به نام خدا

دانشگاه صنعتی شریف
دانشکده مهندسی برق

طراحی سیستم های مبتنی بر ASIC/FPGA

تمرین دوم

امیرحسین یاری
۹۹۱۰۲۵۰۷

۱ فروردین ۱۴۰۳

فهرست مطالب

سوال یک

الف

۱. استفاده از تراشه‌های Block RAM:

- در این روش، می‌توان از بخش‌های Block RAM داخل FPGA برای پیاده‌سازی شیفت رجیستر استفاده کرد. این بخش‌ها از حافظه‌های RAM با سرعت بالا تشکیل شده‌اند که برای ذخیره‌سازی اطلاعات مناسب هستند.
- برای پیاده‌سازی یک شیفت رجیستر ۶۴ تایی با عرض بیت ۶، می‌توان یک Block RAM با ابعاد ۶۴x۶ را استفاده کرد. هر سلول در این Block RAM ۶ بیت اطلاعات را نگهداری می‌کند و ۶۴ سلول برای نگهداری ۶۴ بیت اطلاعات در نظر گرفته می‌شود.
- اتصالات بین ورودی و خروجی‌های شیفت رجیستر و Block RAM باید به گونه‌ای انجام شود که داده‌ها به درستی از ورودی خوانده شده و در خروجی نوشته شوند.

۲. استفاده از تراشه‌های Flip-Flop داخل FPGA:

- در این روش، از تراشه‌های Flip-Flop موجود در داخل FPGA برای پیاده‌سازی شیفت رجیستر استفاده می‌شود. تراشه‌های Flip-Flop از نوع تراشه‌های لاجیکی‌اند که می‌توانند اطلاعات را به صورت همزمان ذخیره کنند و بر اساس سیگنال کلاک ورودی خروجی خود را به تاخیر بیاندازند.
- برای پیاده‌سازی یک شیفت رجیستر ۶۴ تایی با عرض بیت ۶ با استفاده از تراشه‌های Flip-Flop، ۶۴ تراشه Flip-Flop با عرض ورودی ۶ بیت انتخاب می‌شود. این تراشه‌ها به صورت زنجیره‌ای به هم متصل می‌شوند تا یک شیفت رجیستر به وجود آید.
- اتصالات بین ورودی و خروجی‌های شیفت رجیستر و تراشه‌های Flip-Flop باید به گونه‌ای انجام شود که داده‌ها به درستی از ورودی خوانده شده و در خروجی نوشته شوند.

ب

SRL یا Shift Register Look-Up Table (LUT) در FPGAهای زایلینکس، یک نوع حافظه برنامه‌پذیر است که درواقع یک شیفت رجیستر با طول مشخص است که در LUTهای داخلی FPGA پیاده‌سازی شده است. این نوع شیفت رجیستر از بخش‌های ترکیبی FPGA برای ذخیره اطلاعات استفاده می‌کند.

ساختار کلی SRL به صورت زیر است:

- هر SRL شامل تعدادی LUT (Look-Up Table) است که به صورت زنجیره‌ای متصل شده‌اند.
- هر LUT دارای یک ورودی برای داده و یک ورودی برای سیگنال کلاک می‌باشد.

- در حالت عادی، LUTها به صورت ترکیبی عمل می‌کنند و خروجی آنها بر اساس ورودی‌های خود تولید می‌شود.
- اما در حالت SRL، LUTها به صورت ترتیبی (sequential) عمل می‌کنند، به این معنی که ورودی‌های یک LUT به عنوان ورودی به LUT بعدی ارسال می‌شوند. به این ترتیب، یک زنجیره از LUTها یک شیفت رجیستر را پیاده‌سازی می‌کند.

کاربردهای اصلی SRL در FPGAهای زایلینکس عبارتند از:

۱. پیاده‌سازی شیفت رجیستر: برای ایجاد و استفاده از شیفت رجیسترها در طرح‌های FPGA، SRLها می‌توانند به عنوان یک ابزار کارآمد و سریع برای این منظور مورد استفاده قرار گیرند.

۲. پیاده‌سازی منطق ترتیبی: SRLها می‌توانند برای پیاده‌سازی منطق ترتیبی مورد استفاده قرار گیرند، مانند ماشین‌های حالت یا منطق مرتبط با کلاک.

۳. پیاده‌سازی حافظه کوتاه‌مدت: در برخی از موارد، SRLها می‌توانند برای ایجاد حافظه کوتاه‌مدت در FPGA مورد استفاده قرار گیرند، اما این استفاده نیاز به توجه دقیق به ظرفیت و سرعت آنها دارد.

ارتباط SRL با قسمت قبلی (پیاده‌سازی شیفت رجیستر با استفاده از Block RAM یاFlip-Flop) به این صورت است که SRL نیز یکی از روش‌های پیاده‌سازی شیفت رجیستر در FPGAهاست. با استفاده از SRLها، می‌توان شیفت رجیسترها را به صورت کارآمد و سریع پیاده‌سازی کرد، مشابه روش دوم (استفاده از تراشه‌های Flip-Flop داخل FPGA). با این حال، تفاوت اصلی بین این دو روش در این است که SRLها معمولاً درون LUTهای FPGA پیاده‌سازی می‌شوند و از ترکیب LUTهای داخلی برای پیاده‌سازی شیفت رجیستر استفاده می‌کنند، در حالی که روش دوم (استفاده از تراشه‌هایFlip-Flop) از تراشه‌های جداگانه در FPGA استفاده می‌کند.

