به نام خدا

دانشگاه صنعتی شریف
دانشکده مهندسی برق

طراحی سیستم های مبتنی بر ASIC/FPGA

تمرین پنجم

امیرحسین یاری
۹۹۱۰۲۵۰۷

۱۰ خرداد ۱۴۰۳

فهرست مطالب

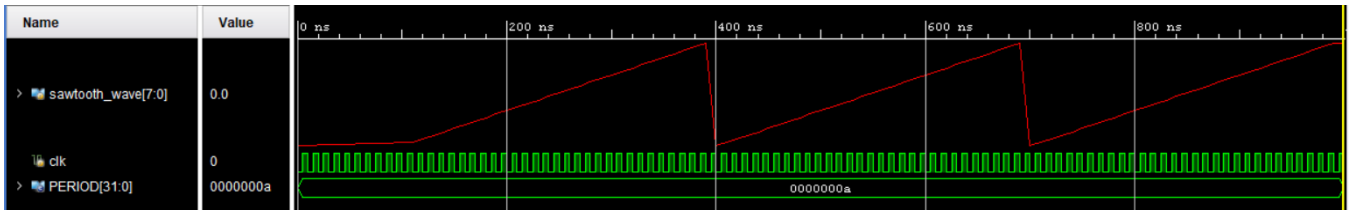
۱ صحت‌سنجی ماژول

الف

تست بنچ نوشته شده برای تولید یک موج دندانه‌اره‌ای (sawtooth) برای ورودی ۸ بیتی ماژول طراحی شده است. در این تست بنچ، دو سیگنال تعریف شده‌اند: sawtooth_wave که ورودی ۸ بیتی است و clk که سیگنال کلاک را نشان می‌دهد.

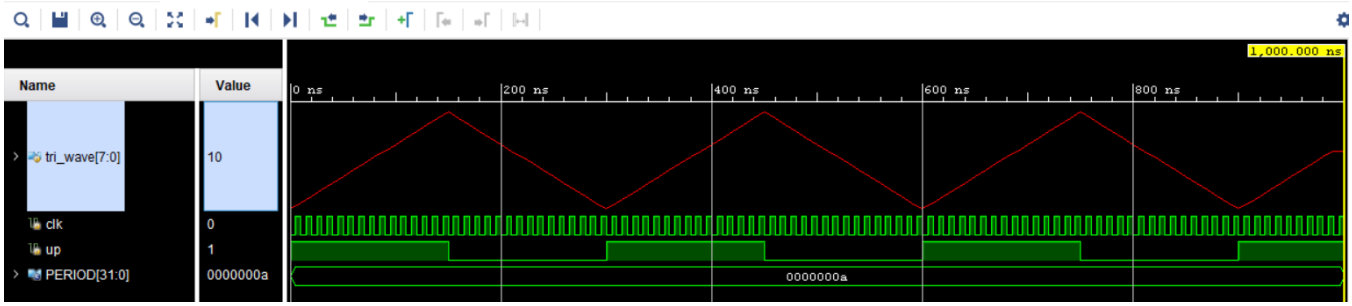
در ابتدا، سیگنال کلاک با پریود ۱۰ نانوثانیه (معادل با فرکانس ۱۰۰ مگاهرتز) تولید می‌شود. سپس، سیگنال sawtooth_wave با مقدار اولیه صفر تنظیم می‌شود. در داخل یک حلقه بی‌نهایت، مقدار sawtooth_wave هر دوره زمانی (PERIOD) یک واحد افزایش می‌یابد. هنگامی که مقدار sawtooth_wave به ۳۰ می‌رسد، مجدداً به صفر مقداردهی می‌شود تا موج دندانه‌اره‌ای ایجاد شود.

در نهایت، یک بخش از تست بنچ برای متوقف کردن شبیه‌سازی بعد از ۱۰۰۰۰ نانوثانیه (۹۰ میلیون سیکل ساعت) اضافه شده است.



ب

تست بنچ نوشته شده برای تولید یک موج مثلثی برای ورودی ۸ بیتی ماژول طراحی شده است. در این تست بنچ، دو سیگنال اصلی tri_wave و clk تعریف شده‌اند. سیگنال ساعت با پریود ۱۰ نانوثانیه تولید می‌شود که معادل فرکانس ۱۰۰ مگاهرتز است. سیگنال tri_wave با مقدار اولیه صفر تنظیم شده و با استفاده از یک فلگ (up) جهت افزایش یا کاهش مقدار آن کنترل می‌شود. در هر دوره زمانی، اگر فلگ up فعال باشد، مقدار tri_wave افزایش یافته و وقتی به ۱۵ برسد، جهت تغییر کرده و شروع به کاهش می‌کند. این فرآیند تکرار شده و یک موج مثلثی با نوسان بین ۰ و ۱۵ ایجاد می‌کند. شبیه‌سازی برای مدت زمان ۹۰۰۰۰۰۰۰ نانوثانیه اجرا شده و سپس متوقف می‌شود.



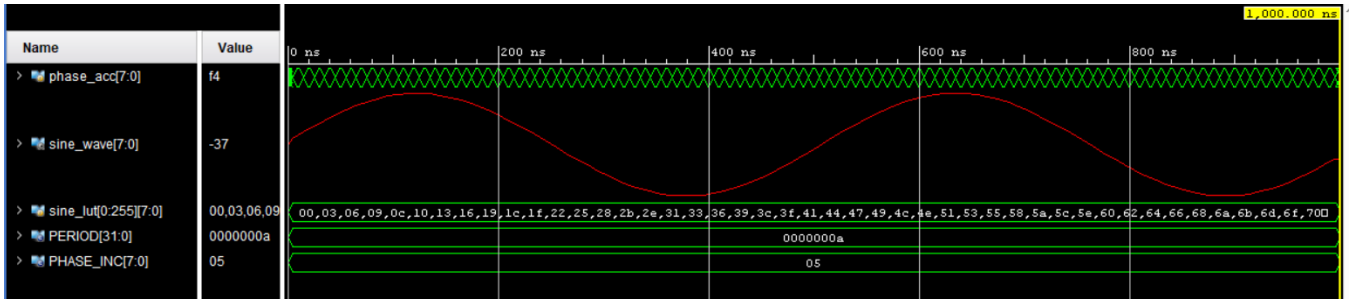
ج

این ماژول از تکنیک DDS برای تولید موج سینوسی استفاده می‌کند. در این تکنیک، یک متغیر به نام phase accumulator به طور ثابت افزایش می‌یابد و مقدار آن به عنوان اندیس برای یک lutاستفاده می‌شود که نمونه‌های موج مورد نظر در آن ذخیره شده‌اند.

