1/25/2023

VHDL Project Report

Marziye Pandi & Mostafa Moghaddas

MASTER STUDENT IN COMPUTER SYSTEMS ARCHITECTURE

STUDENTID: 400721019 & 400721055

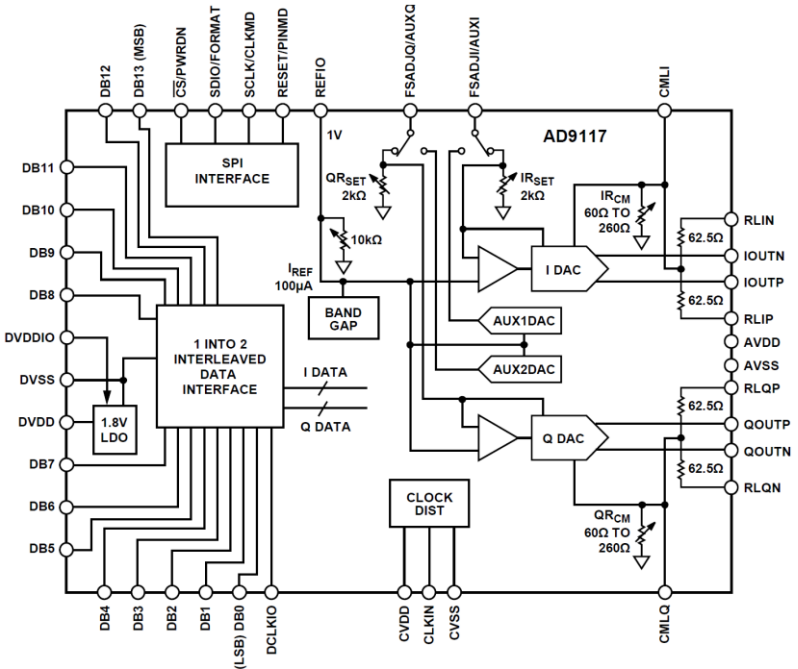
مقدمه:

هدف این پروژه طراحی یک FPGA می‌باشد که در اینجا قرار است به‌عنوان master، با DAC به‌عنوان slave، ارتباط برقرار کند و اطلاعاتی که DAC نیاز دارد را برای آن تولید کرده و در قالب دو مدل SPI و Parallel به آن ارسال و یا از آن دریافت کند. در طراحی SPI باید به این نکته دقت داشت که در هر کلاک امکان ارسال یک بیت دیتا فراهم است و اینکه برخلاف باقی SPI ها، در این مدل از MOSI/MISO استفاده نمی‌شود و روش ارتباط SDIO به‌صورت inout تعریف می‌شود.

FPGA از طریق پروتکل SPI، حداکثر یک دیتای 40 بیتی را برای master ارسال می‌کند یا از آن دریافت می‌کند. این دیتا شامل 8 بیت پرارزش دستورالعمل و 32 بیت دیتا است که این دیتاها هرکدام محتوای رجیسترهایی است که آدرس آن را در 8 بیت دستورالعمل برای تراشه‌ی مقصد فراهم کرده‌ایم. در این پروژه برای شبیه‌سازی از داده‌های دلبخواهی برای testbench استفاده‌شده است که این داده‌ها بر اساس مدل گفته‌شده در datasheet به‌دست‌آمده است.

در ادامه بعد از مقداردهی اولیه به رجیسترها با استفاده از 15 پایه‌ی FPGA، 14 بیت دیتا و یک بیت کلاک برای تراشه‌ی مقصد از طریق FPGA ارسال می‌شود به این دلیل که تراشه‌ی slave دارای رزولوشن 14 بیتی است و برای تبدیل دیجیتال به آنالوگ، نیاز به 14 بیت دیتای دیجیتال دارد که FPGA برای آن فراهم می‌کند.

حال که با روش انجام پروژه آشنا شدیم در ابتدا به طراحی و کد نویسی SPI می‌پردازیم و سپس parallel را بررسی می‌کنیم. همچنین دیاگرام عملکردی DAC را در ادامه باهم می‌بینیم.



پروتکل SPI

در AD9117 دو مرحله برای یک چرخه ارتباطی وجود دارد. فاز اول چرخه دستورالعمل است که در این مرحله یک بایت دستورالعمل در 8 لبه‌ی بالارونده‌ی اول SCLK به DAC منتقل می‌شود که این‌یک بایت دارای اطلاعاتی همچون mode خواندن یا نوشتن در بیت MSB، تعداد بایت انتقال دیتا در بیت‌های 6 و 5 و همچنین آدرس اولین رجیسترکه قرار است program بشود.

بایت دستورالعمل

همان‌طور که گفته شد، این بایت در ابتدای چرخه‌ی ارتباطی است که به دستگاه مقصد یا Slave می‌فهماند که این دیتای واردشده چه‌کاری می‌خواهد انجام دهد.

MSB LSB

DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
R/W’	N1	N0	A4	A3	A2	A1	A0

بیت 7 از بایت دستورالعمل تعیین می‌کند که آیا انتقال داده خواندن یا نوشتن پس از نوشتن بایت دستورالعمل اتفاق می‌افتد. منطق 1 یک عملیات خواندن را نشان می‌دهد. منطق 0 یک عملیات نوشتن را نشان می‌دهد.

N1 و N0 (بیت 6 و بیت 5 از بایت دستورالعمل) تعداد بایت‌هایی را که باید در چرخه انتقال داده منتقل شوند تعیین می‌کنند. اگر 00 باشد، 1 بایت منتقل می‌شود و اگر 01 باشد دو بایت، 10 سه بایت و درنهایت 11 چهار بایت داده منتقل می‌کند.

A4، A3، A2، A1 و A0 تعیین می‌کنند که در طول بخش انتقال داده چرخه ارتباطات به کدام رجیستر دسترسی پیدا می‌شود. برای انتقال چند بایتی، این آدرس بایت شروع است. آدرس‌های رجیستر زیر به‌صورت داخلی توسط AD9117 بر اساس بیت LSBFIRST تولید می‌شوند (رجیستر 0x00، بیت 6).

نقشه رجیستر SPI

در ادامه در جدول زیر، مقادیر اولیه‌ی رجیستر ها و محتوای هر بیت از آن‌ها نشان داده می‌شود.

به دلیل اینکه مقداردهی و تولید دیتا برای تمام رجیسترها دشوار بود دیتا و آدرس آن‌ها به‌صورت دستی در testbench نوشته شد و به دلیل آنکه برای انتقال دیتای Parallel با یک دیتاباس 14 بیتی به محتوای رجیستر Data Control نیاز بود، در testbench فقط مقادیری به‌عنوان ورودی وارد شد که این رجیستر را آدرس‌دهی و مقداردهی کند.

Name	Addr	Default	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
SPI Control	0x00	0x00	Reserved	LSBFIRST	Reset	LNGINS	Reserved			
Power-Down	0x01	0x40	LDOOFF	LDOSTAT	PWRDN	Q DACOFF	I DACOFF	QCLKOFF	ICLKOFF	EXTREF
Data Control	0x02	0x34	TWOS	Reserved	IFIRST	IRISING	SIMULBIT	DCI_EN	DCOSGL	DCODBL
I DAC Gain	0x03	0x00	Reserved		I DACGAIN[5:0]					
IRSET	0x04	0x00	IRSETEN	Reserved	IRSET[5:0]					
IRCML	0x05	0x00	IRCMLEN	Reserved	IRCML[5:0]					
Q DAC Gain	0x06	0x00	Reserved		Q DACGAIN[5:0]					
QRSET	0x07	0x00	QRSETEN	Reserved	QRSET[5:0]					
QRCML	0x08	0x00	QRCMLEN	Reserved	QRCML[5:0]					
AUXDAC Q	0x09	0x00	QAUXDAC[7:0]							
AUX CTLQ	0x0A	0x00	QAUXEN	QAUXRNG[1:0]		QAUXOFS[2:0]			QAUXDAC[9:8]	
AUXDAC I	0x0B	0x00	IAUXDAC[7:0]							
AUX CTLI	0x0C	0x00	IAUXEN	IAUXRNG[1:0]		IAUXOFS[2:0]			IAUXDAC[9:8]	
Reference Resistor	0x0D	0x00	Reserved		RREF[5:0]					
Cal Control	0x0E	0x00	PRELDQ	PRELDI	CALSELQ	CALSELI	CALCLK	DIVSEL[2:0]		
Cal Memory	0x0F	0x00	CALSTATQ	CALSTATI	Reserved		CALMEMQ[1:0]		CALMEMI[1:0]	
Memory Address	0x10	0x00	Reserved		MEMADDR[5:0]					
Memory Data	0x11	0x34	Reserved		MEMDATA[5:0]					
Memory R/W	0x12	0x00	CALRSTQ	CALRSTI		CALEN	SMEMWR	SMEMRD	UNCALQ	UNCALI
CLKMODE	0x14	0x00	CLKMODEQ[1:0]			Searching	Reacquire	CLKMODEN	CLKMODEI[1:0]	
Version	0x1F	0x0A	Version[7:0]							

توضیحات کد نویسی SPI SDIO

پروتکل SPI در FPGA به یک کلاک برای کار کردن نیاز دارد که این کلاک را از سیستم می‌گیرد که به آن CLK_SYS می‌گوییم و آن را به‌صورت ورودی برای SPI تعریف می‌کنیم همچنین برای ارتباط با slave نیاز به یک کلاک است تا بتواند با تراشه مقصد ارتباط برقرار کند که به همین منظور کلاکی به اسم SCLK تعریف کردیم که باید به‌صورت خروجی باشد که این کلاک را یک oscillator برای SPI فراهم می‌کند اما در این پروژه به این دلیل که بنابراین بود که SCLK را در testbench مقدار دهیم به همین دلیل آن را ورودی تعریف کردیم.