سوال دو

الف و ب

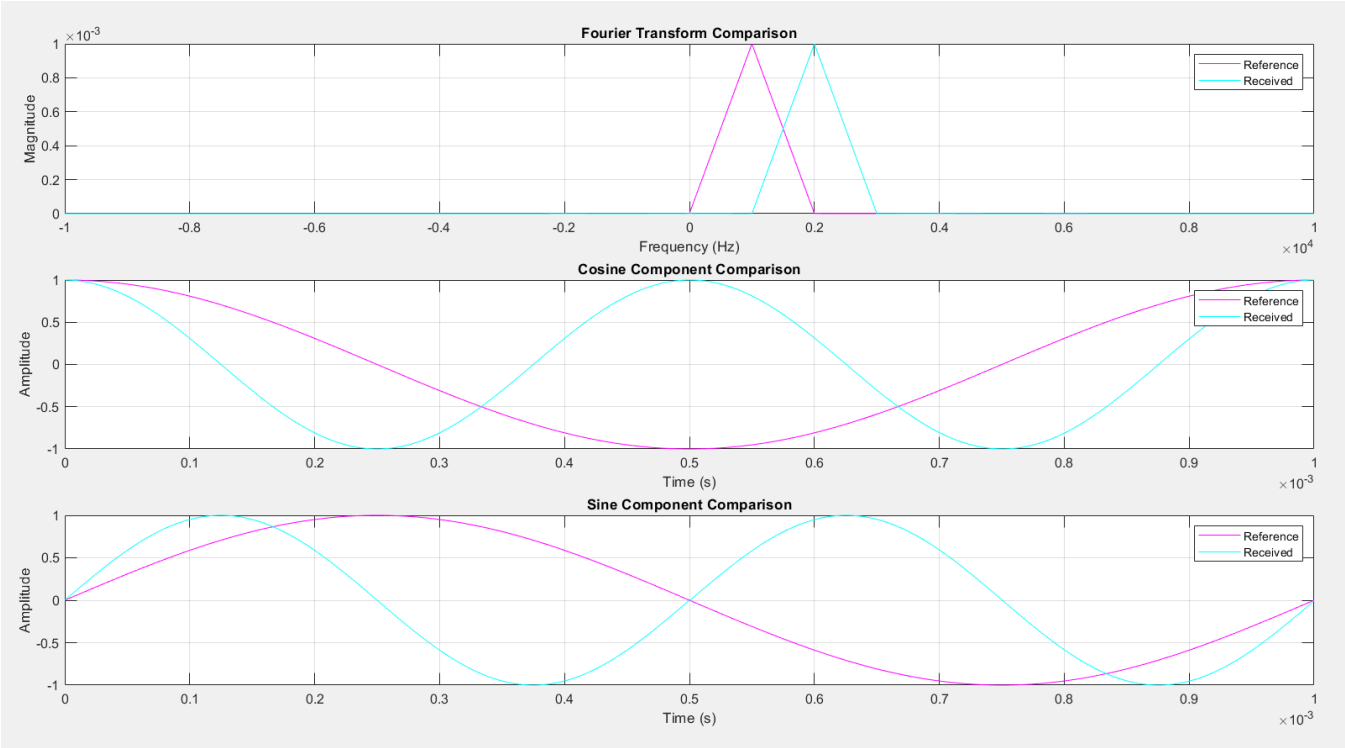
در فایل Signal_Generator.m، یک برنامه MATLAB برای تولید و ذخیره‌ی سیگنال‌های سینوسی و کسینوسی با فرکانس ۱۰۰۰ هرتز با استفاده از تبدیل به نقطه ثابت ارائه شده است. ابتدا پارامترهایی مانند طول واژه، تعداد بیت‌های ممیز، فرکانس نمونه‌برداری و تعداد نمونه‌ها تعیین می‌شوند. سپس بردار زمانی براساس فرکانس نمونه‌برداری و تعداد نمونه‌ها ایجاد می‌شود. سیگنال‌های سینوسی و کسینوسی با استفاده از توابع sin و cos و با در نظر گرفتن فرکانس نمونه‌برداری ایجاد می‌شوند. در نهایت، این سیگنال‌ها به شکل نقطه ثابت تبدیل شده و به فایل‌های مورد نظر با پسوند mem. نوشته می‌شوند. این فرایند با استفاده از یک تابع جداگانه به نام generate_and_write_signal صورت می‌گیرد که سیگنال ورودی را به نقطه ثابت تبدیل می‌کند، آن‌ها را به فرمت باینری تبدیل می‌کند و در نهایت در فایل مورد نظر ذخیره می‌کند.

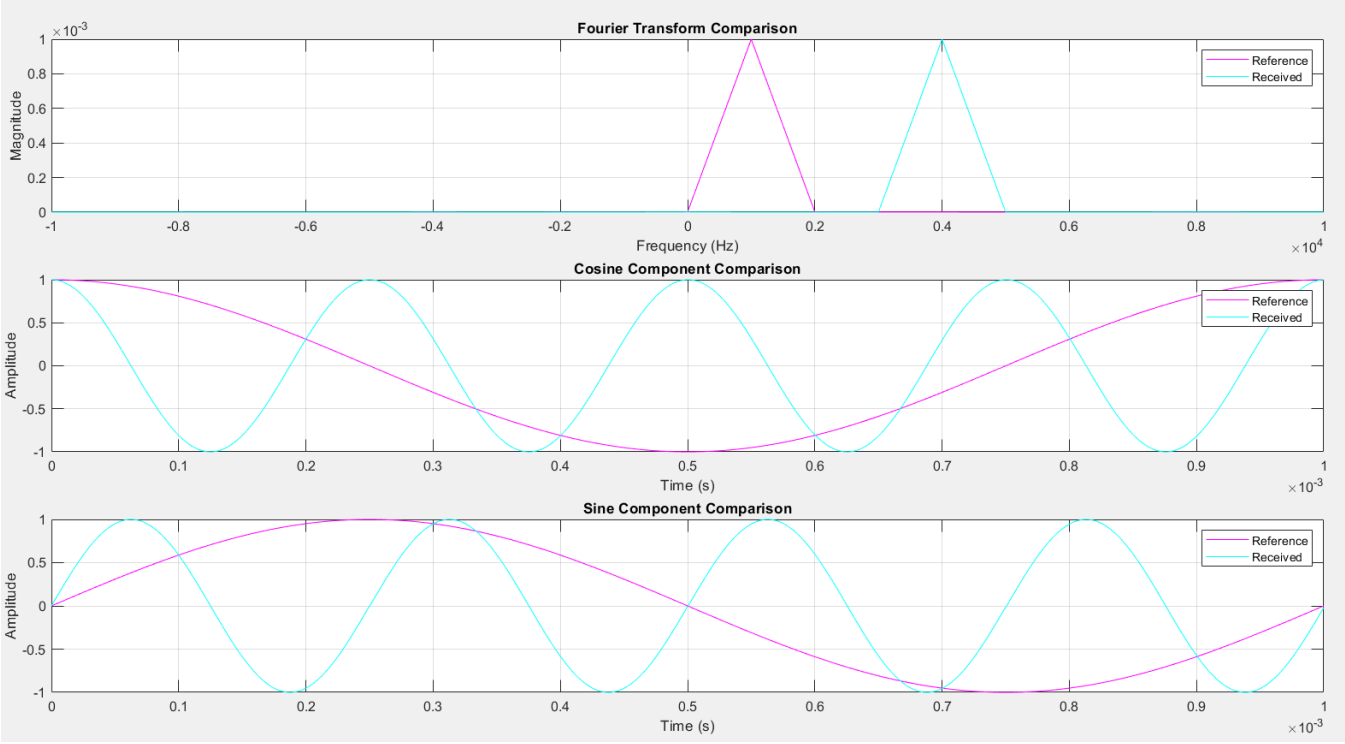
سپس، در فایل Q2.v یک ماژول با نام Q2 تعریف شده است که وظیفه‌ی آن Down Sample کردن سیگنال‌های ورودی با توجه به کنترل داده شده را بر عهده دارد. این ماژول شامل دو حافظه ۱۶ بیتی با تعداد خانه‌های ۱۰۲۴ می‌باشد که به ترتیب برای ذخیره‌ی سیگنال ورودی و خروجی استفاده می‌شوند. با دریافت سیگنال‌های کلاک و ریست و همچنین یک سیگنال ۳ بیتی کنترل، ماژول سیگنال ورودی را Down Sample می‌کند و نمونه‌های حاصل را در حافظه‌ی خروجی memOut قرار می‌دهد. همچنین، اگر مقدار کنترل ورودی به جز ۳، ۲، ۱ و ۴ باشد، سیگنال incorrectControl را فعال می‌کند تا نشان دهد که ورودی کنترل نامعتبر است.