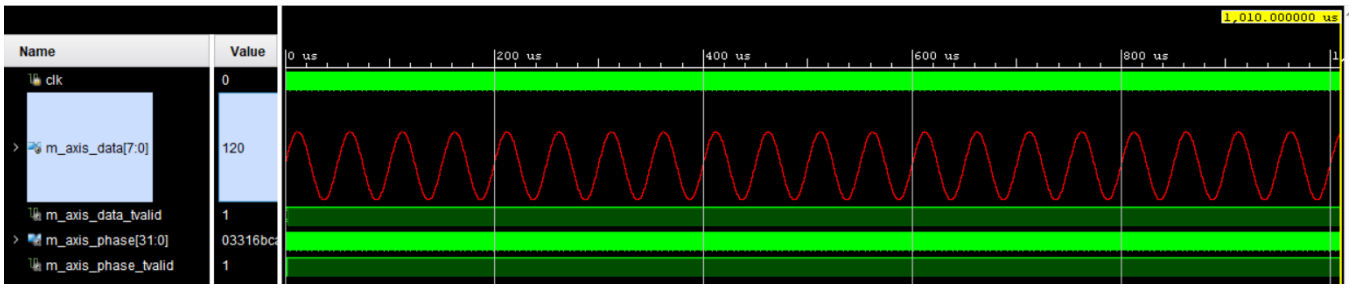
متغیر phase_acc با یک مقدار ثابت PHASE_INC در هر کلاک افزایش می‌یابد. این phase_acc به طور اساسی یک نمایش دیجیتالی از فاز موج سینوسی ایجاد می‌کند.

همچنین، یک lutبه نام sine_lut وجود دارد که نمونه‌های پیش‌پردازش شده موج سینوسی را ذخیره می‌کند. هر نمونه با یک مقدار فاز مشخص متناظر است. با استفاده از مقدار phase_acc به عنوان اندیس در این جدول جستجو، مقدار سینوس متناظر را بازیابی می‌کنیم.

در نهایت، مقدار موج سینوسی به ورودی sine_wave اختصاص داده می‌شود که به عنوان خروجی ماژول عمل می‌کند.



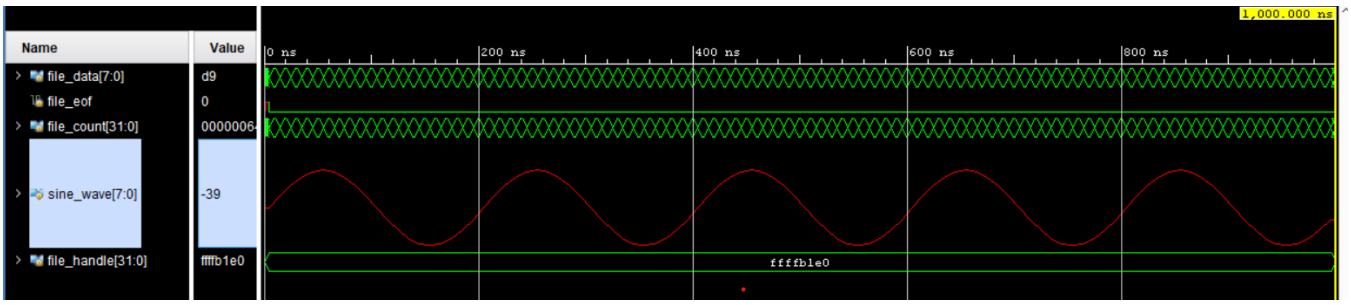
اکنون با استفاده از IP Core نیز سیگنال سینوسی را در ماژول sine_wave_dds_ipcore تولید کردم که خروجی آن را در تصویر زیر مشاهده می‌کنید.



د

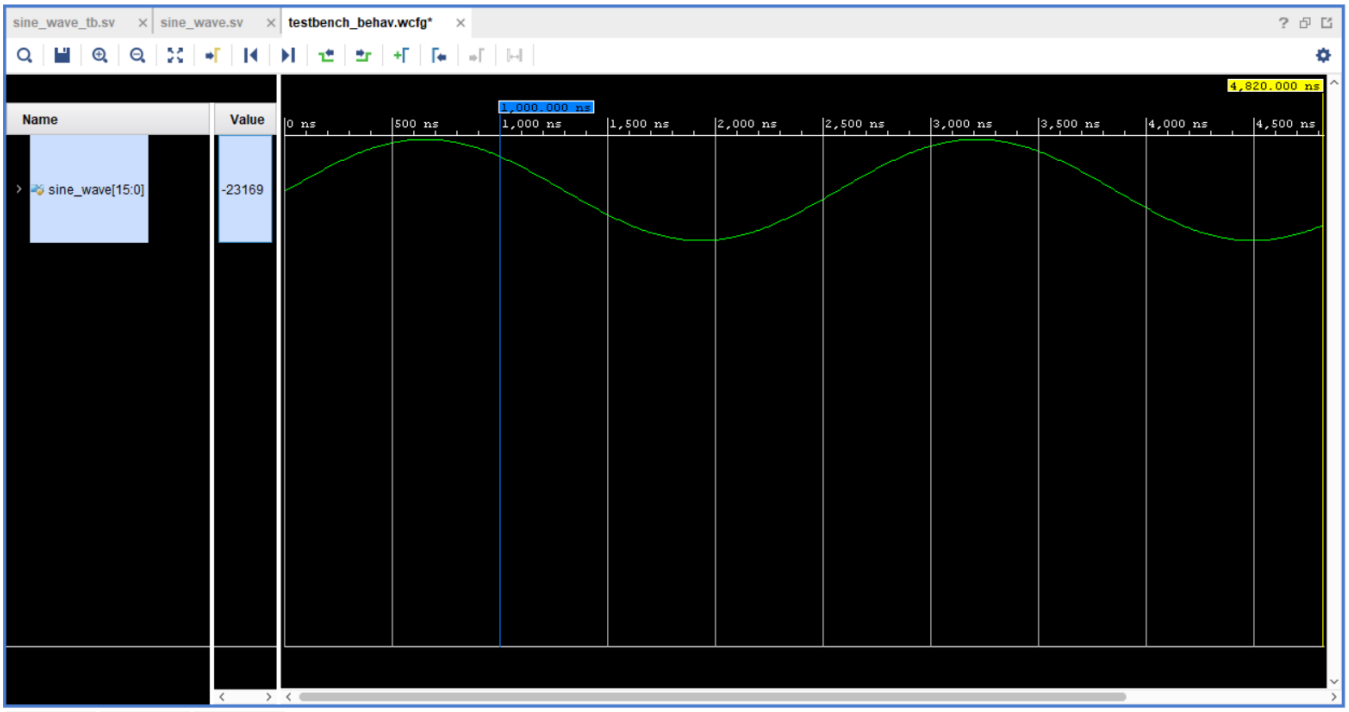
تست‌بنچ نوشته شده به منظور خواندن یک موج سینوسی از یک فایل با نام sinusoidal_signal.txt طراحی شده است. فایل در ابتدای شبیه‌سازی باز می‌شود و اگر فایل قابل باز کردن نباشد، یک پیام خطا نمایش داده می‌شود و شبیه‌سازی متوقف می‌شود. یک سیگنال کلاک ۱۰ نانوثانیه (فرکانس ۱۰۰ مگاهرتز) تولید می‌کنیم. اگر به انتهای فایل رسیده‌باشیم، فلگ پایان فایل تنظیم می‌شود و یک پیام نمایش

داده می‌شود که تعداد کل نمونه‌های خوانده شده را نشان می‌دهد و شبیه‌سازی متوقف می‌شود. داده‌های خوانده شده به خروجی sine_wave اختصاص داده می‌شود.