در ادامه یک سیگنال 40 بیتی تعریف می‌کنیم که همان دیتایی است که FPGA برای تراشه مقصد می‌فرستد یا از آن دریافت می‌کند که در ادامه بیشتر به این دیتا می‌پردازیم. سپس یک سیگنال START تعریف می‌کنیم تا ارتباط را برقرار کنیم.

در ادامه می‌بایست یک سیگنال کنترلی برای SDIO داشته باشیم که این سیگنال مشخص می‌کند چه زمان SDIO در حالت ورودی باشد و یا اینکه چه زمانی این پورت به‌صورت پورت خروجی عمل کند و از این سیگنال در testbench استفاده می‌کنیم. همچنین یک سیگنال CS_n تعریف می‌شود که همان chip select است و

هر زمان که صفر شد نشانگر آن است که FPGA با آن تراشه مشغول ارتباط است و با همین سیگنال CS_n می‌توان چندین تراشه slave را با FPGA مرتبط کرد.

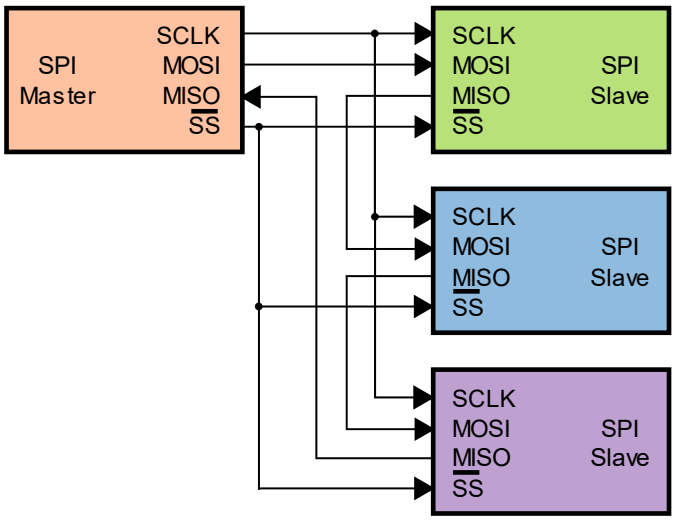


Figure 1 - 1 Master N Slave

درنهایت ENTITY قسمت SPI به شرح زیر تعریف می‌شود که هم در اینجا و هم در کل پروژه تلاش شده با کامنت نویسی‌های متناسب کدها خواناتر شوند و درک آن‌ها راحت‌تر گردد.

```
entity SPI_SDIO is
    Port(
        --inputs
        CLK_SYS         : in    STD_LOGIC;
        SCLK            : in    STD_LOGIC;
        Data_In         : in    STD_LOGIC_VECTOR (39 downto 0);
        Start           : in    STD_LOGIC;
        --outputs
        CS_n            : out   STD_LOGIC;
        RW_CTL          : out   STD_LOGIC;
        --inout
        SDIO            : inout STD_LOGIC
    );
    --
end SPI_SDIO;
```

حال که ENTITY تعریف شد به سراغ نوشتن یک توصیف رفتاری برای این ENTITY می‌رویم و قبل از آن این نکته حائز اهمیت است که بهتر است یک سیگنال میانی برای پورت‌های ورودی و خروجی (به‌جز کلاک‌ها و ورودی خروجی (inout)) تعریف کنیم که با این کار به دو قابلیت دست پیدا می‌کنیم.

1. عمل routing برای پیاده‌سازی بهتر صورت می‌گیرد.
2. باعث می‌شود این سیگنال‌های میانی با کلاک سنکرون شوند که کار طراحی و شبیه‌سازی را ساده‌تر می‌کند.

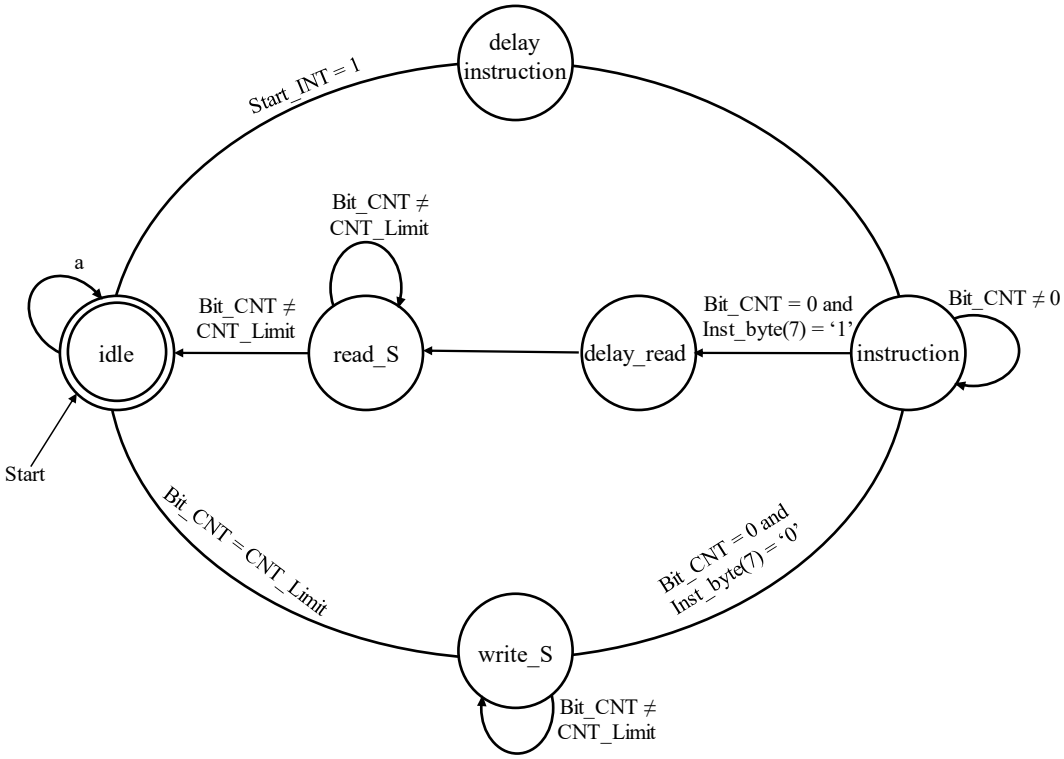
سپس به دلیل نیازمندی‌های داخل طراحی یک سری سیگنال دیگر به‌مانند RX_Data تعریف می‌شود و زمانی از آن استفاده می‌شود که SPI در حالت read قرارگرفته باشد و از slave به‌صورت بیت به بیت دیتا دریافت کند که این دیتا در این رجیستر ذخیره می‌شود. همچنین سیگنالی به نام Inst_byte برای رجیستر کردن دستورالعمل نیاز است که به‌صورت 8 بیتی به‌اندازه بایت دستورالعمل تعریف می‌شود.

یکی دیگر از سیگنال‌هایی که برحسب نیاز تعریف می‌کنیم CNT_Limit است که بر اساس مقادیر درون 8 بیت دستورالعمل مشخص می‌کند چه تعداد بیت باید منتقل شود به همین دلیل با نظر به اینکه ما روش انتقال اطلاعات را به‌صورت MSB first در نظر گرفتیم، پس از 32 به‌صورت نزولی می‌شمارد و اگر قرار به ارسال و دریافت یک بایت دیتا باشد تا 24، دو باید دیتا تا 16، سه بایت دیتا تا 8 و چهار بایت دیتا تا صفر می‌شمارد و به عبارتی حدومرز شمارش را مشخص می‌کند.

