به نام خدا

دانشگاه صنعتی شریف
دانشکده مهندسی برق

طراحی سیستم های مبتنی بر ASIC/FPGA

تمرین اوّل

امیرحسین یاری
۹۹۱۰۲۵۰۷

۵ اسفند ۱۴۰۲

فهرست مطالب

سوال یک

الف

FPGA یک نوع دستگاه الکترونیکی است که قابلیت برنامه‌ریزی و تنظیم مجدد دارد. در واقع، FPGA از یک ترکیب از گیت‌های منطقی و اتصالات برنامه‌ریزی‌شونده تشکیل شده است. این امکان را به کاربر می‌دهد تا تنظیمات و وظایف مختلف را بازنویسی کند، بدون نیاز به تغییرات فیزیکی یا تعویض قطعات.
ASIC به یک نوع مدار یکپارچه اشاره دارد که برای یک وظیفه خاص و بدون امکان برنامه‌ریزی مجدد طراحی شده است. این مدارها بهینه‌سازی شده‌اند تا عملکرد بهتر و مصرف برق کمتری در مقایسه با راه‌حل‌های دیگر ارائه دهند. به عنوان یک راه‌حل خاص، ASIC معمولاً در برنامه‌هایی با تعداد بالا و نیازهای دقیق به عملکرد ویژه مورد استفاده قرار می‌گیرد. با این حال، توسعه و تولید ASIC معمولاً هزینه بالایی دارد و چون برنامه‌ریزی مجدد نمی‌شود، انعطاف‌پذیری کمتری نسبت به FPGA دارد.
پردازنده یک قطعه اصلی در کامپیوترها و دستگاه‌های الکترونیکی است که مسئول اجرای دستورات برنامه‌ها و انجام عملیات محاسباتی است. پردازنده‌ها از معماری‌ها و استانداردهای خاصی برخوردار هستند و با زبان‌های برنامه‌نویسی مختلف کار می‌کنند.

:FPGA

مزایا:

۱. انعطاف‌پذیری: FPGA قابلیت برنامه‌ریزی مجدد برای وظایف یا قابلیت‌های مختلف را فراهم می‌کند و این امکان را به کاربر می‌دهد.

۲. پردازش موازی: FPGA امکان پردازش موازی را فراهم می‌کند، که می‌تواند به اجرای سریع‌تر برخی وظایف نسبت به پردازنده‌های سنتی منجر شود.

۳. پروتوتایپ‌سازی: در پروتوتایپ‌سازی و تست طراحی‌ها خوب است.

۴. زمان تا بازار: زمان توسعه سریع‌تر نسبت به طراحی‌های ASIC سفارشی به دلیل پیش‌تولید شده بودن FPGA و امکان پیکربندی آنها بر اساس نیازهای برنامه است.

معایب:

۱. مصرف برق: مصرف برق معمولاً بالاتر از ASIC ها و پردازنده‌ها برای وظایف مشابه است.

۲. هزینه: FPGAs ممکن است نسبت به پردازنده‌ها برای تولید به مقیاس بزرگ گران‌تر باشند به دلیل خصوصیت قابل‌برنامه‌ریزی آنها.

۳. سرعت کمتر: نسبت به ASIC سرعت کمتری دارد.

۴. فضای بیشتر: نسبت به ASIC فضای بزرگتری را اشغال می‌کند.

پردازنده‌ها:

مزایا:

۱. هزینه مؤثر: پردازنده‌ها برای وظایف عمومی و تولید به مقیاس بزرگ اغلب اقتصادی‌تر هستند.

۲. دسترسی وسیع: به راحتی می‌توان به پردازنده ها دسترسی پیدا کردن و زمان تا بازار کمی دارند.

۳. مصرف برق پایین: معمولاً مصرف برق پایین‌تری نسبت به FPGA و ASIC برای وظایف مشابه دارند.

۴. سهولت استفاده: پردازنده‌ها با زبان‌ها و معماری‌های برنامه‌ریزی استاندارد ارائه شده اند، که آنها را برای توسعه‌دهندگان نرم‌افزاری آسان‌تر می‌کند.

معایب:

۱. تخصص محدود: پردازنده‌ها ممکن است برای وظایف خاص بهینه نشده باشند، که منجر به عملکرد پایین‌تر نسبت به راه‌حل‌های ویژه شده است.

۲. کمترین موازیت: توانایی پردازش موازی محدودتر نسبت به FPGA و ASIC.

ASIC:

مزایا:

۱. عملکرد بالا: ASIC برای یک برنامه خاص طراحی شده‌اند و عملکرد بهینه برای آن وظیفه خاص را فراهم می‌کنند.

۲. مصرف برق کم: ASICها بهینه شده‌اند و این منجر به مصرف کمتر برق برای وظایف خاص می‌شود.

۳. کارایی اقتصادی: در تولید به مقیاس بزرگ، ASIC ها ممکن است به ازای برنامه‌های خاص به صرفه‌تر باشند نسبت به FPGAs یا پردازنده‌ها.

۴. کارایی در فضا: ASICها ممکن است به صورت فشرده‌تر و مؤثر در فضا باشند چون برای هدف خاصی طراحی شده‌اند.

معایب:

۱. هزینه توسعه: ASIC ها هزینه‌های توسعه اولیه بالاتری دارند، به ویژه برای تولیدهای کوچک.

۲. عدم انعطاف‌پذیری: یک بار طراحی و ساخته شده‌اند، ASICها قابل برنامه‌ریزی یا تغییر آسان نیستند که انعطاف‌پذیری آنها را محدود می‌کند.

تفاوت‌ها:

- انعطاف: FPGA بسیار انعطاف‌پذیر و قابل برنامه‌ریزی مجدد است، درحالی که ASIC برای وظایف خاص طراحی شده و ناتوان در برنامه‌ریزی مجدد است.
- زمان توسعه: FPGA زمان توسعه کوتاه‌تری نسبت به ASIC دارد به دلیل خصوصیت قابل‌برنامه‌ریزی‌اش.
- هزینه: پردازنده‌ها معمولاً برای تولید به مقیاس بزرگ از لحاظ هزینه موثرتر هستند، درحالی که ASICها ممکن است برای برنامه‌های خاص به ازای تعداد بالا به صرفه باشند.
- عملکرد: ASIC برای وظایف خاص بهترین عملکرد را ارائه می‌دهد، بعد از آن FPGA و سپس پردازنده.

