
دانشکده مهندسی برق

پروژه درس VHDL

# شبیه سازی پروتکل انتقال داده ADF4360-5

استاد :

دکتر ستار میرزا کوچکی

دانشجو:

بهداد صادقیان پور

دی ۱۴۰۱


بسم الله الرحمن الرحيم

# فهرست مطالب

# فهرست اشکال

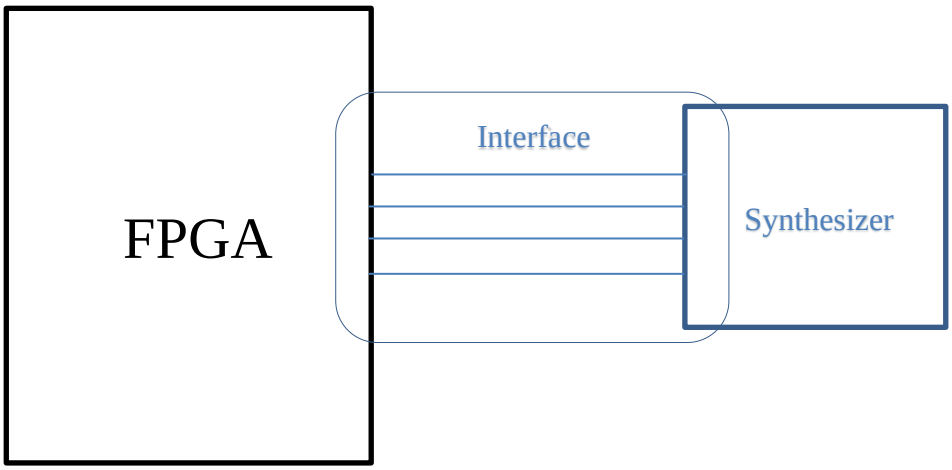
# فهرست جداول

# فصل ۱:

# مقدمه

# ۱-۱- مقدّمه

در ابتدا برای آشنایی کلی با روند پروژه و کارهای صورت‌گرفته شده، توضیح مختصری در مورد پیاده‌سازی اینترفیس و مدارات اینترفیسی داده می‌شود. در حالت کلی به مجموعه تمهیدات سخت‌افزاری و نرم‌افزاری پیاده‌سازی شده، برای دو قطعه دیجیتال، اینترفیس می‌گویند.




شکل (۱-۱) توصیف نحوهٔ ارتباط قطعه جانبی با قطعه دیجیتالی اصلی

در شکل بالا synthesizer (در ادامه توضیح داده خواهد شد)، به‌عنوان یک peripheral یا قطعه جانبی در نظر گرفته شده است. هدف ما از این پروژه توصیف ارتباط بین FPGA و synthesizer است. برای ایجاد این ارتباط باید تمهیداتی در نظر گرفته شود. برای مثال پین‌هایی از هر دو قطعه دیجیتالی به یکدیگر متصل شود و سیگنال‌هایی بین این دو ردوبدل شود. منبع اصلی این طراحی دیتاشیت قطعه جانبی است. در این دیتاشیت اطلاعات کاملی از پروتکل ارتباطی، نحوهٔ اتصال پایه‌ها، وظیفهٔ هر پایه و ... نوشته شده است. برای آشنایی بیشتر در فصل بعدی توضیح مختصری در مورد قطعه جانبی این پروژه یعنی ADF4360-5 که یک نوع synthesizer است داده می‌شود تا درک بهتر و بیشتری از آنچه در ادامه بیان می‌کنیم داشته باشیم.

# فصل ۲:

# پروتکل ارتباطی SPI و انواع آن

## ۲-۱- ارتباط سریال SPI و انواع آن

در حالت کلی دو نوع ارتباط SPI[1] داریم، 3-wire و 4-wire که در اینجا به توضیح هرکدام می‌پردازیم. در ابتدا به توضیح SPI از نوع 4-wire می‌پردازیم که باعث می‌شود درک بهتری نسبت به 3-wire داشته باشیم.

### ۲-۱-۱- ارتباط سریال SPI از نوع 4-wire

پروتکل SPI یک ارتباط داده سریال هماهنگ است که در حالت کاملاً دوطرفه عمل می‌کند. دراین‌ارتباط یک قطعه دیجیتالی به‌عنوان master عمل می‌کند که وظیفهٔ کنترل و صدور فرمان برای برقراری ارتباط را دارد. قطعه یا قطعه‌های دیجیتالی دیگر به‌عنوان slave عمل می‌کنند که وظیفهٔ دریافت فرمان صادر شده از master و دریافت اطلاعات ارسال شده را دارند.

در حالت کلی SPI چهار سیگنال دارد که در زیر به توضیح هر یک از آن‌ها می‌پردازیم.

۱. SCLK : سیگنال کلاک سریال (معمولاً توسط master صادر می‌شود)

۲. MOSI : Master Out – Slave In

۳. MISO : Master In – Slave Out

۴. SS : انتخاب قطعه الکترونیکی جانبی برای برقراری ارتباط

سیگنال SS در صورت‌های دیگری هم از جمله LE[2] و CS[3] می‌تواند ظاهر شود. نکته‌ای که باید به آن توجه داشت این است که سیگنال SS معمولاً به‌صورت فعال-پایین (Active Low) از آن استفاده می‌شود.

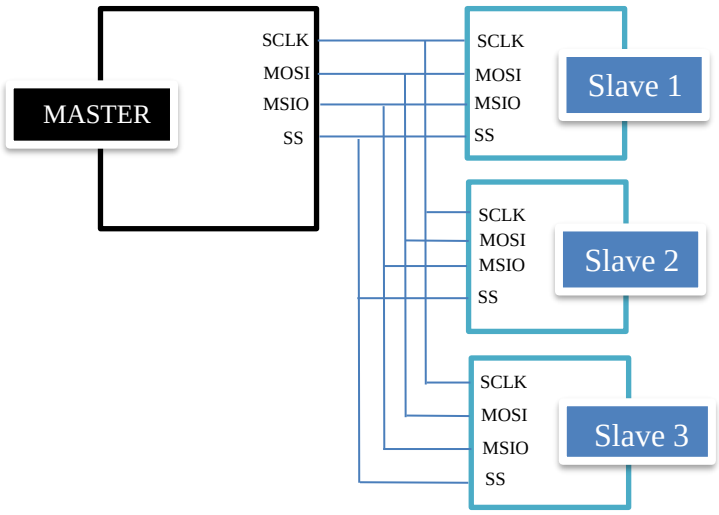
[1] Serial Peripheral Interface

[2] Load Enabel

[3] Chip Select

### ۲-۱-۱-۱- عملکرد

در ارتباط SPI می‌توان یک master با یک یا چند slave ارتباط برقرار کرد. انتخاب اینکه چه slave ای بلید انتخاب شود توسط پایه کنترلی SS صورت می‌گیرد. در زیر نمای کلی این ارتباط را مشاهده می‌کنید.




شکل (۲-۱) نمایش کلی ارتباط بین دستگاه‌های الکترونیکی در SPI نوع 4-wire

برای شروع تبادل داده‌ها، دستگاه master نخست پالس ساعت را با فرکانسی کمتر یا برابر با حداکثر مقداری که دستگاه slave پشتیبانی می‌کند تنظیم می‌کند که معمولاً در حد چند مگاهرتز است. سپس یک سیگنال صفر منطقی از خط انتخاب قطعه به salve می‌فرستد. برای این به دستگاه صفر می‌فرستیم که سطح فعال آن صفر است (Active Low) یعنی سطح خاموش آن، یک منطقی است. اگر نیازی به وقفه بود (مانند در قطعه‌های تبدیل آنالوگ به دیجیتال) دستگاه master باید حداقل به آن میزان صبر کند و سپس پالس ساعت را به Slave بفرستد.

در هر دورهٔ پالس ساعت SPI، یک تبادل داده کاملاً دوطرفه رخ می‌دهد . master داده را از خط MOSI می‌فرستد و Slave نیز آن را از همان خط دریافت می‌کند.slave داده را از خط MISO می‌فرستد و master

نیز آن را از همان خط دریافت می‌کند.

در بالا توضیح مختصری دررابطه‌با نحوه عملکرد پروتکل SPI از نوع 4-wire داده شد. نکته‌ای که باید به آن توجه داشت این است که پروتکل ارتباطی موردنیاز ما برای ADF4360-5 ، SPI و از نوع 3-wire است که به دلیل شباهت این دو نوع با یکدیگر ابتدا 4-wire توضیح داده شد. در بخش بعدی به توضیح 3-wire می‌پردازیم.

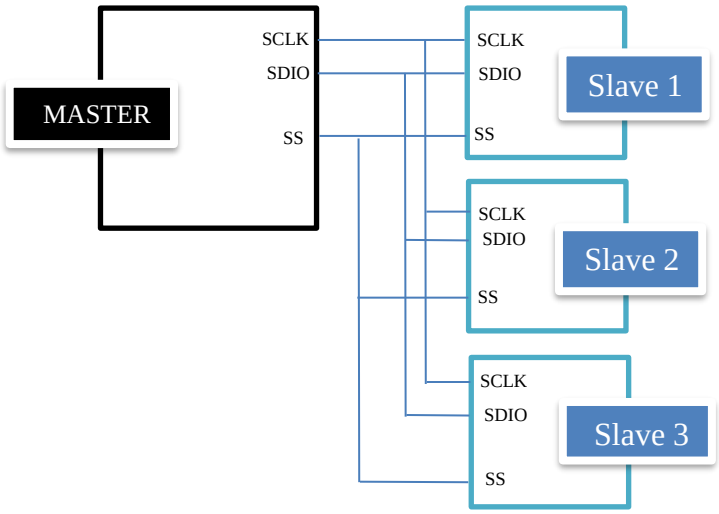
## ۲-۱-۲- ارتباط سریال SPI از نوع 3-wire

همان‌طور که در بخش قبلی گفته شد برای ارتباط 4-wire از ۴ سیگنال یا پایه استفاده می‌کنیم. همان‌طور که از اسم 3-wire مشخص است برای استفاده از این نوع SPI نیاز به ۳ سیگنال یا پایه داریم.

در این پروتکل ارتباطی، ارسال داده و دریافت آن فقط بر روی یک‌پایه صورت می‌گیرد که در قطعات جانبی مختلف این پایه نام متفاوتی دارد؛ ولی معمولاً با نام SDIO[1] آن را می‌شناسیم.

معمولاً به ارتباط 3-wire ارتباط Bidirectional یا دوطرفه نیز گفته می‌شود. در زیر نمای کلی از این پروتکل را مشاهده می‌کنید.

[1] Serial Data I/O




شکل (۲-۲) نمایش کلی ارتباط بین دستگاه‌های الکترونیکی در SPI نوع 3-wire

نکته‌ای که باید به آن توجه داشت وضعیت پایهٔ SDIO در حالت ارسال و یا دریافت اطلاعات است که باید باتوجه‌به دیتاشیت قطعه الکترونیکی در نظر گرفته شود. نکتهٔ دیگر این است که بر خلاف 4-wire که یک ارتباط full-duplex بود، 3-wire یک پروتکل half-duplex است.

## فصل ۳:

# پروتکل ارتباطی ADF4360-5

# ۳-۱- مقدمه

در این فصل به بررسی ADF4360-5 به‌عنوان یک synthesizer می‌پردازیم. سینتزایزر یک قطعه الکترونیک است که محدوده‌ای از فرکانس‌ها را توسط یک فرکانس مرجع تولید می‌کند. از این قطعه در گیرنده‌های رادیویی، تلویزیون‌ها، موبایل‌ها و ... استفاده می‌شود.

سینتزایزر ها به سه دستهٔ کلی تقسیم‌بندی می‌شوند که به ترتیب عبارت‌اند از:

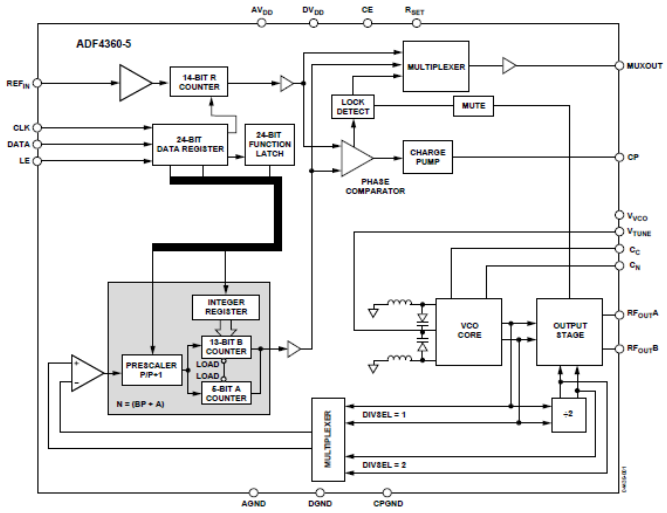
۱. direct analog synthesizer
۲. direct digital synthesizer
۳. indirect digital synthesizer

نوع سوم بیان شده به دو قسمت integer-N و fractional-N تقسیم‌بندی می‌شود که ADF4360-5 از نوع integer-N است.

## ۳-۱-۱- ویژگی‌های ADF4360-5

۱. محدوده فرکانسی خروجی: 1200 MHz to 1400 MHz
۲. ولتاژ تغذیه: 3.0 تا 3.6 ولت
۳. استفاده از پروتکل SPI از نوع 3-wire

در ادامه تصویر کلی از عملکرد واحدهای مختلف synthesizer آورده شده است که بیانگر وظیفهٔ هر واحد بکار گرفته شده در synthesizer است.




شکل (۳-۱) بلوک دیاگرام عملکردی در synthesizer

همان‌طور که در بالا گفته شد، پروتکل ارتباطی ADF4360-5، SPI و از نوع 3-wire است. که موضوع اصلی پروژه هم دررابطه‌با پیاده‌سازی این پروتکل است و در این قسمت از توضیح مداری سینتزایزر خودداری شده است؛ بنابراین تمرکز اصلی این فصل مربوط به نمودارهای زمانی و مسائل مربوط به پیاده‌سازی این پروتکل است.

## ۳-۱- نمودار مشخصات زمانی

یکی از مهم‌ترین بخش‌ها برای طراحی و پیاده‌سازی پروتکل ارتباطی نمودار مشخصات زمانی پالس‌ها است که اطلاعاتی مهمی از جمله فرکانس پالس ساعت و عرض پالس‌های مختلفی که برای برقراری ارتباط نیاز داریم را در اختیار ما قرار می‌دهد. قبل از اینکه به سراغ توصیف پارامترهای زمانی بپردازیم، به معرفی پایه‌های مربوطه برای برقراری ارتباط 3-wire می‌پردازیم.

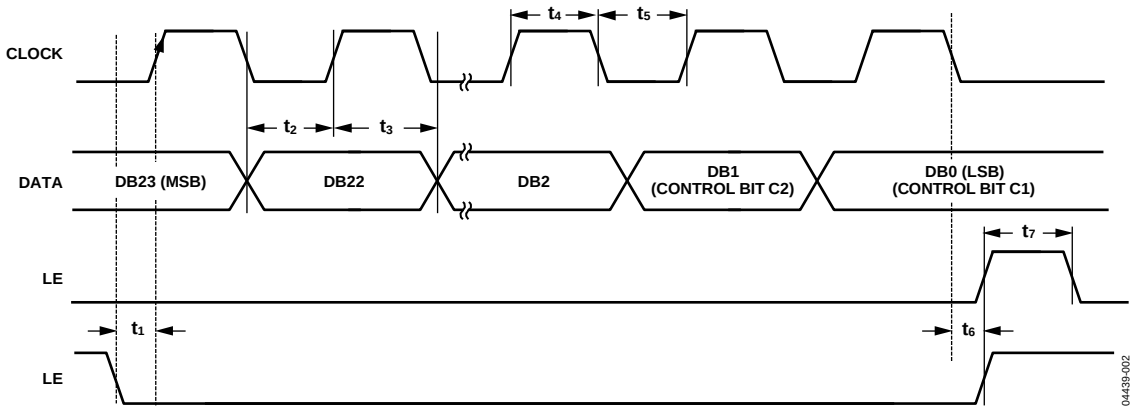
همان‌طور که در بخش‌های قبلی گفته شد برای برقراری ارتباط 3-wire نیازمند سه سیگنال اصلی هستیم. در اینجا این سه سیگنال با نام‌های Clock ، DATA و LE مشخص شده است که در جدول زیر به تعریف هرکدام از این پایه‌ها پرداخته‌ایم.

نکتهٔ مهمی که باید به آن توجه داشت این است که پایهٔ DATA در ADF4360-5 فقط وظیفه ی دریافت اطلاعات از master را دارد و دیتا و یا بیتی ارسال نمی کند.

جدول (۳-۱) توضیحات عملکرد پایه‌های ADF4360-5

| شمارهٔ پایه | Mnemonic | توضیحات |
|---|---|---|
| ۱۷ | CLK | ورودی کلاک. از این پایه برای دریافت سیگنال پالس ساعت تولید شده توسط master استفاده می‌شود. دیتاهای دریافتی با هر پالس ساعت در رجیسترهای مخصوصی ذخیره می‌شوند. |
| ۱۸ | DATA | از این پایه برای دریافت اطلاعات استفاده می‌شود. در ابتدا بیت‌هایی که دریافت می‌شوند به‌صورت بیت‌های باارزش مکانی بالاتر است. (MSB) |
| ۱۹ | LE | زمانی که این پایه صفر فعال است عملیات انتقال دیتا صورت می‌گیرد. در آخر برای ذخیره‌سازی بیت‌های دریافتی در رجیستر مربوطه که با بیت‌های کنترلی انتخاب می‌شود، این پایه باید به یک منطقی برسد. |

بنابراین، با اســـتفاده از جدول بالا به تمام ویژگی‌های پایه‌هایی که برای برقراری ارتباط 3-wire اســـتفاده می‌شوند پی می‌بریم. شکل (۲-۳) به توضیح زمانی شکل موج‌های این پایه‌ها برای برقراری ارتباط 3-wire می‌پردازد.