۵

در این بخش، یک مولد موج سینوسی در SystemVerilog طراحی شده است. این مولد از یک LUT برای ذخیره مقادیر از پیش‌محاسبه شده‌ی سینوسی استفاده می‌کند. ماژول sine_wave_generator شامل یک شمارنده برای تولید اندیس‌های جدول جستجو است و مقادیر سینوسی متناظر با این اندیس‌ها را در هر سیکل کلاک تولید می‌کند. در تست بنچ tb_sine_wave_generator، یک سیگنال کلاک با دوره‌ی ۱۰ واحد زمانی و یک سیگنال ریست تولید می‌شود. همچنین، تست بنچ شامل عملیات فایل برای ذخیره مقادیر تولید شده‌ی موج سینوسی در یک فایل متنی با استفاده از توابع fopen، fdisplay و fclose است. این تست بنچ مقادیر موج سینوسی را در هر لبه‌ی بالارونده‌ی کلاک به فایل sine_wave_output.txt می‌نویسد و پس از ۵۰۰۰ واحد زمانی شبیه‌سازی را خاتمه می‌دهد.



و

با توجه به ماژول waveform_generator_tb که نوشته‌شده، این ماژول یک تست بنچ است که با استفاده از یک پارامتر به نام WAVEFORM_TYPE انواع مختلفی از شکل‌موج‌ها را تولید می‌کند. این پارامتر با تغییر در آن، نوع شکل‌موج تولید شده توسط ماژول را تغییر می‌دهد.

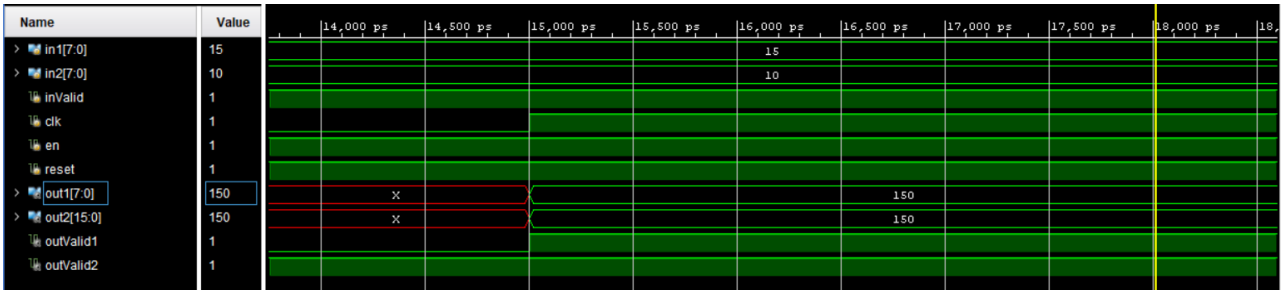
پس اگر مقدار پارامتر WAVEFORM_TYPE را تغییر دهید، ماژول waveform_generator_tb متناسب با مقدار جدید این پارامتر، یکی از ماژول‌های زیر را به عنوان ورودی ماژول اصلی خواهد انتخاب کرد:

۰. sawtooth_tb برای تولید مثلثی
۱. triangular_tb برای تولید مثلثی
۲. sine_wave_dds_ipcore برای تولید سینوسی با استفاده از DDS
۳. sine_wave_file_tb برای تولید سینوسی با استفاده از فایل
۴. tb_sine_wave_generator برای تولید سینوسی با استفاده از system task

۲ صحت‌سنجی ضرب‌کننده‌ها با ۴ سناریو

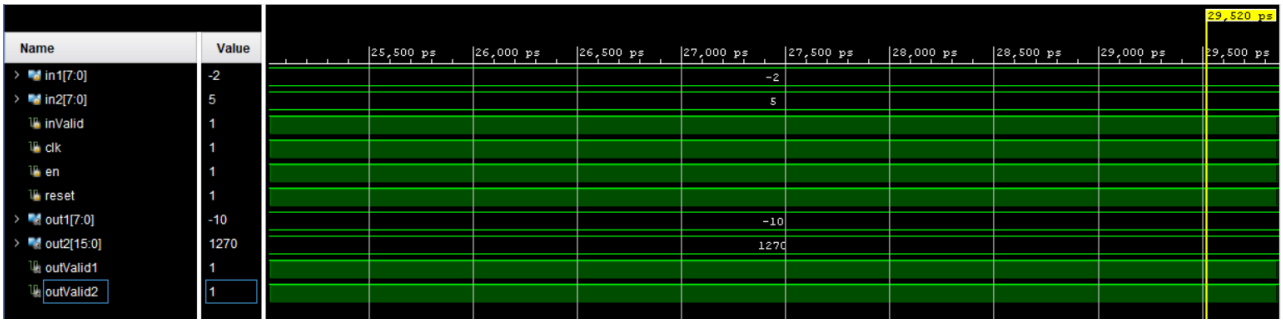
تست بنچ نوشته‌شده شامل چهار سناریوی مختلف برای تست کردن ماژول‌ها است. در ابتدا، سیگنال‌های مورد نیاز مانند clk، reset، en، inValid، in1 و in2 مقداردهی اولیه می‌شوند. سپس چهار سناریوی زیر اجرا می‌شوند:

۱. ضرب unsigned: بررسی می‌شود که آیا ماژول‌ها می‌توانند دو عدد unsigned را به درستی ضرب کنند.



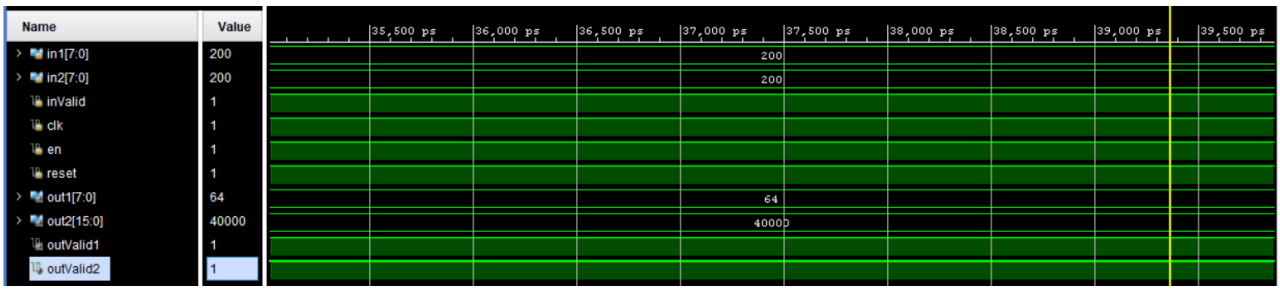
مشاهده می‌کنید که هر دو ماژول به درستی کار می‌کنند.

۲. ضرب signed: بررسی می‌شود که آیا ماژول‌ها می‌توانند دو عدد signed را به درستی ضرب کنند.