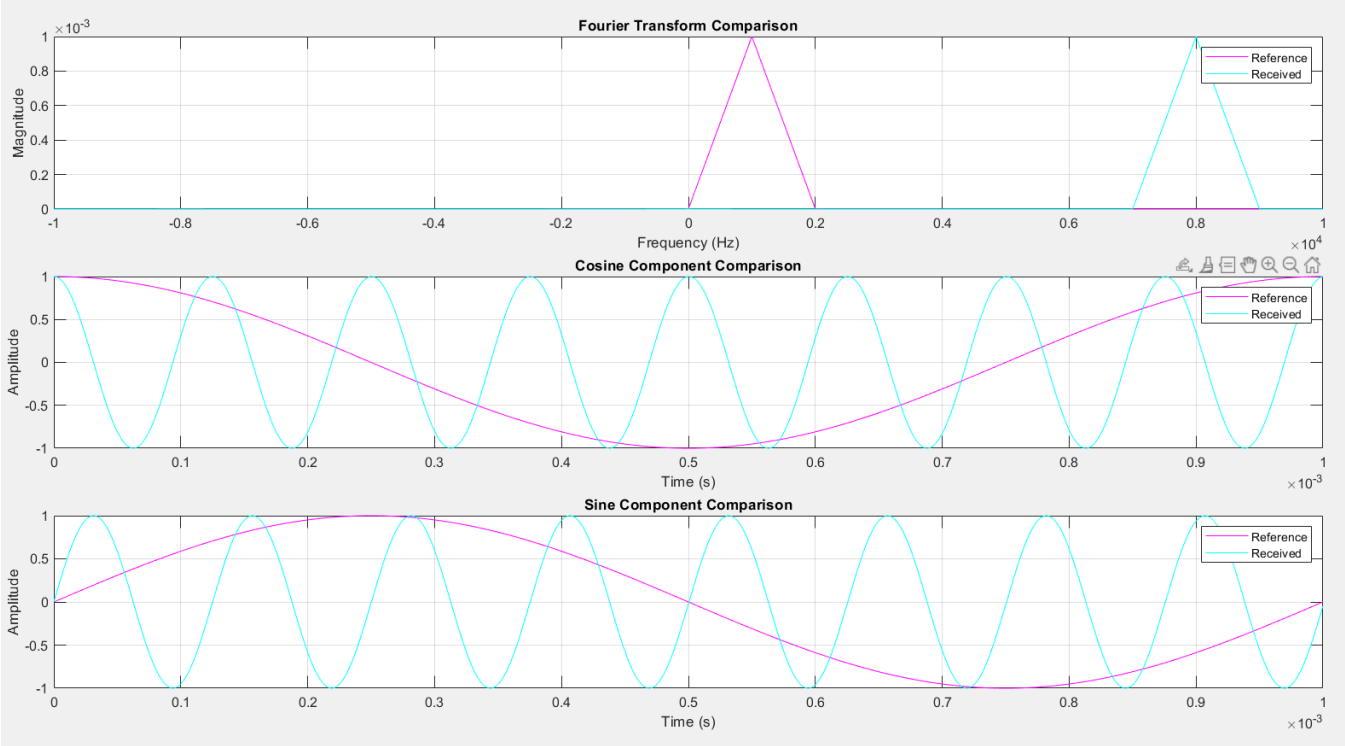
```
module Q2(
    input Clk,
    input Reset,
    input [2:0] Control,
    output reg incorrectControl
);

    parameter memorySize = 1024;

    integer i;
    reg [15:0] memIn [0:memorySize - 1];
    reg [15:0] memOut [0:memorySize - 1];

    always @(posedge Clk) begin
        if (!Reset) begin
            // Reset memory and incorrectControl
            for (i = 0; i < memorySize; i = i + 1)
                memOut[i] <= 0;
            incorrectControl <= 0;
        end else begin
            // Down-sampling logic
            if (Control <= 4 && Control != 0) begin
                for (i = 0; i < memorySize; i = i + 1)
                    memOut[i] <= memIn[(i << (Control - 1)) % memorySize];
                incorrectControl <= 0;
            end else
                incorrectControl <= 1;
        end
    end

endmodule
```

در این بخش، ابتدا با استفاده از تست بنچ Q2_TB، فایل‌های sin.mem وcos.mem را به صورت جداگانه خوانده و برای هر کدام از آن‌ها عملیاتDown Sample را با استفاده از ماژول Q2 انجام می‌دهیم. ابتدا فایل‌ها خوانده شده و سپس با استفاده از حلقه for، برای هر یک از مقادیر کنترلی مقدار Reset به ۰ و سپس به ۱ تغییر می‌دهیم تا ماژول به حالت اولیه برگردد. سپس مقدار کنترل مورد نظر را به ماژول می‌دهیم و پس از گذشت زمانی کافی، خروجی حاصل را در فایل‌های sin_result_[1..4].txt و cos_result_[1..4].txt به صورت جداگانه ذخیره می‌کنیم. این فایل‌ها حاوی نتایجDown Sample شده‌ی سیگنال‌های متناظر با مقادیر کنترلی ۱ تا ۴ هستند. سپس فرآیند تست به پایان می‌رسد و شبیه‌سازی متوقف می‌شود.

در فایل Check_Signal.m با استفاده از متلب، طیف سیگنال‌های کسینوسی و سینوسی را که از فایل‌های cos_result_x.txt و sin_result_x.txt خوانده شده‌اند، رسم می‌کنیم. ابتدا تابع showFourier با تعداد هارمونیک‌های مورد نظر فراخوانی می‌شود. در این تابع، ابتدا داده‌های خوانده شده از فایل‌ها به صورت fixed-point خوانده می‌شوند. سپس پارامترهای مربوط به سیگنال مانند فرکانس و فاصله زمانی محاسبه می‌شوند. سیگنال مرجع به صورت یک سیگنال مختلط ساخته می‌شود. داده‌های خوانده شده از فایل‌ها به شکل double تبدیل شده و با سیگنال مرجع جمع می‌شوند. سپس FFT بر روی سیگنال دریافتی و مرجع انجام می‌شود و نمودارهای مربوط به طیف، مقایسه مؤلفه‌های کسینوسی و سینوسی سیگنال‌ها رسم می‌شوند. در نهایت، فرکانس و زمان محورها، مقادیر محاسبه شده و مقادیر مرجع به عنوان مؤلفه‌های کسینوسی و سینوسی در نمودارها قرار می‌گیرند. خروجی‌ها را در تصاویر زیر مشاهده می‌کنید.