ب

گیرنده مخابراتی

با توجه به نیاز به ایجاد ۲ عدد گیرنده مخابراتی، بهترین راهکار ممکن است استفاده از FPGA باشد. دلایل این انتخاب عبارتند از:

۱. انعطاف پذیری FPGA امکان برنامه‌ریزی و تنظیم مجدد دارد. این امکان به شما اجازه می‌دهد تا طراحی را به سرعت تغییر دهید یا نسخه‌های مختلف را تست کنید بدون نیاز به تغییر در سخت‌افزار. این مورد برای فازهای توسعه و آزمایش بسیار مفید است.

۲. پروتوتایپ سازی: FPGA می‌تواند به عنوان یک ابزار خوب برای ساخت پروتوتایپ از گیرنده مورد نیاز شما باشد. این امکان به شما این اجازه را می‌دهد که طراحی خود را به سرعت تست کنید و نیازهای مخابراتی خاص را تأیید کنید.

۳. کارایی هزینه: در مقایسه با ASIC، توسعه FPGA هزینه کمتری دارد و این مساله در تولید تعداد کمتری محصول به ویژه اهمیت دارد.

تراشه مربوط به محاسبات ریاضی

در مورد تراشه محاسبات ریاضی با تعداد ۲۰۰۰ عدد، بهترین گزینه ممکن است استفاده از ASIC باشد. دلایل این انتخاب عبارتند از:

۱. عملکرد: ASIC ها برای وظایف خاص بهینه‌سازی شده‌اند و در نتیجه معمولاً از نظر عملکرد بهتری نسبت به FPGA یا پردازنده دارند.

۲. مصرف انرژی پایین: تراشه‌های ASIC به طور کلی مصرف انرژی پایین‌تری دارند که در محاسبات ریاضی که نیازمند قدرت پردازش بالا هستند، این موضوع بسیار حیاتی است.

۳. کارایی فضا: برای تعداد بالای تراشه‌ها، ASIC ها از نظر کارایی فضا بهینه‌تر هستند.

ج

گیرنده مخابراتی

با توجه به اهمیت TTM، استفاده از FPGA به عنوان یک راه حل اولیه برای سرعت بخشیدن به توسعه و آزمایش‌ها مناسب است. پس از تست‌ها و اطمینان از صحت طراحی، می‌توانید به سمت توسعه ASIC حرکت کنید. در مقایسه با رقبا، استفاده از FPGA این امکان را می‌دهد که به سرعت به بازار وارد شوید و بعد از آن با توسعه ASIC کیفیت و عملکرد محصول خود را افزایش دهید.

تراشه محاسبات ریاضی

با توجه به تعداد ۲۰۰۰ تراشه مورد نیاز، و اهمیت کارایی فضا و مصرف انرژی، استفاده از ASIC انتخاب مناسبی به نظر می‌رسد. در این حالت نیز، به دلیل تعداد بالای تراشه‌ها، استفاده از ASIC به شما این امکان را می‌دهد که با کاهش هزینه و زمان تولید به سرعت بازار را به خود اختصاص دهید.

سناریوهای مختلف

- اگر رقبا زودتر به بازار وارد شوند: در این صورت، ممکن است نیاز باشد که از FPGA برای توسعه سریع محصول استفاده کنید و سپس به توسعه ASIC بروید تا در مقابل رقبا جلوتر باشید.
- اگر اهمیت ابتدایی بر روی TTM باشد: اولین گام می‌تواند استفاده از FPGA باشد تا به سرعت وارد بازار شوید. سپس، با افزایش فروش و درآمد، می‌توانید به توسعه ASIC برای بهبود عملکرد و کاهش هزینه بپردازید.
- اگر اهمیت بیشتری به عملکرد و بهینه‌سازی دارید: ممکن است توجه بیشتری به توسعه ASIC بپردازید تا از نظر عملکرد، مصرف انرژی و فضا بهینه‌تری داشته باشید. این ممکن است زمان بیشتری برای توسعه نیاز داشته باشد، اما باعث ایجاد محصول با کیفیت بالا و قابل رقابت می‌شود.

د

CPLD

- ساختار: CPLD ها دارای ساختاری با تعداد بالای ماژول‌های منطقی هستند که به صورت ترکیبی برنامه‌ریزی می‌شوند.
- توانایی پردازش: CPLD ها مناسب برای پیاده‌سازی منطق کمپلکس و انجام وظایف با تعداد بالای ورودی/خروجی هستند.
- تاخیرهای مدار: تاخیرهای مدار در CPLD ها معمولاً ثابت هستند و به تعداد ماژول‌های منطقی اعمال می‌شوند.

FPGA

- ساختار: FPGA ها دارای تعداد بالای سلول منطقی و تعداد زیادی تراشه برای پیاده‌سازی منطق برنامه‌ریزی شده هستند.
- توانایی پردازش: FPGA ها قابلیت پیاده‌سازی منطق با تعداد بالای ورودی/خروجی و انجام وظایف با پیچیدگی متوسط تا زیاد را دارا هستند.
- تاخیرهای مدار: تاخیرهای مدار در FPGA ها به صورت قابل تنظیم است و می‌تواند با تغییر تنظیمات برنامه‌پذیری تغییر یابد.

Gate Array

- ساختار: در یک Gate Array، تراشه‌ها بخشی از مدارهای منطقی دارند که پیش‌تر برنامه‌ریزی شده‌اند و بخش دیگری از تراشه برنامه‌ریزی نشده است.
- توانایی پردازش: تراشه‌های Gate Array معمولاً کمترین انعطاف‌پذیری را در مقایسه با CPLD و FPGA دارند.
- تاخیرهای مدار: تاخیرهای مدار در Gate Array ها به صورت ثابت یا با انتخاب یک تراشه مشخص تعیین می‌شود.

تفاوت‌های عمده

- انعطاف‌پذیری: FPGA ها انعطاف‌پذیری بیشتری دارند نسبت به CPLD و Gate Array که برای پروژه‌های پیچیده و تغییرات مکرر در برنامه مفید است.
- توان مصرفی: CPLD ها و FPGA ها معمولاً از توان مصرفی بیشتری نسبت به Gate Array ها برخوردارند.
- قابلیت پیاده‌سازی مدارهای پیچیده: FPGA ها به دلیل ساختار پیچیده‌تر خود، برای پیاده‌سازی مدارهای پیچیده‌تر معمولاً مناسب‌تر هستند.
- هزینه: CPLD ها معمولاً ارزانتر از FPGA ها و Gate Array ها هستند. Gate Array ها نیز معمولاً ارزانتر از FPGA ها هستند، اما انعطاف‌پذیری کمتری دارند.

