجیستر میکنیم تا در استیت بعدی از ان استفاده کنیم.

در استیت بعدی مشابه استیت ADDR_SEQ عمل میکنیم منتها در اینجا بایت ارسال شده مود کاری DAC را مشخص میکند.

```
when command_seq =>

    rw_ctrl_int <= '0';
    busy <= '1';
    if(bit_cnt = 0)then
        sda_int <= '1';
        bit_cnt <= 7 ;
        state <= slave_ack2;
        rw_ctrl_int <= '1';
    else
        bit_cnt <= bit_cnt - 1 ;
        sda_int <= command_byte_r(bit_cnt);
        state <= command_seq ;
    end if;
```

پس از ارسال هشت بیت در این استیت باید تارگت بیت ACK را ارسال کند بنابراین این استیت را در شکل ملاحظه میکنید:

```
when slave_ack2 =>

    rw_ctrl_int <= '0';
    busy <= '1';
    if(addr_rw(0) = '0')then
        if(rw = '1')then
            state <= start_con; -- in reading mode we have to send repeated start
        else
            state <= wr;
            sda_int <= data_tx(bit_cnt);
        end if;
    else
        sda_int <= '1'; --in reading condition sda should be high impedance at first
        rw_ctrl_int <= '1';
        state <= rd ;
    end if ;
```

دلیل صفر بودن سیگنال RW_CTRL_INT مشابه استیت SLAVE_ACK1 میباشد اما یک دستور شرطی اضافه ت نسبت به ان داریم که در حالت خواندن انرا بیشتر توضیح میدهیم. در اینجا فرض بر این است که پس از فرستادن بیت ACK به استیت بعدی یعنی WR میرویم.

```
when wr =>

    rw_ctrl_int <= '0';
    busy <= '1';
    if(bit_cnt = 0)then
        rw_ctrl_int <= '1';
        bit_cnt <= 7 ;
        state <= slave_ack3 ;
    else
        bit_cnt <= bit_cnt - 1;
        sda_int <= data_tx(bit_cnt);
        state <= wr ;
    end if;
```

در این استیت هم هشت بیت دیتایی که در استیت READY در رجیستر DATA_TX ذخیره کرده بودیم را ارسال میکنیم و ACKNOWLEDGE ان هم دقیقا مانند SLAVE_ACK1 میباشد. اما با یک تفاوت :

```
when slave_ack3 =>
    rw_ctrl_int <= '0';
    if(enable = '1')then
        busy <= '1';
        addr_rw <= addres & rw ;
        command_byte_r <= command_byte ;
        data_tx <= data_wr ;
        if(addr_rw = addres & rw and command_byte_r = command_byte)then
           sda_int <= data_wr(bit_cnt) ;
           state <= wr ;
        else
            state <= start_con ;
        end if;
    else
        state <= stop_con;
    end if;
```

اگر در این استیت هنوز مایل باشیم در DAC دیتا را بنویسیم میتوانیم ان را بایک دستور شرطی پیاده سازی کرده و دوباره به استیت WR برگردیم. اما اگر بخواهیم در ادرس دیگری یا مود کاری دیگری را ارسال کنیم باید دوباره به استیت START برویم (REAEATED START) تا این دو بایت را از اول ارسال کنیم.

اگر در این استیت برویم ولی مقدار پورت ENABLE برابر با صفر باشد نشان از این است که میخواهیم به ارسال اطلاعات پایان دهیم. بنابر این پس از ان به استیت STOP_CON میرویم و در انجا سیگنال BUSY برابر با صفر میشود که حاکی از ان است که در حال حاضر ردوبدل اطلاعات صورت نمیگیرد.

حال به توضیح در خصوص بلوک هایی که بصورت CUNCURRENT در حال اجرا میباشند میپردازیم :

```
-----
read_mode <= read_mode_int ;
rw_ctrl <= rw_ctrl_int;
tx <= sda_int when rw_ctrl_int = '0' else 'Z';
-----
```

خط اول این بلوک مربوط به حالت خواندن میباشد که در قسمت مربوط به خودش بررسی میشود.

در خط دوم مقدار رجیستر RW_CTRL_INT که مقدار ان را در بلوک پروسس اصلی تغییر میدادیم بطور کانکارنت در پورت خروجی RW_CTRL ریخته میشود. هدف از استفاده از این سیگنال فرستادن ACKNOWLEDGE در تست بنچ و همچنین فرستادن بیت از تارگت به کنترلر در مود خواندن میباشد که در ادامه به ان پرداخته میشود.

خط اخر این بلوک در حقیقت مقدار رجیستر SDA_INT که در ضمن بلوک اصلی مقدارش مشخص میشود را با یک شرط درون یک رجیستر دیگر میریزیم که مقدار رجیستر RW_CTRL_INT صفرباشد. یعنی زمانی که بخواهیم اطلاعات را به سمت DAC ارسال کنیم مقدار SDA_INT را درون TX میریزیم درغیر این صورت TX های امپدانس میشود. دلیل های امپدانس شدن لاین SDA وقتی کیخواهیم بیتی را دریافت کنیم این است که تارگت ما فقط در صورت Z بود SDA میتواند مقدار لاین را تغییر دهد.

بلوک CUNCURRENT بعدی بصورت زیر است:

```
-----
with state select
    sda_enable <= data_clk_prev     when start_con,
                  not data_clk_prev  when stop_con,
                  tx    when others ;
-----
```

این بلوک در حقیقت بوجود اورنده ی حالت های START CONDITION و STOP CONDITION میباشد.

در صورتی که در این استیت ها باشیم مقدار DATA_CLK و NOT ان را در کلاک قبلی اش درون رجیستر SDA_ENABLE میریزد تا این حالات بوجود ایند. در غیر این صورت مقدار رجیستر TX را که راجبش صصحبت کردیم را داخل این رجیستر میریزد.

بلوک بعدی در حقیقت مقدار دهی باس های اصلی ارتباط I2C یعنی SDA و SCL را انجام میدهد.

```
    scl <= '0' when (scl_en = '1' and scl_clk = '0') else 'Z'
    sda <= '0' when sda_enable = '0' else 'Z';
```

کلاک SCL بدین شکل مقدار دهی میشود که اگر رجیستر SCL_EN که در ضمن بلوک اصلی به مقدار دهی ان پرداختیم اگر برابر با یک شد، و همچنین مقدار SCL_CLK که کلاک مربوط به SCL است و در بلوک اول (CLOCK GENERATOR) به مقدار دهی ان پرداختیم، صفر شد ، در این صورت مقدار باس SCL همصفر میشود و در غیر این صورت مقدارش HIGH AMPEDANCE میشود.

مقدار SDA هم کاملا با توجه به سیگنال داخلی SDA_ENABLE مقداردهی میشود. بصورتی که اگر SDA_ENABLE صفر شد مقدار SDA هم صفر میشود. اگر مقدار دیگری داشت هم مقدار SDA برابر با Z (HIGH AMPEDANCE) میشود.

چرخه خواندن اطلاعات:

خواندن اطلاعات برعکس نوشتن ان در لبه پایین رونده DATA_CLK صورت میگیرد. در دیتا شیت DAC63202 مراحل خواندن اطلاعات از رجیستر های DAC بصورت زیر عنوان شده است:

S	MSB	...	R/W (0)	ACK	MSB	...	LSB	ACK	Sr	MSB	...	R/W (1)	ACK	MSB	...	LSB	ACK	MSB	...	LSB	ACK
	Address byte Section 7.5.2.2.1				Command byte Section 7.5.2.2.2				Sr	Address byte Section 7.5.2.2.1				MSDB				LSDB			
From controller				Target	From controller			Target	From controller				Target	From target			Controller	From target			Controller

همانطور که میبینید چرخه خواندن اطلاعات تفاوت عمده ای با چرخه نوشتن ان دارد. این چرخه تا مرحله ACKNOWLEDGE کردن بایت COMMAND با مرحله خواندن فرقی ندارد اما در SLAVE_ACK2 روند شرطی ای وجود دارد که در شکل زیر میبینید:

```
when slave_ack2 =>

    rw_ctrl_int <= '0';
    busy <= '1';
    if(addr_rw(0) = '0')then
        if(rw = '1')then
            state <= start_con; -- in reading mode we have to send repeated start
        else
            state <= wr;
            sda_int <= data_tx(bit_cnt);
        end if;
    else
        sda_int <= '1'; --in reading condition sda should be high impedance at first
        rw_ctrl_int <= '1';
        state <= rd ;
    end if ;
```

دستور شرطی IF ای در این استیت وجود دارد که مقدار پورت ورودی RW را بررسی میکند واگر صفر بود به روال عادی نوشتن ادامه میدهیم و وارد استیت WR میشویم اما اگر این مقدار یک بود طبق جدول بالا باید دوباره به استیت START_CON برگردیم تا ادرس رجیستری که میخواهیم اطلاعات درونش را بخوانیم را مشخص کنیم.

```
when start_con =>

    rw_ctrl_int <= '0';
    busy <= '1';
    sda_int <= addr_rw(bit_cnt);
    if(rw = '1')then
        addr_rw <= addres & rw ;
        state <= addr_seq;
    else
        state <= addr_seq ;
    end if;
```

در این استیت شرطی وجود داشت که مقدار RW را بررسی میکرد که در حالت نوشتن ان را بررسی نکردیم.