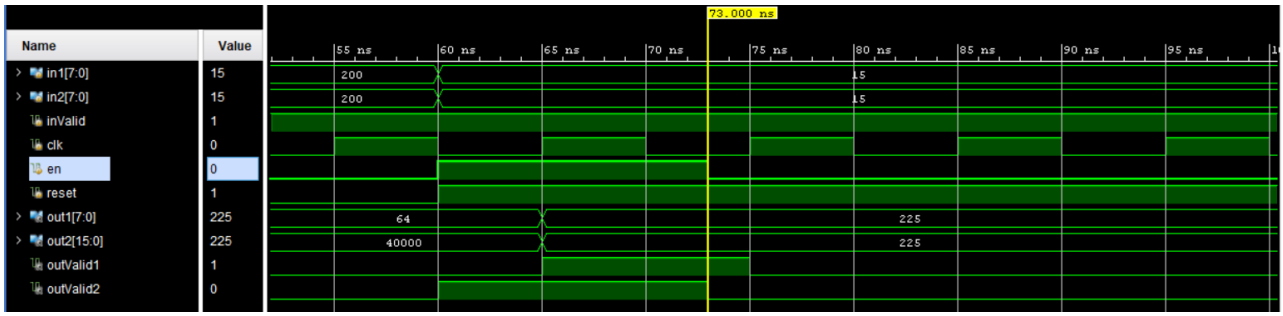
مشاهده می‌کنید که ماژول اول به درستی جواب را محاسبه کرده اما ماژول دوم جواب را نادرست بدست آورده است.

۳. بررسی می‌شود که آیا ماژول‌ها به درستی با اعداد بزرگ که ممکن است باعث overflow شوند، کار می‌کنند.



طبق نتایج بدست آمده ماژول اول overflow کرده و جواب اشتباه می‌دهد اما ماژول دوم به درستی عمل کرده‌است.

۴. عملکرد ماژول‌ها در مواجهه با تغییرات سیگنال‌های enable و reset بررسی می‌شود.



همانطور که مشاهده می‌کنید با صفر شدن en ماژول دوم بلافاصله صفر شده و به عبارتی آسنکرون عمل می‌کند، اما ماژول اول در لبه‌ی کلاک صفر شده و به عبارتی بصورت سنکرون عمل می‌کند.

۳ ترتیب آمدن ورودی‌ها

الف

ماژول ControlSignalDecoder، ماژول خواسته شده در صورت مسئله است.

ب

در ماژول ControlSignalDecoder_tb همانند سناریو گفته‌شده عمل کرده‌ایم و همانطور که انتظار داشتیم مطابق شکل زیر نتایج صحیحی را بدست آوردیم.

```
# out_signal1 = 10, out_signal2 = 01, out_signal3 = 00, out_timing1 =  363, out_timing2 = 1001, out_timing3 = 1923
# out_signal1 = 10, out_signal2 = 01, out_signal3 = 00, out_timing1 =  363, out_timing2 = 1001, out_timing3 = 1924
# out_signal1 = 10, out_signal2 = 01, out_signal3 = 00, out_timing1 =  363, out_timing2 = 1001, out_timing3 = 1925
# out_signal1 = 10, out_signal2 = 01, out_signal3 = 00, out_timing1 =  363, out_timing2 = 1001, out_timing3 = 1926
# out_signal1 = 10, out_signal2 = 01, out_signal3 = 00, out_timing1 =  363, out_timing2 = 1001, out_timing3 = 1927
# out_signal1 = 10, out_signal2 = 01, out_signal3 = 00, out_timing1 =  363, out_timing2 = 1001, out_timing3 = 1928
# out_signal1 = 10, out_signal2 = 01, out_signal3 = 00, out_timing1 =  363, out_timing2 = 1001, out_timing3 = 1929
# out_signal1 = 10, out_signal2 = 01, out_signal3 = 00, out_timing1 =  363, out_timing2 = 1001, out_timing3 = 1930
# out_signal1 = 10, out_signal2 = 01, out_signal3 = 00, out_timing1 =  363, out_timing2 = 1001, out_timing3 = 1931
# out_signal1 = 10, out_signal2 = 01, out_signal3 = 00, out_timing1 =  363, out_timing2 = 1001, out_timing3 = 1932
# out_signal1 = 10, out_signal2 = 01, out_signal3 = 00, out_timing1 =  363, out_timing2 = 1001, out_timing3 = 1933
# out_signal1 = 10, out_signal2 = 01, out_signal3 = 00, out_timing1 =  363, out_timing2 = 1001, out_timing3 = 1934
# out_signal1 = 10, out_signal2 = 01, out_signal3 = 00, out_timing1 =  363, out_timing2 = 1001, out_timing3 = 1935
# out_signal1 = 10, out_signal2 = 01, out_signal3 = 11, out_timing1 =  363, out_timing2 = 1001, out_timing3 = 1936
# ** Note: $stop    : E:/University/Semester8/FPGA/HomeWorks/HW5/Q3/Q3_tb.v(57)
#    Time: 8751 ns  Iteration: 0  Instance: /ControlSignalDecoder_tb
# Break at E:/University/Semester8/FPGA/HomeWorks/HW5/Q3/Q3_tb.v line 57
```

ج

سناریوی خواسته‌شده را در ماژول ControlSignalDecoder2_tb پیاده‌سازی کردیم که نتیجه‌اش را در تصویر زیر مشاهده و صحت عملکرد آن را تایید می‌کنیم.

```
# out_signal1 = 00, out_signal2 = 00, out_signal3 = 00, out_timing1 =    0, out_timing2 =    0, out_timing3 =    0
# out_signal1 = 00, out_signal2 = 00, out_signal3 = 00, out_timing1 =    1, out_timing2 =    1, out_timing3 =    1
# out_signal1 = 00, out_signal2 = 00, out_signal3 = 00, out_timing1 =    2, out_timing2 =    2, out_timing3 =    2
# out_signal1 = 10, out_signal2 = 00, out_signal3 = 00, out_timing1 =    3, out_timing2 =    3, out_timing3 =    3
# out_signal1 = 10, out_signal2 = 00, out_signal3 = 00, out_timing1 =    3, out_timing2 =    4, out_timing3 =    4
# out_signal1 = 10, out_signal2 = 01, out_signal3 = 00, out_timing1 =    3, out_timing2 =    5, out_timing3 =    5
# out_signal1 = 10, out_signal2 = 01, out_signal3 = 00, out_timing1 =    3, out_timing2 =    5, out_timing3 =    6
# out_signal1 = 00, out_signal2 = 00, out_signal3 = 00, out_timing1 =    0, out_timing2 =    0, out_timing3 =    0
# out_signal1 = 00, out_signal2 = 00, out_signal3 = 00, out_timing1 =    1, out_timing2 =    1, out_timing3 =    1
# out_signal1 = 00, out_signal2 = 00, out_signal3 = 00, out_timing1 =    2, out_timing2 =    2, out_timing3 =    2
# out_signal1 = 10, out_signal2 = 00, out_signal3 = 00, out_timing1 =    3, out_timing2 =    3, out_timing3 =    3
# out_signal1 = 10, out_signal2 = 00, out_signal3 = 00, out_timing1 =    3, out_timing2 =    4, out_timing3 =    4
# out_signal1 = 10, out_signal2 = 01, out_signal3 = 00, out_timing1 =    3, out_timing2 =    5, out_timing3 =    5
# out_signal1 = 10, out_signal2 = 01, out_signal3 = 00, out_timing1 =    3, out_timing2 =    5, out_timing3 =    6
# out_signal1 = 10, out_signal2 = 01, out_signal3 = 11, out_timing1 =    3, out_timing2 =    5, out_timing3 =    7
# ** Note: $stop    : E:/University/Semester8/FPGA/HomeWorks/HW5/Q3/Q3_second_tb.v(68)
#    Time: 72 ns  Iteration: 0  Instance: /ControlSignalDecoder2_tb
# Break at E:/University/Semester8/FPGA/HomeWorks/HW5/Q3/Q3_second_tb.v line 68
```