شکل (۳-۲) نمودار مشخصات زمانی

اطلاعات زمانی شکل موج‌های شکل (۳-۲) در جدول آمده است.

جدول (۳-۲) مقادیر زمانی پالس‌ها

| Parameter | Limit at $T_{MIN}$ to $T_{MAX}$ (B Version) | Unit | Test Conditions/Comments |
|---|---|---|---|
| $t_1$ | 20 | ns min | LE Setup Time |
| $t_2$ | 10 | ns min | DATA to CLOCK Setup Time |
| $t_3$ | 10 | ns min | DATA to CLOCK Hold Time |
| $t_4$ | 25 | ns min | CLOCK High Duration |
| $t_5$ | 25 | ns min | CLOCK Low Duration |
| $t_6$ | 10 | ns min | CLOCK to LE Setup Time |
| $t_7$ | 20 | ns min | LE Pulse Width |

در ابتدا به بررسی سیگنال کلاک می‌پردازیم. همان‌طور که در شکل ۳-۲ مشخص است. دوره تناوب سیگنال پالس ساعت برابر است با:

$$t_4 + t_5 = T$$

بنابراین، داریم:

$$25^{ns} + 25^{ns} = 50^{ns}$$

باتوجه‌به رابطهٔ دوره تناوب و فرکانس داریم:

$$f = \frac{1}{T} = \frac{1}{50^{ns}} = 20MHz$$

بنابراین، باتوجه‌به روابط بالا فرکانس کلاک ADF4360-5 برابر با 20 MHz است.

پارامتر زمانی دیگری که مورد بررسی قرار می‌دهیم، LE Setup Time یا $t_1$ است. در واقعیت پالس های مربعی در لحظه به صفر و یک منطقی تبدیل نمی شوند، این تغییر حالت پالس ها با تاخیر همراه است. برای اینکه این تاخیر در پیاده سازی های مدارات دیجیتالی مشکل زا نشود تاخیری، برای اطمینان از حالت اصلی پالس، در نظر می گیرند. برای مثال در اینجا t1 نشان دهنده ی این است که بعد از مدت زمان ۲۰ نانو ثانیه بعد صفر شدن پایه ی LE، سیگنال پالس ساعت می بایست شروع به نوسان کند.

همچنین همین حالت هم برای بیت‌های ارسالی بر روی پایه Data باید در نظر گرفته شود که در جدول بالا با پارامتر t2 نشان‌داده‌شده است.

تعریف دیگری که در اینجا داریم hold time است. hold time در مدارات دیجیتالی عبارت است از حداقل مدت زمانی که می‌بایست دادهٔ ورودی در حالت پایدار خود باقی بماند تا این داده به‌عنوان دادهٔ معتبر از دید کلاک در نظر گرفته شود را hold time می‌گوییم. در اینجا hold time در نظر گرفته شده برای داده برابر است با 10 ns این به این معنا است که داده حداقل به مدت 10 ns باید در حالت پایدار و معتبر خود باقی بماند و بعد سیگنال کلاک لبهٔ مربوطه را فعال کند؛ بنابراین پارامترهای زمانی جدول ۳-۲ تعریف شدند.

نکتهٔ بعدی که باید در نمودارهای زمانی به آن توجه کرد سیگنال داده است، همان‌طور که در نمودار مشخص است، اولین بیت ارسالی از طرف master به‌عنوان بیت باارزش مکانی بیشترین (MSB) در نظر گرفته می‌شود و بعد با هر لبهٔ بالا رنده کلاک بیت بعدی باارزش مکانی کمتر ارسال می‌شود. پارامترهای t2 و t3 مربوط به زمان‌های hold time و setup time است که در بالا توضیح داده شد.

پارامتر زمانی t7 مدت زمانی است که سیگنال LE باید در یک منطقی باقی بملند و بعد از آن دوباره می‌توانیم عملیات ارسال (برنامه‌ریزی رجیستر بعدی) را شروع کنیم.

### ۳-۱-۲- نتیجه‌گیری

برای برقراری ارتباط با ADF4360-5 ابتدا پایهٔ LE را در صـفر منطقی قرار می‌دهیم، سـپس به‌اندازه t1 در این حالت باقی بماند و بعد از آن سـیگنال کلاک فعال شـود و به تعداد ۲۴ لبهٔ بالارونده کلاک بیت‌های داده با رعایت hold time و setup time ارسال شوند و در آخر هم برای ذخیره‌سازی داده‌ها در لچ مربوطه و شروع دوبارهٔ عملیات ارسـال پایه LE به مدت t7 در وضـعیت یک منطقی باقی بماند و دوباره به حالت صـفر منطقی بازگردد.

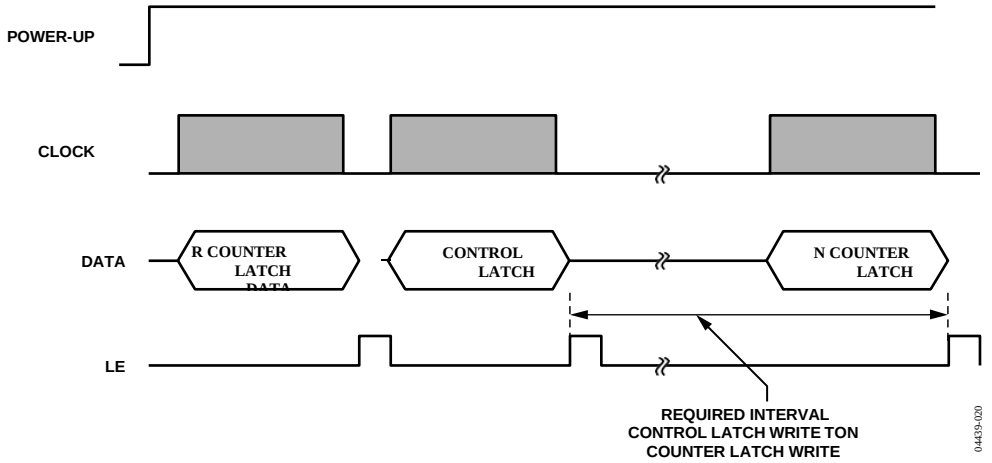
## ۳-۲- توضیحات پیرامون رجیسترهای ADF4360-5

پیاده‌سـازی این پروتکل ارتباطی به‌منظور برنامه‌ریزی رجیسـترهای داخلی ADF4360-5 صـورت می‌گیرد، باتوجه‌به اینکه این پروژه در حد شبیه‌سازی است، و بر روی برد عملی سنتز و پیاده‌سازی نمی‌شود، بنابراین ترتیب بیت های داده ارسالی و تعریف هر کدام از این بیت ها برای ما چندان اهمیتی ندارد تمرکز اصلی این پروژه بر روی پیاده سازی پروتکل ارتباطی 3-wire است، اما در ادامه به توضیح مختصر رجیستر های ADF4360-5 می پردازیم.

ADF4360-5 به طور کلی شامل ۳ رجیستر به نام های Control Latch ، N Counter Latch و R Counter Latch است. هرکدام از این لچ ها ۲۴ بیتی است، بنابراین دادهٔ ارسالی ما برای برنامه‌ریزی این لچ ها هم همان‌طور که در نمودار زمانی مشـخص شـد ۲۴ بیتی اسـت. ترتیب پیشـنهادی ارائه شـده در دیتاشیت به‌صورت زیر است.

۱. R Counter Latch
۲. Control Latch
۳. N Counter Latch

نکته‌ای دیگر که باید به آن توجه کرد نمودار برنامه‌ریزی این لچ ها است که در زیر به آن اشاره شده است.




شکل (۳-۳) نمودار زمانی برنامه‌ریزی لچ ها

همان‌طور که مشـــاهده می‌شـــود بعد از برنامه‌ریزی Conntrol Latch باید یک گپ زمانی در نظر گرفت و سپس N Counter Latch را برنامه‌ریزی کنیم. مدت‌زمان این فاصله زمانی توسط مقدار خازن Cn مشخص می‌شود. جدول زیر بیانگر مقادیر استاندارد برای این مسئله است.

جدول (۳-۳) مقادیر خازن و گپ زمانی

| $C_N$ Value | Recommended Interval Between Control Latch and N Counter Latch | Open-Loop Phase Noise at 10 kHz Offset |
|---|---|---|
| 10 µF | ≥ 5 ms | −88 dBc |
| 440 nF | ≥ 600 µs | −87 dBc |

برای مثال با مقدار خازن ۱۰ میکروفاراد مقدار این فاصلهٔ زمانی بیشتر از 5ms است.

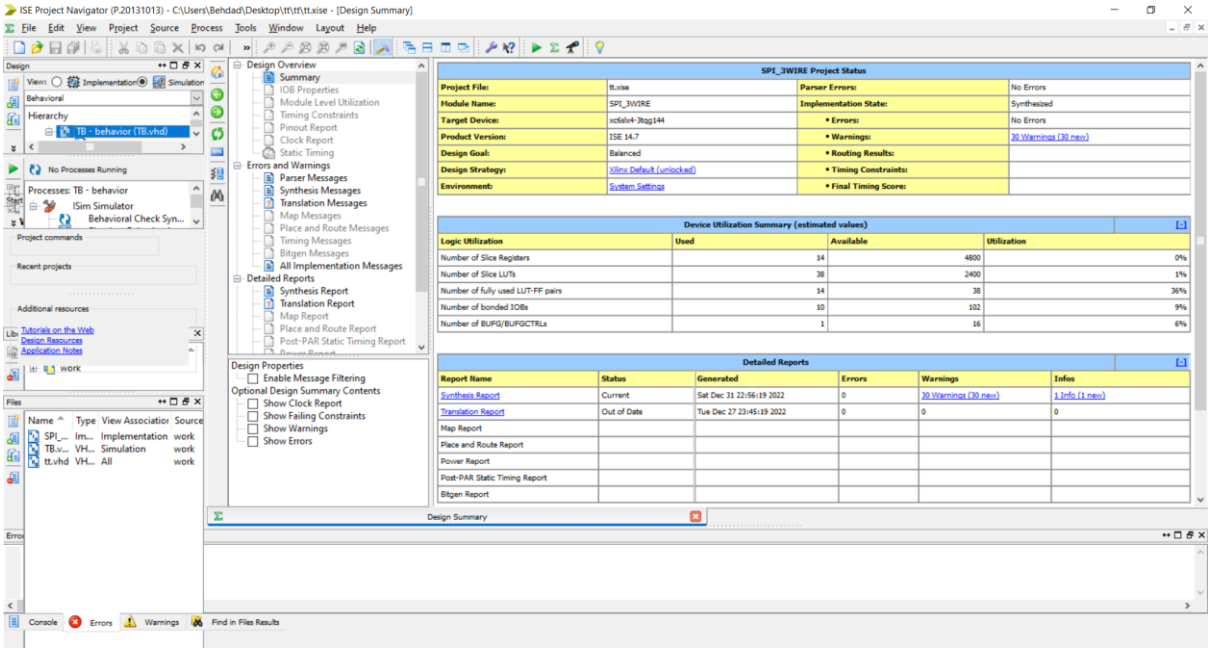
# فصل ۴:

# نرم‌افزار ISE

## ۴-۱- مقدمه

نرم افزار ISE Design Suite محصولی از کمپانی Xilinx است که برای بهینه سازی نیرو و هزینه، از طریق بهره وری طراحی بیشتر، تولید شده است. به کمک نرم‌افزار ISE Design Suite می‌توانید تمام مراحل طراحی و پیاده‌سازی شامل ورود طرح، شبیه‌سازی، سنتز، جانمایی و مسیریابی رو انجام بدهیم. بعد از آن فایل پیکره‌بندی را ایجاد کنیم و FPGA را پروگرام کنیم به کمک نرم‌افزار ISE می‌توانیم انواع تحلیل‌های زمانی و توان مصرفی را برای طرحی که پیاده‌سازی کردیم انجام بدیم. این نرم‌افزار IP Coreها یا کدهای از پیش نوشته شده زیادی را در اختیارتان قرار می‌دهد که می‌توانیم به کمک آنها مدارات بزرگ را سریع‌تر طراحی و تست کنیم.

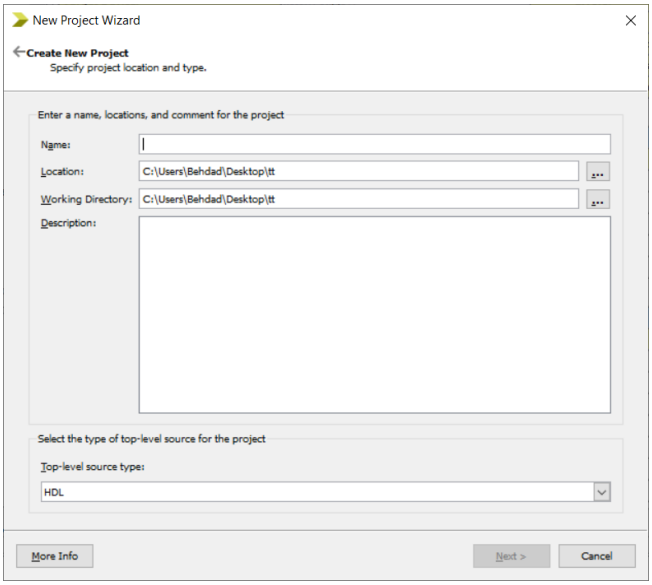
## ۴-۲- آشنایی کوتاه با محیط ISE




شکل (۴-۱) فضای کلی نرم‌افزار ISE

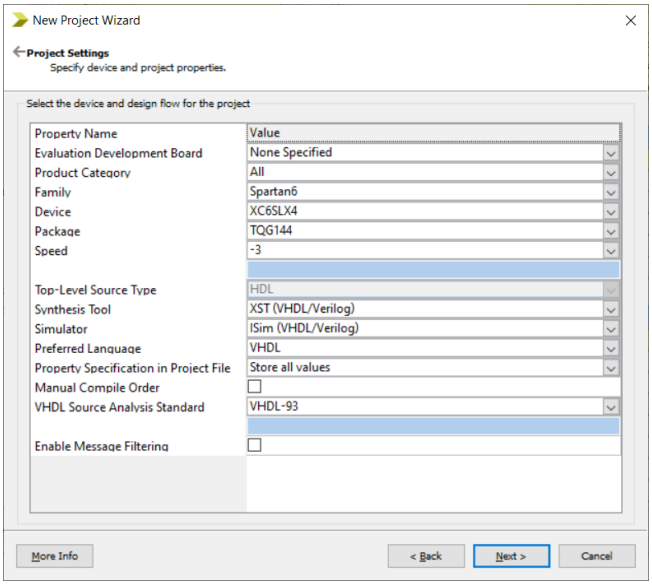
پنجرهٔ Designe summary توضیحات مختصری دربارهٔ پروژهٔ ساخته شده ارائه می‌دهد.

برای ساخت پروژهٔ جدید از مسیر file/new project وارد پنجرهٔ new project wizard می‌شویم.



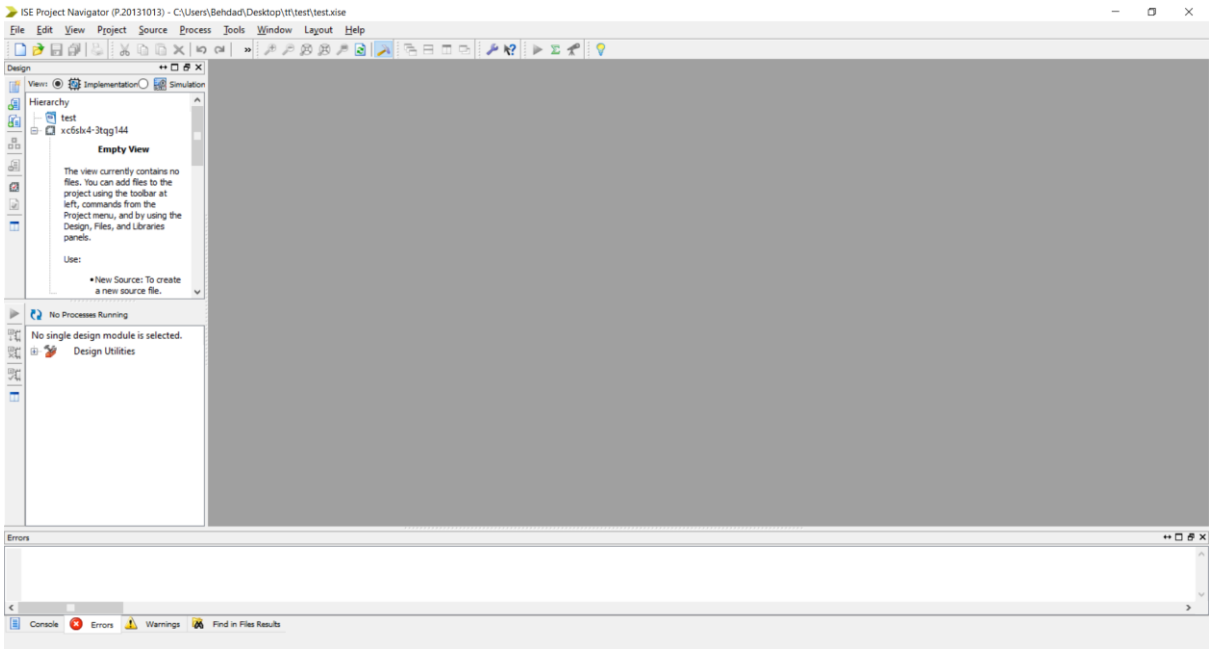

شکل (۴-۲) پنجرهٔ new project wizard

در این پنجره نام پروژه، آدرس ذخیره‌سازی و توضیحات پروژه را مشخص می‌کنیم.