در این شرط اگر RW برابر بایک بود باز هم به استیت ADDR_SEQ میرویم منتها یکبار مقدار کانکتنیت شده ی RW و ADRESS را در رجیستر ADDR_RW ذخیره میکنیم و بعد از ان به استیت ADDR_SEQ میرویم.

```
when addr_seq =>

    rw_ctrl_int <= '0';
    if(bit_cnt = 0)then
        bit_cnt <= 7 ;
        sda_int <= '1'; -- sda is high impedance so slave can send the acknowledge data
        if(addr_rw(0) = '0')then
            state <= slave_ack1 ;
            rw_ctrl_int <= '1';
        else
            state <= slave_ack2 ;
            rw_ctrl_int <= '1';
        end if;
    else
        bit_cnt <= bit_cnt - 1 ;
        sda_int <= addr_rw(bit_cnt - 1);
        state <= addr_seq ;
    end if;
```

در این استیت دوباره هشت بیت ادرس به سمت DAC فرستاده میشود اما ادرس فرستاده شده این بار ادرس رجیستر درون DAC میباشد. شرط IF داخلی موجود در این استیت بیت LSB رجیستر ADDR_RW را بررسی میکند که در حقیقت همان RW میباشد و اگر این مقدار برابر با یک بود این بار به جای اینکه به استیت SLAVE_ACK1 برویم، به SLAVE_ACK2 میرویم(مطابق جدول خواندن اطلاعات).

```
when slave_ack2 =>

    rw_ctrl_int <= '0';
    busy <= '1';
    if(addr_rw(0) = '0')then
        if(rw = '1')then
            state <= start_con; -- in reading mode we have to send repeated start
        else
            state <= wr;
            sda_int <= data_tx(bit_cnt);
        end if;
    else
        sda_int <= '1'; --in reading condition sda should be high impedance at first
        rw_ctrl_int <= '1';
        state <= rd ;
    end if ;
```

در این استیت شرط IF بیرونی ای وجود دارد که قبلا بررسی نشده است. این شرط بیان میدارد که اگر بیت LSB از ADRR_RW برابر با یک شد (که در استیت قبلی ان را یک کردیم) استیت بعدی ما RD خواهد بودپس مقدار RW_CTRL_INT را یک میکنیم چرا که از این به بعد میخواهیم اطلاعات را از DAC بخوانیم.

همچنین SDA_INT را هم یک میکنیم چرا که ازین پس تا نه کلاک دیگر اطلاعات را باید دریافت کنیم که این کار فقط با HIGH AMPEDANCE بودن SDA امکان پذیر خواهد بود.

استیت RD در دولبه پایین رونده وبالا رونده خلاصه میشود.

لبه بالا رونده :

```
when rd =>

    read_mode_int <= '1';
    busy <= '1';
    rw_ctrl_int <= '1'; -- only reading in this state
    if(bit_cnt = 0)then
        bit_cnt <= 7 ;
        data_rd <= data_rx;
        state <= mstr_ack ;
        rw_ctrl_int <= '0'; -- so master can send acknowledge
        if(enable = '1' and addr_rw = addres & rw)then
            sda_int <= '0'; -- meaning that acknowledge in sent to the slave and we want to send another byte to
        else
            sda_int <= '1';
        end if;
    else
        bit_cnt <= bit_cnt - 1;
        state <= rd;
    end if;
```

لبه پایین رونده:

```
when rd =>

    data_rx(bit_cnt) <= sda ;
```

در این استیت همانطور که مشخص است در هر لبه پایین رونده یک بیت موجود در لاین SDA در یکی از بیت های DATA_RX ذخیره شده (BIT_CNT در لبه بالا رونده افزایش می یابد)

در لبه بالا رونده اما دو دستور شرطی وجود دارد. دستور شرطی داخلی میگوید که اگرهمه هشت بیت از DAC ارسال شده بودند (BIT_CNT = 0) و همچنین ENABLE = 1 و ادرسی که داریم برابر با ادرس قبلی بود یعنی میخواهیم دوباره از همان رجیستر بخوانیم. بنابراین ACKNOLEDGE را این بار از سمت MASTER ارسال میکنیم که این کار با صفر کردن لاین SDA_INT محقق میشود. اما اگر ادرس تغییر کرده بود SDA_INT را برابر با یک میگذاریم تا در استیت بعدی که MASTER میخواهد ACKNOWLEDGE را بفرستند مشکلی پیش نیاید.

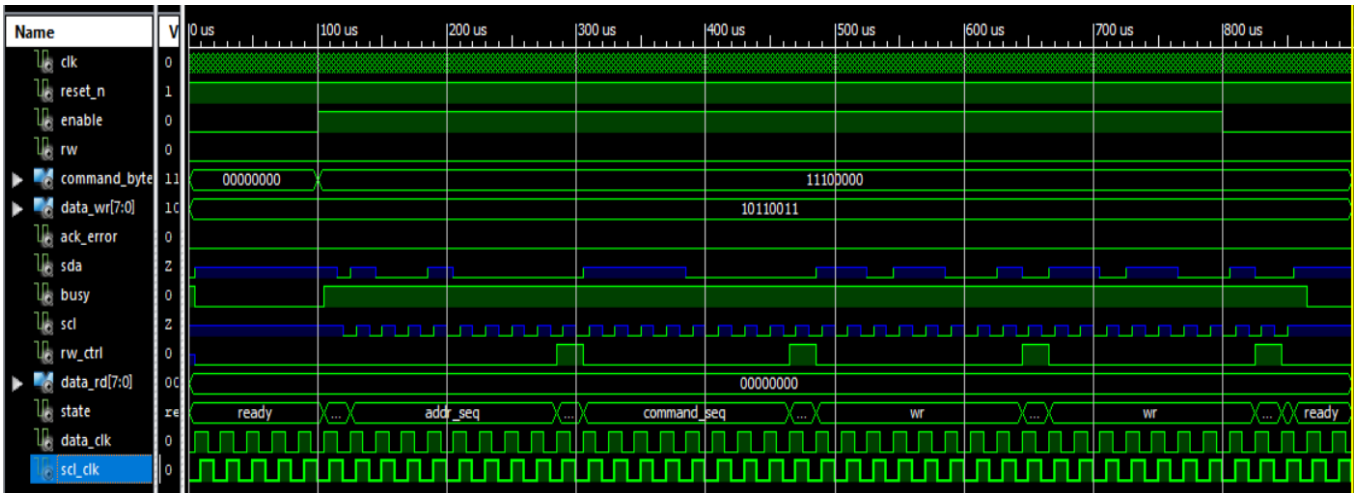
```
when mstr_ack =>

    sda_int <= '0'; -- sending acknowledge to slave
    if(enable = '1')then
        busy <= '0';
        addr_rw <= addres & rw ;
        data_tx <= data_wr ;
        if(addr_rw = addres & rw)then
            sda_int <= '1';
            state <= rd ;
        else
            state <= start_con;
            read_mode_int <= '0';
        end if;
    else
        state <= stop_con;
    end if;

when stop_con =>
    busy <= '0';
    state <= ready;
```

در این استیت مستر باید با صفر کردن SDA_INT بیت ACKNOWLEDGE را به DAC بفرستد.

اما شروطی در این استیت وجود دارد. اول این که اگر در این استیت باید ببینیم که ایا ENABLE هنوز هم یک مانده است یا خیر اگر یک بود یعنی دوباره میخواهیم عملیات خواندن یا نوشتن را انجام دهیم بنابراین کانکتنیت شده ی ADDRES و RW دوباره در ADDR_RW رجیستر میشود. دیتای پورت ورودی هم در DATA_TX رجیستر میشود. بعد از ان باید ببینم ایا هنوز میخواهیم بخوانیم یا اندفعه میخواهیم بنویسیم و اینکه اگر دوباره میخواهیم بنویسیم ایا دوباره از همان رجیستر میخواهیم بخوانیم یا خیر. اگر اادرس یکی بود که دوباره برمیگردیم به استیت RD و هشت بیت دیتای دیگر دریافت میکنیم. اما اگر ادرس تغییر کرده بود باید

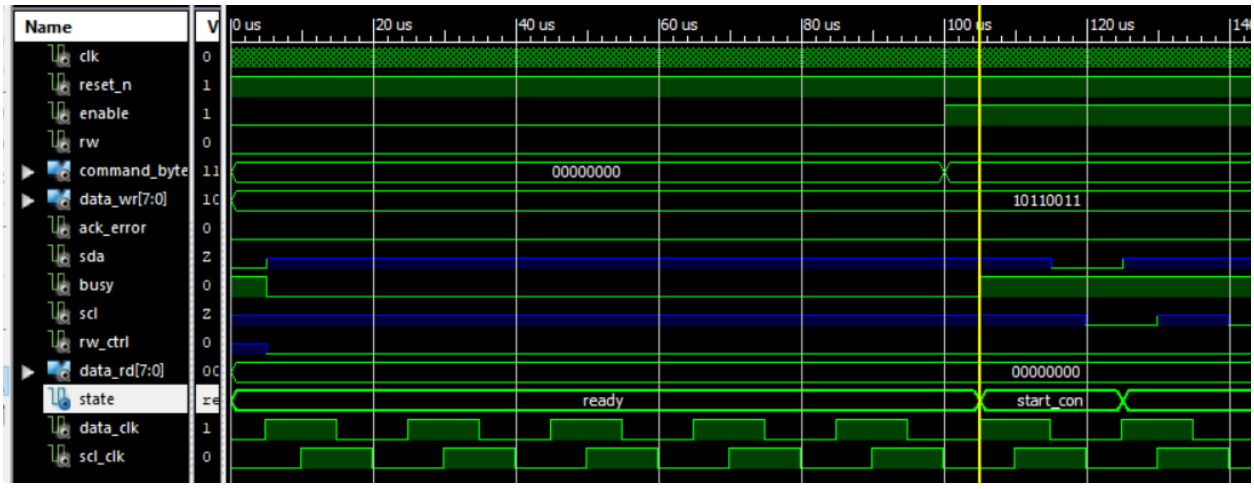
دوباره به استیت START_CON برگردیم (REPEATED START) و از اول مراحل را طی میکنیم(ممکن است در اینجا RW صفر بوده و بخواهیم بخوانیم).