۴ ماژول PacketChecker

در ابتدا ماژول PacketChecker همانند توضیحات داده شده طراحی می‌کنیم.

```
always @(posedge clk or posedge rst) begin
    if (rst) begin
        ram_address <= 8'b0;
        ram_data <= 16'b0;
        ram_en <= 0;
        error <= 0;
    end else begin
        ram_en <= 0;
        error <= 0;
        header = data_in[3:0];
        address = data_in[11:4];
        data = data_in[27:12];

        if (header == 4'hE) begin
            ram_address <= address;
            ram_data <= data;
            ram_en <= 1;
        end else begin
            error <= 1;
        end
    end
end
```

سپس تست‌بنچی جهت صحت‌سنجی مطابق دستورکار می‌نویسیم و صحت عملکرد مدار را تایید می‌کنیم.

UART ۵

الف

پروتکل UART (Universal Asynchronous Receiver/Transmitter) یکی از پروتکل‌های ارتباطی سریال است که برای انتقال داده‌ها بین دستگاه‌های الکترونیکی استفاده می‌شود. این پروتکل به صورت Asynchronous عمل می‌کند، به این معنا که نیازی به سیگنال کلاک مشترک بین فرستنده و گیرنده ندارد.

اجزای اصلی UART

۱. فرستنده (Transmitter): وظیفه تبدیل داده‌های موازی به داده‌های سریال و ارسال آن‌ها را دارد.

۲. گیرنده (Receiver): وظیفه دریافت داده‌های سریال و تبدیل آن‌ها به داده‌های موازی را بر عهده دارد.

نحوه‌ی سیگنالینگ در UART

در پروتکل UART، داده‌ها به صورت بیت به بیت منتقل می‌شوند. هر بیت در یک بازه زمانی مشخص (که به آن بیت تایم گفته می‌شود) ارسال می‌شود. برای هماهنگی بین فرستنده و گیرنده، از یک بیت شروع (Start Bit)، بیت‌های داده (Data Bits)، بیت پریتی (اختیاری) (Parity Bit) و بیت‌های توقف (Stop Bits) استفاده می‌شود.

مراحل سیگنالینگ:

۱. بیت شروع (Start Bit): قبل از ارسال داده، خط انتقال (TX) در حالت بیکاری (Idle) است و معمولاً در سطح ولتاژ بالا (۱ منطقی). فرستنده با قرار دادن خط در سطح ولتاژ پایین (۰ منطقی) به مدت یک بیت تایم، شروع به ارسال داده می‌کند.

۲. بیت‌های داده (Data Bits): پس از بیت شروع، بیت‌های داده به ترتیب از کم ارزش‌ترین بیت (LSB) تا پر ارزش‌ترین بیت (MSB) ارسال می‌شوند. تعداد بیت‌های داده معمولاً بین ۵ تا ۹ بیت است، بسته به تنظیمات پروتکل.

۳. بیت پریتی (اختیاری) (Parity Bit): این بیت برای تشخیص خطا در داده‌ها استفاده می‌شود. اگر از بیت پاریتی استفاده شود، فرستنده یک بیت پاریتی را بر اساس تعداد بیت‌های ۱ در داده محاسبه و ارسال می‌کند. نوع پاریتی می‌تواند زوج (Even) یا فرد (Odd) باشد.

۴. بیت‌های توقف (Stop Bits): برای نشان دادن پایان یک فریم داده، یک یا دو بیت توقف ارسال می‌شوند. این بیت‌ها خط انتقال را به حالت بیکاری (سطح ولتاژ بالا) باز می‌گردانند.

تنظیمات UART

تنظیمات اصلی UART شامل Baud Rate، تعداد بیت‌های داده، بیت پریتی و تعداد بیت‌های توقف می‌باشند. Baud Rate تعداد بیت‌های ارسال یا دریافت شده در هر ثانیه را تعیین می‌کند و باید بین فرستنده و گیرنده یکسان باشد.

ب

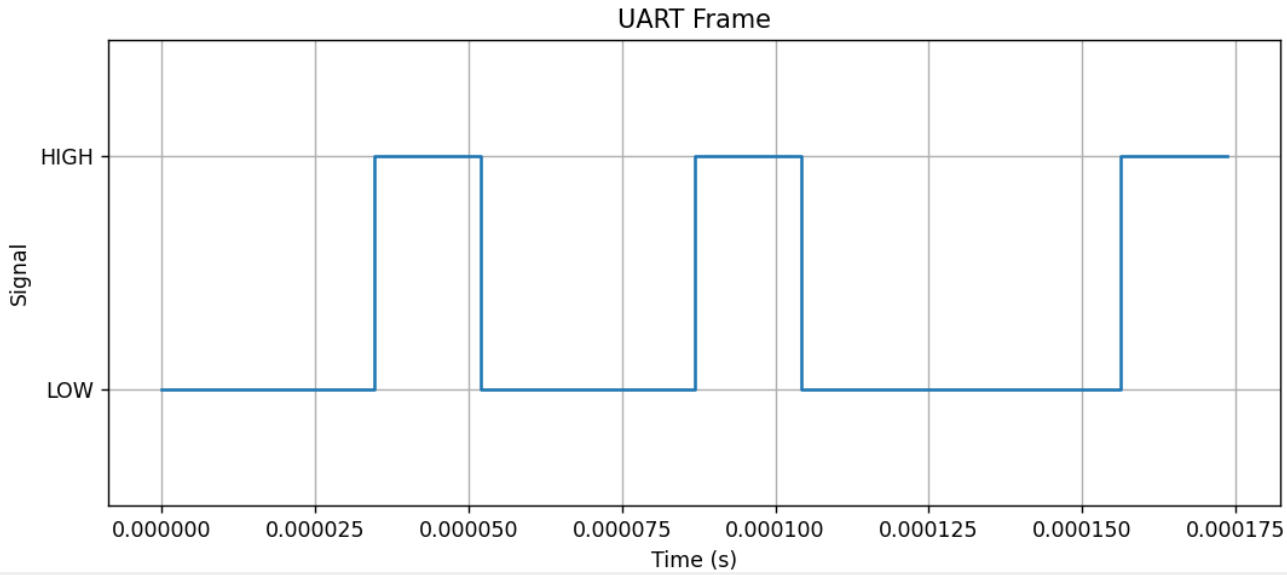
تابع uart_transmit در فایل part2.py یک بایت داده و baud rate را به عنوان ورودی دریافت می‌کند. ابتدا زمان بین هر بیت (bit interval) محاسبه می‌شود. سپس فریم UART که شامل بیت شروع (start bit)، هشت بیت داده (data bits) و بیت توقف (stop bit) است، ساخته می‌شود. این فریم UART به همراه فاصله زمانی بین بیت‌ها به عنوان خروجی بازگردانده می‌شود و در همین حال بیت‌ها یکی یکی با استفاده از time.sleep و فاصله زمانی محاسبه شده ارسال می‌شوند.