Fourier Transform Comparison
Magnitude
Frequency (Hz)
Reference
Received
Cosine Component Comparison
Amplitude
Time (s)
Reference
Received
Sine Component Comparison
Amplitude
Time (s)
Reference
Received


Fourier Transform Comparison
Magnitude
Frequency (Hz)
Reference
Received
Cosine Component Comparison
Amplitude
Time (s)
Reference
Received
Sine Component Comparison
Amplitude
Time (s)
Reference
Received

ج

هسته‌های نرم‌افزاری:

- می‌توانید به وسیلهٔ زبان‌های برنامه‌نویسی مانند Verilog یا VHDL هسته‌های نرم‌افزاری را برای FPGA ایجاد کنید. این هسته‌ها به عنوان ماژول‌های نرم‌افزاری بر روی FPGA قرار می‌گیرند و قابلیت اجرای الگوریتم‌ها و وظایف مختلف را دارند.
- کاربردهای هسته‌های نرم‌افزاری شامل پردازش سیگنال دیجیتال ، کنترل سخت‌افزاری، شبیه‌سازی و پردازش تصویر است.

هسته‌های سخت‌افزاری:

- این هسته‌ها به طور مستقیم در سطح سخت‌افزار FPGA پیاده‌سازی می‌شوند. طراحی آنها معمولاً به زبان‌های RTL مانند Verilog یا VHDL انجام می‌شود.
- هسته‌های سخت‌افزاری به صورت مدارهای منطقی انجام می‌شوند و قابلیت اجرای وظایف با سرعت بالا و با کارایی بالا را دارند.
- کاربردهای هسته‌های سخت‌افزاری شامل رمزنگاری، پردازش تصویر، پردازش سیگنال، مهندسی معکوس و سیستم‌های نظارتی می‌شود.

برخی از هسته‌های نرم‌افزاری و سخت‌افزاری معمول در FPGA شامل موارد زیر هستند:

۱. پردازنده‌های مرکزی (CPU) و پردازنده‌های میانی (MCU): اینها اجازه می‌دهند تا عملیات نرم‌افزاری روی FPGA انجام شود. برای مثال، پردازنده‌های ARM Cortex-M بر روی برخی از FPGAهای شرکت‌هایی مانند Xilinx و Intel اجرا می‌شوند.

۲. پردازش سیگنال: اینها معمولاً برای پردازش سیگنال‌های دیجیتال (DSP) مورد استفاده قرار می‌گیرند. این امکان را فراهم می‌کنند تا الگوریتم‌های پیچیده پردازش سیگنال، مانند فیلترها و FFT، روی FPGA اجرا شوند.

۳. حافظه‌های نرم‌افزاری و سخت‌افزاری: اینها از حافظه‌های داخلی و خارجی برای ذخیره داده‌ها و برنامه‌ها استفاده می‌کنند. حافظه‌های برنامه‌پذیر FPGA می‌توانند برای ذخیره‌سازی برنامه‌های FPGA یا داده‌های ورودی/خروجی مورد استفاده قرار گیرند.

۴. واحدهای ورودی/خروجی (I/O): اینها اتصالات بین FPGA و دیگر قطعات سخت‌افزاری یا نرم‌افزاری را فراهم می‌کنند. آنها به عنوان رابط‌های میان FPGA و قطعات خارجی مانند حسگرها، میکروکنترلرها، و دیگر اجزاء سخت‌افزاری عمل می‌کنند.

DDS به معنی Direct Digital Synthesis است و یک تکنولوژی است که در تولید سیگنال‌های تحولی مستقیم (انالوگ) با فرکانس و فاز قابل تنظیم به کار می‌رود. این تکنولوژی بسیار مفید برای ایجاد

سیگنال‌های تحولی مختلف است که در بسیاری از برنامه‌های الکترونیکی، ارتباطات، و ابزارهای دقیق مورد استفاده قرار می‌گیرد.

زایلینکس یکی از تولیدکنندگان FPGA معروف است که DDS را به عنوان یکی از منابع تولید سیگنال در محصولات خود استفاده می‌کند. در محصولات زایلینکس، DDS به عنوان یک بلاک قابل برنامه‌ریزی موجود است که توسط کاربر بر روی FPGA قابل استفاده و پیکربندی است.

به عنوان مثال، یکی از کاربردهای مهم DDS در محصولات FPGA زایلینکس، ساخت سیگنال‌های فرکانسی مورد نیاز برای انجام آزمایشات الکترونیکی یا ایجاد سیگنال‌های ساعت برای سیستم‌هایی مانند سیستم‌های ارتباطات بی‌سیم است. با استفاده از DDS، می‌توانید فرکانس، فاز و حتی میزان تغییرات دیگر در سیگنال‌ها را با دقت و دقت بالا کنترل کنید.

به عنوان مثال دیگر، DDS می‌تواند در سیستم‌های راداری به عنوان مولتی‌فرکانس سیگنال‌ها برای ارسال و دریافت سیگنال‌های رادار مورد استفاده قرار گیرد. این کاربرد به دقت و قابلیت برنامه‌ریزی بالای DDS بستگی دارد.

مفهوم DDS در این سوال این است که ما از یک سیگنال دیجیتال (به عنوان یک جدول نمونه) برای تولید سیگنال آنالوگ (با استفاده از مبدل دیجیتال به آنالوگ) استفاده می‌کنیم. در اینجا، ما از-fixed point استفاده می‌کنیم که عدد ثابت با دقت مشخصی را نمایش دهد و سپس از این عدد برای تولید سیگنال‌های سینوسی و کوسینوسی با فرکانس‌های مختلف استفاده می‌کنیم.

سوال سه

الف

DSP48 یک بخش مهم و کلیدی در FPGA های شرکت زایلینکس است که برای پردازش سیگنال دیجیتال (DSP) به کار می‌رود. این بلاک‌های DSP48 برای انجام عملیات‌های ریاضی و پردازش سیگنال با دقت بالا طراحی شده‌اند. ساختار کلی این بلاک‌ها شامل اجزای مختلفی است که در زیر توضیح داده می‌شود:

۱. معماری: DSP48 از یک معماری ساده و قابل تنظیم استفاده می‌کند این معماری به طور کلی به اجزایی مانند ضرب‌کننده، جمع‌کننده، شیفت کننده و عملیات های منطقی (AND، OR، XOR، و ...) تقسیم می‌شود.

۲. ماژول‌های ضرب‌کننده: این ماژول‌ها برای انجام عملیات ضرب در DSP48 استفاده می‌شوند. آن‌ها قابلیت انجام ضرب‌های مختلف را با دقت بالا دارند و قابلیت تنظیم پارامترهای مرتبط با ضرب را دارا می‌باشند.

۳. ماژول‌های جمع‌کننده: این ماژول‌ها برای انجام عملیات جمع و تفریق در DSP48 استفاده می‌شوند. آن‌ها می‌توانند جمع و تفریق عددهای دودویی و عددهای سریالی را انجام دهند.

۴. شیفت کننده‌ها: این اجزا برای انجام عملیات شیفت (شیفت به چپ یا به راست) بر روی داده‌های ورودی استفاده می‌شوند.