شبیه سازی پروتکل I2C برای نوشتن اطلاعات:



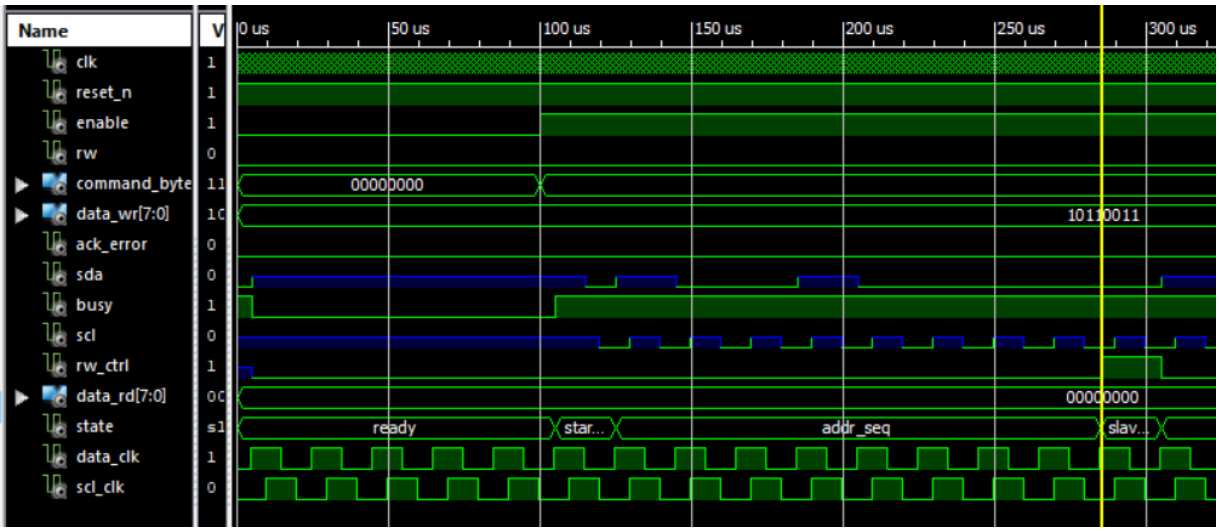
اگر به شبیه سازی دقت کنیم متوجه میشویم که در زمان 100 us پورت ENABLE ورودی یک شده است.

با زوم کردن روی این قسمت به نتایج زیر میرسیم:



همانطور که انتظار داشتیم پس از فعال شدن ENABLE، در اولین لبه ی بالا رونده ی کلاک دیتا(با خط زرد رنگ مشخص شده است) از استیت READY به استیت START_CON رفته ایم و همچنین پورت خروجی BUSY هم بلافاصله یک شده(در ماشین حالت هم در بخش READY پس از یک شدنENABLE بلافاصله BUSY را یک میکردیم).

در استیت START_CON لاین SDA با وجود HIGH AMPEDANCE بودن SCL از Z به صفر تغییر پیدا کرده که همانطور که در بخش CUNCURRENT کد اچ دی ال و همچنین دیتاشیت DAC توضیح داده شد این اتفاق مشخصه اصلی استیت START_CON میباشد. همچنین مشاهده میکنیم که پس از بوقوع پیوستن این استیت SCL شروع میکند به کلاک زدن با فرکانس و فازی دقیقا برابر با SCL_CLK میکند.



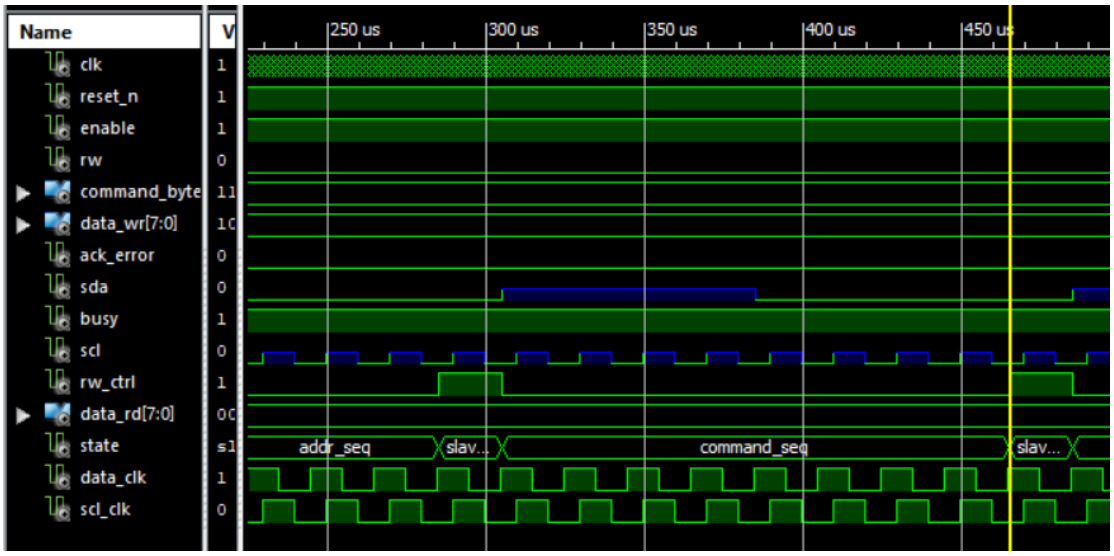
همانطور که در شکل بالا مشاهده میکنید پس از استیت START_CON بلافاصله وارد استیت ADDR_SEQ میشویم. در این استیت مقدار ادرس DAC (۱۰۰۱۰۰۰) و همچنین بیت RW LSB که درانتها به ان کانکتنیت میشود (در اینجا صفر) در باس SDA فرستاده میشود که از شکل هم میتوان متوجه این قسمت شد.

پس از اینکه ادرس بطور کامل فرستاده شده وارد SLAVE_ACK1 (خط زرد رنگ شده ایم) همانطور که در شکل میبینید در این استیت سیگنال RW_CTRL برابر بایک شده و پس از این استیت دوباره به مقدار قبلی خود برگردانده شده. علت این امر همانطور که گفته شده بود این است که ما بتوانیم با استفاده از تست بچ مقدار ACKNOWLEDGE BIT را از سمت DAC به SDA بریزیم. کد VHDL برای تست بنچ این بخش بصورت زیر است.

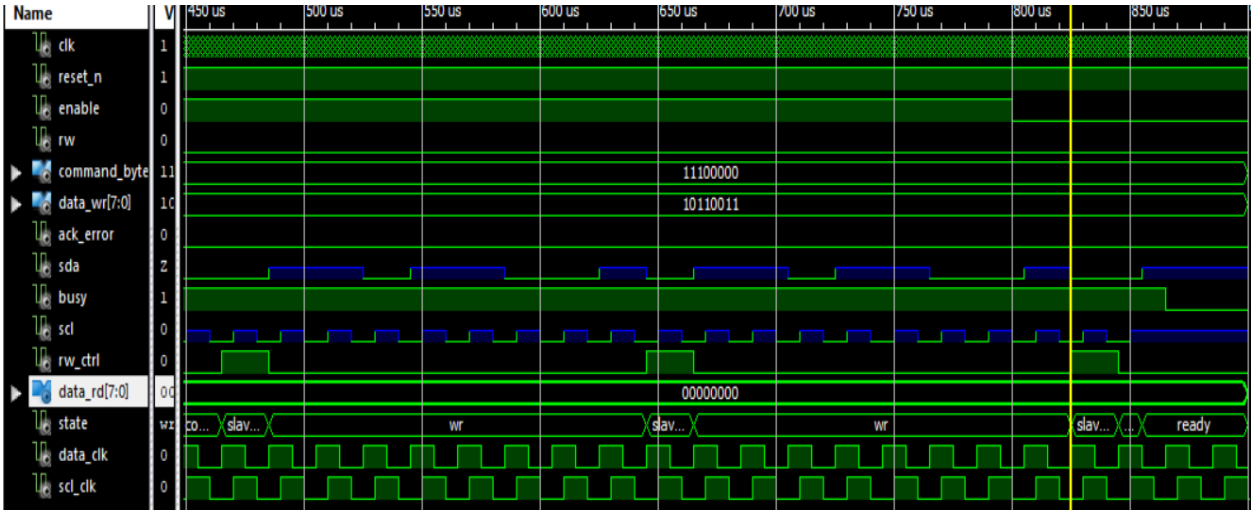
```
sda <= 'Z' when rw_ctrl = '0' else '0';
```

مفهوم این قسمت این است که در صورتی RW_CTRL یک بود از طرف تارگت SDA را صفر میکنیم(ACKNOWLEDGE) اما اگر این طور نبود SDA را ازاد (Z) میگذاریم تا کنترلر مقدار انرا تغییر دهد(دیتا را ارسال کند). مشاهده میکنید که مقدار ACK_ERROR هم صفر باقی مانده که این به معنی درست ارسال شدن ACK از سمت DAC میباشد.

دو استیت بعدی یعنی COMMAND_SEQ و SLAVE_ACK2 دقیقا عملکردی مشابه با دو استیت قبلی دارند پس به انها نمیپردازیم. در شکل زیر این دو استیت را مشاهده میکنیم:



در بقیه ی استیت های موجود همانطور که در قسمت های دیتا شیت و کد VHDL گفته شد باید وارد مرحله ی ارسال اطلاعات (WR) به DAC و در نهایت هم با صفر شدن ENABLE به STOP_CON برویم.