در main ابتدا baud rate به ۱۱۵۲۰۰ تنظیم می‌شود. سپس در یک حلقه بی‌نهایت از کاربر درخواست می‌شود تا یک بایت داده (۰ تا ۲۵۵) وارد کند. اگر کاربر exit وارد کند، حلقه متوقف می‌شود و برنامه خاتمه می‌یابد.

داده‌های ورودی بررسی می‌شوند تا مطمئن شوند که در محدوده ۰ تا ۲۵۵ قرار دارند. اگر ورودی معتبر باشد، تابع uart_transmit برای ارسال داده‌ها فراخوانی می‌شود و سپس تابع plot_uart_frame برای رسم نمودار سیگنال UART استفاده می‌شود.

اگر ورودی نامعتبر باشد، پیام خطای مناسب به کاربر نمایش داده می‌شود.

```
Enter a byte to send (0-255), or 'exit' to quit: 18
UART Frame: [0, 0, 1, 0, 0, 1, 0, 0, 0, 1]
Transmitting bit: 0
Transmitting bit: 0
Transmitting bit: 1
Transmitting bit: 0
Transmitting bit: 0
Transmitting bit: 1
Transmitting bit: 0
Transmitting bit: 0
Transmitting bit: 0
Transmitting bit: 1
Enter a byte to send (0-255), or 'exit' to quit: 274
Please enter a valid byte (0-255).
Enter a byte to send (0-255), or 'exit' to quit: exit
```



ج

در فایل part3.py تابع uart_receive یک فریم UART و baud rate را به عنوان ورودی دریافت می‌کند. ابتدا زمان بین هر بیت محاسبه می‌شود. سپس بیت شروع (start bit) بررسی می‌شود تا مطمئن شویم که برابر با ۰ است. در صورت صحیح بودن، بیت‌های داده (data bits) استخراج شده و به یک بایت داده تبدیل می‌شوند. در نهایت، بیت توقف (stop bit) بررسی می‌شود تا برابر با ۱ باشد. اگر بیت شروع یا توقف نادرست باشد، خطای معتبر نبودن بیت مربوطه صادر می‌شود.

تابع generate_uart_frame برای تولید یک فریم UART از یک بایت داده استفاده می‌شود(همانند بخش قبلی).

در تابع main ابتدا baud rate به ۱۱۵۲۰۰ تنظیم می‌شود. سپس در یک حلقه بی‌نهایت از کاربر درخواست می‌شود تا یک بایت داده (۰ تا ۲۵۵) وارد کند. اگر کاربر exit وارد کند، حلقه متوقف می‌شود و برنامه خاتمه می‌یابد.

داده‌های ورودی بررسی می‌شوند تا مطمئن شویم که در محدوده ۰ تا ۲۵۵ قرار دارند. اگر ورودی معتبر باشد، فریم UART با استفاده از generate_uart_frame تولید می‌شود و سپس توسط uart_receive دریافت و به داده اصلی تبدیل می‌شود.

اگر ورودی نامعتبر باشد، پیام خطای مناسب به کاربر نمایش داده می‌شود.

```
PS E:\University\Semester8\FPGA\HomeWorks\HW5\Q5> python part3.py
Enter a byte to send (0-255), or 'exit' to quit: 18
Generated UART Frame: [0, 0, 1, 0, 0, 1, 0, 0, 0, 1]
Received Data Byte: 18
Enter a byte to send (0-255), or 'exit' to quit: 266
Please enter a valid byte (0-255).
Enter a byte to send (0-255), or 'exit' to quit: exit
PS E:\University\Semester8\FPGA\HomeWorks\HW5\Q5>
```

د

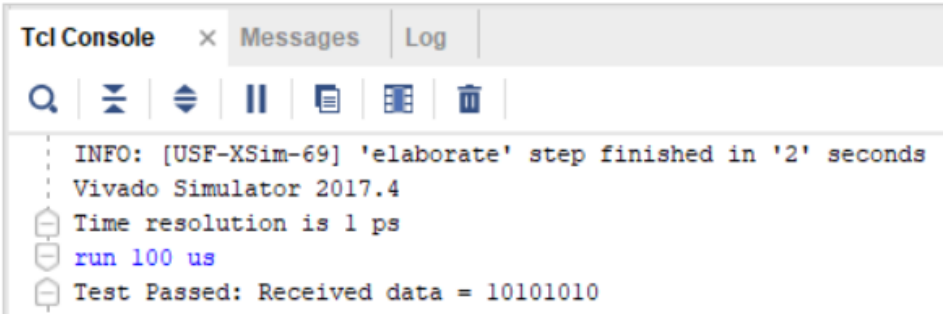
این بخش شامل دو ماژول Verilog برای شبیه‌سازی ارسال و دریافت داده‌ها با استفاده از پروتکل UART است. ماژول uart_tx و uart_rx با baud rate ۱۱۵۲۰۰ و فرکانس کلاک ۵۰ مگاهرتز پیاده‌سازی شده‌اند.

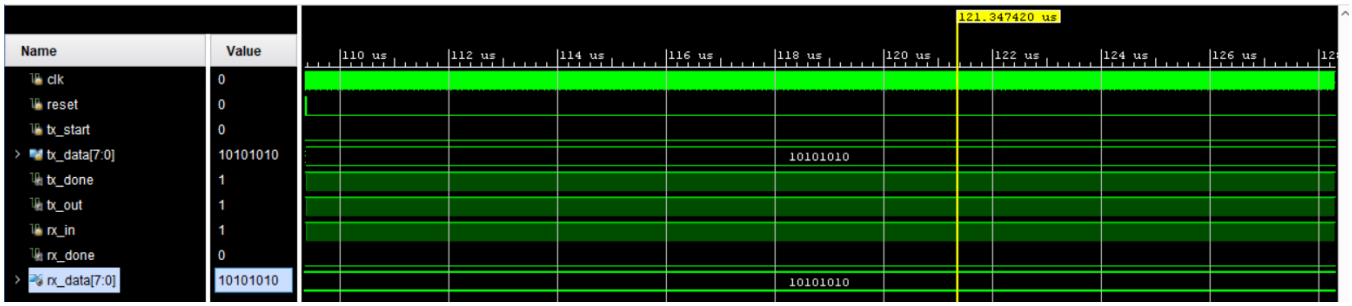
ماژول uart_tx فریم UART را شامل بیت شروع، ۸ بیت داده، و بیت توقف تولید می‌کند. زمانی که tx_start فعال می‌شود، ارسال آغاز شده و بیت‌ها یکی یکی بر روی خط انتقال tx_out ارسال می‌شوند. پس از ارسال همه بیت‌ها، سیگنال tx_done فعال می‌شود.

ماژول uart_rx با تشخیص بیت شروع (۰) در خط دریافت rx_in فعال می‌شود. سپس، بیت‌های داده و بیت توقف را دریافت کرده و در رجیستر قرار می‌دهد. پس از دریافت کامل فریم، سیگنال rx_done فعال شده و داده دریافتی ۸ بیتی در rx_data قرار می‌گیرد.