دیگر سیگنالی که برای محقق شدن ارتباط بین FPGA و تراشه مقصد تعریف می‌شود، سیگنال Tx است که کار آن مشابه عملکرد MOSI در SPI است اما چون در این پروژه ما فقط SDIO در اختیارداریم پس با این سیگنال انتقال دیتا را به Slave فراهم می‌کنیم. دلیل دیگر استفاده از این سیگنال این است که نرم‌افزار Vivado به هنگام check syntax اروری با این مضمون می‌دهد که نمی‌توان هم‌زمان روی یک پورت بنویسیم و بخوانیم.

درنهایت یک سیگنال شمارنده به نام BIT_CNT تعریف می‌کنیم که نهایتاً تا 32 می‌شمارد و وظیفه‌ی کنترل آن بر عهده‌ی CNT_Limit است و به دلیل اینکه این سیگنال را از نوع unsigned تعریف کردیم، از NUMERIC_STD استفاده کرده‌ایم.

به علت اینکه SPI می‌تواند در حالت‌های مختلفی باشد نیاز است که یک ماشین حالت برای آن تعریف گردد که بر اساس رخدادهایی وضعیت آن تغییر کند و وارد حالت‌های دیگر شود که این حالت‌ها در کد مشخص است و زمان استفاده از آن‌ها را نیز در بدنه‌ی process می‌بینیم. همچنین ماشین حالت آن معادل شکل زیر است.



حال که تمام سیگنال‌ها و ماشین حالت را تعریف کردیم، کد آن را در ادامه قرار می‌دهیم.

```
    --SPI Input/Output Register
    signal CS_n_INT     :   STD_LOGIC                       := '1';
    signal RW_CTL_INT   :   STD_LOGIC                       := '0';
    signal Data_In_INT  :   STD_LOGIC_VECTOR (31 downto 0)  := (others => '0');
    signal Start_INT    :   STD_LOGIC                       := '0';
    --

    --SPI Internal Signals
    signal RX_Data      :   STD_LOGIC_VECTOR (31 downto 0)  := (others => '0');
    signal Inst_Byte    :   STD_LOGIC_VECTOR (7 downto 0)   := (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;
    --
```

در مرحله‌ی بعد برای استفاده‌ی درست از SDIO، یک مدار ترکیبی با استفاده از read و write کنترل تشکیل می‌دهیم که نشانگر این است که SDIO در چه زمان در کدام حالت باشد.

SDIO زمانی در حالت خروجی است و سیگنال Tx را به تراشه مقصد می‌فرستد که RW_CTL صفر باشد پس با این کدی که در زیر داریم بدین گونه به SDIO مقدار می‌دهیم.

```
    CS_n        <= CS_n_INT;
    RW_CTL      <= RW_CTL_INT;
    SDIO        <= Tx when RW_CTL_INT = '0' else 'Z';
```

حال که این کد را در اینجا داریم می‌بایست در testbench دقیقاً عکس آن را بنویسیم که این کار به‌درستی انجام‌شده است که در ادامه مقداردهی به آن را می‌بینیم.

```
    sdio <= 'Z' when RW_CTL = '0' else '1',                    '0' after sclk_period,
                                       '1' after sclk_period*2,  '0' after sclk_period*6,
                                       '1' after sclk_period*8,  '0' after sclk_period*11,
                                       '1' after sclk_period*13, '0' after sclk_period*17,
                                       '1' after sclk_period*20, '0' after sclk_period*21,
                                       '1' after sclk_period*22, '0' after sclk_period*26,
                                       '1' after sclk_period*28, '0' after sclk_period*30;
```

یکی از دلایلی که باعث شد رجیسترهای خروجی CS_n و RW_CTL در قسمت concurrent نوشته شوند این است که اگر درون process بود حساس به کلاک می‌شد و با یک کلاک تأخیر به خروجی منتقل می‌شد.

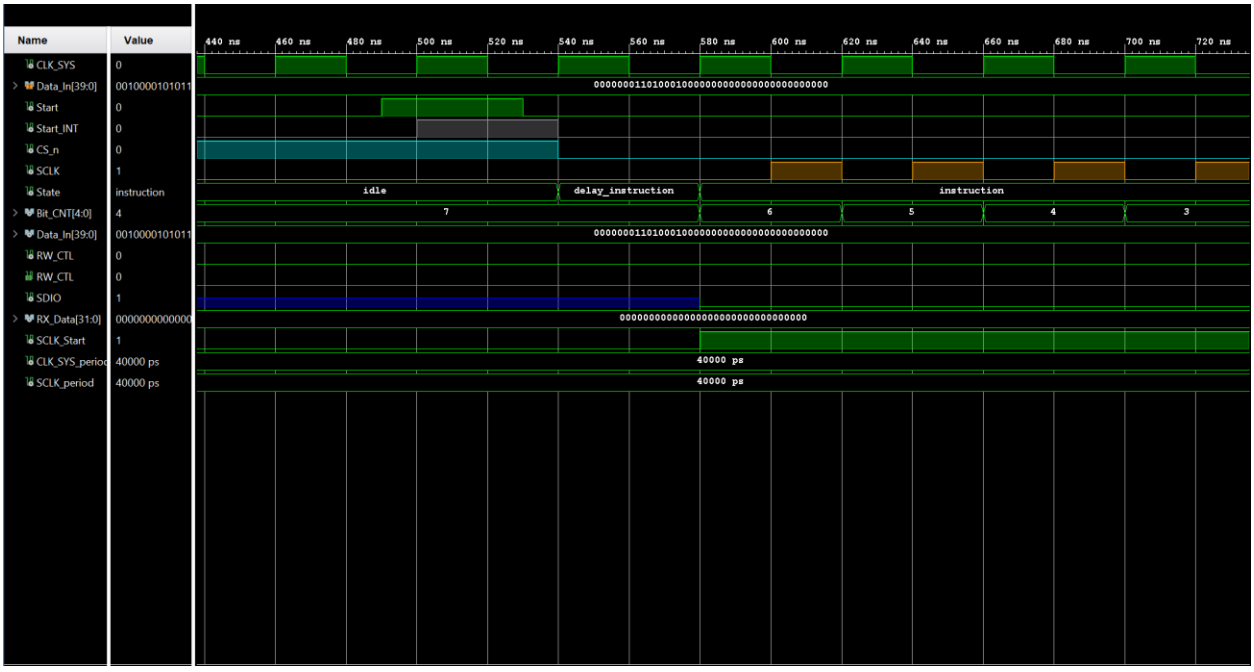
در ادامه وارد process می‌شویم و این process را حساس به کلاک تعریف می‌کنیم و در ادامه شرط حساس به لبه‌ی بالارونده را برای آن می‌گذاریم و بعد رجیسترهای ورودی را مقداردهی می‌کنیم که در اینجا ما دیتای ورودی 40 بیتی را تقسیم کرده و درون دو رجیستر متفاوت ریخته‌ایم به این صورت که بایت دستورالعمل در Inst_byte و دیتا درون Data_In_INT ریخته می‌شود.

برای ساده شدن سخت‌افزار و طراحی از دستور case when به‌جای if استفاده می‌کنیم چراکه if های تودرتو مدار را پیچیده می‌کند به دلیل اینکه در پیاده‌سازی سخت‌افزاری، if به‌صورت سریالی شبیه‌سازی می‌شود درصورتی‌که case موازی است و استفاده از case سرعت مدار را بالا می‌برد به همین دلیل برای پیاده‌سازی ماشین حالت، از Case when استفاده کرده‌ایم.

اولین حالت که SPI در شروع درون آن قرارداد حالت idle یا بیکار است و درون آن‌یک سری سیگنال‌هایی تعریف می‌شود در این دستورات if باید مقداردهی شوند.

ابتدا در حالت بیکار سیگنال‌هایی که حساس به سیگنال شروع نیستند را خارج آن تعریف می‌کنیم و سپس به‌شرط شروع یا start_INT سیگنال State را تغییر می‌دهیم که نشانگر عوض شدن حالت مدار است و در ادامه chip select هم به‌صورت Active low فعال می‌شود اما در میانه‌ی راه قبل از ورود به حالت بعدی یکسری دستوراتی قرار می‌دهیم که بر اساس بیت‌های 5 و 6 دستورالعمل مشخص می‌کند که دیتای ارسالی قرار است چند بایت باشد و بر اساس آن محدوده‌ی CNT_Limit را تعیین کند. در ابتدا این دو بیت را به یک عدد صحیح تبدیل می‌کنیم و بر اساس آن Case when می‌نویسیم که با توجه به مقدار آن، مشخص می‌شود که از 32 تا چه عددی می‌شمارد و آن عدد را مجدد به unsigned 5 بیتی تبدیل می‌کند.