۵

تراشه‌های FPGA شرکت Xilinx از منابع سخت‌افزاری متنوعی تشکیل شده‌اند که قابلیت پیکربندی و برنامه‌ریزی توسط کاربر را دارند. برخی از این منابع عبارتند از:

- بلوک‌های منطقی قابل پیکربندی (CLBs): این بلوک‌ها شامل مجموعه‌ای از مولتیپلکسرها، رجیسترها، مقایسه‌گرها و گیت‌های منطقی هستند که می‌توانند عملیات منطقی، حسابی و شیفت را انجام دهند. این بلوک‌ها می‌توانند به صورت مستقل یا به صورت موازی با یکدیگر کار کنند و تشکیل دهنده‌ی اساسی تراشه‌های FPGA هستند.

- بلوک‌های حافظه (BRAMs): این بلوک‌ها شامل حافظه‌های دو پورته با ظرفیت متفاوت هستند که می‌توانند داده‌ها را ذخیره و بازیابی کنند. این بلوک‌ها می‌توانند به عنوان حافظه‌ی میانی، حافظه‌ی محلی، حافظه‌ی ROM یا FIFO استفاده شوند.
- بلوک‌های DSP : این بلوک‌ها شامل مولتیپلایرها، اکیومولاتورها، افزاینده‌ها و کاهنده‌ها هستند که می‌توانند عملیات پردازش سیگنال دیجیتال را انجام دهند. این بلوک‌ها می‌توانند به عنوان فیلترها، FFT، مدولاتورها، کدک‌ها و غیره استفاده شوند.
- Clocking Resources: این منابع شامل clock devider،clock manager، clock bufferو شبکه‌ی توزیع ساعت هستند که می‌توانند فرکانس و فاز ساعت را تنظیم و توزیع کنند. این منابع می‌توانند به عنوان منبع ساعت داخلی یا خارجی، ساعت مرجع، ساعت متغیر و غیره استفاده شوند.
- منابع ورودی/خروجی (I/O): این منابع شامل پین‌ها، بانک‌ها، بافرها، مبدل‌ها و پروتکل‌های مختلف ورودی/خروجی هستند که می‌توانند ارتباط تراشه‌های FPGA با دستگاه‌های دیگر را برقرار کنند. این منابع می‌توانند به عنوان واسط‌های سریال، موازی، دیفرانسیل، LVDS، PCI Express و غیره استفاده شوند.

سوال دو

الف

طراحی موازی

- تعداد گره‌ها: در طراحی موازی، گره‌ها (عناصر محاسباتی) به صورت موازی و همزمان اجرا می‌شوند.
- سرعت اجرا: به دلیل اجرای موازی، زمان اجرا کاهش می‌یابد و عملیات‌ها با سرعت بیشتری انجام می‌شوند.
- مناسب برای: مسائلی که قابلیت تقسیم دارند و به راحتی قابل پارالل‌سازی هستند.
- مشکلات: مدیریت هماهنگی بین گره‌ها و اطمینان از عدم وجود تداخل یا رزونانس در اجرا.

طراحی همروند

- تعداد گره‌ها: در طراحی همروند، گره‌ها به ترتیب و به صورت همروند اجرا می‌شوند.
- سرعت اجرا: زمان اجرا ممکن است به دلیل اجرای همروند، بیشتر باشد.
- مناسب برای: مسائلی که برای انجام نیاز به ترتیب خاصی دارند یا قابلیت تقسیم کم دارند.
- مشکلات: ممکن است منابع محاسباتی کمی بیافتد، اما مدیریت و هماهنگی آسان‌تر است.

ب

موازی‌سازی در FPGA به وسیله تقسیم مسئله و اجرای همزمان عملیات‌ها در سطوح مختلف ممکن است. در FPGA، موازی‌سازی در چند سطح مختلف قابل انجام است:

۱. سطح ماژول‌ها (Module Level): در این سطح، می‌توانید ماژول‌ها یا بلوک‌های منطقی را به صورت موازی اجرا کنید. هر ماژول می‌تواند یک واحد مستقل از عملکرد را انجام دهد و این ویژگی این اجازه را می‌دهد که عملیات‌های مختلف به صورت همزمان انجام شوند.

۲. سطح بلوک‌های مختلف FPGA (FPGA Block Level): FPGA ها برخی منابع بلوک‌های خاص مانند DSP ، بلوک‌های حافظه، و منطق برنامه‌پذیر (CLBs) دارند. می‌توانید مسائل مختلف را در این بلوک‌ها پیاده‌سازی کرده و موازی‌سازی کنید.

۳. سطح رخدادها (Event Level): در FPGA ها، می‌توانید بر اساس وقایع خاص (مثلاً یک سیگنال فعال شدن) مسئله را به بخش‌های مختلف تقسیم کرده و هر بخش را به صورت موازی اجرا کنید. این می‌تواند با استفاده از اسکیل‌ها (Schedulers) یا ماشین‌های حالت صورت گیرد.

۴. سطح داده‌ها (Data Level): موازی‌سازی در سطح داده‌ها به این معنا است که داده‌ها به صورت همزمان در مسیرهای مختلف پردازش می‌شوند. این می‌تواند با استفاده از بخش‌های موازی پردازشی، پردازش‌های SIMD، یا انجام محاسبات موازی بر روی داده‌های مختلف صورت گیرد.

ج

تفاوت‌های بین GPU و FPGA

۱. توان مصرفی:

- GPUها برای پردازش گرافیک و انجام محاسبات موازی به طور اصلی طراحی شده‌اند. به همین دلیل، توان مصرفی آن‌ها معمولاً بیشتر از FPGAهاست.
- FPGAها به عنوان تراشه‌های برنامه‌پذیر، از لحاظ توان مصرفی قابل برنامه‌ریزی هستند و در برخی موارد می‌توانند توان مصرفی کمتری نسبت به GPU داشته باشند.

۲. زمان:

- GPUها معمولاً برای اجرای وظایف مختلف برنامه‌های گرافیکی طراحی شده‌اند. این بدان معناست که در زمینه‌های دیگر، ممکن است بهینه‌سازی‌های کمتری داشته باشند. همچنین، اگر تغییرات نیازمند بروزرسانی گسترده در سخت‌افزار باشد، زمان آن بیشتر خواهد بود.
- FPGAها به عنوان تراشه‌های برنامه‌پذیر، قابلیت انعطاف بیشتری در اجرای الگوریتم‌ها و برنامه‌های مختلف دارند. به دلیل برنامه‌پذیری بالا، می‌توانند به سرعت با تغییرات نیازمند بروزرسانی و تغییر سازگار شوند.