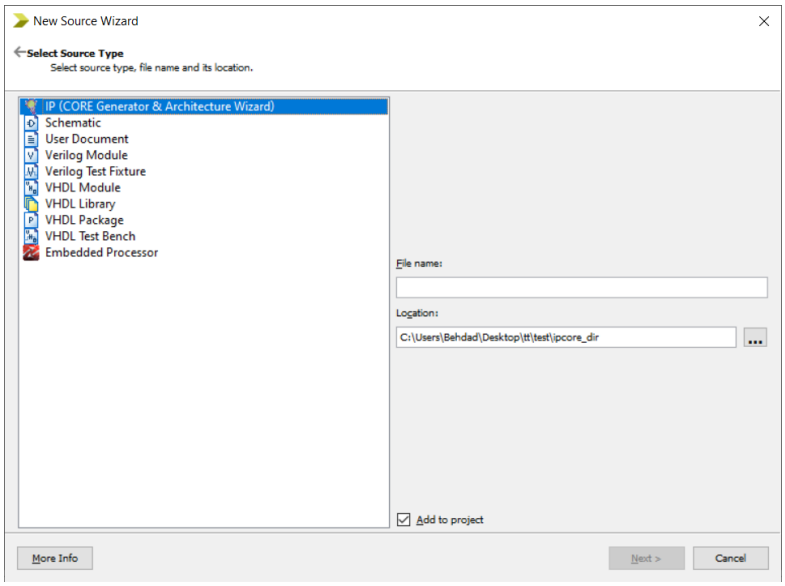

شکل (۴-۳) پنجرهٔ new project wizard

در این پنجره زبان برنامه‌نویسی، مدل و خانوادهٔ FPGA موردنظر و ... را مشخص می‌کنیم.



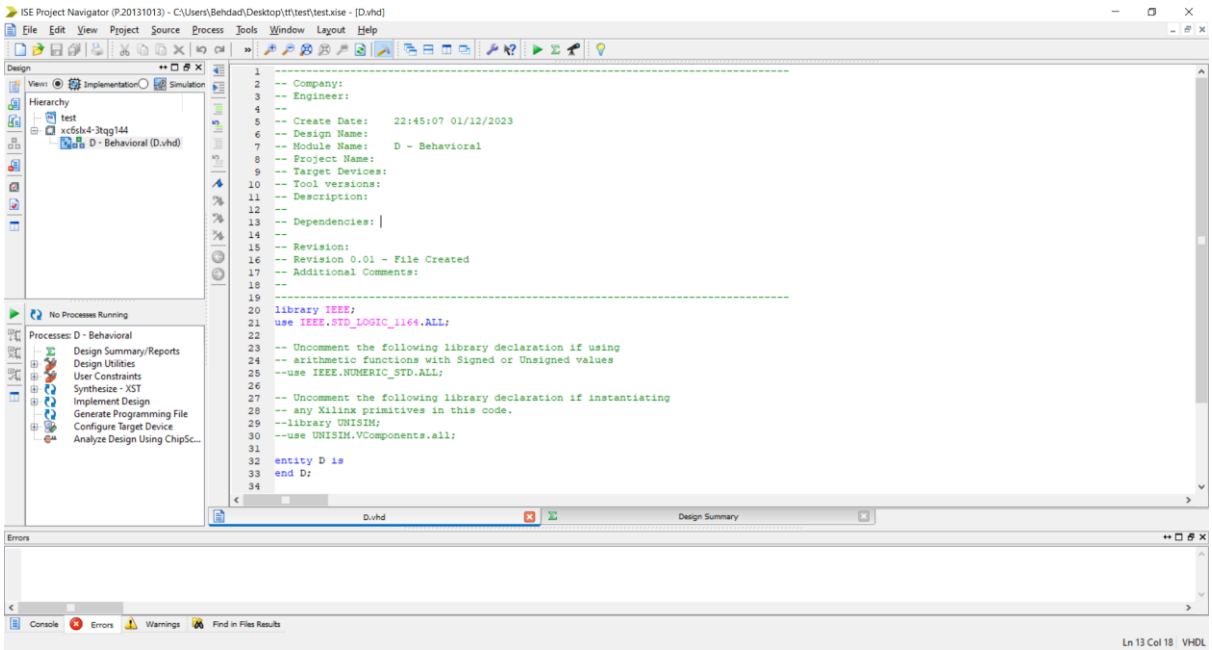

شکل (۴-۴) پنجره اصلی نرم‌افزار ISE

با کلیک راست کردن بر روی قسمت hierarchy و انتخاب گزینهٔ new source وارد پنجره زیر می‌شویم.




شکل (۴-۵) پنجرهٔ new source wizard

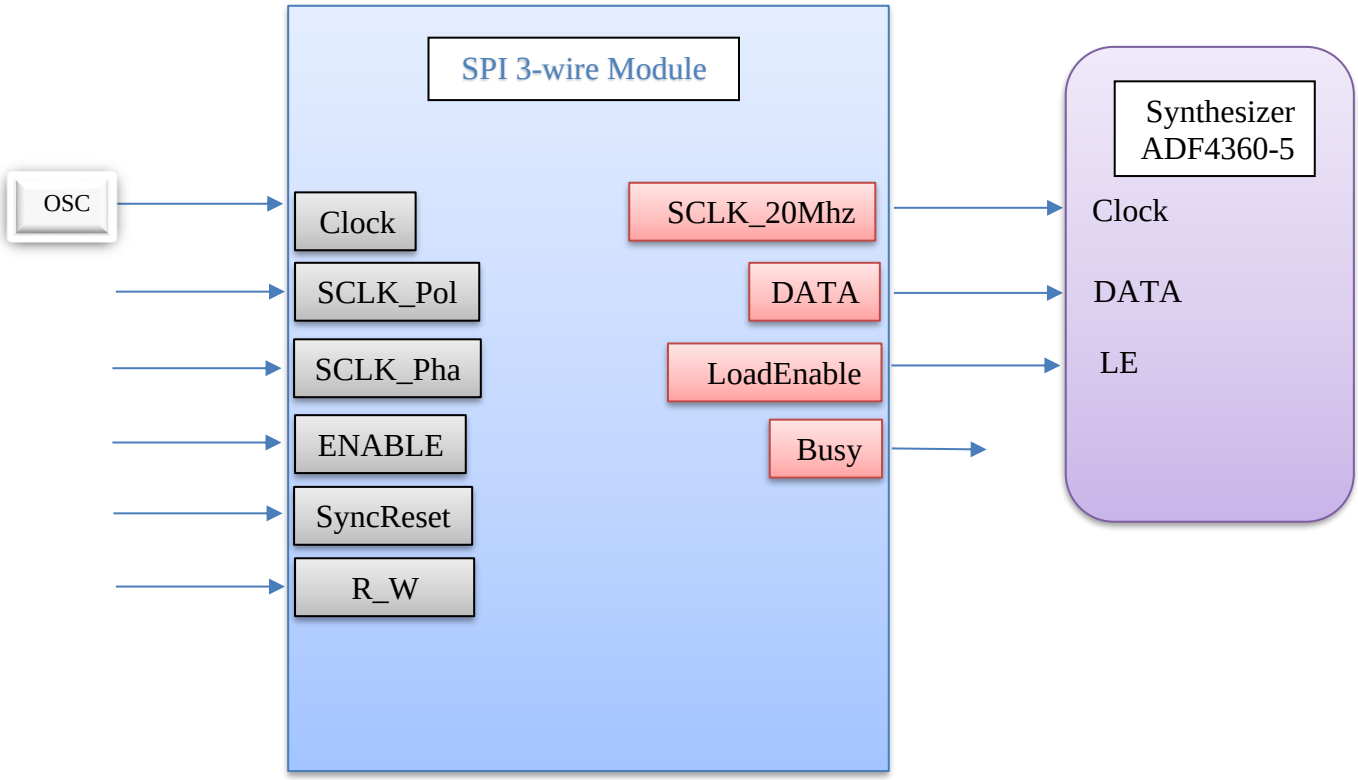
با انتخاب گزینهٔ VHDL Module فایلی با پسوند vhd. نرم‌افزار ایجاد می‌کند که محیط کدنویسی را برای ما فراهم می‌کند.




شکل (۴-۶) محیط کدنویسی به زبان VHDL

## ۴-۳- توصیف و طراحی کلی ماژول SPI به زبان VHDL

در ابتدا به توصیف ماژول SPI طراحی شده می‌پردازیم و هریک از ویژگی‌های آن توضیح می‌دهیم.




شکل (۴-۷) طرح کلی طراحی ماژولSPI 3-wire

همان‌طور که در شکل بالا مشاهده می‌کنید، ماژول SPI طراحی شده، دارای ۶ ورودی و ۴ خروجی است که در جدول زیر عملکرد و وظیفه هرکدام از پورت‌ها توضیح داده شده است.

جدول (۴-۱) توضیح عملکرد پورت‌های ماژول SPI

| نام پورت | نوع پورت | توضیحات |
|---|---|---|
| Clock | ورودی | پایه ورودی کلاک ماژول است که پالس ساعت را از مولد کلاک دریافت می‌کند. |
| SCLK_Pol | ورودی | مقدار اولیه کلاک SPI توسط این پایه مشخص می‌شود. |
| SCLK_Pha | ورودی | فاز کلاک خروجی تولید شده برای پروتکل SPI توسط این پایه مشخص می‌شود. |
| ENABLE | ورودی | برای شروع کار ماژول SPI استفاده می‌شود. اگر این پایه یک منطقی باشد باتوجه‌به شرایط پروتکل، ماژول SPI شروع به کار می‌کند. |
| SyncReset | ورودی | پایه سنکرون با کلاک ریست برای ریست‌کردن ماژول SPI استفاده می‌شود. این پایه active high است. |
| R_W | ورودی | برای انتخاب مُد نوشتن یا خواندن استفاده شده است. اگر یک منطقی باشد از این ماژول برای نوشتن (ارسال داده) و اگر صفر منطقی باشد برای خواندن اطلاعات است. در اینجا فقط برای ارسال داده استفاده می‌شود؛ بنابراین مقدار آن همیشه برابر با یک منطقی در نظر گرفته می‌شود. |
| DATA | خروجی | پورت خروجی داده پروتکل SPI که خروجی در نظر گرفته می‌شود. |
| LoadEnable | خروجی | پورت LE برای انتخاب قطعه جانبی مدنظر که در اینجا ADF4360-5 است. این پایه active low است. |
| Busy | خروجی | پورت خروجی برای نشان‌دادن وضعیت ماژول SPI اگر در حال ارسال داده باشد مقدار این پورت برابر با یک منطقی می‌شود در غیر این صورت صفر منطقی است. |
| SCLK_20MHz_IBUFG | خروجی | پورت خروجی مربوط به کلاک پروتکل SPI است که در اینجا باتوجه‌به دیتاشیت ADF4360-5 فرکانس این کلاک ۲۰ مگاهرتز در نظر گرفته شده است. |

## ۴-۳-۲- پیاده‌سازی اصولی کلاک با استفاده از DCM

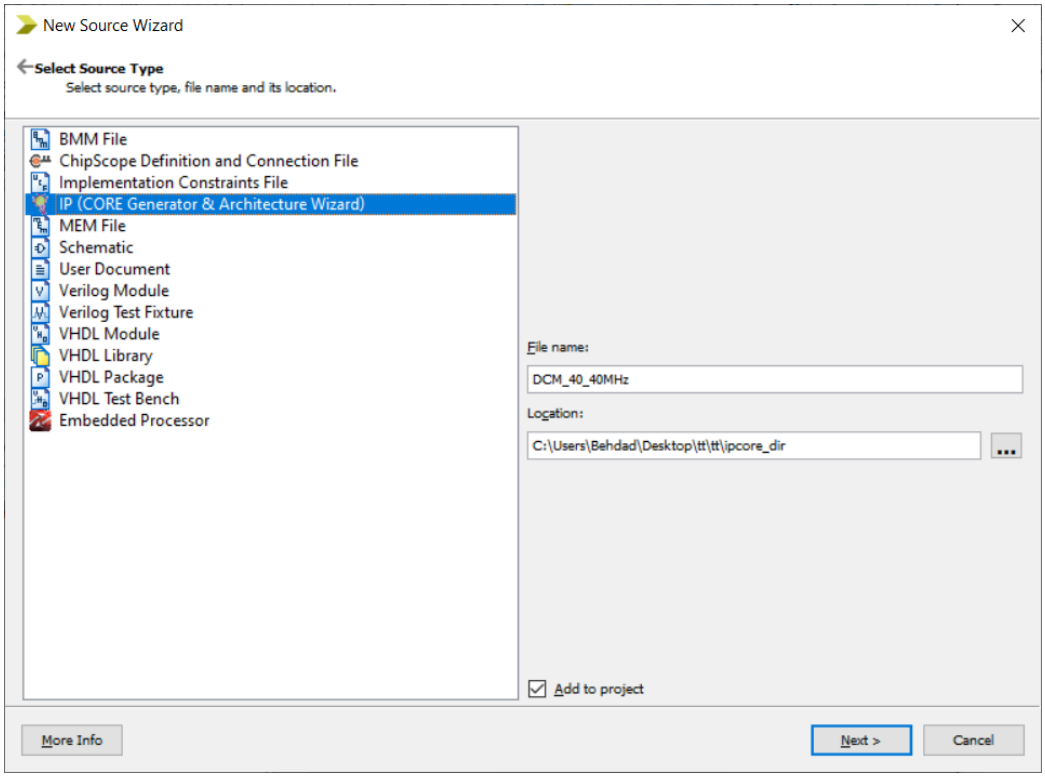
DCM بلوکی است سخت‌افزاری که در FPGAهای شرکت Xilinx تعبیه شده است. با استفاده از این کلاک می‌توانیم فرکانس‌های کلاک جدیدی را از کلاک اصلی مدار، تولید کنیم. فرض کنیم کلاک اسیلاتور روی بورد برابر با ۱۰۰ مگاهرتز است، اما فرکانس کلاک موردنیاز برای ما، ۲۰ مگاهرتز است؛ ازاین‌رو بهتر است برای تولید کلاک ۲۰ مگاهرتز از بلوک سخت‌افزاری DCM استفاده کنیم. هر FPGA دارای تعدادی از بلوک‌های DCM است، هر بلوک DCM ورودی‌ای به‌عنوان کلاک اصلی دارد و خروجی آن کلاک موردنظر ما برای استفاده از درون FPGA است.

نکته‌ای که باید به آن توجه داشت مزیت بلوک DCM تنها تغییر فرکانس کلاک نیست؛ بلکه از این بلوک می‌توان برای کاهش جیتر[1] کلاک استفاده کرد. به همین علت توصیه می‌شود برای تولید کلاک از بلوک DCM استفاده کرد، هرچند که نیازی به تغییر فرکانس در کلاک نباشد. در این پروژه فرض بر این است که کلاک اصلی مدار (کلاک اسیلاتور) برابر است با ۴۰ مگاهرتز و ورودی کلاک ماژول SPI مدنظر ما هم برابر با ۴۰ مگاهرتز است که برای تولید ۴۰ مگاهرتز از بلوک DCM استفاده شده است. بلوک DCM تمام بافرهای مخصوص کلاک که لازم است در مدار داشته باشند، به طور خودکار در مدار قرار می‌دهد، بنابراین نیازی به تعریف بافر نیست.

برای استفاده از DCM می‌بایست از IP Core ها استفاده کرد.

برای استفاده از IPCoreها ابتدا بر روی محیط hierarchy کلیک راست کرده، سپس گزینهٔ New Source را انتخاب می‌کنیم بعد از آن از پنجرهٔ New Source Wizard گزینهٔ IP را انتخاب می‌کنیم و سپس یک نام برای این IP انتخاب می‌کنیم. شکل زیر بیانگر مراحل بالا است.

[1] jitter

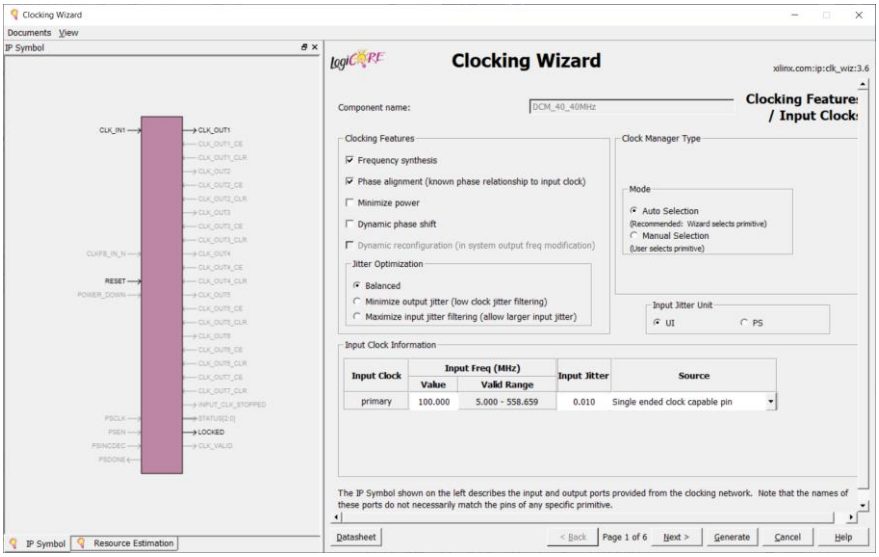



شکل (۴-۸) پنجرهٔ مربوط به IP Core

سپس گزینهٔ Next را می‌زنیم، در پنجرهٔ باز شده مراحل زیر را طی می‌کنیم.