حال در قسمت else اگر شرط دلخواه فراهم نشد همچنان در حالت idle بماند و CS را 1 بگذارد و همچنین حدود CNT_Limit را تا صفر بگذارد.



درون شبیه‌سازی نیز مشخص است که درون حالت idle مانده‌ایم و زمانی که Start_INT مقدار یک گرفته تغییر حالت داده‌ایم.

نکته‌ی مهم: همان‌طور که در کد مشخص است، process حساس به کلاک است و در هر لبه‌ی بالارونده‌ی کلاک تغییرات را مشاهده می‌کند و این تغییرات در لبه‌ی بالارونده‌ی بعدی اعمال می‌شود که در همین شکل هم قابل‌مشاهده است که سیگنال شروع در زمان 500 نانوثانیه یک شده ولی در زمان 540 نانوثانیه وارد حالت بعد

می‌شویم. پس باید دقت شود که اگر در حال انتقال دیتا بودیم این تأخیرها را در نظر بگیریم که دچار Data loss نشویم.

حالت بعدی که مطابق ماشین حالت وارد آن می‌شویم delay_instruction است که وظیفه‌ی آن صرفاً این است که تأخیری ایجاد کند و سپس وارد مرحله‌ی بعد شود. در این State چون یک دستورالعمل در حال انتقال و نوشته شدن روی تراشه‌ی مقصد است، پس RW_CTL باید صفر باشد تا بتوانیم روی آن بنویسیم چون 8 بیت دستورالعمل همیشه در حالت Write است. حال نکته‌ی مهم این است که در همین مرحله برنامه می‌ریزد که اولین بیت inst_byte را به slave منتقل کند و این عملیات در لبه‌ی بالارونده‌ی بعدی اجرایی می‌شود که این گفته نیز در شکل 1 پیداست و در 580 نانوثانیه است که SDIO از حالت Z خارج می‌شود و مقدار صفر را می‌گیرد که مطابق داده‌ای که ارسال کردیم، بیت اول آن صفر است. اما نکته‌ی قابل توجه این است که این دیتا در ثانیه‌ی 580 آماده می‌شود اما به دلیل Setup time و Hold time موجود، با لبه‌ی بالارونده‌ی SCLK که در آن زمان دقیقاً در وسط دیتا است، آن را می‌خواند.

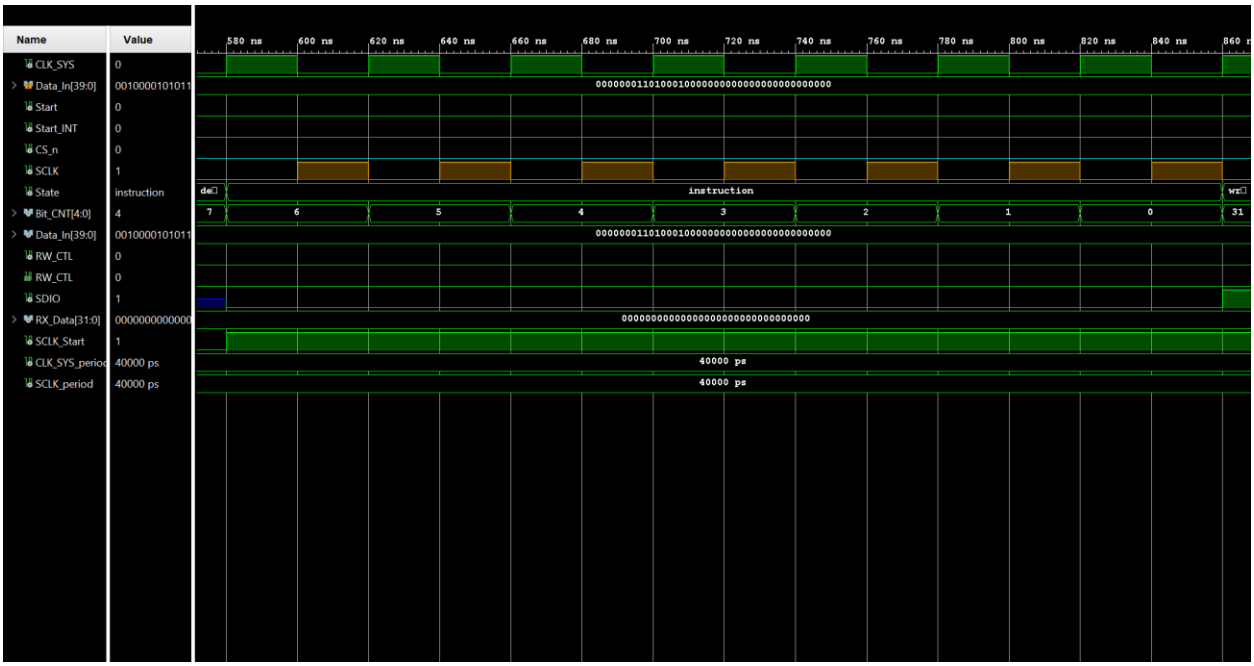
نکته‌ی مهم: اولین بیتی که در هر state مشاهده می‌شود در اصل آخرین بیتی است که از state قبلی باید منتقل می‌شده.

در ادامه در State ای که قرار داریم، از BIT_CNT یک مقدار کم می‌شود چون یک بیت دیتا منتقل‌شده و در ادامه چون گفته شد که تمام سیگنال‌ها می‌بایست درون if مقداردهی شوند، مقدار CNT_Limit را برابر خودش قرار می‌دهیم.

بعدازاین، وارد state بعدی که instruction است می‌رویم و در اینجا باید دیتای inst_byte را به slave ارسال کنیم پس سیگنال Tx را برابر inst_byte قرار می‌دهیم اما باید دقت شود که یک بیت دیتا قبلاً ارسال‌شده پس BIT_CNT را به int تبدیل می‌کنیم که حال مقدار 6 را دارد و این مقدار را به Tx می‌دهیم و در ادامه می‌بایست یک شرطی بگذاریم که تا چه زمان این کار انجام پذیرد که در یک حلقه تا زمانی که شمارنده صفر نشده در این State می‌ماند و تمام inst_byte را به مقصد منتقل می‌کند و زمانی که صفر شد مقدار آن را به 1 بازمی‌گرداند که عملاً مقدار عددی 31 است.

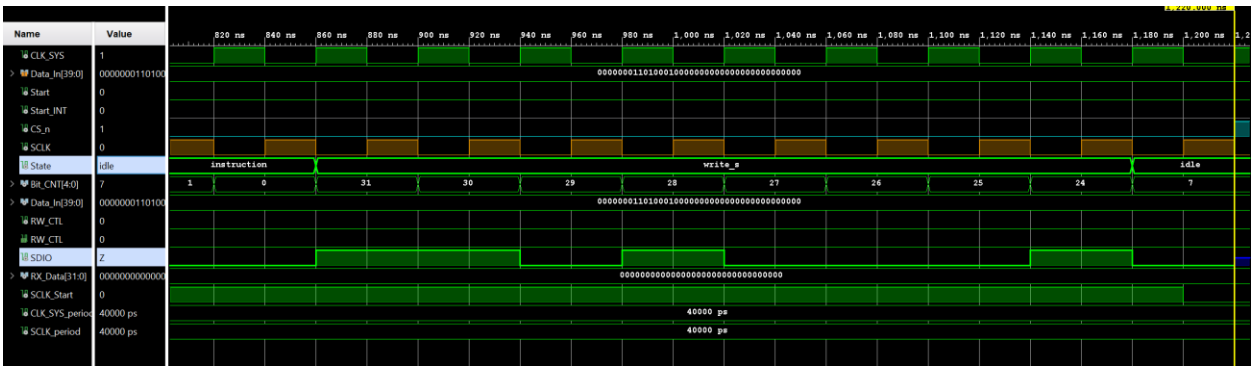
حال در اینجا به یک دوراهی برمی‌خوریم که اگر بیت پرارزش inst_byte صفر یا یک بود به کدام حالت وارد شود. اگر این بیت صفر بود باید وارد مرحله write بشویم و در غیر این صورت نشانه‌ی این است که قرار است عملیات خواندن صورت بپذیرد پس برای خواندن آماده می‌شویم.