۵. عملیات منطقی: این اجزا برای انجام عملیات منطقی (مانند AND، OR، XOR و ...) بر روی داده‌های ورودی استفاده می‌شوند. این منطق‌ها اغلب در کنار عملیات‌های ریاضی مورد استفاده قرار می‌گیرند.

۶. ورودی و خروجی‌ها: DSP48 دارای ورودی‌ها و خروجی‌هایی است که به اجزای مختلف این بلاک‌ها متصل می‌شوند. این ورودی‌ها و خروجی‌ها عموماً به صورت سیگنال‌های دودویی یا سریالی عمل می‌کنند.

ساختار کلی DSP48 را در تصویر زیر مشاهده می‌کنید.

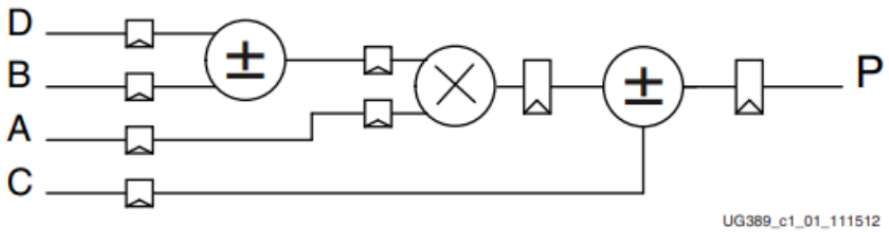


Figure 1-1: **Simplified DSP48A1 Slice with Pre-Adder**

ب

در فایل Q3.v، یک ماژول برای پیاده‌سازی یک ساختار پایپ لاین‌شده با شکل زیر طراحی شده است. این ساختار از جمعاً سه طبقه تشکیل شده است، هرکدام از این طبقات به‌صورت پایپ‌لاین پیاده‌سازی شده است. ورودی‌های این ماژول به طول ۱۸ بیت و خروجی آن به طول ۴۸ بیت است.

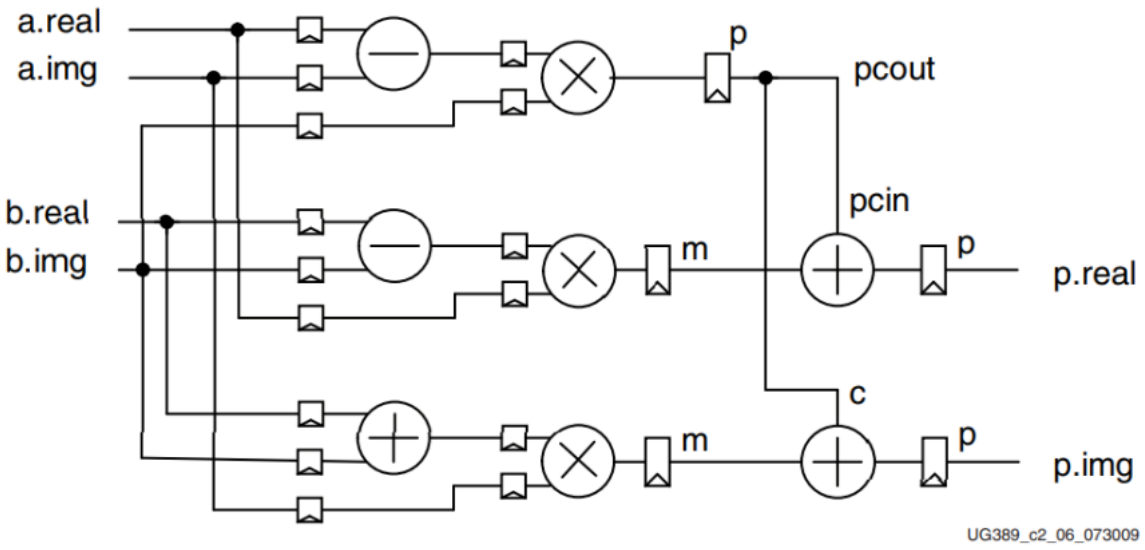
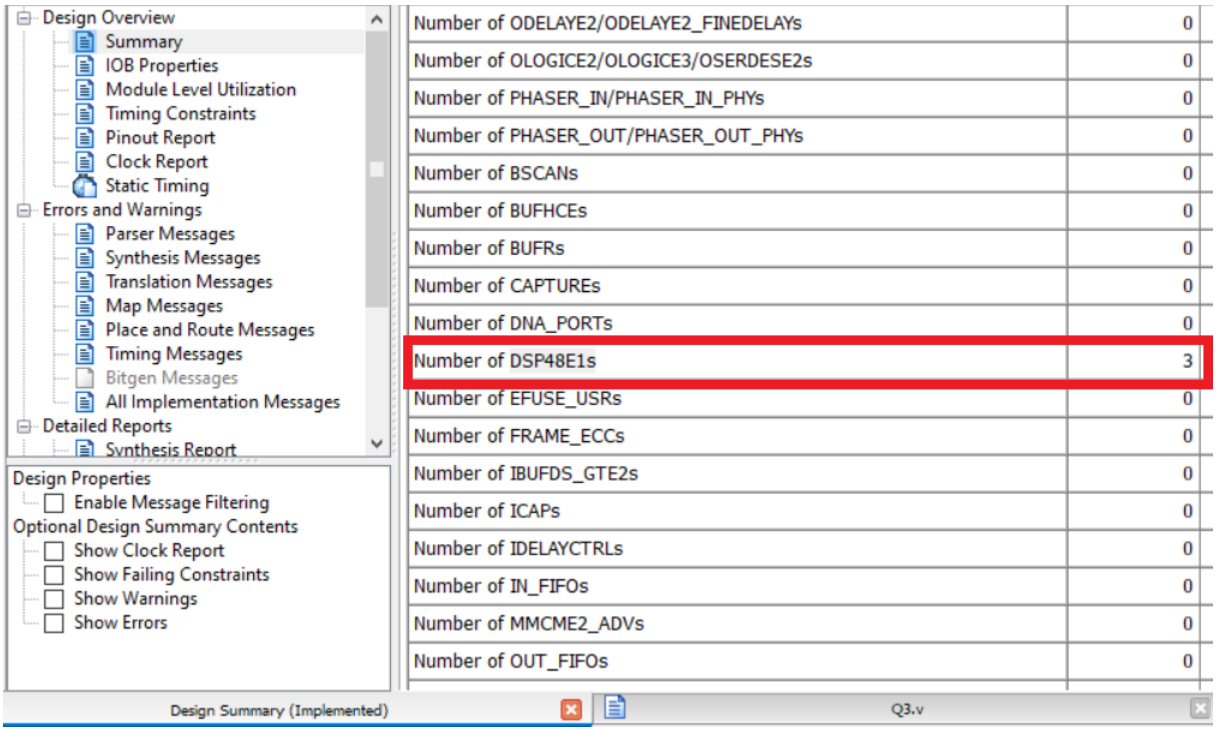