۳. چالش‌های طراحی:

- برنامه‌نویسی برای GPUها به علت محدودیت‌های معماری و ابزارهای نرم‌افزاری خاص، چالش‌برانگیزتر است. همچنین، برنامه‌هایی که مختص GPU هستند، باید بهینه‌سازی شوند تا از توانایی موازی‌سازی GPU به‌طور کامل بهره‌مند شوند.
- برنامه‌نویسی برای FPGAها نیازمند تسلط بر زبان‌های HDL و نیازمندی‌های سخت‌افزاری است. این چالش می‌تواند برنامه‌نویسان را در تحقق طراحی‌های پیچیده محدود کند.

۴. کاربردهای مختلف:

- هاGPU اصولاً برای پردازش گرافیک، شبیه‌سازی علمی، محاسبات علمی و مسائل مشابه به‌خصوص در حوزه یادگیری عمیق مورد استفاده قرار می‌گیرند.
- FPGAها به عنوان تراشه‌های برنامه‌پذیر، در حوزه‌های متنوعی از جمله شبکه‌های مخابراتی، پردازش سیگنال دیجیتال، کنترل سخت‌افزاری، اتوماسیون صنعتی، و حوزه‌های مختلف دیگر به کار می‌روند.

سوال سه

الف

سطح ترانزیستور

- در این سطح، برنامه‌نویس به طور مستقیم بر روی ترانزیستورها و اتصالات آن‌ها تاثیر می‌گذارد. این سطح از دقت و کنترل بالایی برخوردار است و معمولاً برای پروژه‌هایی با پیچیدگی و نیازهای خاص مورد استفاده قرار می‌گیرد.
- Pseudo-Code:

```
// مثال: توگل کردن خروجی یک ترانزیستور
module ToggleTransistor(output Q, input Clock);
  always @(posedge Clock) begin
    Q <= ~Q;
  end
endmodule
```

سطح گیت

- در این سطح، برنامه‌نویس بر روی گیت‌ها، نهادهای منطقی، و اتصالات بین آن‌ها تمرکز دارد. این سطح معمولاً برای پروژه‌های متوسط تا پیچیده مناسب است.
- Pseudo-Code:

```
// مثال: توگل کردن خروجی با استفاده از گیت NAND
module ToggleGate(output Q, input Clock);
  wire Temp;
  assign Temp = ~(Q & Clock);
  assign Q = Temp & Clock;
endmodule
```

سطح جریان داده

- در این سطح، برنامه‌نویس بر روی عملیات‌های منطقی و جریان داده تمرکز دارد. این سطح به برنامه‌نویس این امکان را می‌دهد که به صورت مستقل از ساختار سخت‌افزاری مدارها را پیاده‌سازی کند.
- Pseudo-Code:

```
// مثال: توگل کردن خروجی به صورت جریان داده
module ToggleDataFlow(output Q, input Clock);
  always @(*) begin
    Q = (Clock) ? ~Q : Q;
  end
endmodule
```

سطح رفتاری

- در این سطح، برنامه‌نویس به صورت انتزاعی‌تر و بر اساس رفتار کلی مدارها کد می‌نویسد. این سطح مختص پروژه‌های با پیچیدگی کمتر و انعطاف‌پذیری بیشتر است.
- Pseudo-Code:

```
// مثال: توگل کردن خروجی به صورت رفتاری
module ToggleBehavioral(output reg Q, input Clock);
  always @(posedge Clock) begin
    Q <= ~Q;
  end
endmodule
```

ب

تعداد گیت سطح ترانزیستور

- بستگی به تعداد ترانزیستورها و پیچیدگی مدار دارد.
- در پروژه‌های پیچیده، ممکن است نیاز به کنترل دقیق بر ترانزیستورها باشد و تعداد گیت بیشتری مصرف شود.

تعداد گیت سطح گیت
- از آنجایی که به صورت انتزاعی‌تر است، ممکن است تعداد گیت کمتری نیاز باشد.

تعداد گیت سطح جریان داده
- معمولاً کمتر از سطوح پایین‌تر.
- انتزاعی‌تر و کنترل کمتر بر سخت‌افزار نسبت به سطح گیت.

تعداد گیت سطح رفتاری
- کمترین میزان تعداد گیت.
- انتزاع بالا و کنترل کمتر بر سخت‌افزار نسبت به سطوح پایین‌تر.

سوال چهار

الف

ابزارهای سنتز در فرآیند پیاده‌سازی برنامه‌های FPGA به کار می‌روند. هدف این ابزارها تبدیل مستندات و فایل‌های ورودی، معمولاً نوشته شده به زبان‌هایی مانند VHDL یا Verilog، به مدارهای قابل برنامه‌پذیر در داخل FPGA است. در ادامه، عملکرد و هدف اصلی این ابزارها را توضیح می‌دهیم:

هدف اصلی:
- تبدیل به نت‌لیست : یک نت‌لیست شامل اطلاعات مربوط به گیت‌ها، فلیپ‌فلاپ‌ها، اتصالات و نت‌ها در مدار برنامه‌پذیر است. این نت‌لیست نمایانگر ساختار منطقی و الکتریکی مدار است که باید در FPGA پیاده‌سازی شود.

عملکرد:

۱. تحلیل سنتز : ابزارهای سنتز با تحلیل کدهای ورودی (مثل VHDL یا Verilog) به دقت معماری مدار را درک می‌کنند. این تحلیل شامل شناسایی عناصر مختلف مدار، اتصالات، و رفتارهای مدار است.

۲. تولید نت‌لیست: بر اساس تحلیل کدهای ورودی، ابزار سنتز یک نت‌لیست ایجاد می‌کند که در آن گیت‌ها، فلیپ‌فلاپ‌ها، و اتصالات مدار نمایان می‌شوند.

۳. تهیه مدار RTL: این ابزارها اطلاعات لازم برای تولید مدار در سطح معماری RTL فراهم می‌کنند. سطح RTL نمایانگر عملیات منطقی و انتقال ثبات بین رجیسترها در مدار است.