ماژول تست بنچ uart_tb برای شبیه‌سازی و صحت‌سنجی عملکرد ماژول‌های uart_tx و uart_rx پروتکل UART طراحی شده است. در این تست بنچ، ماژول‌های فرستنده و گیرنده UART با پارامترهای فرکانس کلاک ۵۰ مگاهرتز و baud rate ۱۱۵۲۰۰ به هم متصل شده‌اند، به طوری که خروجی فرستنده tx_out به ورودی گیرنده rx_in وصل شده است.

در این شبیه‌سازی، ابتدا سیگنال‌های اولیه تنظیم می‌شوند و کلاک تولید می‌شود. پس از غیر فعال کردن ریست، یک بایت داده (‘8’b10101010‘) برای ارسال تنظیم می‌شود و سیگنال شروع ارسال tx_start فعال می‌شود. پس از اتمام ارسال و دریافت داده، داده دریافتی با داده ارسالی مقایسه می‌شود. اگر داده دریافتی با داده ارسالی مطابقت داشته باشد، پیام ”Test Passed” نمایش داده می‌شود؛ در غیر این صورت، پیام ”Test Failed” نمایش داده می‌شود.





همانطور که مشاهده می‌کنید ماژول‌های فرستنده و گیرنده به درستی عمل می‌کنند.

۵، و، ز

در این برنامه، یک Test Vector به طول ۲۰۴۸ داده ۸ بیتی تولید می‌شود که شامل اعداد تصادفی از ۰ تا ۲۵۵ است. این داده‌ها به عنوان ورودی‌ها برای تولید فریم‌های UART استفاده می‌شوند. هر فریم UART شامل یک بیت شروع، ۸ بیت داده و یک بیت پایانی است.

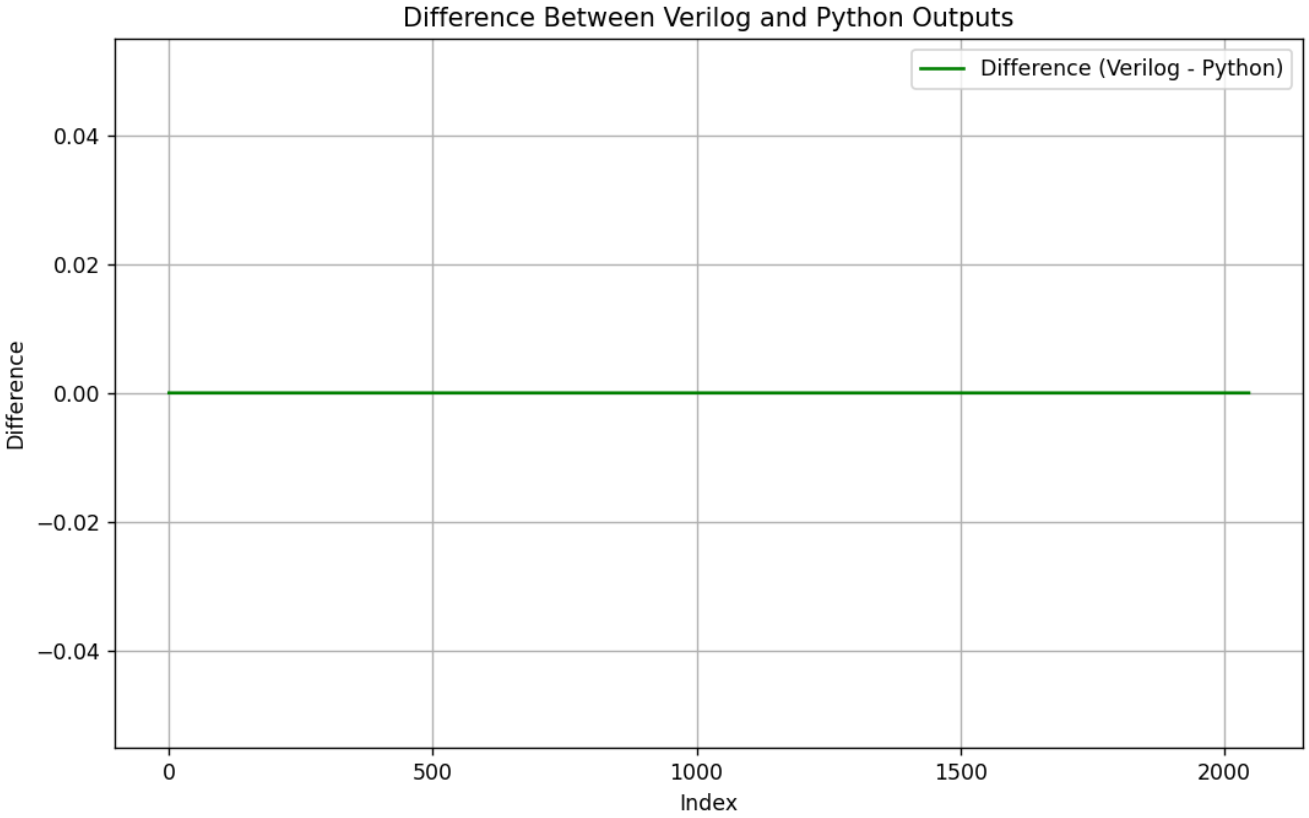
توابع generate_uart_frame و uart_receive به ترتیب برای تولید فریم UART و دریافت داده از فریم UART استفاده می‌شوند. برای هر داده ورودی، یک فریم UART تولید می‌شود و داده متناظر با آن از طریق تابع uart_receive بازیابی می‌شود.

سپس، داده‌های دریافتی از توابع Python و فریم‌های تولید شده به همراه آنها به فایل‌های متنی جداگانه ذخیره می‌شوند. این فایل‌ها شامل خروجی‌های Python در python_outputs.txt و فریم‌های UART در python_uart_frames.txt هستند.