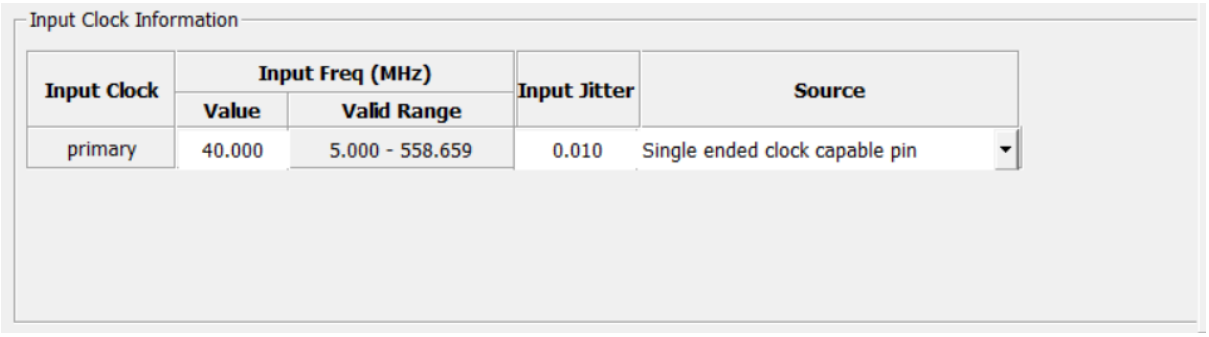
FPGA Features and Design > Clocking > Clocking Wizard ، و سپس گزینهٔ Next را می‌زنیم.

پس از ساخت IPCore مدنظر، پنجرهٔ مربوط به تنظیمات Clocking Wizard باز می‌شود.




شکل (۴-۹) پنجره Clocking Wizard

در این پنجره تنظیمات مختلفی وجود دارد که مربوط به تولید است. برای مثال frequency synthesis برای تولید فرکانس جدید ، Minimize Power برای کاهش توان مصـــرفی ، phase alignment برای تعین فاز کلاک استفاده می‌شود. در قسمت Input Clock Information فرکانس کلاک ورودی را انتخاب می‌کنیم در اینجا فرکانس ورودی 40 مگاهرتز در نظر گرفته شـده اسـت. برای تعیین جیتر کلاک می‌توان به دو صـورت عمل کرد حالت اول UI و حالت دوم PS یا مخفف pico Second اســت که می‌توان زمان جیتر را وارد کرد. در اینجا حالت پیش‌فرض نرم‌افزار در نظر گرفته شـــده اســـت. در قســـمت Source، منبع کلاک DCM را انتخاب می‌کنیم که ۴ حالت مختلف دارد. اگر منبع ورودی به‌صورت تک‌ورودی در نظر گرفته شـده باشـد از گزینهٔ single ended clock capable pin ، اگر به‌صـورت تفاضـلی وارد شـده باشـد از گزینهٔ Differential clock capable pin ، اگر ورودی از خارج FPGA وارد نمی‌شـود و از داخل FPGA تأمین می‌شـود و قبل از اینکه به ورودی DCM وارد شود، از یک بافر عبور داده شده است از گزینه BufG استفاده می‌کنیم و اگر در این حالت از بافر اســتفاده نشــده باشــد از گزینه No Buffer اســتفاده می‌کنیم. در اینجا از گزینهٔ Single ended clock capable pin استفاده می‌کنیم.



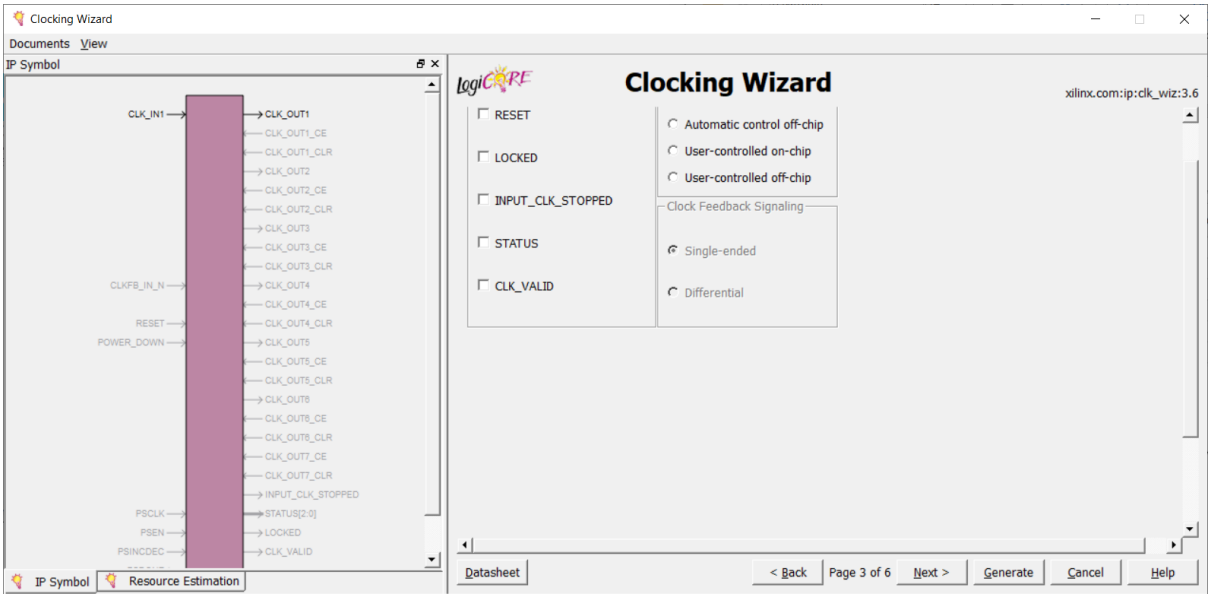

شکل (۴-۱۰) مشخصات کلاک ورودی

در پنجرهٔ بعدی مشـخصـات کلاک خروجی را در نظر می‌گیریم. ابتدا مقدار فرکانس خروجی را وارد می‌کنیم. در اینجا ۴۰ مگاهرتز وارد شده است، فاز کلاک خروجی را صفر در نظر می‌گیریم. Duty Cycle پالس را هم برابر با 50 درصد در نظر می‌گیریم.

| Output Clock | Output Freq (MHz) | | Phase (degrees) | | Duty Cycle (%) | | Drives | Use Fine Ps |
|---|---|---|---|---|---|---|---|---|
| | Requested | Actual | Requested | Actual | Requested | Actual | | |
| CLK_OUT1 | 40.000 | 40.000 | 0.000 | 0.000 | 50.000 | 50.0 | BUFG | ☐ |
| ☐ CLK_OUT2 | 100.000 | N/A | 0.000 | N/A | 50.000 | N/A | BUFG | ☐ |
| ☐ CLK_OUT3 | 100.000 | N/A | 0.000 | N/A | 50.000 | N/A | BUFG | ☐ |
| ☐ CLK_OUT4 | 100.000 | N/A | 0.000 | N/A | 50.000 | N/A | BUFG | ☐ |
| ☐ CLK_OUT5 | 100.000 | N/A | 0.000 | N/A | 50.000 | N/A | BUFG | ☐ |
| ☐ CLK_OUT6 | 100.000 | N/A | 0.000 | N/A | 50.000 | N/A | BUFG | ☐ |

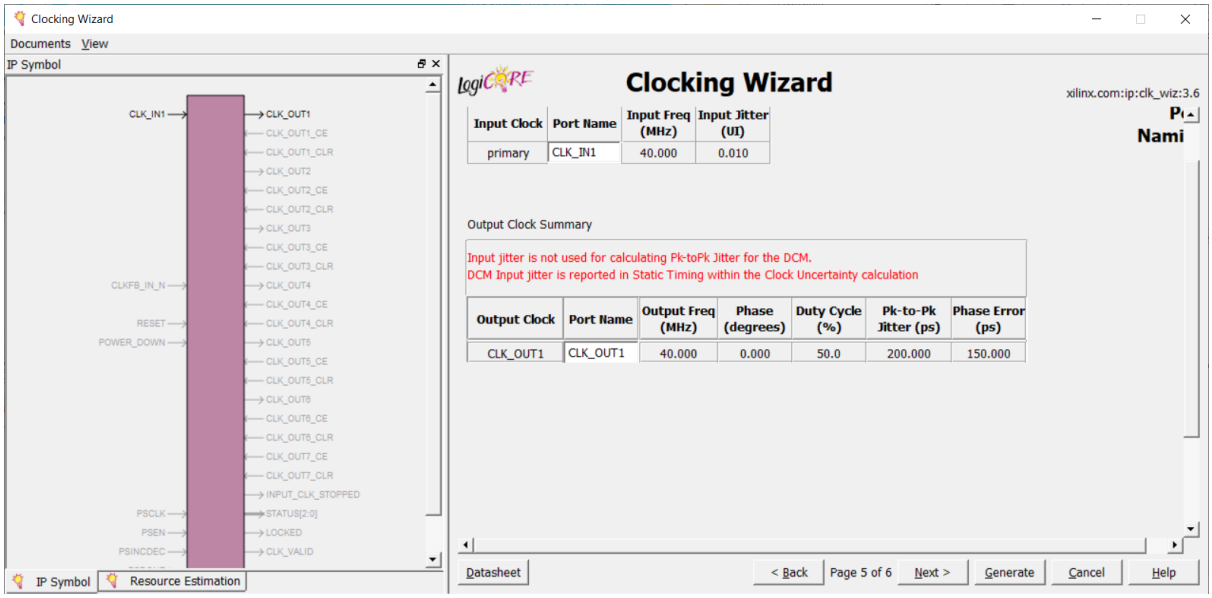
شکل (۴-۱۱) پنجره مشخصات کلاک خروجی

پنجرهٔ بعدی مربوط است به سیگنال‌های کنترلی بلوک DCM که در این جا هیچ‌کدام از این سیگنال‌ها را فعال نمی‌کنیم.




شکل (۴-۱۲) پنجره سیگنال‌های کنترلی بلوک DCM

در پنجرهٔ بعدی نام پورت‌های ورودی و خروجی بلوک DCM را می‌توانیم تغییر دهیم. در اینجا مقادیر پیش‌فرض در نظر گرفته شده است.




شکل (۴-۱۳) پنجره مربوط به نام پورت‌های بلوک DCM

بعدازاین پنجره بر روی گزینهٔ FINISH کلیک می‌کنیم و تنظیمات بلوک DCM به پایان می‌رسد.

بعد از ساخته‌شدن بلوک DCM از گزینهٔ view HDL instantiation مراحل مربوط به نمونه‌سازی آن در کد اصلی را انجام می‌دهیم.

کدهای مربوط به تعریف بلوک DCM:

```
component DCM_40_40MHz
port
 (-- Clock in ports
  CLK_IN1           : in     std_logic;
  -- Clock out ports
  CLK_OUT1          : out    std_logic
 );
end component;
```

کدهای بالا را در قسمت decleration معماری تعریف می‌کنیم ، سپس عملیات Port Map را انجام می‌دهیم.

ورودی بلوک DCM باید پورت کلاک باشد و خروجی آن سیگنال میانی ای که کلاک ۴۰ مگاهرتز تولیدی DCM است؛ بنابراین کلاکی برای ماژول SPI از آن استفاده می‌کنیم همین سیگنال میانی است.

کدهای مربوط به عملیات Port Map:

```
DCM_BLOCK : DCM_40_40MHz
port map
        (-- Clock in ports
        CLK_IN1  => Clock,
        -- Clock out ports
        CLK_OUT1 => Clock_40MHz);
```

بنابراین، کلاکی که برای ماژول SPI از آن استفاده می‌کنیم CLOCK_40MHz است. نکته‌ای که باید به

## ۴-۳-۳- بررسی ماژول SPI در نرم‌افزار ISP

### ۴-۳-۳-۱- تعریف کتابخانه

در این ماژول از کتابخانه‌های استاندار مربوط به IEEE استفاده شده است. کتابخانهٔ UNSIM برای استفاده از اعداد signed و unsigned و همچنین استفاده از IBUFG است.

کدهای مربوط به تعریف کتابخانه:

```
--DEFINE LIB--
library IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;
--For IBUFG--
LIBRARY UNISIM;
use UNISIM.vcomponents.all;
--
```

### ۴-۳-۳-۲- تعریف موجودیت

کدهای مربوط به موجودیت ماژول SPI :

```
entity SPI_3WIRE IS
    --define generic parameters
    GENERIC(
        DataWidth_CMD : INTEGER := 24 ;     -- DATA width for transmit
        SlavesNum     : INTEGER := 1        --Slaves Number
    );

    --define inputs and outputs
    PORT(
        --INPUT--
        CLOCK           : IN   STD_LOGIC; --Module clock
        SCLK_Pol        : IN   STD_LOGIC; --initiate SPI CLOCK polarity
        SCLK_Pha        : IN   STD_LOGIC; --SPI Clock Phase

        Enable          : IN   STD_LOGIC; --enable SPI Module
        SyncReset       : IN   STD_LOGIC; --synchronous  RESET for SPI Module
        R_W             : IN   STD_LOGIC; --Read / wirte // this is optional. in this module
                                          --we use R_W just for write. according to datasheet

        --OUTPUT--
        DATA             : OUT  STD_LOGIC := 'Z';                        --SPI DATA Line according to datasheet
                                                                         --this pin is JUST INPUT
        LoadEnable       : OUT  STD_LOGIC_VECTOR(SlavesNum - 1 downto 0); --SPI ENABLE SLaves
        Busy             : OUT  STD_LOGIC;                               --optional: show SPI module Status
        SCLK_20MHz_IBUFG : OUT  STD_LOGIC                                --spi Clock

    );
end SPI_3WIRE;
```

در کدهای بالا، ابتدا موجودیتی به نام SPI_3WIRE تعریف شـــده اســـت و ســـپس در بدنهٔ آن به تعریف پورت‌های موردنظر پرداخته‌ایم.

در بخش generic ، متغیر DataWidth_CMD برای مشخص‌کردن تعداد داده‌های ارسالی تعریف شده است. متغیر SlavesNum برای مشخص‌کردن تعداد synthesizerهای متصل شده به FPGA تعریف شده است، در اینجا تعداد برابر با یک در نظر گرفته شده است.

در بخش Port به تعریف پورت‌ها پرداخته شده است. در جدول زیر توضیحات مربوط به هر پایه نوشته شده است.

جدول (۴-۲) مشخصات پورت‌های ماژول SPI

| نام پورت | جهت | نوع | توضیحات |
|---|---|---|---|
| Clock | ورودی | STD_LOGIC | کلاک اصلی ماژول SPI |
| SCLK_Pol | ورودی | STD_LOGIC | تعیین پلاریتهٔ کلاک |
| SCLK_pha | ورودی | STD_LOGIC | تعیین فاز کلاک |
| Enable | ورودی | STD_LOGIC | تعیین وضعیت فعال و غیرفعال بودن ماژول SPI .<br>فعال: ۱ غیرفعال: ۰ |
| SyncReset | ورودی | STD_LOGIC | ریست سنکرون با کلاک<br>فعال: ۱ غیرفعال: ۰ |
| R_W | ورودی | STD_LOGIC | تعیین حالت خواندن یا نوشتن. در اینجا تنها از حالت نوشتن استفاده می‌شود.<br>نوشتن: ۱ خواندن: ۰ |
| Data | خروجی | STD_LOGIC | پایه ارسال داده ماژول SPI |
| LoadEnable | خروجی | STD_LOGIC_VECTOR | پایه Load Enable برای انتخاب Synthesizer موردنظر |
| Busy | خروجی | STD_LOGIC | پایه‌ای برای نمایش وضعیت ماژول SPI |
| SCLK_20MHz_IBUFG | خروجی | STD_LOGIC | کلاک خروجی ماژول SPI |

### ۴-۳-۳-۳- تعریف معماری

ابتدا معماری‌ای با نام Behaviroal مربوط به موجودیت SPI_3WIRE تعریف می‌کنیم.

```
architecture behaviroal of SPI_3WIRE IS
```

سپس به تعریف سیگنال‌های میانی می‌پردازیم. در اینجا به‌ازای هر پورت تعریف شده در موجودیت، یک سیگنال میانی تعریف شده است. علت این کار را در ادامه به آن می‌پردازیم.

تعریف سیگنال SCLK برای تولید کلاک SPI :

```
signal SCLK                : STD_LOGIC                    :='0';
```

تعریف سیگنال‌های میانی:

```
--define INTERNAL Signal for all PORTS except CLOCK PORT(REGISTER PORT)
signal Enable_BUF       : STD_LOGIC                := '1';       --active low enable
signal SyncReset_BUF    : STD_LOGIC                := '1';       --active low SyncReset
signal R_W_BUF          : STD_LOGIC                := '1';       --R_W = '1' -> for write and R_W = '0' -> for read
signal LoadEnable_BUF   : STD_LOGIC_VECTOR(SlavesNum - 1 downto 0)   := (OTHERS => '1'); --LOAD ENABLE BUFFER
Signal DATA_BUF         : STD_LOGIC := 'Z';
signal Busy_BUF         : STD_LOGIC := '0';                                                --BUSY BUFFER
--internal signal
signal DATA_CMD         : STD_LOGIC_VECTOR(DataWidth_CMD -1 downto 0) := (OTHERS => '1');--DATA to transmit
signal Count            : unsigned(1 downto 0)                        := "00";
signal ADR              : INTEGER                                     := 0;
signal SCLK_Pol_BUF     : STD_LOGIC                                    := '0';
signal SCLK_Pha_BUF     : STD_LOGIC                                   := '0';
signal Counter          : INTEGER                                     := 23;
signal Clock_40MHz      : STD_LOGIC                                    := '0';
```

تعریف تایپ‌های موردنیاز:

```
--type
type FSM is (Idle,WriteStatus,InitDelay,FinalDelay,FinalState,CtrlDelay); --FSM State
signal State : FSM                                                                  := Idle;
type Registers IS (R , C , N);                                          --Synthesizer Register
signal que   : Registers                                                            := R;
```

در این قسمت دو نوع تایپ تعریف شده است. تایپ FSM که به ماشین حالت مربوط می‌شود تایپ دوم مربوط می‌شود به رجیسترهای داخلی ADF4360-5 که در اینجا R , C و N به ترتیب مخفف رجیستر ControlLatch ، R Latch و N Latch است. در ادامه از هر کدام سیگنالی با این تایپ ها تعریف کرده ایم.