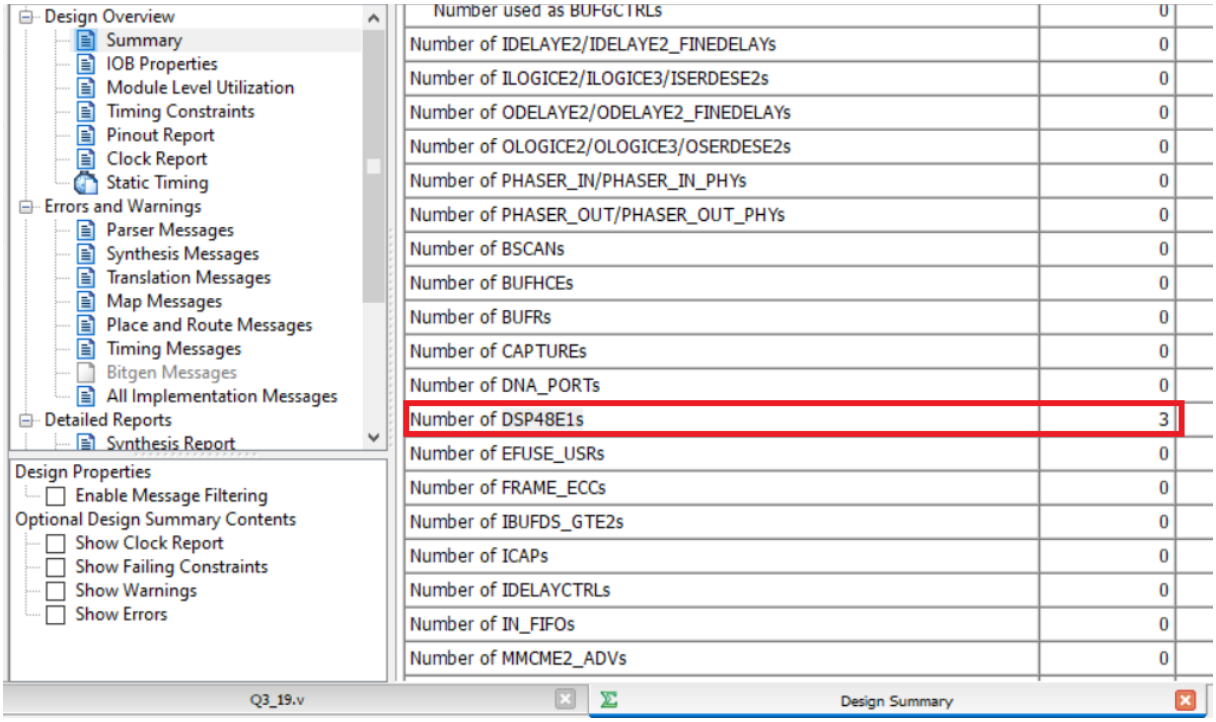
Figure 2-6: **Three-Multiplier Complex Version**

انتخاب این ساختار به دلیل کمینه بودن تعداد DSP ها که به سه واحد محدود شده است، صورت گرفته است.



ج

تعداد DSP ها برای ورودی ۱۹ بیتی تغییری نمی‌کند، زیرا همانطور که در کد ماژول Q3 مشاهده می‌کنید خروجی ۴۸ بیتی در نظر گرفته شده است. پس در این حالت هم تعداد DSP ها سه تا می باشد.



د

در فایلQ3_TB.v ، ما تست‌های مربوط به این ماژول را قرار داده‌ایم. در این تست، ما ۱۰۰ ورودی تصادفی تولید کرده و آنها را به ماژول ارسال می‌کنیم. سپس خروجی را با نتایج مورد انتظار مقایسه می‌کنیم. پس از اینکه تمامی تست‌ها پاس شوند، عبارت All Tests Passed! را چاپ می‌کنیم.

```
# Test          91:
#    A = (  -9570 + -105219j)
#    B = ( -77189 +   29583j)
# ----------------------------------------------------------------
# Test          92:
#    A = ( -26109 +   76259j)
#    B = (  17181 +   81329j)
# ----------------------------------------------------------------
# Test          93:
#    A = ( 106308 +  -82795j)
#    B = ( -52000 +  -34579j)
# ----------------------------------------------------------------
# Test          94:
#    A = (   5458 +   23544j)
#    B = (  46989 +  -15278j)
# ----------------------------------------------------------------
# Test          95:
#    A = ( 124804 +   64838j)
#    B = ( -93556 +   32144j)
# ----------------------------------------------------------------
# Test          96:
#    A = (  49431 +   90474j)
#    B = (  87940 +  -17238j)
# ----------------------------------------------------------------
# Test          97:
#    A = (  69500 + -130464j)
#    B = (  19898 +  -40821j)
# ----------------------------------------------------------------
# Test          98:
#    A = (   -346 +  -57819j)
#    B = (  99122 +  -64350j)
# ----------------------------------------------------------------
# Test          99:
#    A = ( -33719 +  -85996j)
#    B = ( -16323 +     843j)
# ----------------------------------------------------------------
# All Tests Passed!
# ** Note: $stop    : E:/University/Semester8/FPGA/HomeWorks/HW2/Q3/Q3_TB.v(77)
#    Time: 1012 ns  Iteration: 0  Instance: /Q3_TB
# Break at E:/University/Semester8/FPGA/HomeWorks/HW2/Q3/Q3_TB.v line 77
```

سوال چهار

الف

این ماژول یک sequence detector را پیاده‌سازی می‌کند که دریافت یک رشته بیتی به طول ۸ بیت (۱۰۱۱۰۱۱۰) را تشخیص می‌دهد. ماژول شامل یک ماشین حالت است که دو حالت دارد: S1 و S2. در حالت S1، ماشین در حالت اولیه قرار دارد و اگر ورودی معتبر باشد، بیت‌های ورودی به یک شیفت‌رجیستر اضافه شده و در صورتی که الگوی ۱۰۱۱۰۱۱۰ به دست آمده باشد، به حالت S2 منتقل می‌شود؛ در غیر این صورت در همان حالت باقی می‌ماند. در حالت S2، ماشین در حالت نهایی قرار دارد و همواره بیت خروجی را به ۱ تنظیم می‌کند.

همچنین ماژول دارای یک شیفت‌رجیستر است که در هر کلاک، بیت ورودی جدید را دریافت کرده و به آن اضافه می‌کند. اگر الگوی موردنظر (۱۰۱۱۰۱۱۰) در شیفت‌رجیستر مشاهده شود، ماشین به حالت S2 منتقل می‌شود. در هر حالت، بیت خروجی براساس حالت ماشین تنظیم می‌شود، به این ترتیب که در حالت S1 بیت خروجی به ۰ تنظیم می‌شود و در حالت S2 به ۱. در صورتی که ریست فعال شود، ماشین به حالت اولیه باز می‌گردد و شیفت‌رجیستر و بیت خروجی صفر می‌شوند.