۴. بهینه‌سازی: ابزارهای سنتز معمولاً قابلیت بهینه‌سازی مدار را دارا هستند. این بهینه‌سازی شامل کاهش تعداد گیت‌ها، بهبود عملکرد، و کاهش مصرف منابع FPGA می‌شود.

۵. تعیین منابع FPGA: ابزار سنتز به کاربر اطلاعات مربوط به میزان منابع FPGA مورد نیاز برای پیاده‌سازی مدار را ارائه می‌دهد. این اطلاعات شامل تعداد و نوع گیت‌ها، فلیپ‌فلاپ‌ها، و اتصالات است.

۶. راه‌اندازی تاخیر: ابزارهای سنتز مسئولیت رسیدن به تاخیرهای زمانی مشخص شده را دارند. آن‌ها سعی می‌کنند با بهینه‌سازی‌ها و تغییرات در مدار، به تاخیرهای زمانی اهداف مورد نظر برسند.

۷. تولید فایل‌های خروجی:

- Bitstream: این فایل شامل اطلاعات بایت به بایت برنامه FPGA است. این فایل برای بارگذاری برنامه به FPGA استفاده می‌شود.
- Constraints: فایل‌های مربوط به محدودیت‌ها و تنظیمات مدار که توسط برنامه‌نویس ارائه می‌شوند.

ب

پس از فرایند سنتز مدار، مرحله Place and Route به عنوان یکی از مراحل مهم در پیاده‌سازی FPGA آغاز می‌شود. این مرحله شامل دو فعالیت اصلی است:

۱. Place:

- در این مرحله، موقعیت فیزیکی هر عنصر مدار (گیت‌ها، فلیپ‌فلاپ‌ها و ...) در ناحیه فیزیکی FPGA تعیین می‌شود. این شامل تخصیص مکان‌های فیزیکی بر روی FPGA و تعیین اتصالات بین عناصر است.
- هدف این مرحله این است که فاصله‌های کوتاه تری بین عناصر مدار وجود داشته باشد تا بهینه‌سازی در مصرف منابع و حداقل کردن تاخیرهای زمانی ممکن باشد.

۲. Route

- پس از Place، اتصالات بین عناصر مدار را مسیریابی می‌کند. این به معنای ایجاد مسیرهای فیزیکی (Routing) برای اتصالات بین گیت‌ها و دیگر عناصر مدار است.
- در این مرحله، باید بهینه‌سازی در ترکیب مسیرها صورت گیرد تا به تاخیرهای زمانی مقبول برسیم.

ارتباط با محدودیت‌های زمانی و فرکانسی

۱. محدودیت‌های زمانی:

- این محدودیت‌ها توسط برنامه‌نویس در فایل‌های محدودیت‌ها (Constraints) تعیین می‌شوند. این محدودیت‌ها شامل تاخیرهای حداکثر و حداقل بین عناصر مدار، تاخیرهای سیگنال‌ها، و اطلاعات مربوط به پروتکل‌های خاص می‌شوند.

- Place and Route باید این محدودیت‌ها را رعایت کند تا مدار بر اساس نیازهای زمانی برنامه‌نویس بهینه شود.

۲. محدودیت‌های فرکانسی:

- این محدودیت‌ها نیز توسط برنامه‌نویس در فایل‌های محدودیت تعیین می‌شوند و مشخص می‌کنند چه تعداد تاخیر گیت در یک دوره سیگنال (Clock Period) مجاز است.
- Place and Route باید سعی کند با تعیین بهینه‌سازی در قرارگیری و مسیریابی، فرکانس سیستم را به حداکثر ممکن افزایش دهد تا سرعت عملکرد FPGA بهینه شود.

ج

زبان توصیف سخت‌افزار

- عملکرد: در زبان‌های توصیف سخت‌افزار مانند VHDL و Verilog، حلقه‌ها به عنوان فرآیندها یا ماشین‌های حالت توصیف می‌شوند. این حلقه‌ها به شکل همروند توصیف می‌شوند، یعنی فرآیندها به صورت همزمان اجرا می‌شوند. در واقع، ماشین‌های حالت در داخل FPGA به صورت موازی و همزمان اجرا می‌شوند.
- پیاده‌سازی: در زمان سنتز، ابزارهای سنتز این حلقه‌ها را به منابع سخت‌افزاری مختلف تخصیص می‌دهند و مدارهای موازی ایجاد می‌کنند.

زبان‌های برنامه‌نویسی

- عملکرد: در زبان‌های برنامه‌نویسی مانند C++ یا Python، حلقه‌ها به عنوان بلوک‌های متوالی از دستورات توصیف می‌شوند. اجرای دستورات درون یک حلقه به ترتیب صورت می‌گیرد و به صورت متوالی است.
- پیاده‌سازی: برنامه‌های نوشته شده با زبان‌های برنامه‌نویسی در محیط‌های اجرایی متوالی اجرا می‌شوند. به این معنا که هر دستور یکی پس از دیگری اجرا می‌شود و اجرای یک دستور منتظر اتمام دستور قبلی نمی‌ماند.

د

منطق LUT-Based

- توضیح: در منطق LUT-Based، از یک LUT برای پیاده‌سازی منطق استفاده می‌شود. LUT یک جدول است که هر سلول آن مقدار خروجی متناظر با یک ترکیب ورودی‌هایش را نگهداری می‌کند.
- استفاده: با انتخاب درست ترکیب ورودی‌ها، می‌توان تمام منطق‌های مختلف را در یک LUT پیاده‌سازی کرد. این روش برای پیاده‌سازی منطق‌های پیچیده و با الگوهای متغیر مناسب است.

منطق MUX-Based

- توضیح: در منطق MUX-Based، از MUXها برای پیاده‌سازی منطق استفاده می‌شود. MUX یک ترکیب کننده ورودی‌های متعدد به یک خروجی است.
- استفاده: با استفاده از ترکیب‌های مختلف ورودی‌ها و مدیریت ورودی‌های MUX، می‌توان منطق‌های مختلف را پیاده‌سازی کرد. این روش برای پیاده‌سازی منطق‌های ساده و با الگوهای تصمیم‌گیری مناسب است.