مسئله‌ای که هست این است که SCLK در ابتدا صفر است و تا زمانی که CS صفر نشده SCLK تولید نمی‌شود که در توان صرفه‌جویی شود و بعدازآن کلاک SCLK زده می‌شود.



در اینجا نیز مشخص است که در state ای که قرار داریم تمام inst_byte از طریق SDIO منتقل می‌شود که چون ما در testbench آن دیتایی که در عکس مشخص است را به ورودی داده‌ایم همان‌ها در هر کلاک SCLK به slave منتقل می‌شوند.

چون در این مثال، بعدازاین مرحله وارد استیت write_s می‌شود ما هم کد این مرحله را بررسی می‌کنیم. مطابق inst_byte، چون یک بایت دیتا می‌خواهد به مقصد منتقل شود، پس شرطی می‌گذاریم که BIT_CNT به‌اندازه‌ی یک بایت دیتا می‌شمارد و دیتا را وارد slave می‌کند که این کار مشابه کارهایی هست که قبل‌تر نیز انجام داده‌ایم.



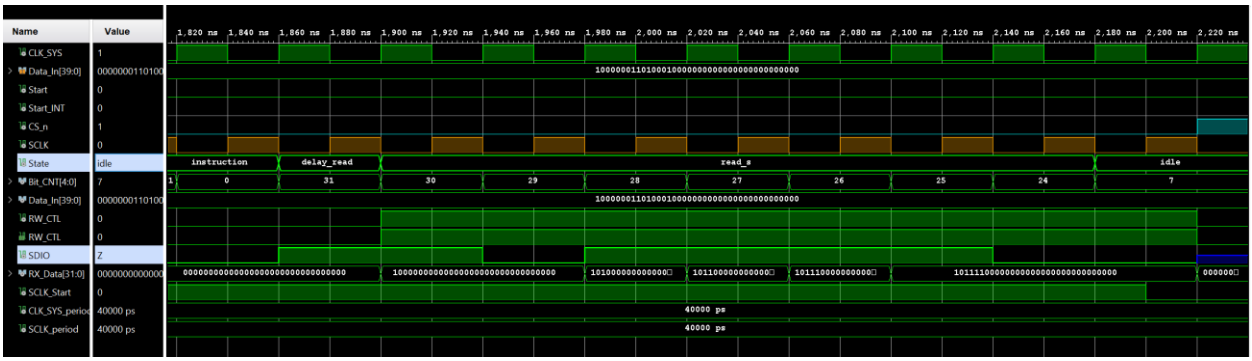
زمانی که وارد write_S state شده ایم به عبارتی در حال نوشتن روی SDIO هستیم که این دیتارا به DAC بفرستد. در زمان 860 نانوثانیه وارد حالت بعد شدیم و گفته بودیم که اخرین بیت حالت قبلی در این زمان از SDIO به سمت DAC میرود پس اولین بیت دیتا که میخواهد ارسال شود در زمان 860 توسط FPGA دیده میشود و در لبه ی بالارونده ی بعدی ارسال میشود که همانطور که مشخص است دیتای 10100010 قرار است منتقل شود و اولین بیت آن در زمان 900 نانوثانیه منتقل میشود که در آن زمان SDIO مقدار یک را داراست و بعد ازآن صفر و مجددا یک میشود که نشانه ی این است که به درستی دیتا را به DAC میفرستد.

زمانی که عملیات این state به پایان رسید یعنی زمانی که شمارنده به‌اندازه‌ی محدودیتی که بر اساس تعداد بایت‌ها گذاشتیم شمارد، SPI وارد حالت اولیه‌ی خود یعنی idle می‌شود و همچنینCS یک می‌شود و منتظر ارسال دیتای جدید می‌ماند.

اگر به دیتا توجه شود، می‌بینیم که دیتای ما 00000001101000100000000000000000000000000 است که 8 بیت اول آن 00000001 است که مطابق datasheet، بیت اول نشانه‌ی write و دو بیت بعدی نشانه‌ی انتقال یک بایت دیتا و 5 بیت بعدی آدرس اولین رجیستری است که مقداردهی می‌شود که در اینجا رجیستر Power-Down مدنظر است که در ادامه به آن مقدار 10100010 داده‌شده درصورتی‌که مقدار پیش‌فرض آن 01000000 است درحالی‌که ما مقدار دیگری به آن می‌دهیم که مقادیر رجیستر های LDOOFF LDOSTAT PWRDN Q DACOFF I DACOFF QCLKOFF ICLKOFF EXTREF است که ما در testbench فقط برای آزمودن، مقادیر دلبخواهی داده‌ایم و این آدرس‌دهی‌ها ارزش چندانی ندارند.

در Testbench دیتای بعدی که وارد کردیم بیت اول آن یک است که به عبارتی SPI را در حالت read می‌برد. در این حالت قبل از اینکه وارد مرحله‌ی خواندن برویم ابتدا به حالت delay_read می‌رویم و در اینجا مقدار Tx را Z قرار می‌دهیم که با دیتای صفر یا یک اشتباه نشود و چون در این حالت SDIO ورودی است پس در testbench مقداری برای آن مشخص می‌کنیم.

پیش‌تر نیز گفته بودیم که در این حالت همه‌ی کارها برعکس پیش می‌روند و از SDIO اطلاعات بیت به بیت می‌آید و ما آن را درون RX_Data ذخیره می‌کنیم که این کار را مجدداً با استفاده از شمارنده‌ی BIT_CNT و CNT_Limit انجام می‌دهیم و اولین بیت در همین مرحله برای ارسال به FGPA آماده می‌شود و در لبه‌ی بالارونده‌ی بعدی FGPA آن را می‌خواند و سپس به مرحله‌ی بعدی که read_S است وارد می‌شویم.



به دلیل اینکه در این حالت SDIO که چندی پیش خروجی بود حال نقش ورودی دارد پس سیگنال کنترلی RW_CTL یک می‌شود که نشانه‌ی این است که این پورت حال به‌صورت ورودی عمل می‌کند و دیتای خود را از testbench می‌گیرد که اگر به تغییرات شکل موج SDIO توجه شود همان دیتایی که در testbench تولید می‌شود را در هر لبه‌ی کلاک می‌گیرد و درون RX_Data می‌ریزد

این مرحله نیز مشابه write_S است با این تفاوت که در حال خواندن از slave است و مانند آن تا زمانی که دیتا تمام نشده آن را می‌خوانیم و درنهایت وارد حالت idle می‌شویم و منتظر دیتای جدید می‌مانیم.

مواردی که شبیه‌سازی شد درنهایت با گفته‌های datasheet مقایسه شد که قرار این بود که شکل زیر در شبیه‌سازی به دست آید.

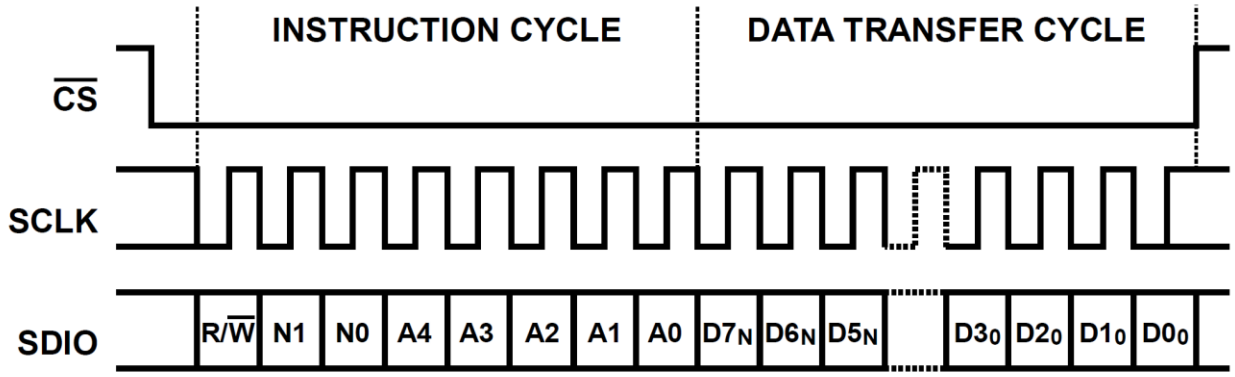


Figure 85. Serial Register Interface Timing, MSB First Write

شکل فوق برای حالتی است که قرار است write صورت بپذیرد که اگر با شکل‌های شبیه‌سازی‌شده مقایسه شود مشخص می‌شود که برای دستورات نوشتن دقیقاً به همین شکل صورت می‌پذیرد.

همچنین برای دستورات خواندن نیز مطابق شکل بعد است که قابل‌مشاهده است.