ب

در تصویر زیر صحت عملکرد ماژول قسمت قبل را مشاهده می‌کنید.

```
VSIM 65> run -all
# Time =                  5000, bit_in = 0, bit_out = 0, sequence = 00000000
# Time =                 15000, bit_in = 0, bit_out = 0, sequence = 00000000
# Time =                 25000, bit_in = 0, bit_out = 0, sequence = 00000000
# Time =                 35000, bit_in = 0, bit_out = 0, sequence = 00000000
# Time =                 45000, bit_in = 1, bit_out = 0, sequence = 00000000
# Time =                 55000, bit_in = 0, bit_out = 0, sequence = 00000001
# Time =                 65000, bit_in = 1, bit_out = 0, sequence = 00000010
# Time =                 75000, bit_in = 1, bit_out = 0, sequence = 00000101
# Time =                 85000, bit_in = 0, bit_out = 0, sequence = 00001011
# Time =                 95000, bit_in = 1, bit_out = 0, sequence = 00010110
# Time =                105000, bit_in = 1, bit_out = 0, sequence = 00101101
# Time =                115000, bit_in = 0, bit_out = 0, sequence = 01011011
# Time =                125000, bit_in = 1, bit_out = 0, sequence = 10110110
# Time =                135000, bit_in = 1, bit_out = 1, sequence = 01101101
# Time =                145000, bit_in = 0, bit_out = 0, sequence = 11011011
# Time =                155000, bit_in = 0, bit_out = 0, sequence = 10110110
# Time =                165000, bit_in = 0, bit_out = 1, sequence = 01101100
# Time =                175000, bit_in = 0, bit_out = 0, sequence = 11011000
# Time =                185000, bit_in = 0, bit_out = 0, sequence = 10110000
# ** Note: $stop    : E:/University/Semester 8/FPGA/HomeWorks/HW2/Q4/Q4_Moore_TB.v(59)
#    Time: 190 ns  Iteration: 0  Instance: /testbench
# Break at E:/University/Semester 8/FPGA/HomeWorks/HW2/Q4/Q4_Moore_TB.v line 59
```

ج

در Q4_Mealy ماشین حالت میلی را پیاده‌سازی کردم و با تست بنچ طراحی شده صحت آن را بررسی کردم. نتیجه را در تصویر زیر مشاهده می‌کنید.

```
VSIM 2> run -all
# Time =                5000, bit_in = 0, bit_out = 0, shift_reg = 00000000
# Time =               15000, bit_in = 0, bit_out = 0, shift_reg = 00000000
# Time =               25000, bit_in = 0, bit_out = 0, shift_reg = 00000000
# Time =               35000, bit_in = 0, bit_out = 0, shift_reg = 00000000
# Time =               45000, bit_in = 1, bit_out = 0, shift_reg = 00000001
# Time =               55000, bit_in = 0, bit_out = 0, shift_reg = 00000010
# Time =               65000, bit_in = 1, bit_out = 0, shift_reg = 00000101
# Time =               75000, bit_in = 1, bit_out = 0, shift_reg = 00001011
# Time =               85000, bit_in = 0, bit_out = 0, shift_reg = 00010110
# Time =               95000, bit_in = 1, bit_out = 0, shift_reg = 00101101
# Time =              105000, bit_in = 1, bit_out = 0, shift_reg = 01011011
# Time =              115000, bit_in = 0, bit_out = 1, shift_reg = 10110110
# Time =              125000, bit_in = 1, bit_out = 0, shift_reg = 01101101
# Time =              135000, bit_in = 1, bit_out = 0, shift_reg = 11011011
# Time =              145000, bit_in = 0, bit_out = 1, shift_reg = 10110110
# Time =              155000, bit_in = 0, bit_out = 0, shift_reg = 01101100
# Time =              165000, bit_in = 0, bit_out = 0, shift_reg = 11011000
# Time =              175000, bit_in = 0, bit_out = 0, shift_reg = 10110000
# Time =              185000, bit_in = 0, bit_out = 0, shift_reg = 01100000
# ** Note: $stop    : E:/University/Semester 8/FPGA/HomeWorks/HW2/Q4/Q4_Mealy_TB.v(59)
#    Time: 190 ns  Iteration: 0  Instance: /Q4_Mealy_TB
# Break at E:/University/Semester 8/FPGA/HomeWorks/HW2/Q4/Q4_Mealy_TB.v line 59
```

د

در دو pdf موجود در پوشه‌ی مربوط به این سوال با جزییات کامل منابع مصرفی این دو پیاده‌سازی را می‌توانید مشاهده کنید. در زیر برخی از این تفاوت ها را بیان می‌کنم.

1. Slice Logic Utilization: Mealy uses 10 slice registers while Moore uses 9 slice registers. Mealy uses 9 LUTs for logic while Moore uses 2 LUTs for logic.
3. LUT-FF pairs used: Mealy uses 10 LUT-FF pairs while Moore uses 8 LUT-FF pairs.
4. Number of unique control sets: Mealy uses 2 unique control sets while Moore uses 1 unique control set.
5. Number of routing stages: Mealy shows higher route-through LUT usage (3) compared to Moore (0).
6. Average fanout of non-clock nets: Mealy has a higher average fanout of 3.25 compared to 2.53 for Moore.

سوال پنج

الف

کد MATLAB (data_generator) یک الگوریتم برای تولید و ذخیره‌سازی داده‌های ۱۱ بیتی و داده‌های ۱۵ بیتی با استفاده از کد همینگ را ارائه می‌دهد.
در این کد، ابتدا صد داده تصادفی از داده‌های ۱۱ بیتی تولید می‌شود. سپس برای هر مجموعه از این داده‌ها، بیت‌های parity با استفاده از کد همینگ محاسبه می‌شود. بیت‌های parity در موقعیت‌های ۱، ۲، ۴ و ۸ از داده‌های ۱۵ بیتی قرار می‌گیرند. داده‌های ۱۱ بیتی متناظر نیز در قسمت‌های باقی‌مانده از داده‌های ۱۵ بیتی قرار می‌گیرند.
در مرحله بعدی، داده‌های ۱۱ بیتی و ۱۵ بیتی به ترتیب در فایل‌های data_11_bit.txt و data_15_bit.txt ذخیره می‌شوند.
این الگوریتم به این ترتیب عمل می‌کند که ابتدا داده‌های تصادفی تولید شده را دریافت کرده، بیت‌های parity را برای هر مجموعه از داده‌ها محاسبه و به داده‌های ۱۵ بیتی اضافه می‌کند، سپس داده‌های ۱۱ بیتی و ۱۵ بیتی را در فایل‌ها ذخیره می‌کند.