استیت های SLAVE_ACK3 که در شکل مشاهده میکنید پس از ارسال هر هشت بیت دیتا به DAC به وقوع میپیوندند و خط SDA را صفر میکنند. دیتای ارسال شده در تست بنچ بصورت زیر است:

```
data_wr <= "11011001"  ;
```

در پایان این شبیه سازی هم مشاهده میکنیم که پس از صفر شدن ENABLE کماکان پروسه انتقال متوقف نمیشود و باید حتما بیت ACK توسط DAC ارسال شود. پس از اینکه ACK ارسال شد وارد استیت

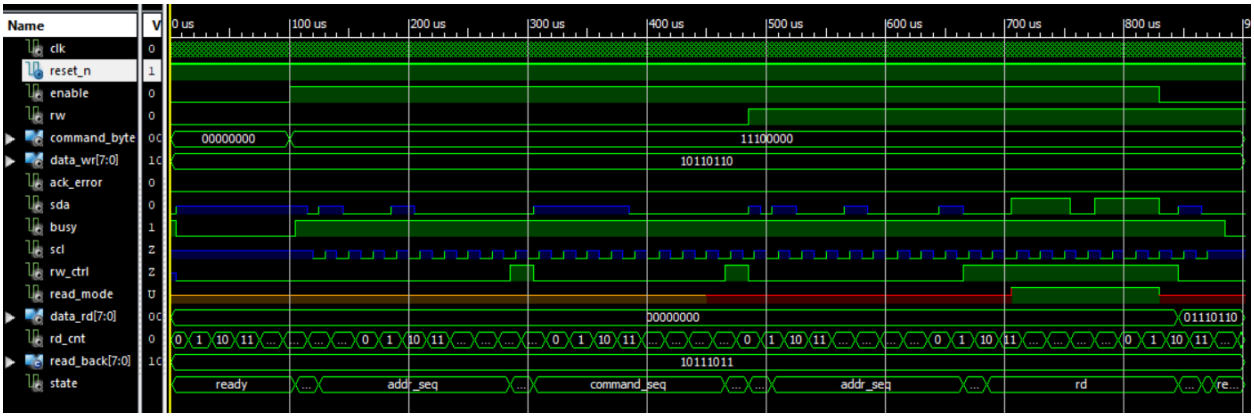
STOP_CON میشویم. در این استیت همانطور که گفته شد استیت به حالت READY رفته و پس از ان سیگنال BUSY صفر میشود که با توجه به شبیه سازی این موضوع هم قابل در است.

کد اچ دی ال بخش تست بنچ هم بشکل زیر است:

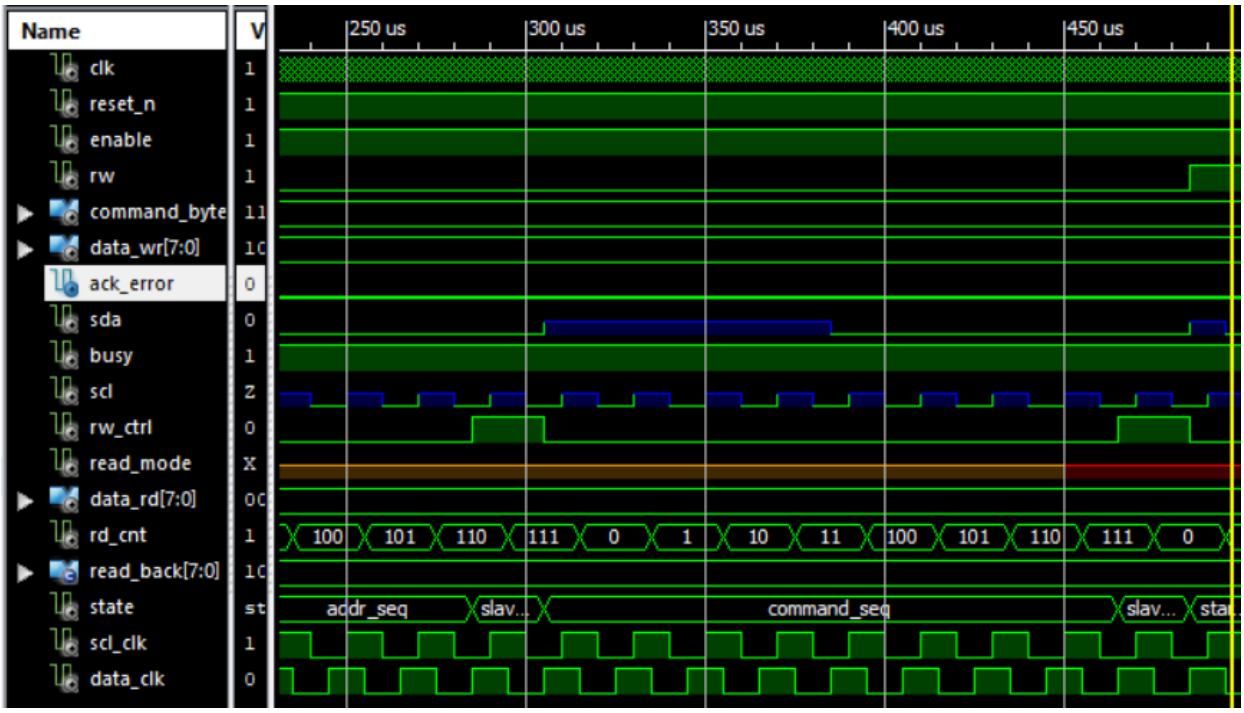
```
34
35  ENTITY i2c_test IS
36  END i2c_test;
37
38  ARCHITECTURE behavior OF i2c_test IS
39
40      -- Component Declaration for the Unit Under Test (UUT)
41
42      COMPONENT I2C_Protocol
43      PORT(
44           clk : IN  std_logic;
45           reset_n : IN  std_logic;
46           enable : IN  std_logic;
47           busy : OUT  std_logic;
48           ack_error : INOUT  std_logic;
49           sda : INOUT  std_logic;
50           scl : OUT  std_logic;
51           rw : IN  std_logic;
52           rw_ctrl : OUT  std_logic;
53           command_byte : IN  std_logic_vector(7 downto 0);
54           data_wr : IN  std_logic_vector(7 downto 0);
55           data_rd : OUT  std_logic_vector(7 downto 0)
56          );
57      END COMPONENT;
58
59
60     --Inputs
61     signal clk : std_logic := '0';
62     signal reset_n : std_logic := '0';
63     signal enable : std_logic := '0';
64     signal rw : std_logic := '0';
65     signal command_byte : std_logic_vector(7 downto 0) := (others => '0');
66     signal data_wr : std_logic_vector(7 downto 0) := (others => '0');
67
68      --BiDirs
69     signal ack_error : std_logic;
70     signal sda : std_logic;
71
72      --Outputs
73     signal busy : std_logic;
74     signal scl : std_logic;
75     signal rw_ctrl : std_logic;
76     signal data_rd : std_logic_vector(7 downto 0);
```

```
80
81  BEGIN
82
83       -- Instantiate the Unit Under Test (UUT)
84     uut: I2C_Protocol PORT MAP (
85             clk => clk,
86             reset_n => reset_n,
87             enable => enable,
88             busy => busy,
89             ack_error => ack_error,
90             sda => sda,
91             scl => scl,
92             rw => rw,
93             rw_ctrl => rw_ctrl,
94             command_byte => command_byte,
95             data_wr => data_wr,
96             data_rd => data_rd
97           );
98          clk <= not clk after 20 ns ;
99          enable <= '1' after 100 us , '0' after 800 us;
00          data_wr <= "11011001"  ;
01          command_byte <= "11100000" after 100 us;
02          sda <= 'Z' when rw_ctrl = '0' else '0';
03          reset_n <= '1';
04
05  END;
```

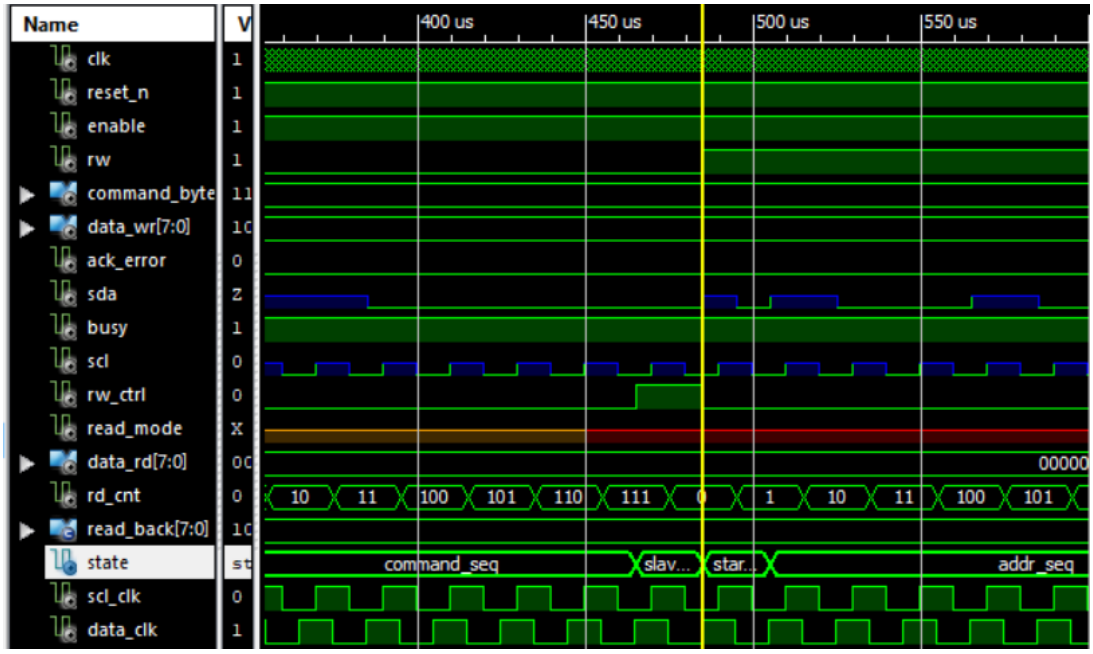
شبیه سازی پروتکل I2C برای خواندن اطلاعات :



در بخش خواندن اطلاعات مراحل تا قبل از SLAVE_ACK2 با حالت نوشتن برابر است:

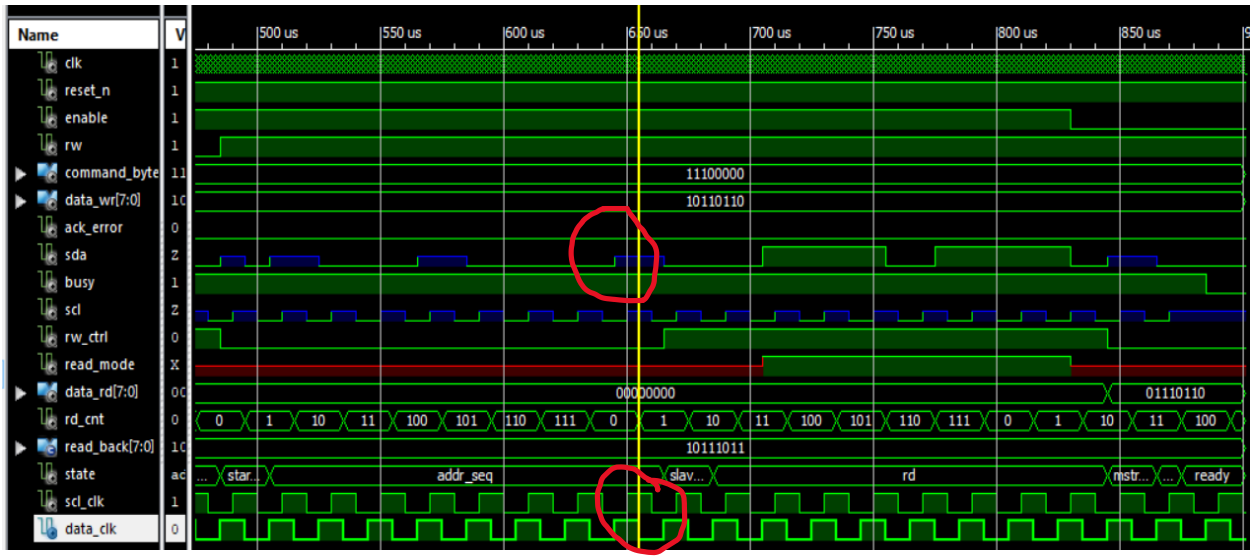


بعد از این که هشت بیت ادرس و هشت بیت کامند در مرحله اول ارسال شد و ACK این دو بایت هم توسط DAC فرستاده شد همانطور که در کد VHDL بخش خواندن دیدیم، در SLAVE_ACK2 باتوجه به مقدار بیت RW وارد مرحله بعدی میشویم. همانطور که دیدیم در حالت خواندن این بیت در استیت SLAVE_ACK2 برابر بایک خواهد بود که در شکل زیر نیز مشاهده میشود(خط زرد رنگ):



پس از اینکه متوجه شدیم RW برابر بایک است بلافاصله دوباره به استیت START_CON میرویم که به اصطلاح به ان repeated start condition میگویند.

پس از این استیت نوبت به ارسال ادرس رجیستر مورد نظر در DAC میرسد. بیت LSB در این ادرس هم باید برابر با یک باشد چون میخواهیم یک دستور خواندن ارسال کنیم:



همانطور که در شکل هم مشخص شده در اخرین لبه پایین رونده از DATA_CLK که بیت RW هم در این کلاک مشخص میشود، لاین SDA برابر با Z شده که تایید کننده این موضوع است که بیت RW برابر بایک

میباشد. همچنین پس از اینکه استیت ADDR_SEQ در این مرحله به اتمام رسید مشاهده میکنیم که سیگنال RW_CTRL تا پایان استیت RD برابر باید شده که گویای این است که پس از این در استیت های SLAVE_ACK2 و RD فقط میخواهیم از DAC بخوانیم که اگر به بخش خواندن کد VHDL هم برویم در کد نویسیس هم این موضوع رعایت شده.

سیگنال READ_MODE خروجی ای بود که در ماژول اصلی ان را تعریف کردیم مشاهده میشود که در مدت زمان ارسال اطلاعات از DAC این سیگنال دیگر نامعلوم (U) نیست و مقدارش یک است. یک بودن این مقدار به ما کمک میکند تا در تست بنچ به راحتی بتوانیم مقادیری را از طریق SDA به سمت FPGA ارسال کنیم.

کد VHDL تست بنچ در بخش خواندن اطلاعات بصورت زیر است:

```
clk <= not clk after 20 ns;
rw <= '1' after 485 us  ;
read_mode <= '1' after 450 us, '0' after 830 us ;
enable <= '1' after 100 us , '0' after 830 us;
data_wr <= "10110110" ;
reset_n <= '1';
command_byte <= "11100000" after 100 us;
sda <= 'Z'                when rw_ctrl = '0' else
       read_back(rd_cnt) when ( read_mode = '1') else '0' ;
```

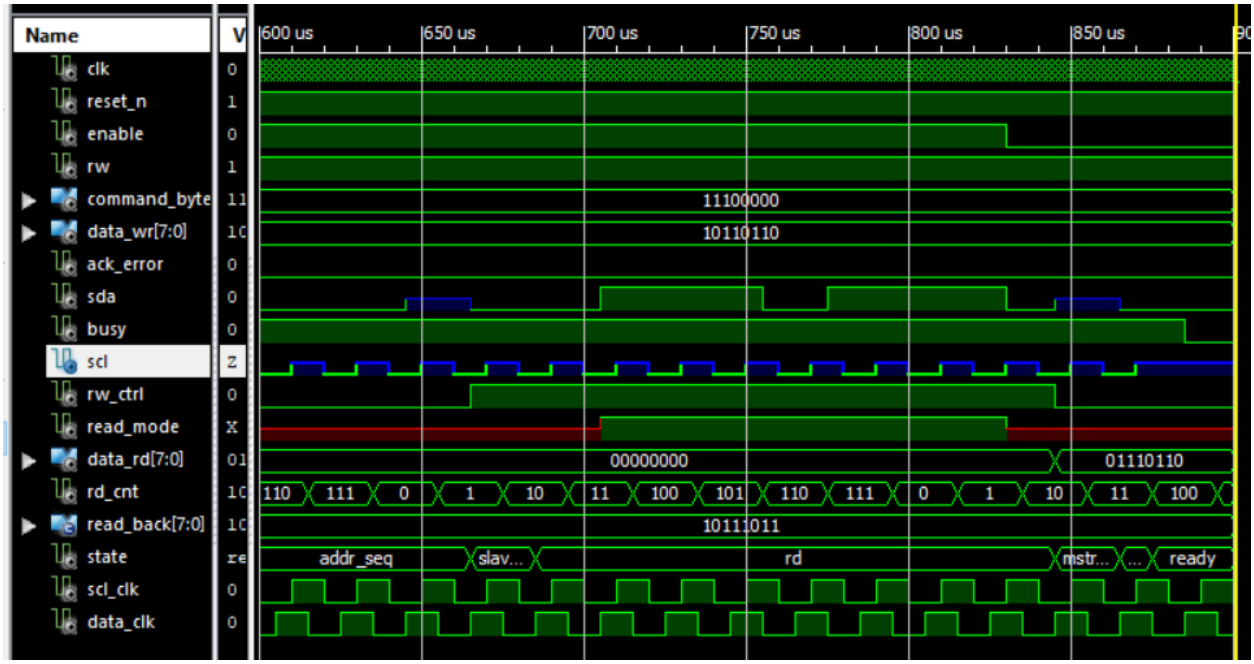
بجز مقدار دهی به SDA بقیه سیگنال ها با توجه به شبیه سازی کاملا واضح هستند.

برای مقدار دهی به SDA از سمت DAC از دستور شرطی WHEN-ELSE استفاده کردیم. به شکلی که اگر RW_CTRL صفر بود DAC لاین SDA را ازاد میگذارد تا FPGA مقدار ان را تعیین کند. اما اگر این مقدار صفر نبود و میخواستیم دیتا را بصورت بالعکس انتقال دهیمبه شرط یک بودن READ_MODE (یعنی اگر در استیت RD باشیم) مقدار سیگنال READ_BACK که بصورت CONSTANT در خود تست بنچ انرا مقدار دهی کردیم را با یک کانتر RD_CNT در لبه های پایین رونده ی کلاک SCL در برروی لاین SDA قرار میدهیم. این شگنال بصورت زیر است که در شبیه سازی هم دقیقا به همین شکل در لاینSDA قرار گرفته است.

```
constant read_back : std_logic_vector(7 downto 0) := "01110111";
```

اگر هم READ_MODE برابر با یک نبود یعنی میخواهیم ACK را ارسال کنیم پس SDA را صفر میکنیم.

در پایان شبیه سازی این بخش به استیت MSTR_ACK میرسیم و پس از ان به موجب صفر بودن ENABLE ارسال اطلاعات متوقف شده و به استیت READY برمیگردیم. مطابق شکل زیر :



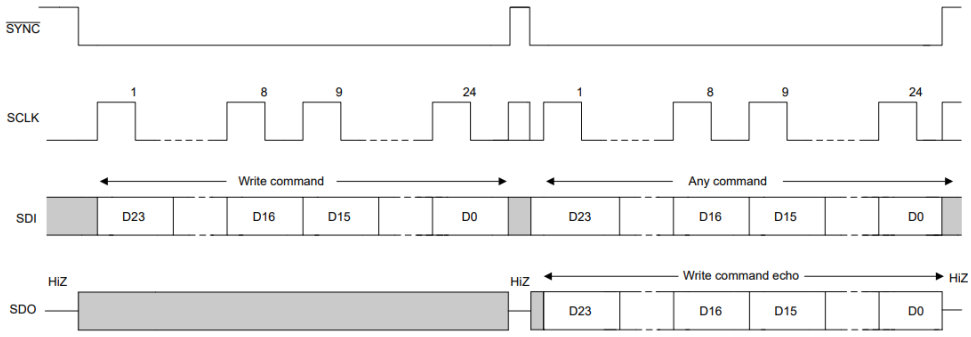
(پروتکل SPI برای DAC63202)

مشخصات دیتا شیت برای پروگرام کردن ارتباط SPI :

ارتباط SPI را میتوان هم از طریق پروتکل سه سیمه ی SPI (فقط برای نوشتن اطلاعات) و هم از طریق پروتکل چهار سیمه ی SPI (برای وقتی که میخواهیم اطلاعات رجیستر های داخلی DAC را بخوانیم) بکار برد.