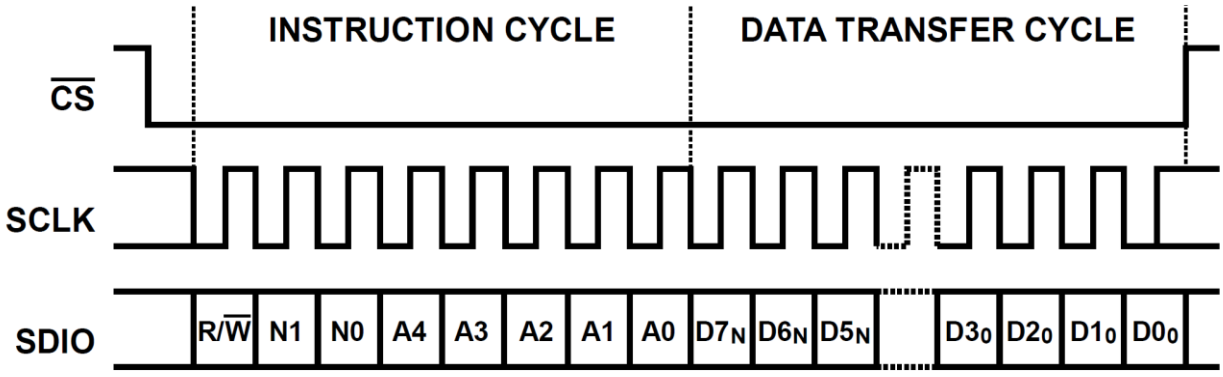


Figure 86. Serial Register Interface Timing, MSB First Read

توضیحات testbench

در testbench تمام سیگنال‌هایی که در کد اصلی تعریف‌شده‌اند را با مقدار آن‌ها در قسمت declaration تعریف کرده‌ایم که با کامنت گذاری ماهیت هرکدام از آن‌ها مشخص است.

ازآنجایی‌که در Datasheet گفته شده سیستم با کلاک 25MHz کار می‌کند پس در testbench این مقادیر را 40 نانوثانیه تعریف کردیم.

پیش‌تر گفته بودیم که برای صرفه‌جویی در توان، بهتر است SCLK زمانی که CS_n صفر بود شروع به کلاک زدن کند به همین منظور یک سیگنال میانی به نام SCLK_Start در testbench فراهم می‌کنیم که وظیفه‌ی آن همین است که در زمان صفر شدن CS فعال شود که مقادیر آن به ازای زمان‌هایی که دیتا برای انتقال می‌آید در testbench نوشته‌شده است. حال از همین سیگنال برای تولید کلاک SCLK استفاده‌شده است که فقط زمانی مقادیر SCLK تغییر می‌کند که این سیگنال فعال باشد و در غیر این صورت کاری انجام نمی‌دهد.

همچنین در اینجا CLK_SYS هم مقداردهی می‌شود که صفر و یک شدن آن نیز در قالب کد زیر انجام شود.

```
    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;
```

یکی دیگر از سیگنال‌هایی که باید آن را مقداردهی می‌کردیم سیگنال start بود که با توجه به مدت‌زمان طول کشیدن انتقال اطلاعات و بررسی فواصل زمانی، زمان‌های مناسبی برای آن‌ها انتخاب شد که در testbench قابل‌مشاهده است. اما به دلیل اینکه این سیگنال با کلاک سنکرون نیست، یک سیگنال دیگر از روی آن ساختیم که این سیگنال به‌صورت سنکرون عمل می‌کند که باعث شبیه‌سازی بهتر مدار می‌شود.

همچنین یک سری دیتا در فواصل زمانی مختلف برای آزمودن انتقال اطلاعات از یک بایت تا 4 بایت در Testbench تعریف‌شده است و زمان‌بندی‌ها بر اساس زمان انجام این انتقال‌ها در نظر گرفته‌شده است.

```
data_in_pro: process
 begin
     data_in <= "00000010001001000000000000000000000000000",
                "10000010001001000000000000000000000000000" after 01500ns,
                "00100010001001001010001000000000000000000" after 02620ns,
                "10100010001001001010001000000000000000000" after 03880ns,
                "01000010001001001101110010100010000000000" after 05460ns,
                "11000010001001001101110010100010000000000" after 07120ns,
                "01100010001001000100100111011100101000010" after 08760ns,
                "11100010001001000100100111011100101000010" after 10840ns;
     wait;
 end process data_in_pro;
```

درنهایت مهم‌ترین بخش testbench نوشتن مدار ترکیبی برای SDIO است مشابه چیزی که در کد اصلی نوشته‌شده بود. درصورتی‌که RW_CTL برابر صفر باشد به این معناست که SDIO خروجی‌ست و از کد اصلی برای آن مقدار فراهم می‌شود پس در اینجا Z می‌شود و اگر 1 بود شروع به دریافت اطلاعات می‌کند که این اطلاعات دلخواه درون testbench تعریف‌شده است که بر اساس فواصل SCLK تعریف می‌شود.

Databus موازی یا parallel

مطابق مطالب گفته‌شده در datasheet، یک باس موازی 14 بیتی دیتای موردنیاز DAC را فراهم می‌کند که وظیفه‌ی ما این است که این مسیر را برای DAC از طریق FPGA فراهم کنیم.

همان‌طور که مشخص است این DAC دو کاناله ست که برای جلوگیری از نویز، دیتا را در دو کانال بررسی می‌کند برای همین ما نیز دو کانال دیتا را وارد منطقی به اسم ODDR کنیم که خروجی را Double data rate کند و از این قابلیت برای ارتباط با DAC استفاده می‌کنیم و همچنین کلاک DCLKIO را FPGA برای DAC فراهم می‌کند پس به همین دلیل با یک ODDR دیگر این کلاک را تولید می‌کنیم و به عبارتی clock forwarding انجام می‌دهیم که یکی از قابلیت‌های ODDR ها است. همچنین خروجی هرکدام از

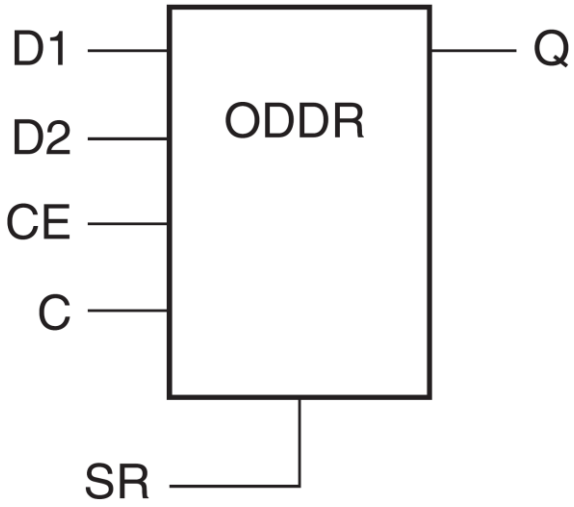
ODDR ها وارد بافرهای خروجی می‌شود که برای پیاده‌سازی سخت‌افزاری، مناسب است که خروجی FPGA از این طریق به PAD وارد شود که این کار جلوی خطاهای پیش‌بینی‌نشده را می‌گیرد. کد موارد گفته شده به صورت زیر است که با استفاده از Language template های آماده vivado نوشته شده است.

```
DCLKIO_ODDR_inst : ODDR
    generic map(
        DDR_CLK_EDGE => "OPPOSITE_EDGE",     -- "OPPOSITE_EDGE" or "SAME_EDGE"
        INIT         => '0',                 -- Initial value for Q port ('1' or '0')
        SRTYPE       => "SYNC")              -- Reset Type ("ASYNC" or "SYNC")
    port map (
        Q            => DCLKIO_INT,          -- 1-bit DDR output
        C            => DCLK_125MHz_P_90_I, -- 1-bit clock input
        CE           => '1',                 -- 1-bit clock enable input
        D1           => '0',                 -- 1-bit data input (positive edge)
        D2           => '1',                 -- 1-bit data input (negative edge)
        R            => '0',                 -- 1-bit reset input
        S            => '0'                  -- 1-bit set input
    );


--OBUF instantiation for DCLKIO
 OBUF_inst : OBUF
 generic map (
    DRIVE           => 12,
    IOSTANDARD      => "DEFAULT",
    SLEW            => "SLOW")
 port map (
    O               => DCLKIO_O,            -- Buffer output (connect directly to top-level
port)
    I               => DCLKIO_INT           -- Buffer input
 );

--ODDR instantiation for I and Q Data
 Data_Bus_Gen : for i in 13 downto 0 generate

   Data_Bus_ODDR_inst : ODDR
   generic map(
      DDR_CLK_EDGE  => "OPPOSITE_EDGE",     -- "OPPOSITE_EDGE" or "SAME_EDGE"
      INIT          => '0',                 -- Initial value for Q port ('1' or '0')
      SRTYPE        => "SYNC")              -- Reset Type ("ASYNC" or "SYNC")
   port map (
      Q             => Data_Bus_Int(i),     -- 1-bit DDR output
      C             => DCLK_125MHz_P_I,     -- 1-bit clock input
      CE            => '1',                 -- 1-bit clock enable input
```

```
      D1            => I_DAC_I(i),          -- 1-bit data input (positive edge)
      D2            => Q_DAC_I(i),          -- 1-bit data input (negative edge)
      R             => '0',                 -- 1-bit reset input
      S             => '0'                  -- 1-bit set input
   );

 end generate Data_Bus_Gen;                 -- End of ODDR_inst instantiation

 --OBUF instantiation for I and Q Data
 Data_Bus_OBUF_Gen : for i in 13 downto 0 generate

   OBUF_inst : OBUF
   generic map (
      DRIVE         => 12,
      IOSTANDARD    => "DEFAULT",
      SLEW          => "SLOW")
   port map (
      O             => Data_Bus_O(i),       -- Buffer output (connect directly to top-level
port)
      I             => Data_Bus_Int(i)      -- Buffer input
   );

   end generate Data_Bus_OBUF_Gen;
```