تفاوت و کاربردها

- تفاوت اصلی:
  - در LUT-Based، از جدول جستجو برای ذخیره و بازیابی مقادیر منطقی استفاده می‌شود.
  - در MUX-Based، از ترکیب ورودی‌های یک یا چند MUX برای تولید خروجی استفاده می‌شود.
- استفاده در پیاده‌سازی:
  - LUT-Based معمولاً برای پیاده‌سازی منطق‌های پیچیده و متغیر استفاده می‌شود.
  - MUX-Based معمولاً برای پیاده‌سازی منطق‌های ساده و بهره‌گیری از تصمیم‌گیری ساده مناسب است.
- کاربردها:
  - LUT-Based در برنامه‌های پردازش سیگنال و الگوریتم‌های پیچیده استفاده می‌شود.
  - MUX-Based در سنسورها، مدارهای کنترل، و برنامه‌های ساده‌تر منطقی مورد استفاده قرار می‌گیرد.
- مصرف منابع:
  - LUT-Based ممکن است به مصرف منابع FPGA بیشتری نیاز داشته باشد.
  - MUX-Based معمولاً نیاز کمتری به منابع دارد اما محدودیت‌هایی در پیاده‌سازی منطق پیچیده دارد.

سوال پنج

```
module Timer(
    input wire clk,      // 50 MHz clock input
    input wire rst_n,    // Active-low synchronous reset
    input wire [5:0] min, // 6-bit input for minutes (0-59)
    input wire [5:0] sec, // 6-bit input for seconds (0-59)
    output reg done      // Output signal indicating timer completion
);
```

این ماژول تایمر یک تایمر شمارش معکوس را با استفاده از یک سیگنال ساعت و ورودی‌های دیگر، مانند مدت زمان (دقیقه و ثانیه) و سیگنال ریست فعال‌ـپایین ایجاد می‌کند.

```
reg [5:0] counter_min; // Counter for minutes
reg [5:0] counter_sec; // Counter for seconds
reg [24:0] clock_counter; // Counter for clock cycles
```

این تایمر با استفاده از یک شمارنده برای دقیقه‌ها (counter_min) و یک شمارنده برای ثانیه‌ها (counter_sec) عمل می‌کند. همچنین یک شمارنده کلی (clock_counter) نیز برای شمارش تعداد سیکل‌های ساعت استفاده می‌شود.

```
always @(posedge clk or negedge rst_n) begin
    if (~rst_n) begin
        // Reset on active-low reset signal
        counter_min <= min;
        counter_sec <= sec;
        clock_counter <= 0;
        done <= 1'b0;
    end else begin
        // Increment clock counter on each clock cycle
        clock_counter <= clock_counter + 1;

        // Check if one second has elapsed
        if (clock_counter == 50000000) begin
            // Reset clock counter
            clock_counter <= 0;

            // Count down
            if ((6'b0 == counter_min) && (6'b0 == counter_sec)) begin
                // If target time is reached, set done signal
                done <= 1'b1;
            end else if (counter_sec == 6'b0) begin
                // Decrement minute counter when seconds reach 0
                counter_min <= counter_min - 1;
                counter_sec <= 59;
            end else begin
                // Decrement second counter
                counter_sec <= counter_sec - 1;
            end
        end
    end
end
```

در هر لبه صعودی سیگنال ساعت یا لبه نزولی سیگنال ریست (rst_n)، مقدار شمارنده‌ها و شمارنده کلی مقداردهی مجدد می‌شوند. اگر ریست فعال‌ـپایین باشد، مقادیر ورودی (دقیقه و ثانیه) به شمارنده‌ها منتقل می‌شوند و شمارنده کلی صفر می‌شود. همچنین سیگنال done به ۰ تنظیم می‌شود.

در غیر این صورت، با هر سیکل ساعت، شمارنده کلی افزایش می‌یابد. اگر شمارنده کلی به تعداد ۵۰ میلیون (برابر با یک ثانیه) برسد، زمان یک ثانیه گذشته شده و شمارنده کلی صفر می‌شود. در این حالت، اگر زمان به پایان نرسیده باشد (counter_min و counter_sec هرکدام برابر با صفر نباشد)، شمارنده ثانیه کاهش می‌یابد. اگر شمارنده ثانیه به صفر برسد، شمارنده دقیقه کاهش می‌یابد و شمارنده ثانیه به ۵۹ تنظیم می‌شود. اگر زمان به پایان رسیده باشد (هر دو شمارنده برابر با صفر باشند)، سیگنال done به ۱ تنظیم می‌شود، نشان‌دهنده اتمام تایمر است.

```
`timescale 1ns / 1ps

module Timer_tb;

  reg clk;
  reg rst_n;
  reg [5:0] min;
  reg [5:0] sec;
  wire done;

  // Instantiate the Timer module
  Timer timer_inst (
    .clk(clk),
    .rst_n(rst_n),
    .min(min),
    .sec(sec),
    .done(done)
  );

  // Clock generation
  always #10 clk = ~clk;

  // Counter for seconds
  reg [23:0] sec_counter;
```

تست بنچ Timer_tb برای ارزیابی عملکرد ماژول Timer ایجاد شده است. این تست بنچ شامل تولید سیگنال‌های ورودی مانند سیگنال ساعت (clk)، سیگنال ریست (rst_n)، و مقادیر مقداردهی اولیه برای دقیقه و ثانیه (min و sec) است.

در تست بنچ، یک شمارنده برای ثانیه‌ها (sec_counter) نیز ایجاد شده است که با استفاده از سیگنال ساعت شمارش افزایش می‌یابد. در این تست بنچ، دقیقه و ثانیه اولیه به ترتیب به ۱ و ۳۰ تنظیم شده‌اند.

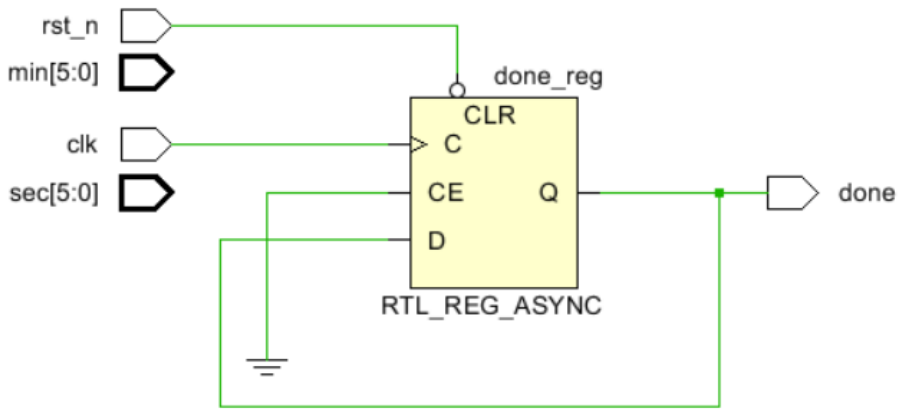
```
// Initial values
initial begin
  clk = 0;
  rst_n = 1;
  min = 6'd1; // Initial value for minutes
  sec = 6'd30; // Initial value for seconds
  sec_counter = 0;

  // Apply reset
  #10 rst_n = 0;
  #10 rst_n = 1;

  // Monitor 'done' signal and finish simulation when it becomes 1
  wait(done == 1);
  $stop;
end

// Display signals every 1 second
always @(posedge clk) begin
  sec_counter <= sec_counter + 1;

  if (sec_counter == 50000000) begin
    $display("Time: %02d:%02d", timer_inst.counter_min, timer_inst.counter_sec);
    sec_counter <= 0;
  end
end
```