تعریف رجیسترهای ADF4360-5 :

```
--Registers
Constant R_CounterlLatch : STD_LOGIC_VECTOR(23 downto 0):= "110111000011101010111100";   --ADF4360-5 REG ->
R Counter Latch
Constant ControlLatch    : STD_LOGIC_VECTOR(23 downto 0):= "101000101110101011110001";     --ADF4360-5 REG
-> Control Latch
Constant N_ControlLatch  : STD_LOGIC_VECTOR(23 downto 0):= "101011110101011110001010";     --ADF4360-5 REG
-> N Counter Latch
```

برای برنامه‌ریزی رجیسترهای ADF4360-5، هرکدام از این رجیسترها را به‌صورت عدد ثابت constant تعریف کرده‌ایم. هرکدام از این رجیسترها از نوع STD_LOGIC_VECTOR ؛ ۲۴ بیتی و دارای مقدار اولیه‌ای دلخواه هستند. به دلیل اینکه این پروژه به‌صورت عملی بر روی بورد پیاده‌سازی نمی‌شود؛ بنابراین مقدار اولیهٔ این رجیسترها برای ما اهمیتی ندارد و تنها برای نشان‌دادن درستی ارسال این داده‌ها به آن مقدار داده شده است.

تعریف بلوک DCM:

```
component DCM_40_40MHz
port
       (-- Clock in ports
       CLK_IN1           : in     std_logic;
       -- Clock out ports
       CLK_OUT1          : out    std_logic
       );
end component;
--
```

همان‌طور که گفته شد برای کلاک ورودی مدار از بلوک DCM استفاده شده است. این بلوک DCM توسط IPCoreها ساخته شده است که در بخش‌های قبلی به توضیح آن پرداخته‌ایم.

بلوک DCM آخرین قسمتی بود که در decleration معماری behavioral تعریف کردیم. بعد از آن وارد بدنهٔ اصلی معماری می‌شویم. بعد از کلمهٔ begin معماری به تعریف واحدهای دیگر می‌پردازیم.

نمونه‌سازی از IBUFG:

```
--INSTANT IBUFG for SPI Clock--
IBUG_inst : IBUF
generic map(
        IBUF_LOW_PWR => TRUE, --low Power
        IOSTANDARD => "DEFAULT")
port map (
        O => SCLK_20MHz_IBUFG,
        I => SCLK
);
```

در این ماژول برای ساخت کلاک SPI از سیگنالی با نام SCLK استفاده کردیم. برای اینکه ویژگی‌هایی بر روی این سیگنال صورت بگیرد تا بتوان از آن به‌عنوان کلاک استفاده کرد بهتر است که از ویژگی IBUFG استفاده کرد. ورودی این نمونه را SCLK قرار می‌دهیم و خروجی آن را SCLK_MHz_IBUFG قرار می‌دهیم. بنابراین کلاک SPI، پایه خروجی SCLK_MHz_IBUFG است.

نمونه‌سازی بلوک DCM:

```
--DCM Block --
DCM_BLOCK : DCM_40_40MHz
port map
        (-- Clock in ports
        CLK_IN1  => Clock,
        -- Clock out ports
        CLK_OUT1 => Clock_40MHz);
```

ورودی این بلوک را Clock قرار می‌دهیم و خروجی آن که به‌عنوان کلاک از آن استفاده می‌کنیم سیگنالی با نام Clock_40MHz استفاده می‌کنیم.

بنابراین، سیگنال کلاک اصلی ما در اینجا Clock_40Mhz است.

- توجه: برای اینکه بتوانیم در شبیه‌سازی نتایج بهتری داشته باشیم، بهتر است که از سیگنال کلاک استفاده کنیم. در اینجا تنها به این نکته اشاره شده است که در واقعیت بهتر است که برای تولید کلاک از بلوک DCM استفاده کنیم.

عملیات رجیستر:

```
Enable_BUF      <= Enable;
R_W_BUF         <= R_W;
SyncReset_BUF   <= SyncReset;
Data           <= Data_BUF;
Busy           <= Busy_BUF;
LoadEnable(ADR)<= LoadEnable_BUF(ADR);
SCLK_Pol_BUF    <= SCLK_Pol;
```

هرکدام از پورت‌های ورودی و خروجی را به جزء پورت کلاک به سیگنال میانی مربوطه متصل می‌کنیم. به این کار رجیستر کردن پورت‌ها می‌گویند. این عمل باعث می‌شود که FPGA نزدیک‌ترین و سریع‌ترین مسیر را برای هرکدام از این پورت‌ها انتخاب کند. تمام دستورات بالا در محیط concurrent معماری صورت می‌گیرد. بعد از آن به سراغ process می‌رویم.

توضیحات مربوطه به process:

در لیست حساسیت process تنها، سیگنال کلاک را می‌نویسیم، تا این process تنها به سیگنال کلاک حساس باشد.

```
process(Clock)
```

سپس به تعریف متغیرهایی که در این process نیاز داریم، می‌پردازیم.

```
variable counter_delay : integer := 0;
variable counter_flag  : boolean := false;
variable CtrlD         : integer := 0;
-Registers Flag
variable R_Latch       : boolean := false;
variable C_Latch       : boolean := false;
variable N_Latch       : boolean := false;
```

در ادامه به کاربرد هریک از این متغیرها می‌پردازیم.

بعد تعریف متغیرها و بعد از کلمهٔ begin شرط بالاروندهٔ سیگنال کلاک را با استفاده از دستور زیر چک می‌کنیم.

```
--rising edge Clock
if(Clock'EVENT and Clock = '1') then   --Check CLOCK SYSTEM Rising EDGE
```

سپس با به بررسی سیگنال ریست که یک سیگنال سنکرون با لبهٔ بالارونده کلاک است می‌پردازیم.

```
if(SyncReset_BUF = '1') then
    Busy_BUF     <= '1';   --Status: busy
    DATA_BUF     <= 'Z';   --SPI  DATA
    LoadEnable_BUF(ADR) <= '1';--DONT SELECT Slave
    State        <= Idle; --Reset State
    SCLK         <= SCLK_Pol_BUF; --Set SPIClock Polarity
    Counter      <= 23; --Reset Data Transmision Counter
    que          <= R ;--Reset que to first register
else
```

در بلوک if-else بالا، ابتدا شرط فعال‌بودن ریست چک شده است که در صورت فعال‌بودن تمام پایه‌های ارسال و کنترلی به حالت اولیه بر می‌گردند.

در ادامه به تغییراتی که بعد از فعال‌شدن ریست رخ می‌دهد می‌پردازیم.

Busy_BUF: برای نشان‌دادن وضعیت ماژول SPI است که در اینجا فرض شده است اگر حالت ریست فعال باشد این پایه برابر با یک منطقی شود.

Data_Buf: به پین ارسال داده مربوط می‌شود که فرض کردیم اگر داده‌ای ارسال نشود این پین برابر با ‘z’ یا High-impedance باشد.

LoadEnable_BUF(ADR): اگر برابر با یک باشد یعنی synthesizer ای انتخاب نمی‌شود. ADR آدرس slave مدنظر است که در اینجا به صفر در نظر گرفته شده است.

State: با استفاده از این سیگنال حالت FSM را کنترل می‌کنیم که در اینجا بعد از ریست‌شدن می‌بایست State مقدار حالت اولیه یعنی Idle را به خود بگیرد.

SCLK: مقدار اولیه کلاک را که با استفاده از سیگنال SCLK_Pol_BUF تعیین می‌کنیم به SCLK تخصیص می‌دهیم.

Counter: این متغیر برای کنترل تعداد بیت ارسالی صورت می‌گیرد. در اینجا چون ۲۴ بیت ارسال می‌شود؛ بنابراین مقدار آن را برابر با ۲۳ (از ۰ تا ۲۳) که در مجموع ۲۴ بیت می‌شود قرار می‌دهیم. در این قسمت می‌توانستیم از متغیر DataWidth_CMD نیز استفاده کنیم تا کدنویسی حالت عمومی‌تر پیدا کند.

بعد از حالت ریست که اولویت بالاتری داشت به بیان حالت‌های مختلف ارسال می‌پردازیم. در اینجا برای ارسال از FSM یا ماشین حالت استفاده شده است که ۶ حالت مختلف دارد که به توضیح هرکدام می‌پردازیم.

حالت اولیه این ماشین برابر Idle است مقدار متغیر حالت یعنی State هم در ابتدا این حالت را دارد. در این حالت چون داده‌ای ارسال نمی‌شود؛ بنابراین فرض کردیم که پایه Data به‌صورت High-impedance است. Busy_BUF برابر با صفر است؛ یعنی ماژول SPI آزاد است و داده‌ای ارسال نمی‌کند. مقدار Counter برابر با ۲۳ قرار می‌گیرد به این دلیل که در ابتدا هیچ داده‌ای ارسال نشده است و تازه ماژول در حالت شروع به کار است. سپس با بلوک if-else به چک‌کردن پایهٔ Enable می‌پردازیم. اگر Enable برابر با ۱ بود آنگاه حالت State به InitDelay تغییر می‌کند، Busy_Buf ۱ می‌شود به معنای مشغول بودن، LoadEnable برابر با صفر می‌شود به معنای انتخاب slave موردنظر و در آخر فاز کلاک در بافر آن قرار می‌گیرد.

اگر پایهٔ Enable فعال نبود حالت State همان idle باقی می‌ماند و LoadEnable برابر با ۱ می‌شود تا salve مدنظر انتخاب نشود.

توضیحات بالا مربوط به بلوک زیر است.

```
case State is
  --IDLE State--
  when Idle =>
    Data_BUF <= 'Z';                    --In IDLE State -> DATA <= '0'
    Busy_BUF <= '0';                    --In IDLE State The SPI module Is not Busy
    Counter  <= 23;                     --Reset Data Transmision Counter
    if(Enable_Buf = '1')then            --In IDLE State Check Enable Pin
      State               <= InitDelay; --move to InitDelay
      Busy_BUF            <= '1';
      LoadEnable_BUF(ADR) <= '0';      --
      SCLK_Pha_BUF        <= SCLK_Pha; --Set SPI Clock phase to Clock phase buffer
    else
      State          <= Idle;            --stay in Idle State
      LoadEnable_Buf <= (others => '1'); --deactive slaves load enable Pins
    end if;
```

حال به بیان حالت بعدی یعنی InitDelay می‌پردازیم.

این حالت در اصل برای رعایت Setup time و Hold time است. یعنی بعد از صفرشدن LoadEnable طبق دیتاشیت باید زمانی بعد دیتا شروع به ارسال کند؛ بنابراین این زمان را با استفاده از این حالت بررسی کرده‌ایم. کد زیر بیانگر این حالت است.

```
--Init Delay--
when InitDelay =>
LoadEnable_BUF(ADR) <= '0';          --Select Target Slave
State               <= WriteStatus; --move to WriteStatus
Busy_BUF            <= '1';          --In Init State The SPI module Is  Busy
--SEND BLOCK--             SEND according to Register Flag
case que is
  when R => Data_BUF <= R_CounterlLatch(Counter); --send  R Latch DATA
  when C => Data_BUF <= ControlLatch(Counter);    --send  C Latch DATA
  when N => Data_BUF <= N_ControlLatch(Counter);  --send  C Latch DATA
end case;
--
Counter <= Counter - 1;
```

همان‌طور که مشاهده می‌شود حالت بعدی WriteStatus است که به متغیر State تخصیص‌داده‌شده است.

نکته‌ای که باید به آن توجه داشت ترتیب ارسال داده و برنامه‌ریزی رجیسترها است. باتوجه‌به دیتاشیت و مطالب بیان شده ترتیب پیشنهادی برای برنامه‌ریزی به‌صورت زیر است.

R Latch

Control Latch

N latch

یعنی ابتدا R Latch ، سپس Control Latch و در اخر هم N latch برنامه‌ریزی شود.

این ترتیب به‌صورت متغیرهایی از جنس Boolean کنترل شده است. que متغیر کنترلی از جنس Boolean است که سه مقدار کلی می‌تواند داشته باشد. چون در حالت قبل یعنی Idle متغیر que برابر با R بود؛ بنابراین در یک کلاک بعد از آن یعنی حالت InitDelay در بلوک case که مربوط به ارسال داده می‌شود ابتدا بیت‌های مربوط به R Latch ارسال می‌شود.

نکته‌ای که باید به آن توجه کنیم این است که تمامی مقادیر سیگنال‌هایی که در Process مقدار جدید برای آن‌های تخصیص داده می‌شود در یک کلاک بعد مقدار سیگنال تغییر می‌کند. (پایان Process)

بنابراین، اولین بیت هر رجیستر در حالت InitDelay ارسال می‌شود؛ چون در یک کلاک بعد یعنی زمانی که

State در حالت WriteStatus است بیت MSB بر روی Data قرار بگیرد.

حال به بیان مهم‌ترین حالت یعنی WriteStatus می‌پردازیم.

```
--Write State--
when WriteStatus =>
if(Enable_BUF = '1' AND R_W_BUF = '1' ) THEN --check enable and R_W Pins
  SCLK         <= NOT SCLK;                  --generate SPI CLOCK
  SCLK_Pha_BUF <= NOT SCLK_Pha_BUF;          --SET SPI CLOCK phase
end if;
if(Counter >= 0 AND SCLK_Pha_BUF = '1' AND counter_flag = false) then --Check clo
--SEND BLOCK--                              --SEND according to Register Flag
  case que is
    when R => Data_BUF <= R_CounterlLatch(Counter); --send  R Latch DATA
    when C => Data_BUF <= ControlLatch(Counter);    --send  C Latch DATA
    when N => Data_BUF <= N_ControlLatch(Counter);  --send  C Latch DATA
  end case;
  --
  Counter <= Counter - 1; --
elsif (Counter = 0) then  --that means SEND DATA finished
  counter_flag := true;
  if(counter_delay = 1) then --MAKE DELAY FOR SEND LAST BIT
    --SEND BLOCK--                      SEND according to Register Flag
    case que is
      when R => Data_BUF <= R_CounterlLatch(Counter); --send  R Latch DATA
      when C => Data_BUF <= ControlLatch(Counter);    --send  C Latch DATA
      when N => Data_BUF <= N_ControlLatch(Counter);  --send  C Latch DATA
    end case;
  --
  end if;
  counter_delay := counter_delay + 1;
  if(counter_delay = 5)then
    counter_flag  := false;
    counter_delay := 0;
    State         <= FinalDelay; --Move to FinalDelay State
    if que = C then
      State <= CtrlDelay; --
    end if;
    Counter  <= 23; --SET Counter TO 23
    Data_BUF <= 'Z';
    SCLK     <= '0'; --turn off SPI clock after send Data
  end if;
```

کلاک اصلی ماژول برابر با ۴۰ مگاهرتز است، یعنی ورودی ماژول ۴۰ مگاهرتز اما نکته‌ای که باید به آن توجه داشت این است که کلاک خروجی ماژول یا همان کلاک SPI برابر با ۲۰ مگاهرتز است. کلاک ۲۰ مگاهرتز توسط Toggle کردن سیگنال SCLK ساخته شده است. با هر لبهٔ بالارونده کلاک اصلی، مقدار SCLK معکوس می‌شود. یعنی در هر تناوب کلاک اصلی سیگنال SCLK یک مقدار ثابت دارد؛ بنابراین فرکانس SCLK نصف فرکانس اصلی می‌شود؛ بنابراین سیگنال SCLK برابر با ۲۰ مگاهرتز است.

بعد از چک‌کردن مقدار Enable، فاز SCLK و فلگ شمارش یک بیت از داده مدنظر را ارسال می‌کنیم و از

شمارنده یک واحد کم می‌کنیم. Counter_flag برای کنترل ارسال بیت آخر استفاده شده است که در ادامه توضیح خواهد داده شد.

بلاک SEND ترتیب ارسال داده‌های رجیسترهای مختلف را چک می‌کند که در اینجا مقدار آن برابر است با R یعنی اولین رجیستر در حال برنامه‌ریزی است. نکته‌ای که باید به آن توجه داشت این است گه اگر Counter برابر با صفر شود مقدار دو بیت آخر به دلیل تغییر در حالت state ارسال نمی‌شوند؛ بنابراین باید تأخیری در این حالت ایجاد کرد که علاوه بر عدم متوقف کردن سیگنال اصلی کلاک حالت State هم تغییر نکند.