Bit position		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11
Parity bit coverage	p1	✗		✗		✗		✗		✗		✗		✗		✗
	p2		✗	✗			✗	✗			✗	✗			✗	✗
	p4				✗	✗	✗	✗					✗	✗	✗	✗
	p8								✗	✗	✗	✗	✗	✗	✗	✗
	p16															

ب

کد وریلاگ خواسته شده ماژول با نام DataVerifier است که داده‌های ورودی ۱۵ بیتی را دریافت می‌کند. این ماژول برای بررسی صحت داده‌ها از بیت‌های parity استفاده می‌کند که با استفاده از آن‌ها اعتبار داده‌ها را بررسی می‌کند. در صورتی که داده‌ها صحیح باشند، این ماژول داده‌های ۱۱ بیتی را در خروجی خود قرار می‌دهد و سیگنال valid را به مدت یک کلاک فعال می‌کند. اما اگر داده‌ها نامعتبر باشند، داده‌های ورودی ۱۵ بیتی را دور می‌اندازد و به مقدار خروجی error یک مقدار اضافه می‌کند. این ماژول توسط یک کلاک کنترل می‌شود که نیاز است که از خارج از ماژول به آن ارسال شود.

```
reg [3:0] parity_bits;
reg [10:0] data_temp;
reg valid_temp;

// Calculate parity bits
always @* begin
    parity_bits[0] = data_in[0] ^ data_in[2] ^ data_in[4] ^ data_in[6] ^ data_in[8] ^ data_in[10] ^ data_in[12] ^ data_in[14];
    parity_bits[1] = data_in[1] ^ data_in[2] ^ data_in[5] ^ data_in[6] ^ data_in[9] ^ data_in[10] ^ data_in[13] ^ data_in[14];
    parity_bits[2] = data_in[3] ^ data_in[4] ^ data_in[5] ^ data_in[6] ^ data_in[11] ^ data_in[12] ^ data_in[13] ^ data_in[14];
    parity_bits[3] = data_in[7] ^ data_in[8] ^ data_in[9] ^ data_in[10] ^ data_in[11] ^ data_in[12] ^ data_in[13] ^ data_in[14];
end

// Check parity
always @* begin
    if (parity_bits == 4'b0000) begin
        // Parity is correct
        data_temp = {data_in[2], data_in[6:4], data_in[14:8]};
        valid_temp = 1;
        error = 0;
    end
    else begin
        // Parity is incorrect
        valid_temp = 0;
        error = 1;
    end
end

// Output data and valid signal with one clock delay
always @(posedge clk) begin
    if (valid_temp) begin
        data_out <= data_temp;
        valid <= 1;
    end
    else begin
        data_out <= 11'b0;
        valid <= 0;
    end
```

ج

در تست بنچ نوشته شده در فایل Q5_TB.v ابتدا فایل text مربوط به داده‌های ۱۵ بیتی را باز می‌کنیم و سپس خط به خط دیتای ورودی را استخراج می‌کنیم. پس از تاخیر زمانی اندکی، دیتای ورودی و خروجی و valid بودن آن را نمایش می‌دهیم. همانطور که انتظار داشتیم، تمامی دیتاها معتبر است.

```
# data_in: 1001110010011l0, data_out: 11001001110, valid: 1
# data_in: 1100011001010l0, data_out: 00101100011, valid: 1
# data_in: 1100000000111l1, data_out: 10011100000, valid: 1
# data_in: 1111010101010l0, data_out: 00101111010, valid: 1
# data_in: 0011000001011l0, data_out: 10100011000, valid: 1
# data_in: 1111111101010l0, data_out: 00101111111, valid: 1
# data_in: 0101111111110l0, data_out: 01110101111, valid: 1
# data_in: 0111001000l0011, data_out: 00010111001, valid: 1
# data_in: 1001110001l0110, data_out: 10111001110, valid: 1
# data_in: 1000111010000l0, data_out: 01001000111, valid: 1
# data_in: 0101111100l1011, data_out: 00010101111, valid: 1
# data_in: 1001011001011l1, data_out: 10101001011, valid: 1
# data_in: 1000111011l0100, data_out: 11111000111, valid: 1
# data_in: 1100101000001l0, data_out: 10001100101, valid: 1
# data_in: 1101100011l0101, data_out: 11111101100, valid: 1
# data_in: 1100101011000l0, data_out: 01101100101, valid: 1
# data_in: 1101110111011l0, data_out: 11101101110, valid: 1
# data_in: 0001110101111l1, data_out: 10110001110, valid: 1
# data_in: 1010011011111l1, data_out: 01111010011, valid: 1
# data_in: 1110000101001l1, data_out: 10101110000, valid: 1
# data_in: 0011010110100l1, data_out: 01010011010, valid: 1
# data_in: 1100001110100l0, data_out: 01011100001, valid: 1
# data_in: 0101101011001l0, data_out: 11100101101, valid: 1
# data_in: 1101000100011l0, data_out: 10001101000, valid: 1
# data_in: 0111011110001l0, data_out: 11000111011, valid: 1
# data_in: 0110000000000l1, data_out: 00000110000, valid: 1
# data_in: 0110001101101l0, data_out: 10110110001, valid: 1
# data_in: 0000101001100l1, data_out: 00110000101, valid: 1
# data_in: 0001000101110l1, data_out: 00110001000, valid: 1
# data_in: 1111101000110l1, data_out: 00011111101, valid: 1
# data_in: 1111101000110l1, data_out: 00011111101, valid: 1
# ** Note: $stop    : E:/University/Semester 8/FPGA/HomeWorks
#    Time: 2525 ns  Iteration: 0  Instance: /DataVerifier_TB
# Break at E:/University/Semester 8/FPGA/HomeWorks/HW2/Q5/Q5_
```

د

اکنون مطابق خواسته‌ی دستورکار چند داده‌ی آخر را نویزی کردم. مشاهده می کنید که به درستی نامعتبر بودن دیتاها تشخیص داده شده است.

```
# data_in: 100101100101101, data_out: 10101001011, valid: 1
# data_in: 100011101110100, data_out: 11111000111, valid: 1
# data_in: 110010100000100, data_out: 10001100101, valid: 1
# data_in: 110110001110101, data_out: 11111101100, valid: 1
# data_in: 110010101100010, data_out: 01101100101, valid: 1
# data_in: 110111011101110, data_out: 11101101110, valid: 1
# data_in: 000111010111101, data_out: 10110001110, valid: 1
# data_in: 101001101111001, data_out: 01111010011, valid: 1
# data_in: 111000010100101, data_out: 10101110000, valid: 1
# data_in: 001101011010001, data_out: 01010011010, valid: 1
# data_in: 110000111010010, data_out: 01011100001, valid: 1
# data_in: 010110101100110, data_out: 11100101101, valid: 1
# data_in: 1101000100011110, data_out: 10001101000, valid: 1
# data_in: 011101111000100, data_out: 11000111011, valid: 1
# data_in: 011000000000011, data_out: 00000110000, valid: 1
# data_in: 111000110110110, data_out: 00000000000, valid: 0
# data_in: 100010100110001, data_out: 00000000000, valid: 0
# data_in: 100100010111011, data_out: 00000000000, valid: 0
# data_in: 011110100011011, data_out: 00000000000, valid: 0
# data_in: 011110100011011, data_out: 00000000000, valid: 0
# ** Note: $stop    : E:/University/Semester 8/FPGA/HomeWorks
#    Time: 2525 ns  Iteration: 0  Instance: /DataVerifier_TB
# Break at E:/University/Semester 8/FPGA/HomeWorks/HW2/Q5/Q5_
```