حالت نوشتن اطلاعات :

چرخه زمانی نوشتن اطلاعات در DAC بصورت زیر میباشد:



از شکل قابل استنباط است که اطلاعات بصورت ۲۴ بیتی در باس های SDI و SDO ارسال میشوند و در طول ارسال باید به اندازه ۲۴ کلاک پورت خروجی SYNC که معادل Chip select است باید صفر شود(active low). از شکل متوجه میشویم که اکوی هر دستور نوشتن که در لاین MOSI به سمت DAC میرود در چرخه ی بعدی از طریق لاین MISO به سمت کنترلر بر میگردد.

بررسی کد VHDL برای حالت نوشتن اطلاعات :

پورت های ورودی و خروجی در ارتباط SPI بشکل زیر میباشند:

```
port(
        --input signals
        clk_sys             : in    std_logic                       := '0'; -- 50 Mhz
        serial_data_in      : in    std_logic_vector(23 downto 0)   := (others => '0'); -- data to be transmited including addres and rw bit
        start_bit           : in    std_logic                       := '0'; -- start a transmission cycle (24 bit)
        Miso_line           : in    std_logic                       := 'Z'; -- SDO (optional)
        sdo_en              : in    std_logic                       := '0'; -- sdo line enabling

        --output signals
        Mosi_line           : out   std_logic                       := '0'; --SDI
        sync_n              : out   std_logic                       := '1'; -- enabling the spi communication protocol
        cs_n                : out   std_logic                       := '1';  --selects one of the target devices (one device here)
        sclk                : out   std_logic                       := '0'; --serial clk for data transmission (CLK POL = 0)
        read_back_data_out  : out   std_logic_vector(23 downto 0)   := (others => '0');
        error_sig           : out   std_logic                       := '0'
    );
```

پورت ورودی C

LK_SYS کلاک ورودی سیستم است که دیتا های ما به صورت سنکرون با این کلاک انتقال پیدا میکند.

SERIAL_DATA_IN دیتای ورودی سیستم است که ۲۴ بیت عرض دیتای ان میباشد که در جدول زیر مشخص شده است:

BIT	FIELD	DESCRIPTION
23	R/$\overline{W}$	Identifies the communication as a read or write command to the address register: R/$\overline{W}$ = 0 sets a write operation. R/$\overline{W}$ = 1 sets a read operation
22-16	A[6:0]	Register address: specifies the register to be accessed during the read or write operation
15-0	DI[15:0]	Data cycle bits: If a write command, the data cycle bits are the values to be written to the register with address A[6:0]. If a read command, the data cycle bits are *don't care* values.

START_BIT یک بیت دیتای ورودی است که با یک شدن ان انتقال اطلاعات اغاز میشود.

لاین MISO هم یکی از لاین های اصلی ارتباط SPI است که دیتا هایی که از رجیستر DAC خوانده میشود در این باس قرار میگیرد.

SDO_ENیک پورت ورودی است که در حقیقت به یکی از پین های DAC متصل است و در صورتی که بخواهیم اطلاعات رجیستر های DAC را بخوانیم باید این پورت فعال باشد.

لاین MOSI هم یکی از باس های اصلی ارتباط SPI است که دیتا هایی که میخواهیم به سمت تارگت بفرستیم از طریق ایت لاین جابجا میشود.

SCLK باس کلاک خروجی در ارتباط SPI است که 180 درجه نسبت به CLK_SYS اختلاف درجه دارد. دلیل این اخلاف ایت است که در دیتاشیت DAC63202 عناون شده بود که اطلاعات در لبه پایین رونده ی کلاک SCLK فرستاده میشود بنابراین با این اختلا ف خیلی راحت در بدنه اصلی کدمان میتوانی از لبه بالا رونده کلاک سیستم استفاده کنیم.

پورت DATA_READBACK_OUT خروجی کنترلر است که دیتایی که از DAC خوانده میشود را پس از ۴۸ کلاک در خود نشان میدهد.

ERROR_SIG یک پورت خروجی است که در صورتی که در ماشین حالت طراحی شده در پرئتکل SPI اتفاق غیر منتظره ای رخ دهد این سیگنال یک میشود.

سیگنال های داخلی که اکثرا رجیستر های مرتبط با پورت های ورودی خروجی خستند بشکل زیر میباشند:

```
--internal signals:
signal start_bit_INT      : std_logic           := '0';
signal sync_n_INT         : std_logic           := '1';
signal serial_data_in_INT : std_logic_vector(23 downto 0)    := (others => '0');
signal Mosi_line_INT      : std_logic           := '0';
signal cs_n_INT           : std_logic           := '1';
signal sclk_start         : std_logic           := '0';
signal R_W                : std_logic           := '0';
signal read_back_data     : std_logic_vector(23 downto 0)    := (others => 'Z');
signal read_back_data_r   : std_logic_vector(23 downto 0)    := (others => 'Z');
--controll signal:
signal bit_cnt            : unsigned(4 downto 0):= "10111";

--states:
type states is (idle , delay_inst , inst , write_st , read_command , read_data , error_st , delay_cs);
signal state              : states              := idle ;
```

ماشین حالت طراحی شده دارای استیت های مختلفیست که در بخش بلوک اصلی به انها میپردازیم.

بلوک کانکارنت بصورت زیر میباشد:

```
cs_n <= cs_n_INT ;
sync_n <= sync_n_INT;
Mosi_line <= Mosi_line_INT ;
read_back_data_out <= read_back_data_r(15 downto 0) ;
---
```

در این بلوک سیگنال های چیپ سلکت که در ضمن بلوک پروسس اصلی از انها استفاده میکنیم بصورت کانکارنت در پورت خروجی مربوط به خودشان ریخته میشوند.

سیگنال MOSI_LINE_INT هم که در ضمن بلوک پروسس اصلی مقادیر دیتا هایی که میخواهیم بفرستیم را درونش قرار میدهیم بصورت کانکارنت در پورت MOSI_LINE قرار میگیرد.

در نهایت با استفاده از یک دستور WITH_SELECT سیگنال SCLK را با شرط فعال بودن سیگنال SCLK_START که در ضمن بلوک پروسس اصلی مقدارش را مشخص میکنیم با یک تاخیر ۲۰ نانو ثانیه(اختلاف فاز ۱۸۰ درجه) بوجود می اوریم.

حال به بررسی ماشین حالت میپردازیم که حساس به لبه بالا رونده کلاک سیستم (لبه پایین رونده ی SCLK) میباشد.

حالت اول IDLE نام دارد که در ان هنوز شروع به انتقال اطلاعات نکرده ایم :

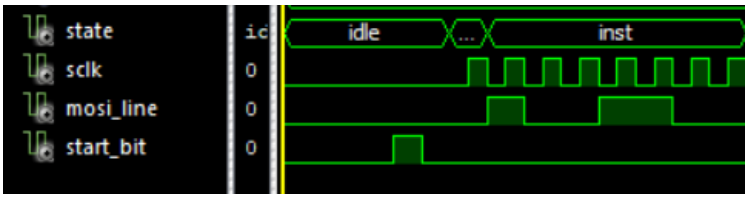
```
when idle =>
     start_bit_INT <= start_bit ;
     Mosi_line_INT <= '0';
     bit_cnt  <= "10111";
     sync_n_INT <= '1';
     cs_n_INT <= '1';
     sclk_start <= '0';
     R_W <= '0';

    if(start_bit_INT = '1')then
        serial_data_in_INT <= serial_data_in;
        sclk_start <= '1';
        state <= delay_inst ;
        cs_n_INT <= '0';
        sync_n_INT <= '0';
    else
        state <= idle ;
        cs_n_INT <= '1';
        sync_n_INT <= '1';
    end if;
```

در این استیت با هر لبه کلاک مقدار STATR_BIT را رجیستر میکنیم و با یک شدن این مقدار رجیستر شده وارد استیت بعدی میشویم. همچینین با یک شدن ان یکبار SERIAL_DATA_IN را رجیستر میکنیم و سیگنال SCLK_START را هم یک میکنیم ات از اینجا به بعد SCLK به تارگت ارسال شود. مقادیر چیپ سلکت را هم فعال میکنیم.

در غیر این صورت در استیت IDLEمیمانیم و چیپ سلکت ها غیر فعال میمانند.

```
when delay_inst =>

    state <= inst ;
    cs_n_INT <= '0';
    sync_n_INT <= '0';
    Mosi_line_INT <= serial_data_in_INT(to_integer(bit_cnt)); -- MSB bit of serial data input is passed to mosi line (R/W)
    bit_cnt <= bit_cnt - 1;
    R_W <= serial_data_in_INT(to_integer(bit_cnt)) ; --register the R/W bit
```

بعد از ان به استیت DELAY_INST میرویم. این حالت در حقیقت تاخیری در فرستادن دیتا برای ما ایجاد میکند که در تست بچ میتوانیم ان را به خوبی مشاهده کنیم.



همانطور که در تست بنچ مشاهده میشود در صورتی که این استیت نباشد بیت MSB از SERIAL_DATA_IN_INT ازبین میرود و از بیت بعدی دیتا ها ارسال میشوند.

پس از این استیت به INST میرویم :

```
when inst =>

    cs_n_INT <= '0';
    sync_n_INT <= '0';
    Mosi_line_INT <= serial_data_in_INT(to_integer(bit_cnt));
    if(bit_cnt /= "10000")then

        state <= inst ;
        bit_cnt <= bit_cnt - 1 ;
    else
        if(R_W = '0')then
            state <= write_st ;
        else
            state <= read_command ;
        end if ;
```

در این استیت در حقیقت هشت بیت (به علاوه یک بیتی که در DELAY_INST ارسال شد) ارسال میشوند که با استفاده از BIT_CNT تعداد بیت های فرستاده شده را کنترل میکنیم. در صورتی که هشت بیت ارسال شده باشد با استفاده از دستور شرطی بیت MSB را که در حقیقت در استیت قبلی فرستاده بودیم را کنترل میکنیم که در صورتی که صفر باشد وارد مرحله‌ی WRITE_ST میشویم.

```
when write_st =>

    cs_n_INT <= '0';
    sync_n_INT <= '0';
    Mosi_line_INT <= serial_data_in_INT(to_integer(bit_cnt));
    if(bit_cnt /= "00000")then
        bit_cnt <= bit_cnt -1;
        state <= write_st;
    else
        bit_cnt <= "10111";
        state <= delay_cs ;
    end if;
```

در این استیت خیلی ساده تا صفر شدن مقدار BIT_CNT به ارسال اطلاعات ادامه میدهیم و در صورت صفر شدن BIT_CNT ان را ریست کرده و به استیت DELAY_CS میرویم.