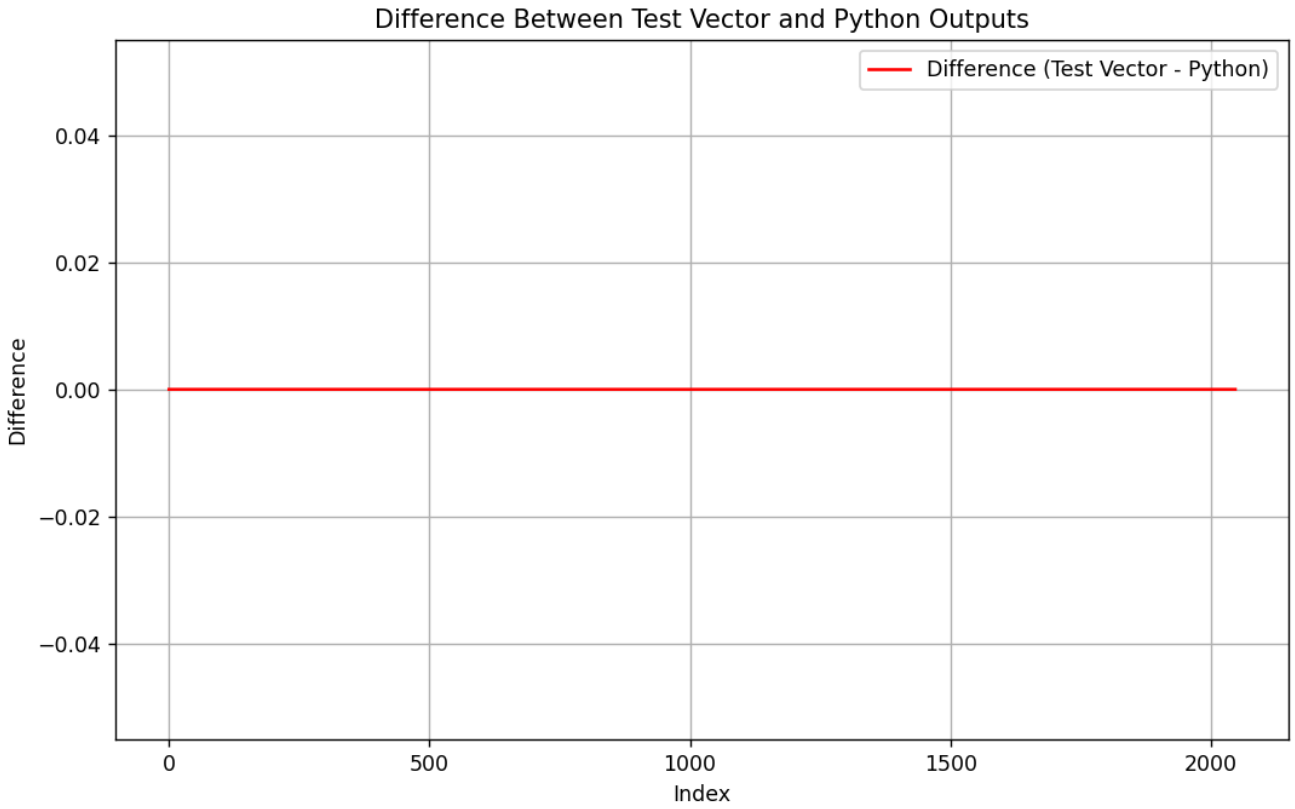
ماژول uart_test_vector_tb یک testbench برای ماژول‌های UART می‌باشد.

در بخش initial ابتدایی، فایل test_vector.txt به همراه verilog_outputs.txt برای ذخیره نتایج باز می‌شود. سپس از طریق یک حلقه تمام تست وکتورها خوانده شده و به ماژول UART_TX ارسال می‌شوند. سپس منتظر تمام شدن ارسال و دریافت می‌ماند و داده دریافت شده در فایل خروجی نوشته می‌شود.

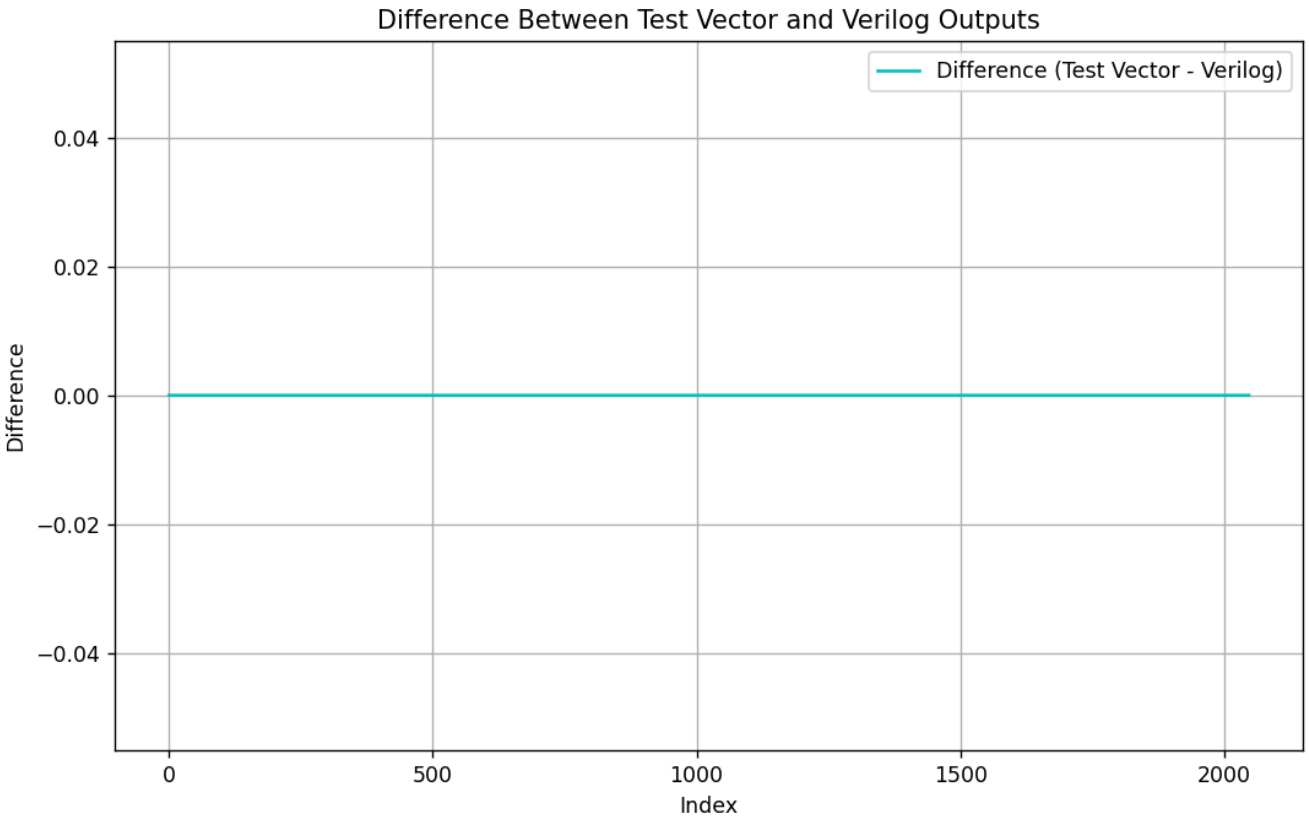
در آخر در فایل compare_result.py اختلاف نتایج بدست آمده از وریلاگ و پایتون را نمایش می‌دهیم که در شکل زیر مشاهده می‌کنید. صفر بودن این نمودار نشان دهنده‌ی صحت کار انجام شده‌است.



همچنین اختلاف نتایج بدست آمده از تست وکتور و پایتون را نمایش می‌دهیم که در شکل زیر مشاهده می‌کنید. صفر بودن این نمودار نشان دهنده‌ی صحت کار انجام شده‌است.



همچنین اختلاف نتایج بدست آمده از تست وکتور و وریلاگ را نمایش می‌دهیم که در شکل زیر مشاهده می‌کنید. صفر بودن این نمودار نشان دهنده‌ی صحت کار انجام شده‌است.


Difference Between Test Vector and Verilog Outputs
Difference (Test Vector - Verilog)
0.04
0.02
0.00
−0.02
−0.04
Difference
0
500
1000
1500
2000
Index