همچنین در تست بنچ یک مانیتور برای نمایش زمان باقی‌مانده تا اتمام تایمر (counter_min و counter_sec از ماژول Timer) هر ثانیه نمایش داده می‌شود. تست بنچ تا زمانی ادامه پیدا می‌کند که سیگنال done از ماژول Timer برابر با ۱ شود، و سپس فرآیند stop اجرا شده و شبیه‌سازی به پایان می‌رسد.
خروجی را در تصویر زیر مشاهده می‌کنید.

```
# Time:  1:30
# Time:  1:29
# Time:  1:28
# Time:  1:27
# Time:  1:26
# Time:  1:25
# Time:  1:24
# Time:  1:23
# Time:  1:22
# Time:  1:21
# Time:  1:20
# Time:  1:19
# Time:  1:18
# Time:  1:17
# Time:  1:16
# Time:  1:15
# Time:  1:14
# Time:  1:13
# Time:  1:12
# Time:  1:11
# Time:  1:10
# Time:  1: 9
# Time:  1: 8
# Time:  1: 7
# Time:  1: 6
# Time:  1: 5
# Time:  1: 4
# Time:  1: 3
# Time:  1: 2
# Time:  1: 1
# Time:  1: 0
# Time:  0:59
# Time:  0:58
# Time:  0:57
# Time:  0:56
# Time:  0:55
# Time:  0:54
# Time:  0:53
# Time:  0:52
```

```
# Time:  0:34
# Time:  0:33
# Time:  0:32
# Time:  0:31
# Time:  0:30
# Time:  0:29
# Time:  0:28
# Time:  0:27
# Time:  0:26
# Time:  0:25
# Time:  0:24
# Time:  0:23
# Time:  0:22
# Time:  0:21
# Time:  0:20
# Time:  0:19
# Time:  0:18
# Time:  0:17
# Time:  0:16
# Time:  0:15
# Time:  0:14
# Time:  0:13
# Time:  0:12
# Time:  0:11
# Time:  0:10
# Time:  0: 9
# Time:  0: 8
# Time:  0: 7
# Time:  0: 6
# Time:  0: 5
# Time:  0: 4
# Time:  0: 3
# Time:  0: 2
# Time:  0: 1
# Time:  0: 0
# ** Note: $stop    : E:/University/Semester 8/FPGA/HomeWorks/HW1/Q5_code/Q5_testbench.v(40)
```

بلاک دیاگرام مدار را در تصویر زیر مشاهده می‌کنید.



سوال شش

```
module SimpleALU(
    input signed [3:0] operand1,
    input signed [3:0] operand2,
    input [1:0] control,
    output reg signed [7:0] result
);

    reg signed [3:0] temp_operand1;  // Temporary variable for division

    always @*
    begin
        case (control)
            2'b00:   // Addition
                result = operand1 + operand2;
            2'b01:   // Multiplication
                result = operand1 * operand2;
            2'b10:   // Subtraction
                result = operand1 - operand2;
            2'b11: begin   // Division
                result = 8'sb0;
                // Division using repeated subtraction
                temp_operand1 = operand1;
                if (operand2 != 0) begin
                    while ((temp_operand1 >= operand2) || (temp_operand1 <= -operand2)) begin
                        temp_operand1 = temp_operand1 - operand2;
                        result = result + 1;
                    end
                end
            end
        endcase
    end

endmodule
```

این ماژول یک واحد پردازشی ساده برای انجام عملیات‌های جمع، ضرب، تفریق، و تقسیم بر روی دو عدد ۴ بیتی فراهم می‌کند. ورودی‌های این ماژول شامل دو عدد ۴ بیتی برای عملیات (operand1 و operand2) و دو بیت کنترلی برای تعیین نوع عملیات مورد نظر (control) است. همچنین یک خروجی ۸ بیتی (result) برای نتیجه عملیات انجام شده در ماژول وجود دارد.

در داخل بلاک always @* از یک مورد case استفاده شده است تا بسته به مقدار control، عملیات جمع، ضرب، تفریق یا تقسیم انجام شود. در صورتی که control برابر با 2'b00 باشد، عمل جمع انجام می‌شود و نتیجه در result قرار می‌گیرد. اگر control برابر با 2'b01 باشد، عمل ضرب انجام می‌شود. در صورتی که control برابر با 2'b10 باشد، عمل تفریق انجام می‌شود. و در نهایت، اگر control برابر

با 2'b11 باشد، عمل تقسیم انجام می‌شود.

برای عمل تقسیم، از یک متغیر موقت temp_operand1 برای نگهداری operand1 استفاده شده است. سپس با استفاده از یک حلقه while و انجام تفریق مکرر، نتیجه تقسیم در result ذخیره می‌شود. توجه شود که در صورتی که operand2 برابر با صفر باشد، تقسیم به صورت غیرمعتبر در نظر گرفته شده و مقدار result به صفر تنظیم می‌شود.

```
// Test cases
initial begin
    // Addition
    operand1 = 4;
    operand2 = -5;
    control = 2'b00;
    #10 $display("Addition: %0d + %0d = %0d", operand1, operand2, result);

    // Multiplication
    operand1 = -3;
    operand2 = 7;
    control = 2'b01;
    #10 $display("Multiplication: %0d * %0d = %0d", operand1, operand2, result);

    // Subtraction
    operand1 = 2;
    operand2 = 7;
    control = 2'b10;
    #10 $display("Subtraction: %0d - %0d = %0d", operand1, operand2, result);

    // Division
    operand1 = 6;
    operand2 = 3;
    control = 2'b11;
    #10 $display("Division: %0d / %0d = %0d", operand1, operand2, result);

    // Division 2
    operand1 = 5;
    operand2 = 3;
    control = 2'b11;
    #10 $display("Division: %0d / %0d = %0d", operand1, operand2, result);

    $stop;
end
```

این کد، یک ماژول تست بنچ برای ماژول SimpleALU است. در این تست بنچ، ماژول ALU با ورودی‌های operand1 و operand2 و control از نوع reg و یک خروجی result از نوع wire نمایش داده شده است.