یکی دیگر از نکات مهم در حین طراحی این است که ODDR ها ورودی و خروجی تک‌بیتی دارند پس برای تولید دیتاهای 14 بیتی که رزولوشن DAC ما است، می‌بایست از 14 ODDR موازی برای این کار استفاده کنیم که هرکدام از این‌ها، یک بیت دیتا برای ما تولید می‌کردند.

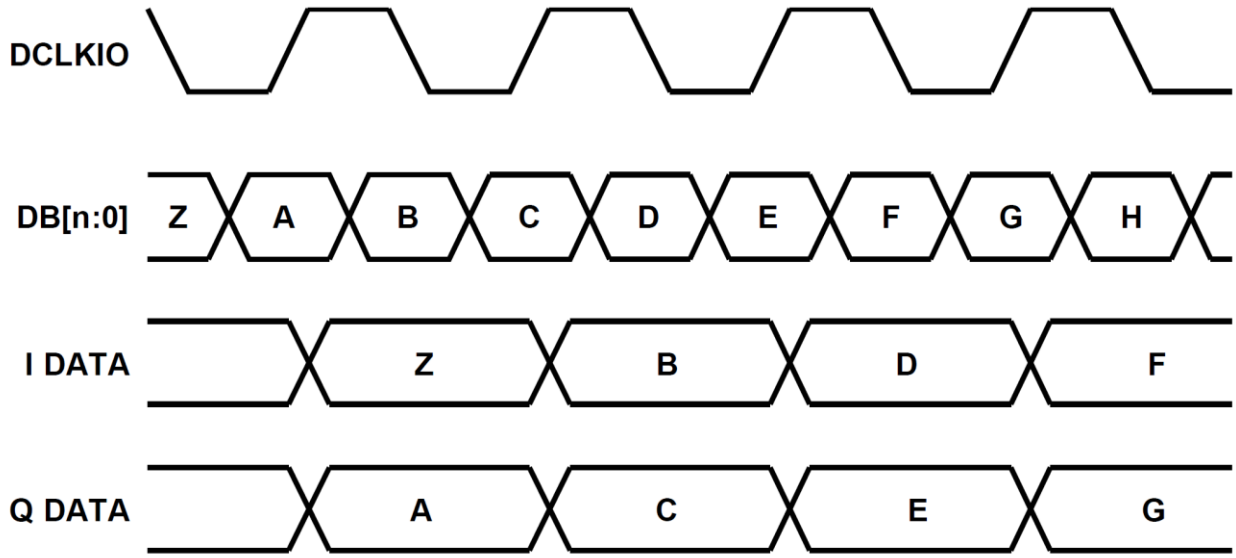
ازآنجایی‌که تراشه‌ی FPGA ما، از سری Xilinx 7 بود و در محیط Vivado شبیه‌سازی انجام می‌شد، مطابق Datasheet سری 7 این شرکت، معماری ODDR با سری پیش تفاوت‌هایی کرده که یکی از آن‌ها عدم وجود دو کلاک برای این سری است و فقط با یک کلاک کار می‌کند که شکل آن را در پایین می‌بینیم.



اما نکته‌ی حائز اهمیت این است که در اینجا ما یک کلاک برای ODDR هایی داریم که دیتا را generate می‌کنند و یک کلاک هم باید برای تولید DCLKIO تولید کنیم که این دو باهم اختلاف‌فاز 90 درجه دارند و به همین منظور با استفاده از قابلیت clock wizard شبیه‌ساز Vivado، یک کلاک 125MHz را به این wizard دادیم و سه کلاک از آن دریافت کردیم. یک کلاک دقیقاً مشابه کلاک ورودی و 125MHz برای ODDR هایی که دیتا را منتقل می‌کنند و یک کلاک 125MHz با اختلاف‌فاز 90 درجه برای تولید DCLKIO و همچنین یک کلاک 25MHz برای clk_sys و SCLK در SPI که کلاک متفاوتی به نسبت ODDR ها دارند.

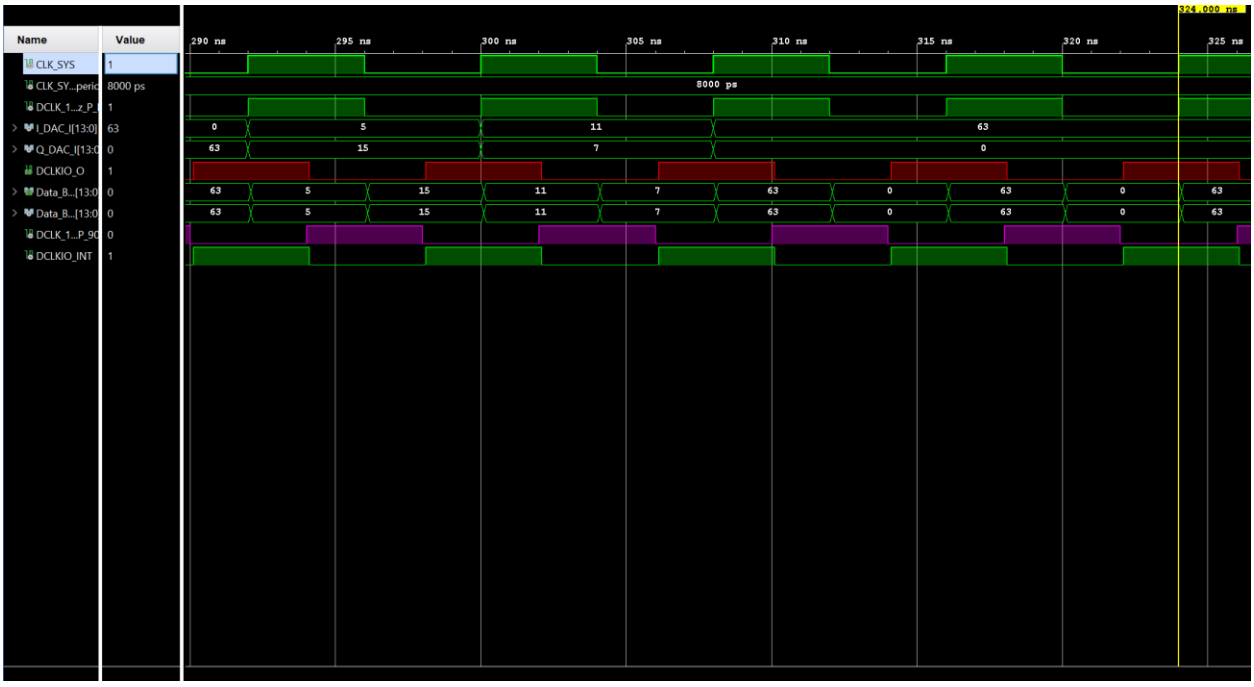
طبق مطالب گفته‌شده در datasheet، 4 مدل متفاوت برای شبیه‌سازی و انتقال دیتا به DAC وجود دارد که با دو رجیستر IFIRST و IRISING نوع آن مشخص می‌شود که ما در testbench مقادیر و آدرس رجیسترها را طوری تعیین کردیم که این دو رجیستر به ترتیب مقادیر 1 و صفر را داشته باشند که تداعی‌کننده‌ی شکل زیر برای ارسال دیتا است.

