به نام خدا

گزارش پروژه اول کودیزاین

دکتر عبدلی

شادی نیکویی

ترم پائیز ۴۰۴

بخش اول(طراحی booth multiplier)

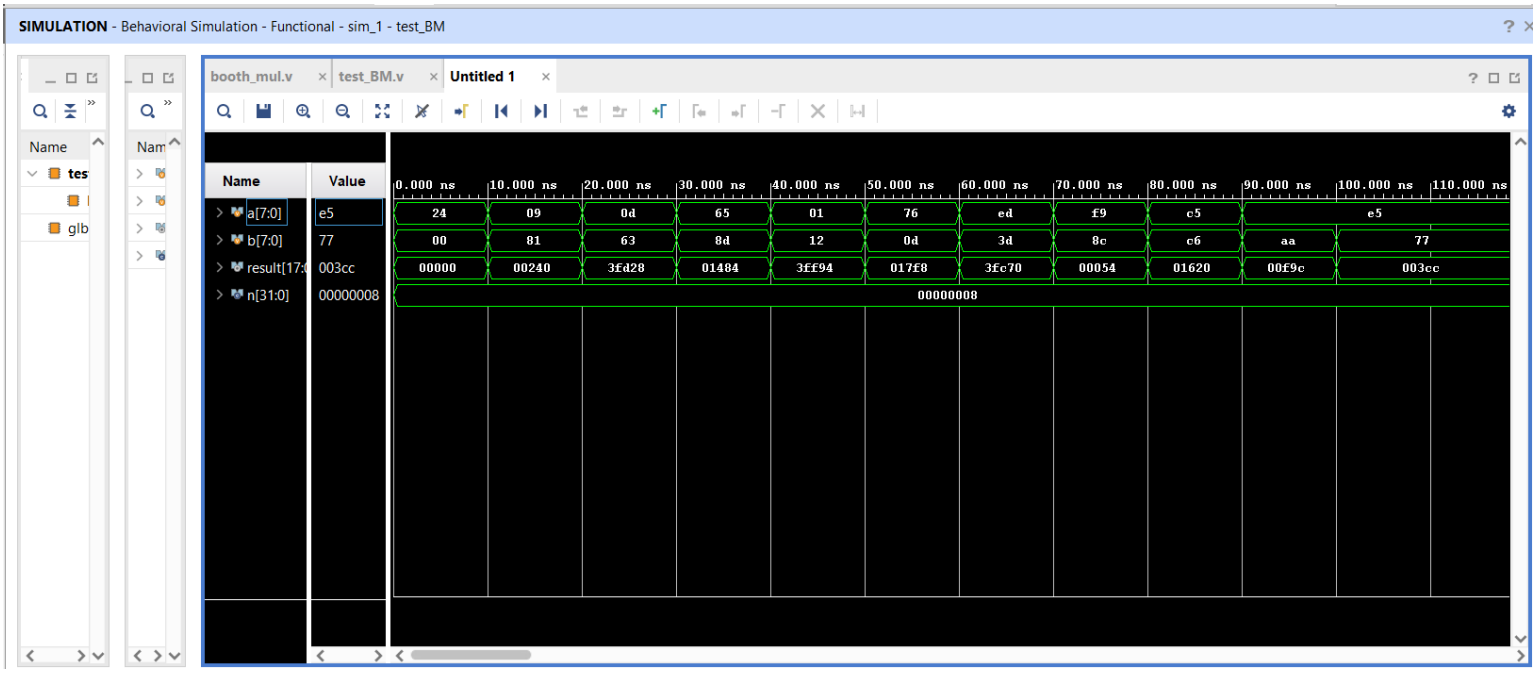
در این بخش با استفاده از وریلاگ یک توصیف در سطح RTL از این نوع ضرب کننده داشتم. چون مدار combinational است از کلاک نه در ماژول اصلی و نه در تست بنچ استفاده نشد. در عوض برای شبیه‌سازی از تاخیر های زمانی استفاده کردم. در ماژول اصلی با توجه به جدول داده شده در پی دی اف تمرین از یک for loop استفاده کردم که بتواند سه بیت سه بیت روی ویکتور عدد دوم یا همان b حرکت کند و آن سه بیت را با overlap یک بیتی به هم کانکت کند. برای همین شماره گام حلقه را ۲ گذاشتم. سپس داخل حلقه از case استفاده کردم(شرط کیس، وضعیت سه بیت کانکت شده است) تا شروط جدول را پیاده سازی کنم. در نهایت در انتهای حلقه دو بیت شیفت به چپ برای هر parshal prodouct در نظر گرفتم(طبق خود الگوریتم). در بخشی از الگوریتم آمده که اگر تعداد بیت های عدد دوم زوج باشد که تعداد حاصل ضرب های جزئی برابر n/2 است و در غیر این صورت باید ساین اکستند کنیم؛ من برای این کار من ترجیح دادم از همان اول ساین اکستند را انجام بدم. چون هر دو حالت را پوشش میدهد.

برای تست بنچ هم به جای استفاده از اعداد دستی از یک سیستم تسک در خود وریلاگ به نام $random استفاده کردم که یک عدد شبه رندوم ۳۲ بیتی تولید میکند. چون ماژول را به فرم پارامتریک نوشتم هر عددی را میتواند دریافت کند(در صورت نیاز به اعداد بیشتر از ۳۲ بیت باید دو تا دالر رندوم را با هم کانکات کنم. برای کمتر از ان هم که فقط بخش کم ارزش را نگه میدارد).

همان طور که قبل‌تر هم گفتم چون مدار ترکیبی هست از کلاک استفاده نکردم و صرفا از تاخیر برای شبیه سازی استفاده کردم.