در اینجا این تغییر با استفاده از متغیر counter_flag و counter_delay صورت گرفته است. زمانی که counter صفر می شود به دلیل تاخیر در ارسال ( تاخیر process) و تغییر در حالت State دو بیت آخر (LSB) ارسال نمی شوند بنابراین حدودا می بایست ۵ کلاک اصلی تاخیر داشته باشیم تا این دو بیت ارسال شوند. بنابراین با صفر شدن counter مقدار counte_flag برابر با true می شود تا دیگر از مقدار counter کاسته نشود و مقدار معتبر بر روی DATA قرار گیرد. زمانی که مقدار Counter_delay برابر با ۱ شده است می بایست بیت آخر ارسال شود که این شرط توسط بلوک if کنترل می شود.همچنین یک واحد به counter_delay اضافه می شود زمانی که counter_delay به عدد ۵ رسید، یعنی تاخیر مد نظر اعمال شد و داده های مورد نظر ارسال شدند اکنون می بایست حالت State تغییر کند؛ نکته ای که باید به آن توجه کرد این است که با توجه به دیتاشیت بعد از برنامه ریزی ControlRegiste ، مدت زمانی به عنوان تاخیر اعمان کنیم و سپس به برنامه ریزی N LATCH بپردازیم. این مدت زمان تاخیر با مقدار خازن CN که در قسمت های قبلی توضیح داده شده است، متفاوت است. در اینجا مقدار تاخیر حدودا 500 میکرو ثانیه در نظر گرفته شده است. بنابراین اگر در حال برنامه ریزی Control latch بودیم یعنی مقدار que برابر با C بود باید حالت بعدی برابر با CtrlDelay باشد. در این بلاک توسط متغیری تاخیر مد نظر ایجاد شده است. اگر que برابر با C نبود به حالت FinalDelay تغییر حالت می دهیم. شکل (۳-۳) بیان گر تاخیر در برنامه ریزی رجیستر ها است. بلوک CtrlDelay:

```
--CtrlDelay --generate Delay with CtrlDelay Variable--
when CtrlDelay =>
CtrlD := CtrlD + 1;
Case CtrlD is
  When 1 =>
    LoadEnable_BUF(ADR) <= '1';
  when 2 =>
    LoadEnable_BUF(ADR) <= '0';
  when 20 =>
    CtrlD := 0;
    State <= FinalDelay;
  when others =>

end case;
```

در این حالت نباید بعد از برنامه‌ریزی Control Latch پایه LoadEnable برابر با یک شـود؛ بنابراین توسـط بلوک case بالا کنترل شده است.

دو حالت بعدی با نام‌های FinalDelay و FinalState حالت‌های پایانی ارسـال هسـتند. برای درنظرگرفتن تأخیرهای مدنظر همچنین مقداردهی به que برای انتخاب رجیستر بعدی از این stateهای استفاده می‌کنیم.

```
when FinalDelay =>
State <= FinalState; --Move to Final State
if que /= C then
  LoadEnable_BUF(ADR) <= '1'; --DESELECT SLAVE
end if;
--Final State--
when FinalState =>
State <= Idle; --Move to Idle State
case que IS
  when R => que <= C;
  when C => que <= N;
  when N => que <= R;
end case;
```

به مدت یک کلاک در حالت FinalDelay تأخیر ایجاد می‌کنیم؛ اگر que برابر با C نبود آنگاه loadEnable باید ۱ شود همان‌طور که در بالا اشاره شد.

در حالت Final state هم توسـط بلوک case ترتیب برنامه‌ریزی را رعایت می‌کنیم و همچنین State را به حالت اولیه مقداردهی می‌کنیم.

## ۴-۳-۴- کدهای ماژول SPI

```
--DEFINE LIB--
library IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;
--For IBUFG--
LIBRARY UNISIM;
use UNISIM.vcomponents.all;
--

entity SPI_3WIRE IS
  --define generic parameters
  GENERIC(
    DataWidth_CMD : INTEGER := 23 ; -- DATA width for transmit
    SlavesNum     : INTEGER := 1    --Slaves Number
  );

  --define inputs and outputs
  PORT(
    --INPUT--
    CLOCK     : IN STD_LOGIC; --Module clock
    SCLK_Pol  : IN STD_LOGIC; --initiate SPI CLOCK polarity
    SCLK_Pha  : IN STD_LOGIC; --SPI Clock Phase
    Enable    : IN STD_LOGIC; --enable SPI Module
    SyncReset : IN STD_LOGIC; --synchronous  RESET for SPI Module
    R_W       : IN STD_LOGIC; --Read / wirte // this optional. in this module
                              --we use R_W just for write. according to
datasheet

    --OUTPUT--
    DATA : OUT STD_LOGIC := 'Z';                                    --SPI
DATA Line according to datasheet
                                                                    --this
pin is JUST INPUT
    LoadEnable       : OUT STD_LOGIC_VECTOR(SlavesNum - 1 downto 0); --SPI
ENABLE SLaves
    Busy             : OUT STD_LOGIC;                               --
optional: show SPI module Status
    SCLK_20MHz_IBUFG : OUT STD_LOGIC                                --spi
Clock
  );
end SPI_3WIRE;
```

```
architecture behaviroal of SPI_3WIRE IS
  signal SCLK : STD_LOGIC := '0';
  --define INTERNAL Signal for all PORTS except CLOCK PORT(REGISTER PORT)
  signal Enable_BUF     : STD_LOGIC                                    := '1';
--active low enable
  signal SyncReset_BUF  : STD_LOGIC                                    := '1';
--active low SyncReset
  signal R_W_BUF        : STD_LOGIC                                    := '1';
--R_W = '1' -> for write and R_W = '0' -> for read
  signal LoadEnable_BUF : STD_LOGIC_VECTOR(SlavesNum - 1 downto 0) := (OTHERS
=> '1'); --LOAD ENABLE BUFFER
  Signal DATA_BUF       : STD_LOGIC                                    := 'Z';
  signal Busy_BUF       : STD_LOGIC                                    := '0'; --
BUSY BUFFER
                                                                         --
internal signal
  signal Count        : unsigned(1 downto 0) := "00";
  signal ADR          : INTEGER              := 0;
  signal SCLK_Pol_BUF : STD_LOGIC            := '0';
  signal SCLK_Pha_BUF : STD_LOGIC            := '0';
  signal Counter      : INTEGER              := 23;
  signal Clock_40MHz  : STD_LOGIC            := '0';
  --type
  type FSM is (Idle,WriteStatus,InitDelay,FinalDelay,FinalState,CtrlDelay); -
-FSM State
  signal State : FSM := Idle;
  type Registers IS (R , C , N); --Synthesizer Register
  signal que : Registers := R;
  --Registers
  Constant R_CounterlLatch : STD_LOGIC_VECTOR(23 downto 0) :=
"110111000011101010111100"; --ADF4360-5 REG -> R Counter Latch
  Constant ControlLatch    : STD_LOGIC_VECTOR(23 downto 0) :=
"101000101110101011110001"; --ADF4360-5 REG -> Control Latch
  Constant N_ControlLatch  : STD_LOGIC_VECTOR(23 downto 0) :=
"101011110101011110001010"; --ADF4360-5 REG -> N Counter Latch

--DCM BLock Decleratin--
  component DCM_40_40MHz
    port
    (-- Clock in ports
      CLK_IN1 : in std_logic;
      -- Clock out ports
      CLK_OUT1 : out std_logic
    );
  end component;
--
begin
```

```
--INSTANT IBUFG for SPI Clock--
  IBUG_inst : IBUF
    generic map(
      IBUF_LOW_PWR => TRUE, --low Power
      IOSTANDARD   => "DEFAULT")
    port map (
      O => SCLK_20MHz_IBUFG,
      I => SCLK
    );
  --  DCM Block --
  DCM_BLOCK : DCM_40_40MHz
    port map
    (-- Clock in ports
      CLK_IN1 => Clock,
      -- Clock out ports
      CLK_OUT1 => Clock_40MHz);

  Enable_BUF      <= Enable;
  R_W_BUF         <= R_W;
  SyncReset_BUF   <= SyncReset;
  Data            <= Data_BUF;
  Busy            <= Busy_BUF;
  LoadEnable(ADR) <= LoadEnable_BUF(ADR);
  SCLK_Pol_BUF    <= SCLK_Pol;

  process(Clock)
    variable counter_delay : integer := 0;
    variable counter_flag  : boolean := false;
    Variable CtrlD         : integer := 0;
    --Registers Flag
    variable R_Latch : boolean := false;
    variable C_Latch : boolean := false;
    variable N_Latch : boolean := false;
  begin
    --rising edge Clock
    if(Clock'EVENT and Clock = '1') then --Check CLOCK SYSTEM Rising EDGE
                                         --check sync. reset -> active high
      if(SyncReset_BUF = '1') then
        Busy_BUF            <= '1';          --Status: busy
        DATA_BUF            <= 'Z';          --SPI  DATA
        LoadEnable_BUF(ADR) <= '1';          --DONT SELECT Slave
        State               <= Idle;         --Reset State
        SCLK                <= SCLK_Pol_BUF; --Set SPI Clock Polarity
        Counter             <= 23;           --Reset Data Transmision Counter
        que                 <= R ;           --Reset que to first register
      else
        case State is
          --IDLE State--
          when Idle =>
            Data_BUF <= 'Z';                    --In IDLE State -> DATA <= '0'
            Busy_BUF <= '0';                    --In IDLE State The SPI module Is not Busy
```

```
     Data_BUF <= 'Z';                    --In IDLE State -> DATA <= '0'
            Busy_BUF <= '0';                    --In IDLE State The SPI module Is not Busy
            Counter  <= 23;                     --Reset Data Transmision Counter
            if(Enable_Buf = '1')then            --In IDLE State Check Enable Pin
              State               <= InitDelay; --move to InitDelay
              Busy_BUF            <= '1';
              LoadEnable_BUF(ADR) <= '0';      --
              SCLK_Pha_BUF        <= SCLK_Pha; --Set SPI Clock phase to Clock phase buffer
            else
              State          <= Idle;            --stay in Idle State
              LoadEnable_Buf <= (others => '1'); --deactive slaves load enable Pins
            end if;

          --Init Delay--
          when InitDelay =>
            LoadEnable_BUF(ADR) <= '0';         --Select Target Slave
            State               <= WriteStatus; --move to WriteStatus
            Busy_BUF            <= '1';         --In Init State The SPI module Is  Busy
                                                --SEND BLOCK--                          SEND
according to Register Flag
            case que is
              when R => Data_BUF <= R_CounterlLatch(Counter); --send  R Latch DATA
              when C => Data_BUF <= ControlLatch(Counter);    --send  C Latch DATA
              when N => Data_BUF <= N_ControlLatch(Counter);  --send  C Latch DATA
            end case;
            --
            Counter <= Counter - 1;
          --Write State--
          when WriteStatus =>
            if(Enable_BUF = '1' AND R_W_BUF = '1' ) THEN --check enable and R_W Pins
              SCLK         <= NOT SCLK;                  --generate SPI CLOCK
              SCLK_Pha_BUF <= NOT SCLK_Pha_BUF;          --SET SPI CLOCK phase
            end if;
            if(Counter >= 0 AND SCLK_Pha_BUF = '1' AND counter_flag = false) then --Check clock
phase and counter
                                                                                --SEND BLOCK--
--SEND according to Register Flag
              case que is
                when R => Data_BUF <= R_CounterlLatch(Counter); --send  R Latch DATA
                when C => Data_BUF <= ControlLatch(Counter);    --send  C Latch DATA
                when N => Data_BUF <= N_ControlLatch(Counter);  --send  C Latch DATA
              end case;
              --
              Counter <= Counter - 1; --
            elsif (Counter = 0) then  --that means SEND DATA finished
              counter_flag := true;
              if(counter_delay = 1) then --MAKE DELAY FOR SEND LAST BIT
                                         --SEND BLOCK--                    SEND according to
Register Flag
                case que is
                  when R => Data_BUF <= R_CounterlLatch(Counter); --send  R Latch DATA
                  when C => Data_BUF <= ControlLatch(Counter);    --send  C Latch DATA
                  when N => Data_BUF <= N_ControlLatch(Counter);  --send  C Latch DATA
```

```
            end case;
            --
            end if;
            counter_delay := counter_delay + 1;
            if(counter_delay = 5)then
              counter_flag  := false;
              counter_delay := 0;
              State         <= FinalDelay; --Move to FinalDelay State
              if que = C then
                State <= CtrlDelay; --
              end if;
              Counter  <= 23; --SET Counter TO 23
              Data_BUF <= 'Z';
              SCLK     <= '0'; --turn off SPI clock after send Data
            end if;
          end if;
        --CtrlDelay --generate Delay with CtrlDelay Variable--
        when CtrlDelay =>
          CtrlD := CtrlD + 1;
          Case CtrlD is
            When 1 =>
              LoadEnable_BUF(ADR) <= '1';
            when 2 =>
              LoadEnable_BUF(ADR) <= '0';
            when 20 =>
              CtrlD := 0;
              State <= FinalDelay;
            when others =>

          end case;
        --Final Delay--
        when FinalDelay =>
          State <= FinalState; --Move to Final State
          if que /= C then
            LoadEnable_BUF(ADR) <= '1'; --DESELECT SLAVE
          end if;
        --Final State--
        when FinalState =>
          State <= Idle; --Move to Idle State
          case que IS
            when R => que <= C;
            when C => que <= N;
            when N => que <= R;
          end case;
      end case;
    end if;
  end if;

end process;

end architecture;
```

# فصل ۵:

# شبیه‌سازی و نتایج

برای شبیه‌سازی از نرم‌افزار ISIM استفاده شده است. ابتدا باید فایل TestBench را به پروژه اضافه کنیم. برای این کار از قسمت hierarchy کلیک راست کرده و سپس از قسمت NewSource فایل تست بنچ را اضافه می‌کنیم. قسمت‌های مهم تست بنچ تعیین فرکانس کلاک و process مقداردهی ورودی‌ها است.

## نمودارها و کدهای پروژه

```
-- Clock process definitions
CLOCK_process :process
begin
 Clock <= '1';
 wait for CLOCK_period/2;
 Clock <= '0';
 wait for CLOCK_period/2;
end process;
```

ثابت CLOCK_period تعیین شده است. باتوجه‌به فرکانس کلاک اصلی که برابر با ۴۰ مگاهرتز است دورهٔ تناوب کلاک را برابر با 25 ns در نظر می‌گیریم.

```
constant CLOCK_period  : time := 25 ns;
```

حال در Process شبیه‌سازی، مقادیر ورودی را مقداردهی می‌کنیم. در شبیه‌سازی ابتدایی فرض می‌کنیم پایهٔ ریست به مدت ۱۰۰ نانوثانیه فعال می‌شود؛ بنابراین در این حالت می‌بایست داده‌ای ارسال نشود و همچنین Busy_BUF باید ۱ شود. بعد از ۱۰۰ نانوثانیه مقدار پورت‌های R_W و Enable ۱ می‌شوند همچنین ریست را غیرفعال می‌کنیم. مقدار و فاز اولیه کلاک راه هم برابر با ۰ در نظر می‌گیریم.

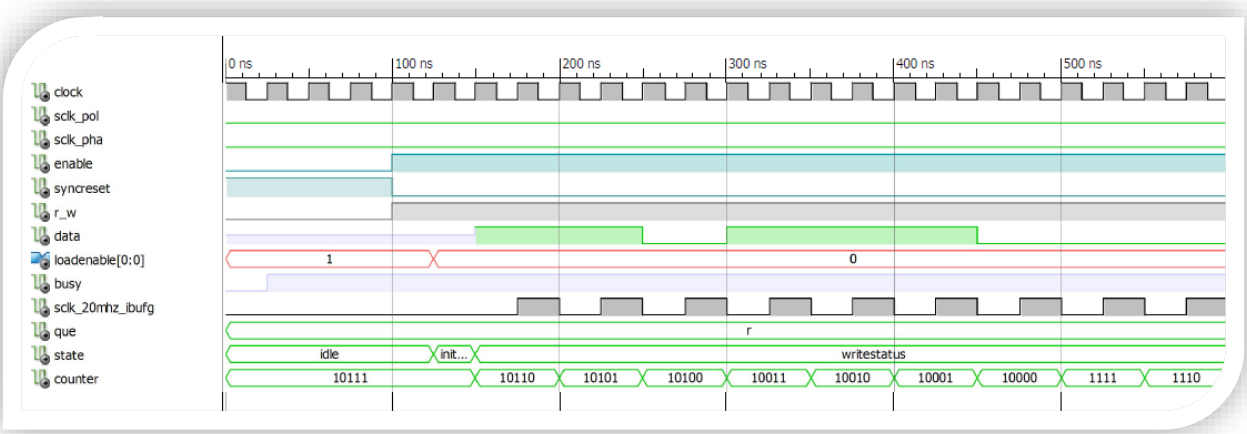
```
-- Clock process definitions
CLOCK_process : process
begin
  Clock <= '1';
  wait for CLOCK_period/2;
  Clock <= '0';
  wait for CLOCK_period/2;
end process;

-- Stimulus process
stim_proc : process
begin

  -- insert stimulus here
  -- hold reset state for 100 ns.
  SyncReset <= '1';
  wait for 100 ns;
  R_W       <= '1';
  Enable    <= '1';
  SyncReset <= '0';
  SCLk_Pol  <= '0';
  SCLK_Pha  <= '0';

  wait;
end process;
```

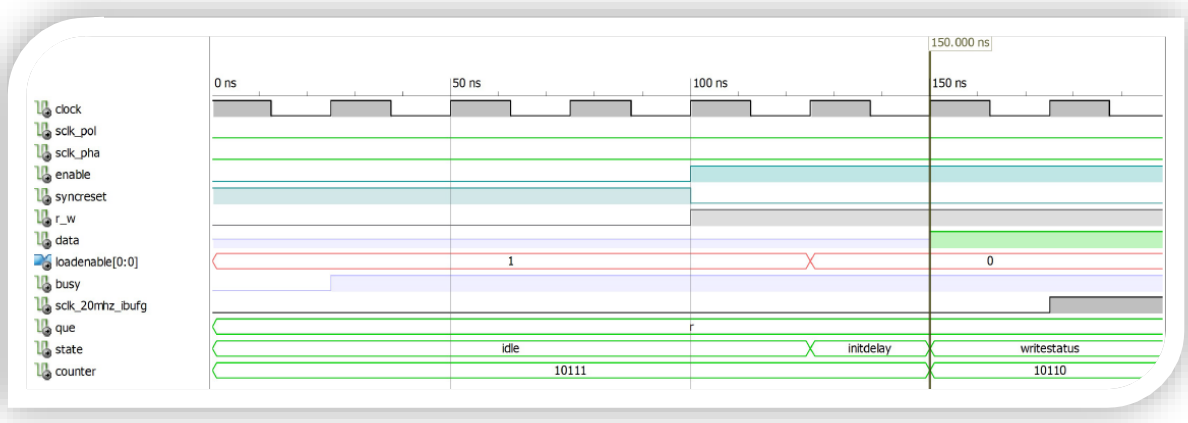
باتوجه‌به کدهای بالا، بعد از ۱۰۰ نانوثانیه ماژول به ترتیب شروع به ارسال بیت‌های رجیسترهای R latch ، C latch و N Latch کند.