مطابق شکل، دیتایی که IDATA به ODDR ارسال می‌شود در لبه‌ی پایین‌رونده‌ی DCLKIO برداشته می‌شود و دیتای QDATA در لبه‌ی بالارونده توسط DAC خوانده می‌شود که این مدل انتقال دیتا با موفقیت شبیه‌سازی شد و تصاویر شبیه‌سازی نیز در ادامه آورده شده است.



NOTES:
1. DB[n:0], WHERE n IS 7 FOR THE AD9114, 9 FOR THE AD9115, 11 FOR THE AD9116, AND 13 FOR THE AD9117.

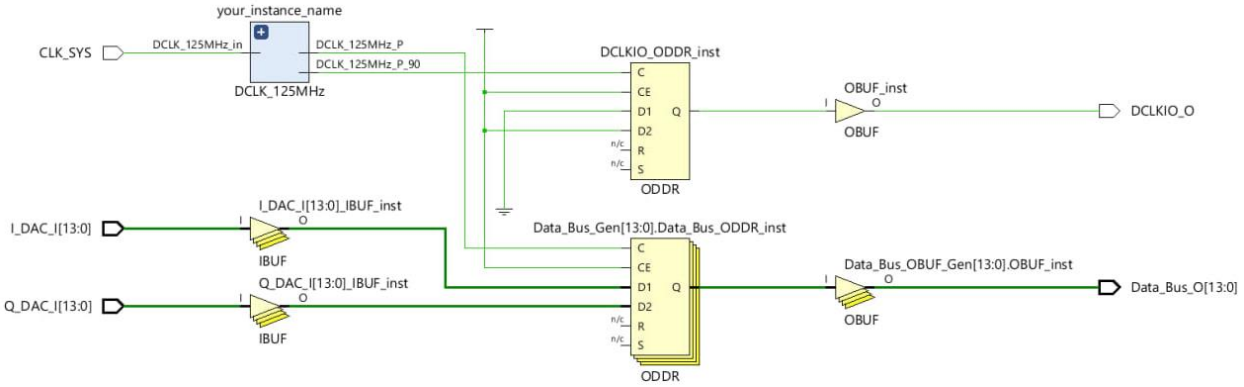
Figure 91. Timing Diagram with IFIRST = 1, IRISING = 0



در این شکل نیز مشخص است که دیتا در مرکز DCLKIO آماده می‌شود و زمان Setup time را رعایت می‌کند و همچنین تا زمانی بعد از خورده شدن کلاک نیز مقدار خود را hold می‌کند. همچنین شایان‌ذکر است

که مطابق عکس ابتدا در هر کلاک DCLKIO، QDATA مقدار خود را در خروجی می‌گذارد و سپس در لبه‌ی پایین‌رونده، IDATA ست که به خروجی می‌رود.

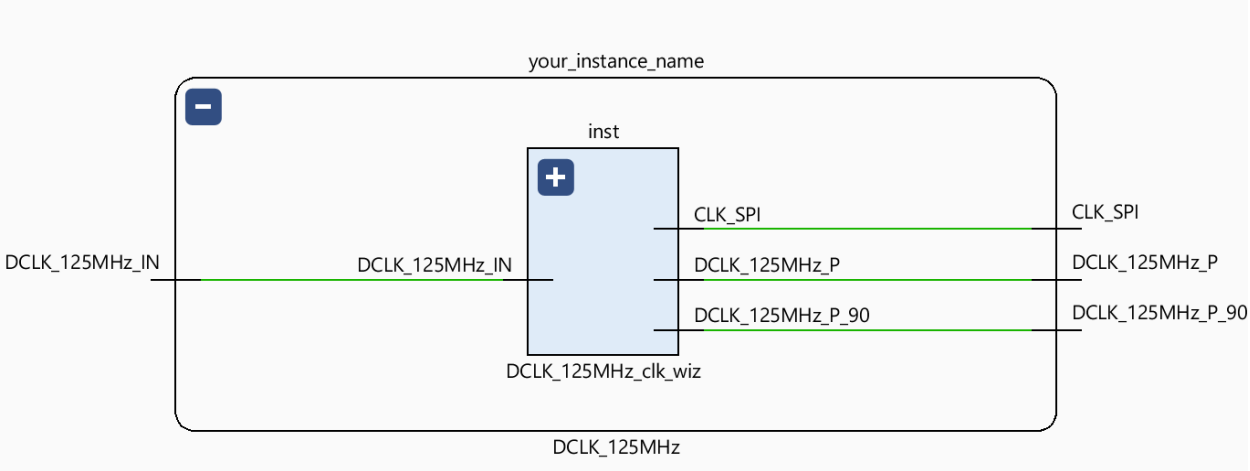
برای تولید ODDR از کد از پیش تعریف‌شده‌ی درون شبیه‌ساز Vivado استفاده کردیم اما به دلیل اینکه 14 عدد ODDR دقیقاً یکسان نیاز بود، با استفاده از دستور generate، 14 عدد ODDR تولید شد و مورداستفاده قرار گرفت که شکل آن در زیر پیداست.



فایل نهایی

در ابتدا برای پیچیده نشدن طراحی و کد نویسی، SPI و Parallel جداگانه نوشته و شبیه‌سازی شدند و پس از انجام تست‌ها و شبیه‌سازی‌ها و اطمینان از صحت هرکدام، یک کد دیگر که حاوی هر دو پروتکل بود، نوشته شد که در قالب فایل آپلود شده، قابل‌مشاهده است.

تفاوت این کد با کدهای بررسی‌شده در این است که سیستم با یک کلاک 125MHz کار می‌کند و این کلاک وارد یک MMCM شده و سه کلاک جدید برای ما تولید می‌کند که یکی برای ODDR های انتقال دیتا، یکی برای ODDR تولیدکننده‌ی DCLKIO و دیگری برای CLK_SYS مرتبط با SPI است که این کلاک با فرکانس 25MHz کار می‌کند.



ازآنجایی‌که SPI در اصل آماده‌سازی DAC برای دریافت اطلاعات از طریق مقداردهی به رجیسترها است، ما نیز در testbench مقادیر را طوری تغییر دادیم که اولین دیتای ارسال‌شده برای DAC، شامل آدرس رجیستر سوم باشد که آدرس 00010 را دارد و چون مدل انتقال دیتا این‌گونه است که در لبه‌ی بالارونده اول، دیتای QDATA را برمی‌دارد و سپس در لبه‌ی پایین‌رونده IDATA منتقل می‌شود پس رجیسترهای IFIRTS و IRISING باید به ترتیب مقدار 1 و صفر را داشته باشند که این شرط در testbench رعایت شده است.

مورد بعدی این است که در این پیاده‌سازی، به دلیل اینکه از ابتدا CLK_SYS برای SPI آماده نیست و این MMCM است که باید این کلاک را تولید کند، پس با شبیه‌سازی SPI که قبل‌تر دیدیم مقداری تفاوت دارد اما هردوی آن‌ها دقیقاً مانند هم عمل می‌کنند و نتایج شبیه‌سازی در این پیاده‌سازی نیز مانند شبیه‌سازیSPI و شبیه‌سازی Parallel است که در ادامه در شکل می‌بینیم.

اگر به این شکل دقت شود، می‌بینیم که CLK_SYSn کلاک اصلی سیستم است که از آن سه کلاک تولید می‌شود و هرکدام برای موارد مختلفی استفاده می‌شوند. اولین آن DCLK_P است که برای ODDR های IDATA و QDATA ست و مشاهده می‌شود که در هر کلاک سایکل، هم IDATA و هم QDATA به DAC منتقل می‌شود که خاصیت double data rate در آن رعایت شده است.

نتیجه‌گیری:

هدف اصلی این پروژه طراحی یک FPGA بود که به‌عنوان تراشه‌ی master با DAC که در نقش slave است ارتباط برقرار کرده و اطلاعاتی که DAC نیاز دارد را برای آن تولید کند و در قالب دو پروتکل SPI باهدف کانفیگ کردن رجیسترهای DAC و parallel برای ارسال دیتای موردنیاز DAC جهت تبدیل دیتای دیجیتال به آنالوگ به آن ارسال و یا از آن دریافت کند. در این پروژه توانستیم هدف اصلی دیتاشیت را برآورده کرده و FPGA خواسته‌شده را طراحی کنیم. نتایج testbench نیز نشان داد که FPGA به‌درستی طراحی‌شده و با توجه به ورودی‌ها، خروجی‌های صحیحی را مشاهده می‌کنیم.