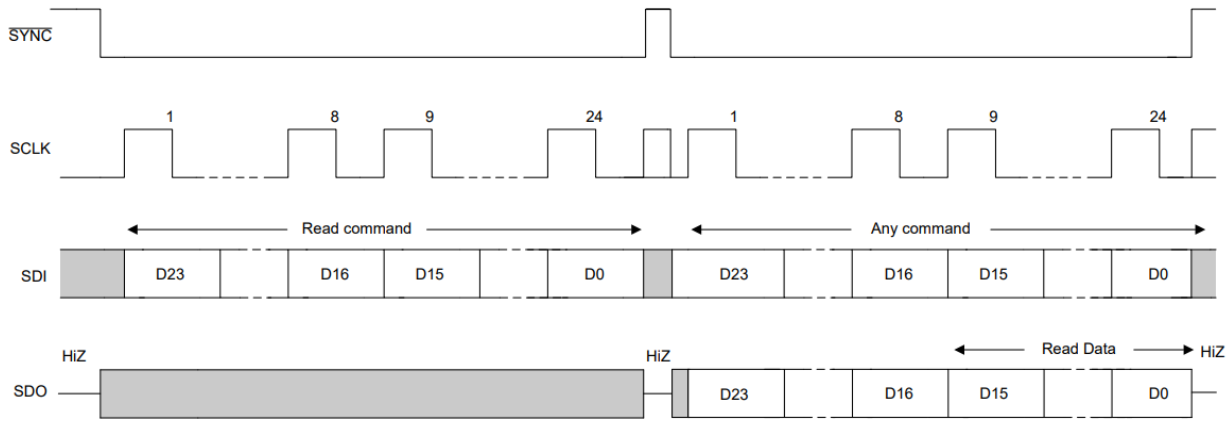
```
when delay_cs =>
    cs_n_INT <= '0';
    state <= idle ;
    Mosi_line_INT <= '0';
    bit_cnt <= "10111";
    sync_n_INT <= '0';
```

مکانیزم عملکرد این استیت هم مشابه DELAY_INST میباشد و برای جلو گیری از هدر رفتن اطلاعاتی است که با یک کلاک تاخیر وارد باس MOSI میشوند.

پس از این استیت دوباره وارد استیت IDLE میشویم و منتظر میمانیم تا پورت START_BIT دوباره یک شده و سپس به انتقال دوباره اطلاعات میپردازیم.

حالت خواندن اطلاعات :

پروسه خواندن اطلاعات در پروتکل SPI برای DAC63202 یک پروسه ی دو مرحله ای است. در شکل زیر این مراحل را مشاهده میکنید:



همانطور که مشاهده میکنید تا کلاک ۲۴ ام فرایند تقریبا مشابه حالت خواندن است. اما این ۲۴ بیت فرستاده شده با حالت خواندن فرق اساسی ای دارد. اگر 24 بیت READ COMMAND فرستاده شده به DAC را بررسی کنیم ۸ بیت اول ادرس FACTORY PRESET مربوط به DAC هست که بصورت دیفالت برابر با ۱۰۰۱۰۰ میباشد. بیت MSB که در سمت چپ ادرس وجود دارد همخواندن و نوشتن اطلاعات را مشخص میکند. ۸ بیت دوم مود کار DAC برای ارسال اطلاعات و ۸ بیت اخر هم ادرس رجیستری که میخواهیم اطلاعات درونش را بخوانیم را مشخص میکند.

در مرحله دوم ۲۴ بیت دیگر البته این بار از لاین SDO به سمت کنترلر فرستاده میشود که ۱۶ بیت LSB ان دیتایی خوانده شده از ان رجیستر میباشد.

بررسی کد VHDL برای حالت خواندن اطلاعات:

این مرحله تا بعد از استیت INST یکسان است بنابراین تا اینجای کار ۸ بیت ادرس DAC را ارسال میکنیم. در دستور شرطی IF داخل استیت INST در صورتی که مقدار R_W برابر بایک بود باید در استیت بعدی به READ_COMMAND برویم.

```
when read_command =>

    cs_n_INT <= '0';
    sync_n_INT <= '0';

    if(sdo_en = '1')then
        Mosi_line_INT <= serial_data_in_INT(to_integer(bit_cnt));
        if(bit_cnt /= "00000")then
            bit_cnt <= bit_cnt -1;
            state <= read_command ;
        else
            bit_cnt <= "10111";
            state <= read_data ;
        end if ;
    else
        state <= error_st;
    end if;
```

بدنه ی این استیت همانند استیت WRITE_ST میباشد منتها به جای اینکه دیتا را ارسال کنیم مود و ادرس رجیستر را ارسال میکنیم. اگر تمام ۱۶ بیت ارسال شد به استیت بعدی یعنی READ_DATA برمیگردیم. در صورتی که در استیت READ_COMMAND یعنی در استیت قبلی مقدار R_W برابر یک بوده اما در این استیت مقدار پورت SDO_EN که یکی از پین های DAC است برابر با یک نباشد به استیت ERROR_ST میرویم.

```
when error_st =>
    cs_n_INT <= '1';
    sync_n_INT <= '1';
    read_back_data <= (others =>'Z');
    state <= idle;
    error_sig <= '1';
```

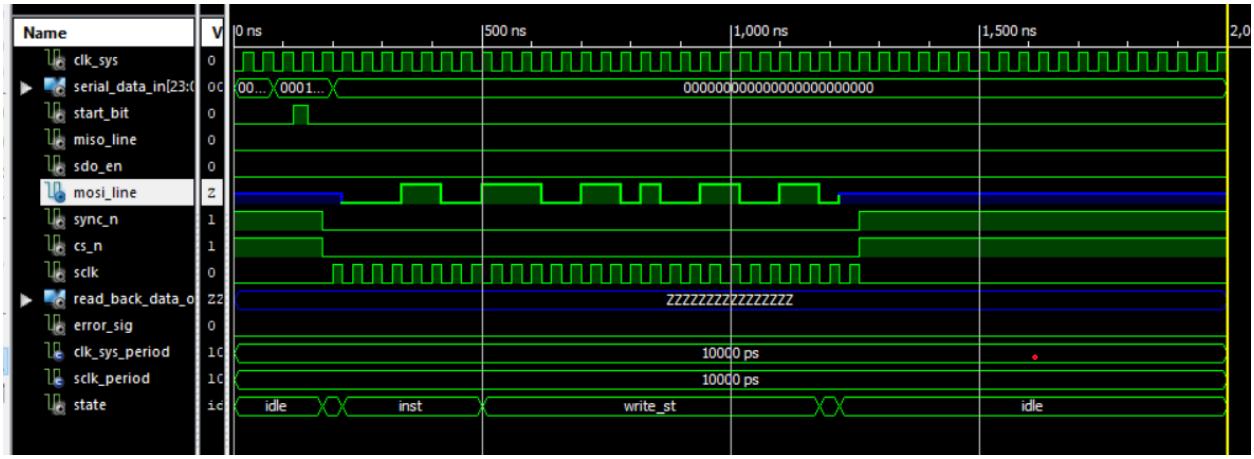
در این استیت چیپ سلکت ها غیر فعال شدهو کل بیت های DATA_READ_BACK برابر با Z میشوند.
همچنین پورت مربوط به ارور یک شده و وارد استیت IDLE میشویم.

در استیت READ_DATA شروع به خواندن اطلاعات از لاین SDO میکنیم.

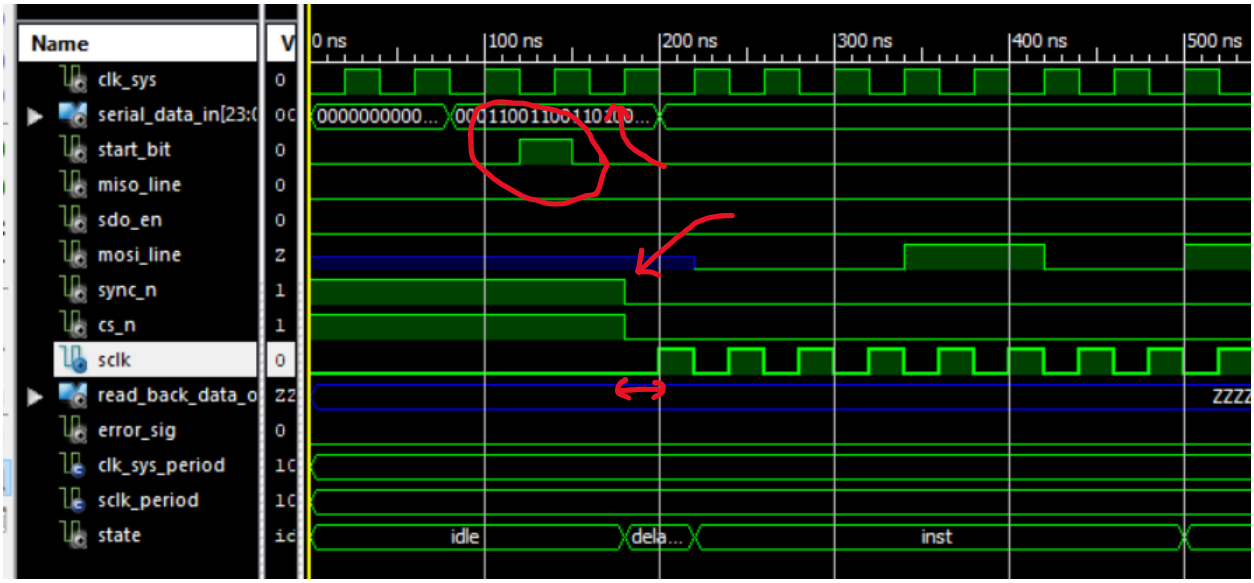
```
when read_data =>
    if(bit_cnt /= "00000")then
        read_back_data(to_integer(bit_cnt)) <= Miso_line;
        bit_cnt <= bit_cnt - 1;
        state <= read_data ;
    else
        read_back_data_r <= read_back_data ;
        read_back_data <= (others => 'Z');
        bit_cnt <= "10111";
        state <= delay_cs ;
    end if;
```

در اینجا در هر کلاک مقدار لاین MISO_LINE را خوانده و درون یکی از بیت های رجیستر READ_BACK_DATA قرار میدهیم(شماره بیت را با شمارنده BIT_CNT کنترل میکنیم). پس از اینکه ۲۴ بار از لاین MISO دیتا ها را خوانده و در رجیستر READ_BACK_DATA قرار دادیم. مقدار این رجیستر را در رجیستر دیگری با همان ابعاد قرار میدهیم و خود رجیستر READ_BACK_DATA را ریست میکنیم به HIGH AMPEDANCE. همزمان بصورت CUNCURRENT ، ۱۶ بیت LSB از رجیستر READ_BACK_DATA_R را درون پورت READ_BACK_DATA_OUT قرار میدهیم.