در بخش initial begin، تست‌های مختلف برای ماژول SimpleALU انجام شده‌اند. هر تست شامل

تعیین مقادیر operand1، operand2 و control برای یک عملیات خاص است و سپس با استفاده از دستور display نتیجه عملیات در خروجی نمایش داده شده است. بین هر تست از دستور 10# برای ایجاد تاخیر استفاده شده است.

تست‌ها شامل جمع، ضرب، تفریق و تقسیم هستند. نتایج این تست‌ها به ترتیب در یک فاصله زمانی ۱۰ واحد نمایش داده می‌شوند. پس از اجرای تست‌ها، دستور stop برای اتمام شبیه‌سازی استفاده شده است.

خروجی‌های حاصل شده در تصویر زیر قابل مشاهده می‌باشد.

```
# Addition: 4 + -5 = -1
# Multiplication: -3 * 7 = -21
# Subtraction: 2 - 7 = -5
# Division: 6 / 3 = 2
# Division: 5 / 3 = 1
```

سوال هفت

```
module ShiftRegister (
  input wire clk,          // Clock input
  input wire rst,          // Reset input
  input wire shift_left,   // Shift left control signal
  input wire shift_right,  // Shift right control signal
  input wire load,         // Load control signal
  input wire latch,        // Latch control signal
  input wire [7:0] data_in, // Input data
  output reg [7:0] data_out // Output data
);
  always @ (latch, data_in) begin
    if (latch) begin
      // Latch the input data if latch is active
      data_out <= data_in;
    end
  end
  always @(posedge clk or posedge rst) begin
    if (rst) begin
      // Reset the shift register
      data_out <= 8'b0;
    end else begin
      // Shift left if shift_left is active
      if (shift_left) begin
        data_out <= data_out << 1;
      end
      // Shift right if shift_right is active
      else if (shift_right) begin
        data_out <= data_out >> 1;
      end
      // Load the input data if load is active
      else if (load) begin
        data_out <= data_in;
      end
    end
  end
endmodule
```

این ماژول یک شیفت رجیستر با عملکرد چندگانه است که با ورودی‌های مختلف کنترل می‌شود. ورودی‌های این ماژول شامل سیگنال‌های clk (سیگنال کلاک)، rst (سیگنال ریست)، shift_left (سیگنال شیفت به چپ)، shift_right (سیگنال شیفت به راست)، load (سیگنال لود)، latch (سیگنال لچ) و data_in (ورودی داده به طول ۸ بیت) هستند. خروجی این ماژول نیز یک رجیستر ۸ بیتی به نام data_out است.

در بلوک always @ (latch, data_in) اگر سیگنال latch فعال باشد، داده ورودی به data_out لچ می‌شود.

در بلوک always @(posedge clk or posedge rst)، در صورتی که سیگنال rst فعال باشد، مقدار data_out صفر قرار داده می‌شود و رجیستر ریست می‌شود. در غیر این صورت، اگر سیگنال shift_left فعال باشد، داده درون رجیستر به سمت چپ شیفت می‌یابد. اگر سیگنال shift_right فعال باشد، داده درون رجیستر به سمت راست شیفت می‌یابد. همچنین، اگر سیگنال load فعال باشد، داده ورودی به data_out بارگذاری می‌شود.

```
// Test scenario
initial begin
  clk = 0;
  rst = 1;
  shift_left = 0;
  shift_right = 0;
  load = 0;
  latch = 0;
  data_in = 8'b10101010;

  // Release reset
  #10 rst = 0;

  // Load data
  #10 load = 1;
  #10 load = 0;
  $display("After loading: data_out = %b", dut.data_out);

  // Shift left
  #10 shift_left = 1;
  #10 shift_left = 0;
  $display("After shifting left: data_out = %b", dut.data_out);

  // Shift right
  #10 shift_right = 1;
  #10 shift_right = 0;
  $display("After shifting right: data_out = %b", dut.data_out);

  // Latch data
  #10 latch = 1;
  #10 latch = 0;
  $display("After latching: data_out = %b", dut.data_out);

  // End simulation
  #10 $stop;
end
```

این Testbench برای ماژول ShiftRegister طراحی شده است. در این تست‌بنچ، یک شیفت‌رجیستر با استفاده از ورودی‌های کنترلی مختلف از جمله کلاک (clk)، ریست (rst)، شیفت به چپ (shift_left)، شیفت به راست (shift_right)، لود (load) و لچ (latch) آزمایش می‌شود.

در این تست‌بنچ، ابتدا ورودی‌های clk و rst تعریف شده‌اند. سپس از ماژول ShiftRegister به نام

dut به عنوان یک instance استفاده شده است. ورودی‌ها و خروجی‌های ماژول به این نمونه متصل شده‌اند.

سپس یک سیگنال کلاک با استفاده از always #5 clk = clk ایجاد شده و در هر چرخه کلاک، مقدار clk به مقدار معکوس خود تغییر می‌یابد.

در بلوک initial، مقادیر اولیه برای ورودی‌ها تعیین شده‌اند. پس از ۱۰ واحد زمانی، سیگنال rst از ۱ به ۰ تغییر کرده و ریست آزاد شده است. سپس داده با استفاده از سیگنال load به داخل رجیستر لود شده و مقدار خروجی (data_out) نمایش داده می‌شود. سپس با فعال کردن سیگنال‌های shift_left و shift_right به ترتیب، شیفت به چپ و سپس شیفت به راست انجام شده و مقدار خروجی پس از هر عملیات نمایش داده می‌شود. در نهایت با فعال کردن سیگنال latch، داده درون رجیستر لچ شده و مقدار نهایی خروجی نمایش داده می‌شود.

در تصویر زیر خروجی شبیه‌سازی شده را مشاهده می‌کنید.

```
# After loading: data_out = 10101010
# After shifting left: data_out = 01010100
# After shifting right: data_out = 00101010
# After latching: data_out = 10101010
```

بلاک دیاگرام مدار را در تصویر زیر مشاهده می‌کنید.