شکل (۵-۱) خروجی شماره ۱

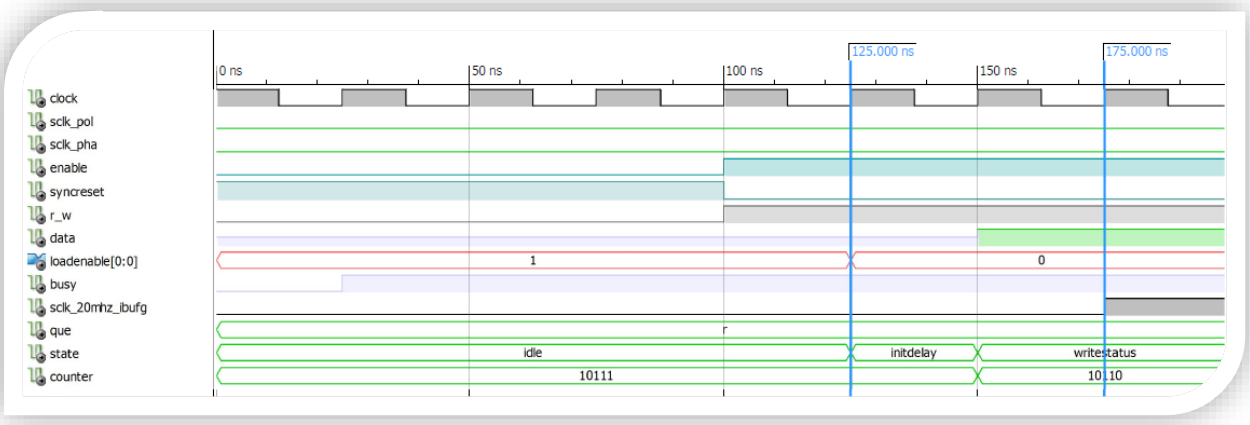
همان‌طور که در شکل (۵-۱) مشخص است تا ۱۰۰ نانوثانیه ابتدایی، مدار در حالت ریست قرار دارد. در این حالت فرض کردیم پایه Busy یک شود. بعد از ریست‌شدن ماژول در زمان ۱۰۰ نانوثانیه ریست غیرفعال شده و پایه enable و R_W به‌منظور شروع عملیات ارسال فعال می‌شوند.

بعد از اینکه لبهٔ بالارونده در وضعیت enable = 1 و R_W = 1 برای اولین‌بار فعال شد، متغیر State به حالت InitDelay تغییر حالت می‌دهد و اولین بیت در این حالت ارسال می‌شود. نکته‌ای که باید به آن توجه داشت این است که اولین بیت ارسالی یا همان MSB در کلاک بعدی ارسال می‌شود. شکل (۵-۲) زمان ارسال اولین بیت بر روی پایه Data را نشان می‌دهد.



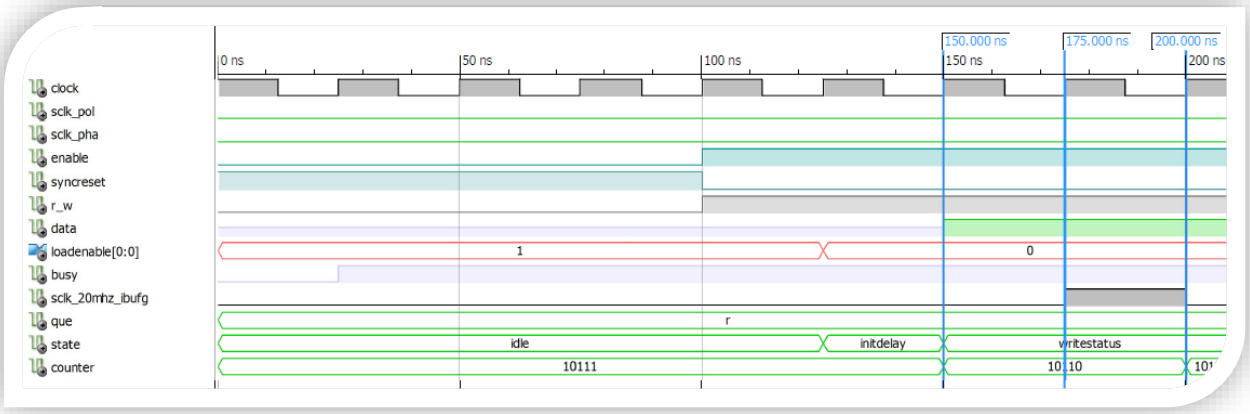

شکل (۵-۲) خروجی شماره ۲

قبل از زمان ۱۵۰ نانوثانیه پایه Data فرض شده است در حالت high-impedance قرار دارد. باتوجه‌به دیتاشیت setup-time ، LoadEnable برابر با ۲۰ نانوثانیه است؛ یعنی بعد از گذشتن حداقل ۲۰ نانوثانیه باید کلاک SPI اولین لبهٔ بالارونده خود را فعال کند که در شکل (۵-۳) رعایت این تایمینگ را مشاهده می‌کنید.




شکل (۵-۳)  خروجی شماره ۳

نکته دیگری که باید توجه داشت، setup time و hold time دیتا است. که باتوجه‌به دیتاشیت حداقل این مقادیر برابر با ۱۰ نانوثانیه است. شکل (۵-۴) رعایت این تامینگ را نشان می‌دهد.

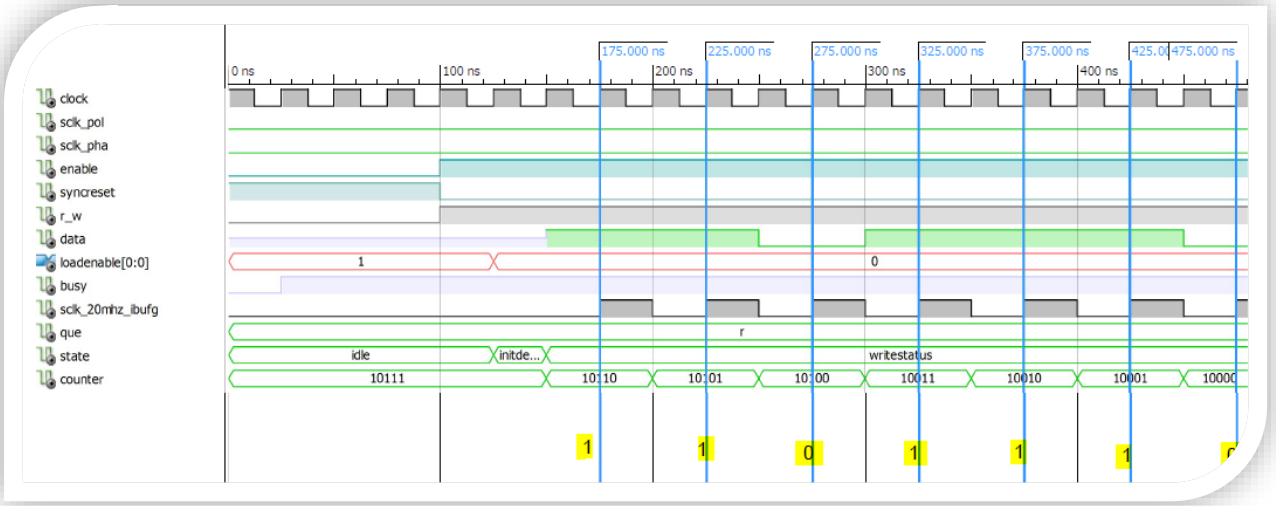



شکل (۵-۴)  خروجی شماره ۴

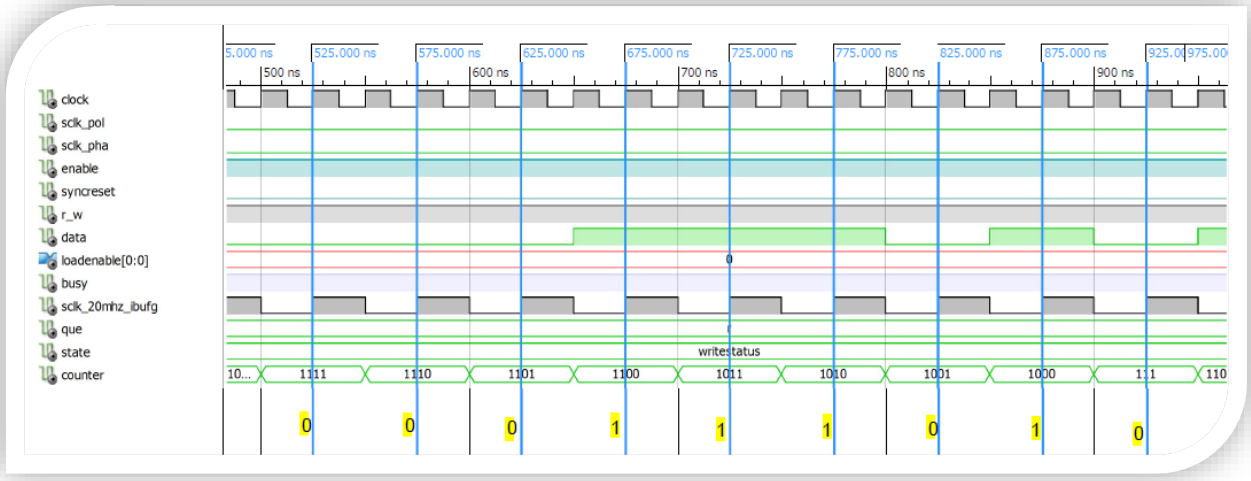
بعد از بررسی مقادیر زمانی حال به ارسال داده می‌پردازیم.

همانطور که از مقدار que مشخص است اولین رجیستری که باید برنامه ریزی شود R Latch است که مقدار آن را برابر با "۱۱۰۱۱۱۰۰۰۰۱۱۱۰۱۰۱۰۱۱۱۱۰۰" در نظر گرفتیم. حال به بررسی این مقادیر بر روی پایه

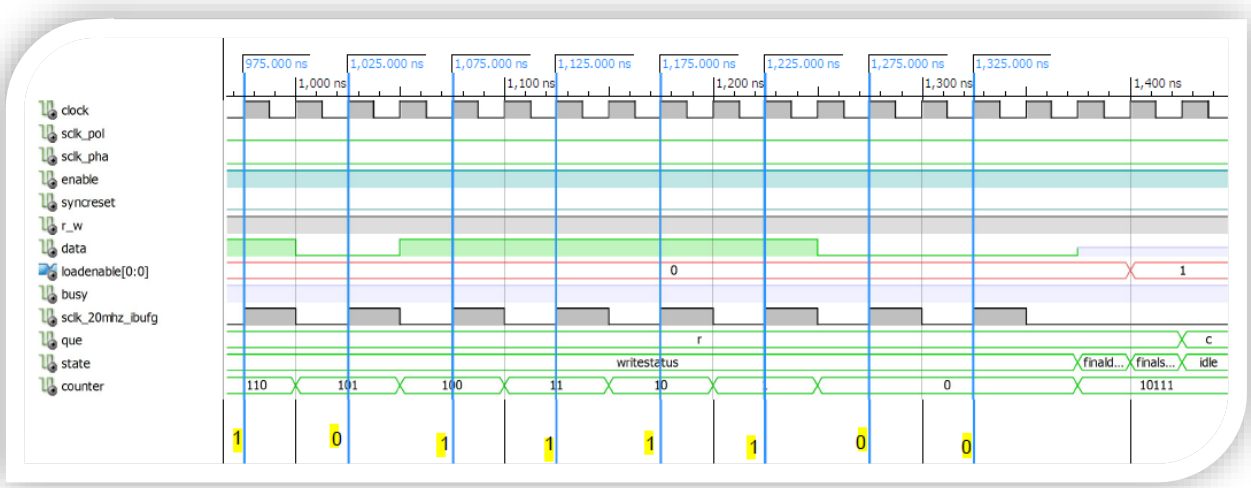
DATA می پردازیم. با هر لبه ی بالا رونده SCLK_20mhz_ibufg می بایســـت یک بیت از ســـمت چپ (MSB) بر روی پایه Data قرار گیرد. در شکل‌های زیر نتایج ارسال این رجیستر را مشاهده می‌کنید.




شکل (۵-۵) خروجی شماره ۵



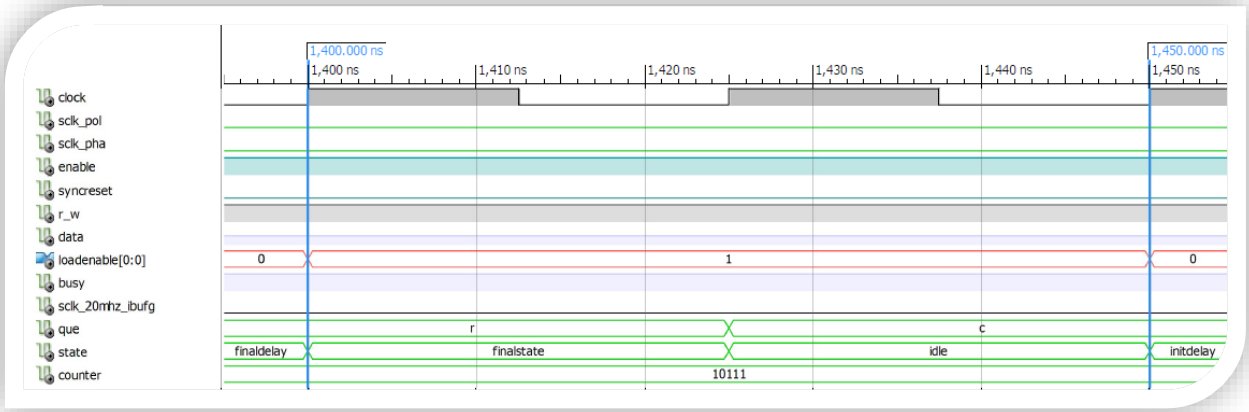

شکل (۵-۶) خروجی شماره ۶




شکل (۵-۷) خروجی شماره ۷

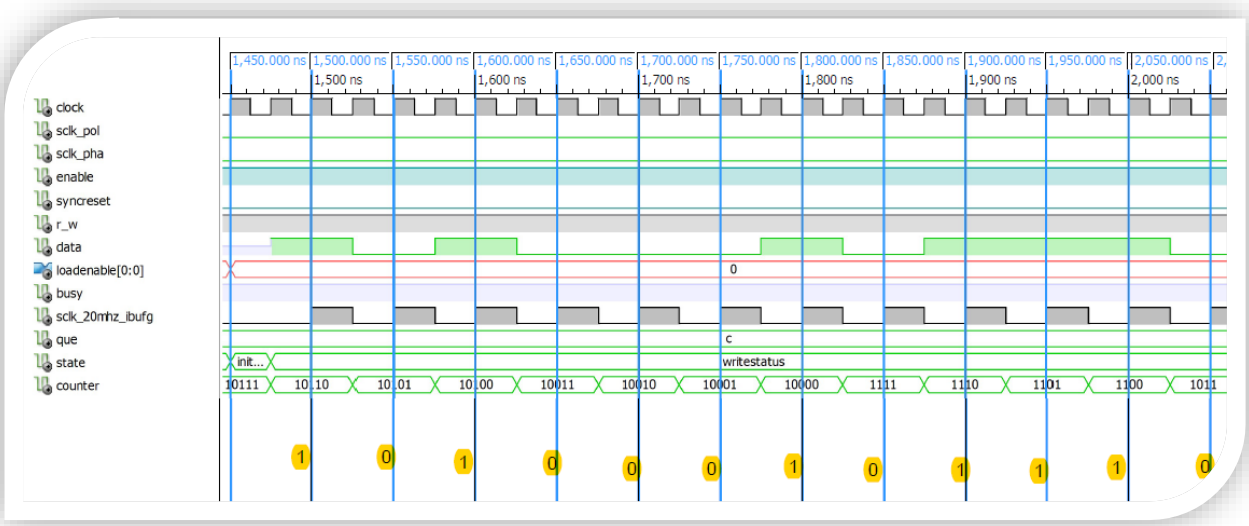
همان‌طور که در شکل‌های ۵، ۶ و ۷ مشاهده می‌کنید دیتای R Latch به‌درستی ارسال شده است.

بعد از ارسال می‌بایست پایه LoadEnable به مدت ۲۰ نانوثانیه ۱ شود و سپس ۰ شود و بیت‌های مربوط به رجیستر بعدی ارسال شود. در شکل (۵-۸) رعایت این تامینگ را مشاهده می‌کنید.