شبیه سازی حالت نوشتن :



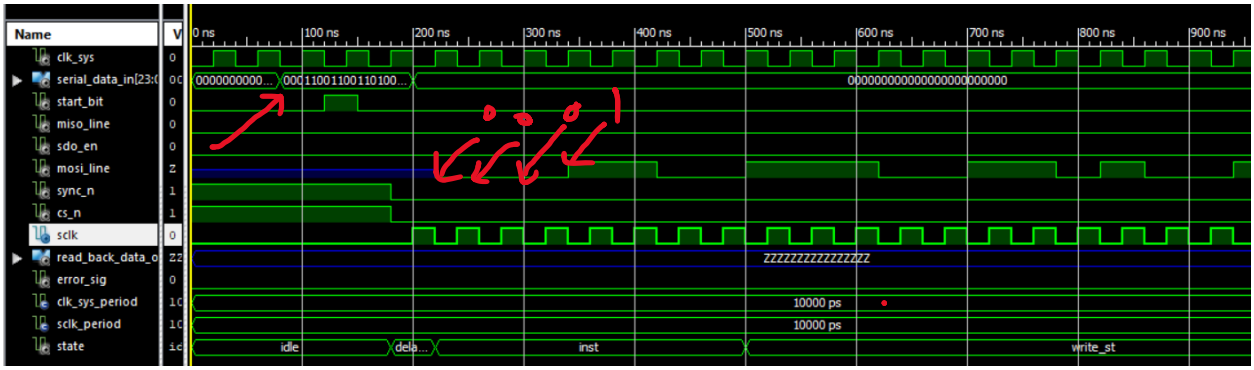
در ابتدا مشاهده میکنیم که سیگنال START_BIT به مدت کوتاهی یک شده است:



در همین لحظه چیپ سلکت های ما فعال شده (یک میشوند) مطابق شکل بالا. همچنین SCLK با تاخیر ۲۰ نانو ثانیه نسبت به CLK_SYS شروع به کلاک زدن میکند.

پس از اینکه START_BIT را دیدیم ، در لبه بالا رونده کلاک CLK_SYS وارد استیت بعدی DELAY_INST میشویم. همانطور که در بخش کد هم گفته شد در این استیت در حقیقت تاخیری در فرستادن اطلاعات در پروسه اصلی بوجود می اوریم تا اطلاعات ارسالی بدلیل تاخیرات بوجود امده در پروسس از بین نروند.

در ادامه پروسه خواندن شاهد استیت های INST که ارسال ادرس در ان صورت میگیرد و همچنین WRITE_ST که ارسال دیتا در ان صورت میگیرد و همچنین DELAY_CS که به دلیل مشابه با DELAY_INST برای جلوگیری از هدر رفتن اطلاعات قرار داده شده است را میبینیم:

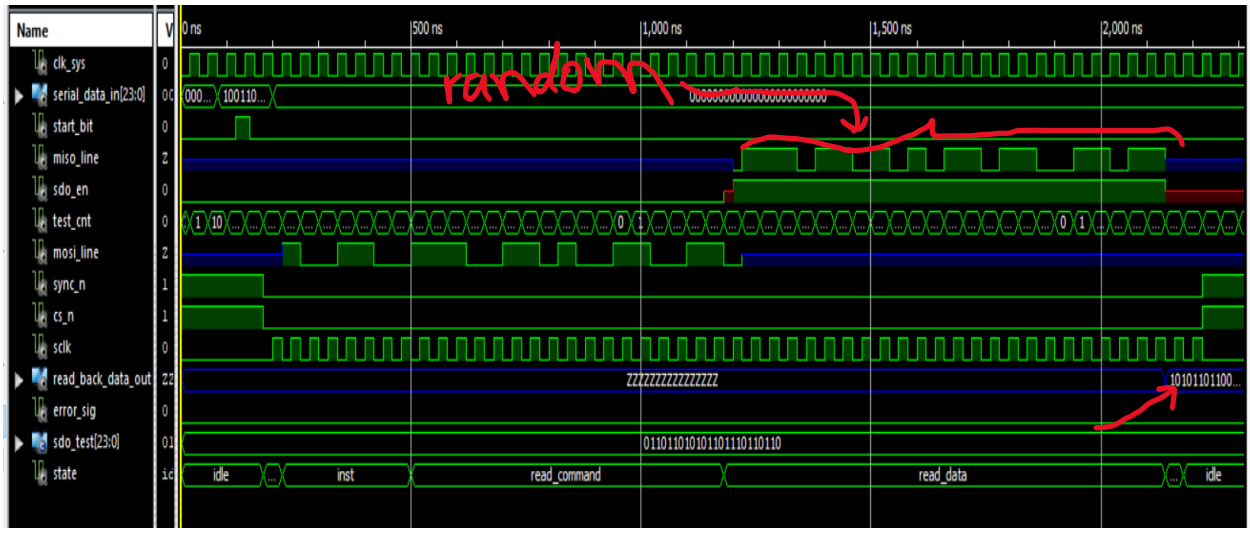


همانطور که در شکل بالا میبینید اطلاعات در پورت ورودی SERIAL_DATA_IN که در اخرین کلاک از استیت IDLE رجیستر شده اند به ترتیب پس از استیت DELAY_INST در لاین MOSI (در هر لبه پایین رونده SCLK) قرار گرفته اند.

همچنین مشاهده میشود که در یک سیکل ارسال اطلاعات در کل ۲۷ بار کلاک SCLK زده شده است.

۲۴ کلاک برای ۲۴ بیت ارسالی + ۲ کلاک برای DELAY_INST و DELAY_CS + ۱ کلاک که برای اطمینان در پایان زده میشود و تاثیری در ارسال اطلاعات ندارد.

شبیه سازی حالت خواندن :



همانطور که از شکل پیداست در یک سیکل برای دریافت دیتا از رجیستر های DAC باید ۵۱ کلاک SCLK زده شود.

پورت ورودی خروجی SDO_EN که در کدمان تعریف کردیم پس از پایان ارسال اطلاعات موردنیاز به DAC (ادرس خود DAC و ادرس رجیستر DAC) یک میشود(بخش کد اچ دی ال برای خواندن اطلاعات را نگاه کنید) و پس از ان میتوانیم در تست بنچ خودمان دیتا هایی را به لاین MISO فرستاده تا از درستی ارسال اطلاعات بطور بالعکس نیز مطلع شویم.

```
 -- Instantiate the Unit Under Test (UUT)
uut: spi_protocol PORT MAP (
       clk_sys => clk_sys,
       serial_data_in => serial_data_in,
       start_bit => start_bit,
       Miso_line => Miso_line,
       sdo_en => sdo_en,
       test_cnt => test_cnt,
       Mosi_line => Mosi_line,
       sync_n => sync_n,
       cs_n => cs_n,
       sclk => sclk,
       read_back_data_out => read_back_data_out,
       error_sig => error_sig
     );

     clk_sys <= not clk_sys after 20 ns ;
     serial_data_in <= "10011001100110100110011O" after 80 ns , (others => '0') after 200 ns ;
     start_bit <= '1' after 120 ns , '0' after 150 ns ;
     sdo_en <= '1' after 1200 ns;
     Miso_line <= sdo_test(test_cnt) when (sdo_en = '1') else 'Z';
```

در تست بنچ حالت خواندن تنها اطلاعاتی که با حالت نوشتن فرق دارند بصورت زیر هستند:

بیت MSB در SERIAL_DATA_IN برابر بایک است (در کل اچ در ال این بیت را در رجیستر R_W ذخیره کردیم و باتوجه به مقدار ان در استیت INST به استیت بعدی رفتیم) .

SDO_EN بعد از ارسال ادرس های مورد نیاز به DAC یک میشود (بعد از READ_COMMADN) تا ارسال اطلاعات در لاین MISO از DAC به سمت FPGA اغاز شود.

در خط اخر تست بنچ هم یک دیتای رندوم که در رجیستر SDO_TEST ۲۴ بیتی ذخیره کرده بودیم را با استفاده از یک کانتر و با شرط یک بودن SDO_EN درون MISO میریزیم.

حالت خواندن تا اخر استیت READ_COMMAND با حالت نوشتن یکی است با اینکه اطلاعات ارسال شده در READ_COMMAND در حقیقت ادرس رجیستر های DAC میباشند. اما بعد از این استیت متوجه میشویم که لاین MOSI بصورت HIGH AMPEDANCE در می اید و اطلاعات رندومی که ساخته بودیم در لاین MISO شروع به قرار گرفتن میکنند.

در نهایت هم اطلاعات فرستاده شده (همانطور که در شکل نشان داده شده است) درون پورت خروجی READ_BACK_DATA_OUT قرار میگیرند.