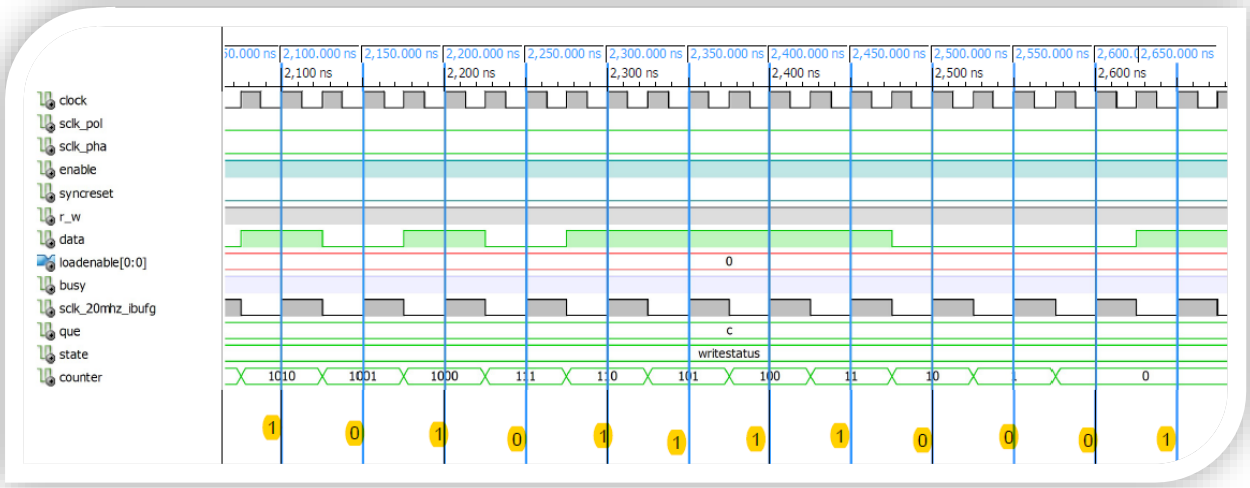



شکل (۵-۸) خروجی شماره ۸

بعد از ارسال رجیستر R Latch باید رجیستر Control Latch برنامه‌ریزی شود. همان‌طور که مشاهده می‌شود مقدار que برابر با C شده است. در شکل‌های زیر به بررسی ارسال دیتاهای Control Latch می‌پردازیم. مقدار Control Latch برابر با "۱۰۱۰۰۰۱۰۱۱۱۰۱۰۱۰۱۱۱۱۰۰۰۱" است.



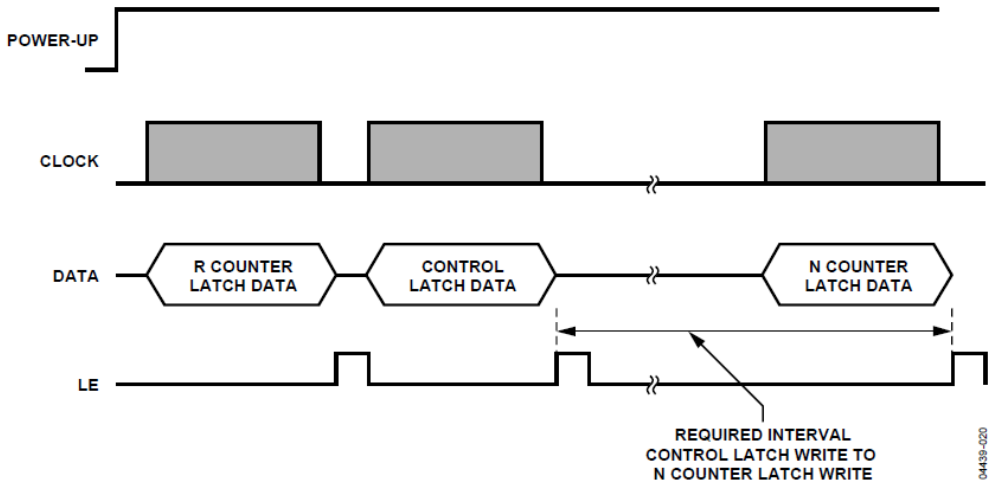

شکل (۵-۹) خروجی شماره ۹




شکل (۵-۱۰) خروجی شماره ۱۰

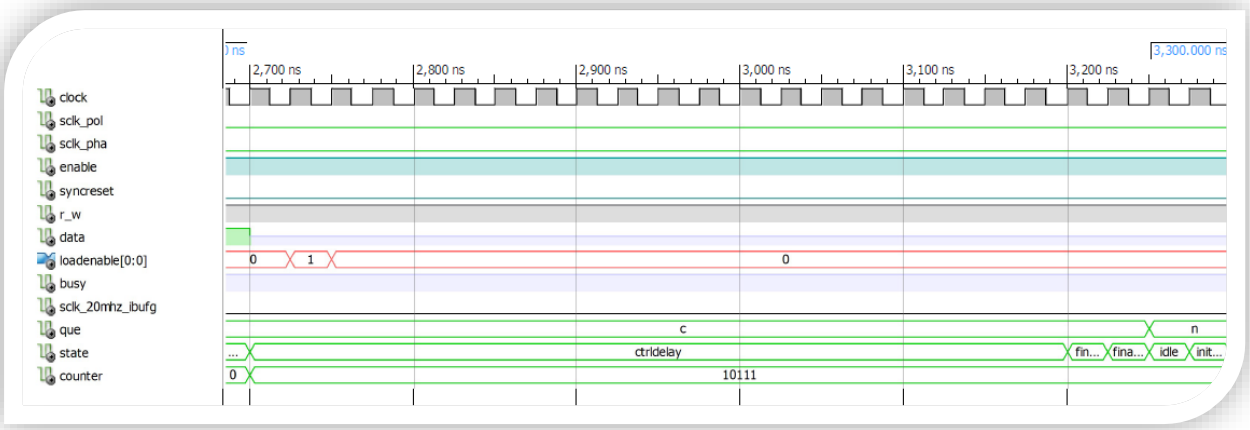
همان‌طور که در شکل‌های بالا مشاهده می‌کنید دیتای Control Latch به‌درستی ارسال شده است.

نکته‌ای که باید به آن توجه کرد گپ زمانی بین برنامه‌ریزی ControlLatch و N Latch باید ایجاد شـــود که شکل (۵-۱۱) این موضوع را نشان می‌دهد.




شکل (۵-۱۱) تایمینگ برنامه ریزی Latch ها

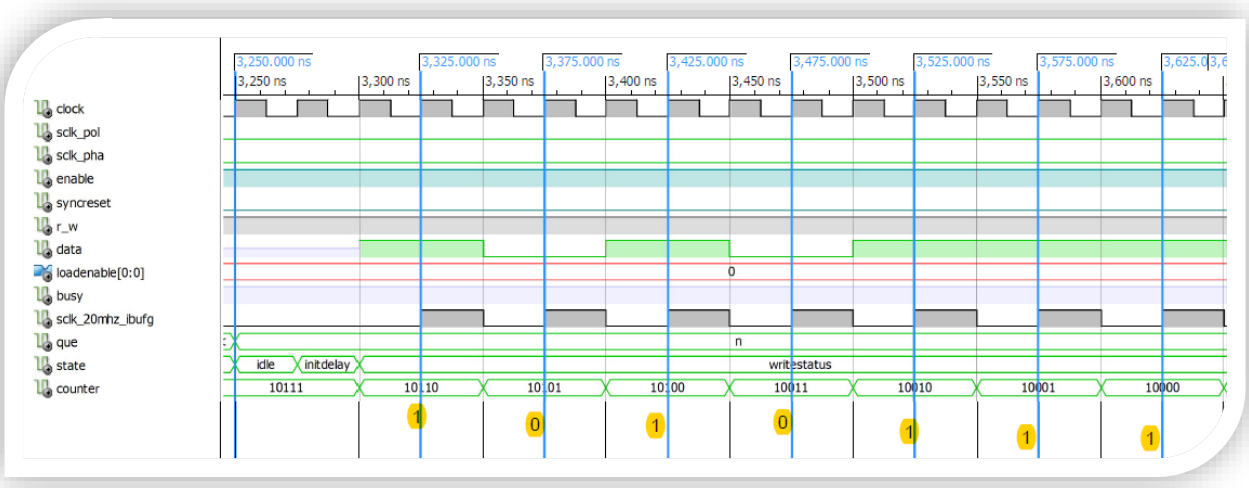
شکل (۵-۱۲) تایمینگ شبیه سازی شده را نشان می دهد.



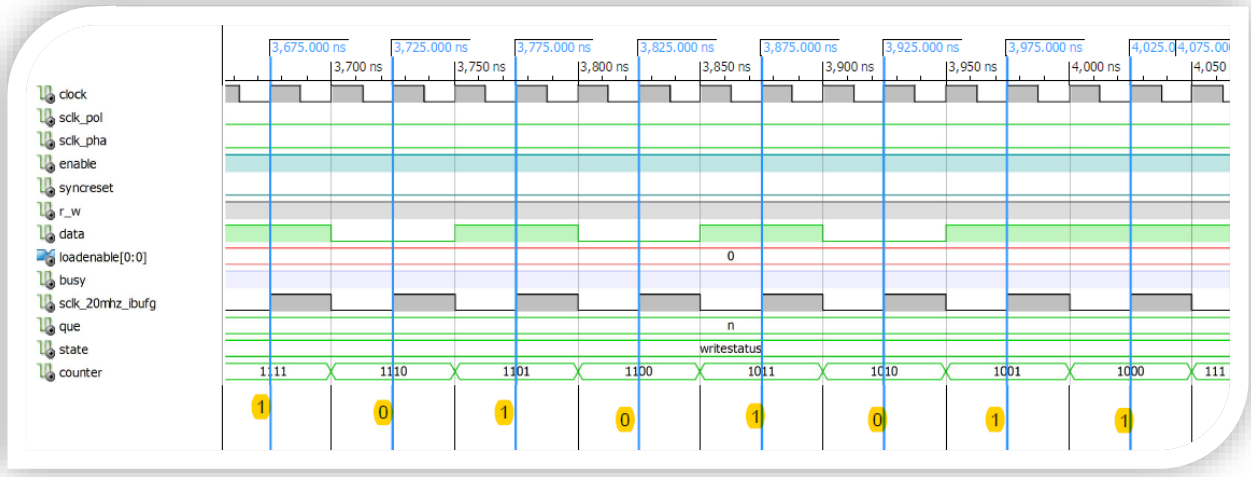

شکل (۵-۱۲) خروجی شماره ۱۱

همانطور که گفته شـده تاخیر بین Control Latch و N Latch را توسـط حالت CtrlDelay ایجاد کردیم که در شکل (۵-۱۲) این موضوع مشخص شده است.

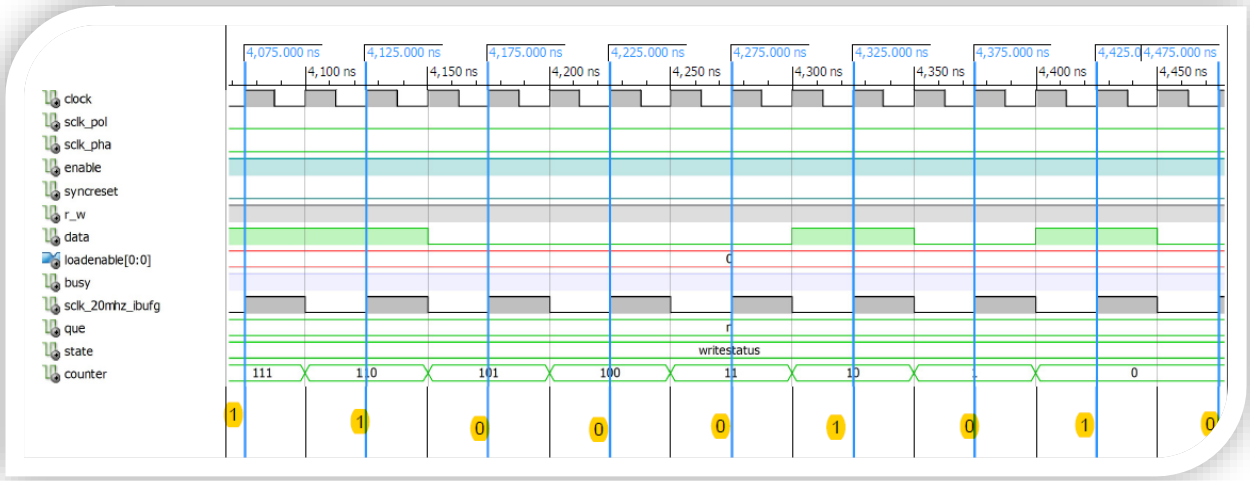
حال به بررسی ارسال بیت‌هایی رجیستر N latch می‌پردازیم. مقدار آن برابر با "۱۰۱۰۱۱۱۱۰۱۰۱۰۱۱۱۱۰۰۰۱۰۱۰" است.




شکل (۵-۱۳) خروجی شماره ۱۲



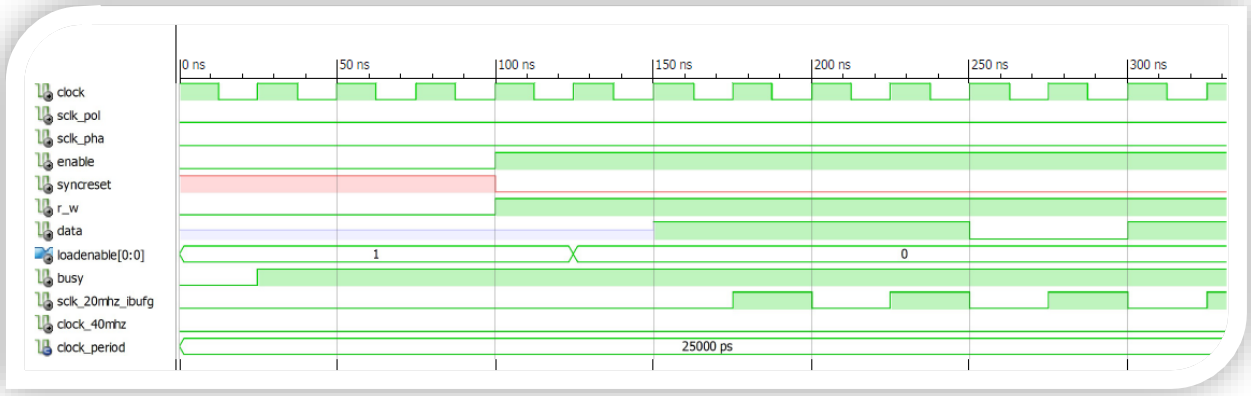

شکل (۵-۱۴) خروجی شماره ۱۳



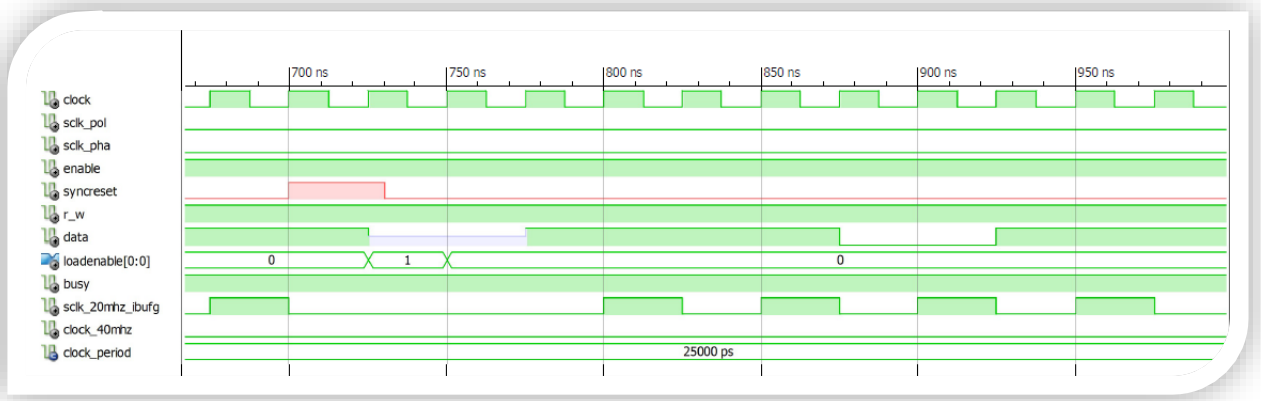

شکل (۵-۱۵) خروجی شماره ۱۴

همان‌طور که در شـکل‌های بالا مشـاهده شـد بیت‌های ارسـالی معتبر هسـتند. بعد از پایان ارسـال می‌توانیم عملیات ارسال را متوقف کنیم؛ اما در اینجا این عملیات مجدد صورت می‌گیرد.

در ادامه به بررسـی حالت بحرانی ای می‌پردازیم. فرض کنید ۵۰۰ نانوثانیه از زمان ارسـال گذشـته اسـت و در یک‌لحظه ریست فعال می‌شود حال انتظار داریم بعد از غیرفعال‌شدن ریست ارسال دوباره صورت گیرد. در ادامه به بررسی این موضوع می‌پردازیم.




شکل (۵-۱۶) خروجی شماره ۱۵




شکل (۵-۱۷) خروجی شماره ۱۶

همان‌طور که در شکل‌های بالا مشاهده می‌کنید بعد از فعال‌شدن دوباره ریست ارسال داده‌های R Latch دوباره از اول شروع می‌شود.

- کدهای تست بنچ در فایلی با پسوند TB قرار گرفته‌اند.

نکته ای که باید به آن توجه داشت فایل ucf پروژه است.( برای پیاده سازی عملی بر روی بورد.) کد های مربوط به UCF:

```
NET "Clock" LOC = "p85"
```

```
NET "Clock" TNM_NET = Clock;
TIMESPEC P_Clock = PERIOD "Clock" 40 MHz HIGH 50%
```

در کد بالا مشخص شده است که سیگنال کلاک به کدام پایه متصل شود همچنین dutycycle این سیگنال هم مشخص شده است.