خروجی حاصل ضرب هم باید حداقل دو برابر سایز اعداد ورودی باشد که من برای اطمینان از وجود کری از ۲ مازاد بر ان هم استفاده کردم. کد تست بنچ و ماژول اصلی در فایل زیپ آمده اند.

در نهایت خروجی سیمولیشن به صورت زیر درآمده است که خروجی به فرمت هگز است:



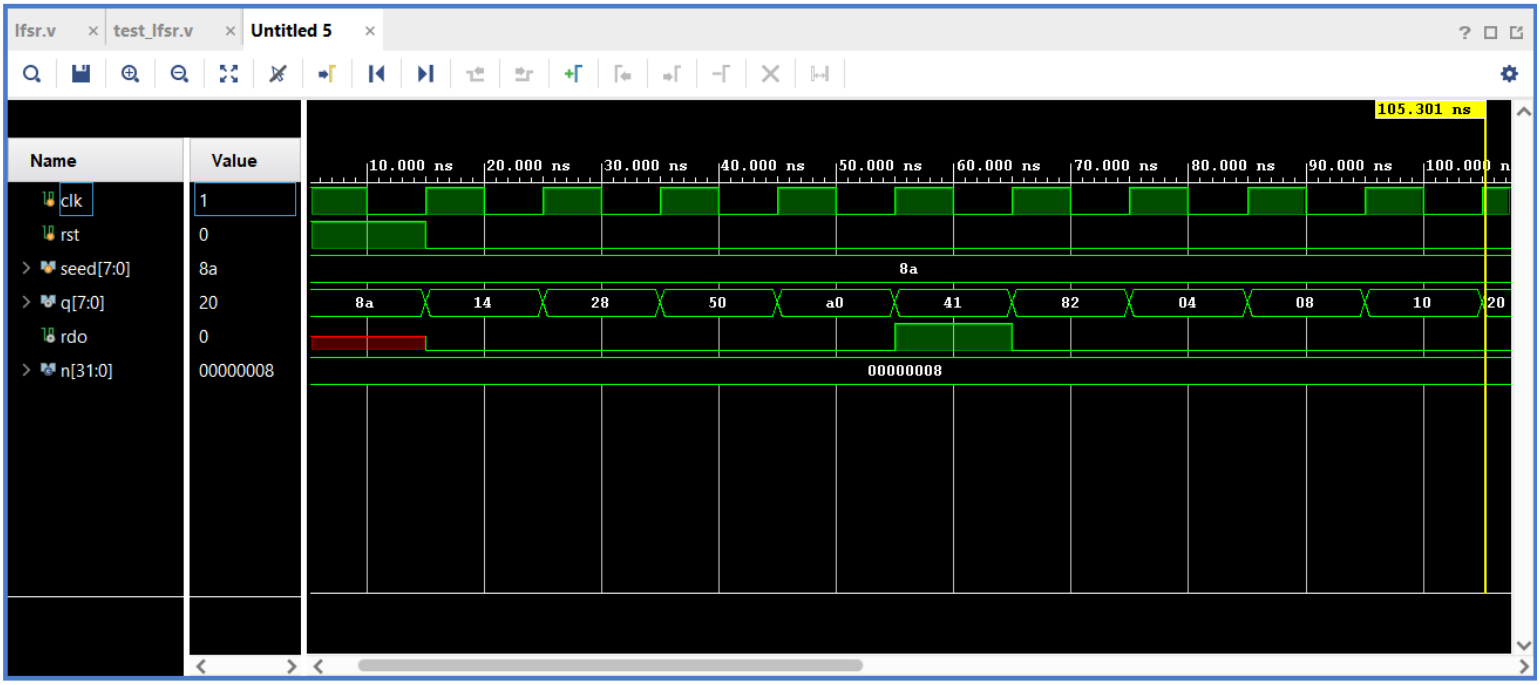
در ادامه بخش اول از خواسته شده که با متلب کدی بنویسیم تا همین الگوریتم ضرب بوث را پیاده سازی کرده و این بخش پروژه ضرب دو ورودی در بازه‌ی ([-1,+1]) را با نمایش Fixed-Point هشت‌بیتی Q2.6 پیاده‌سازی می‌کند. برای این کار ابتدا هر ورودی با ضرب در (۶^۲) کوانتیزه شده و به یک عدد صحیح int8 در نمایش مکملِ دو تبدیل می‌شود. سپس هسته‌ی Booth Radix-4 روی همین اعداد صحیح مقیاس‌یافته اجرا می‌شود: در هر گام با بازکُدگذاری سه‌تایی ({b_{i+2},b_{i+1},b_i})یکی از مقادیر ({0, A, 2A}) انتخاب و در آکیومولاتور با شیفت ۲بیتی جمع می‌شود؛ حاصل در نهایت یک int16 با مقیاس (2^{12}) معادل (Q4.12) است. خروجی با شیفت حسابی و گرد کردن به Q2.6 برگردانده می‌شود و برای ارزیابی، با مرجع MATLAB fi همان قالب (Q2.6) مقایسه می‌شود. معیارهای MAE/MSE نشان می‌دهند که هسته‌ی بوثِ صحیح، پس از اعمال همان محدودیت‌های کوانتیزه‌سازی (گرد کردن/اشباع)، هم‌ارز مرجع fi عمل می‌کند و اختلاف فقط ناشی از قواعد کوانتیزه‌سازی Q2.6 است، نه خطای الگوریتمی ضرب. این سازوکار به‌سادگی به دقت‌های بالاتر (مثلاً Q2.14 در ۱۶ بیت) نیز قابل تعمیم است.

بخش دوم(طراحی و شبیه سازی یک LFSR):

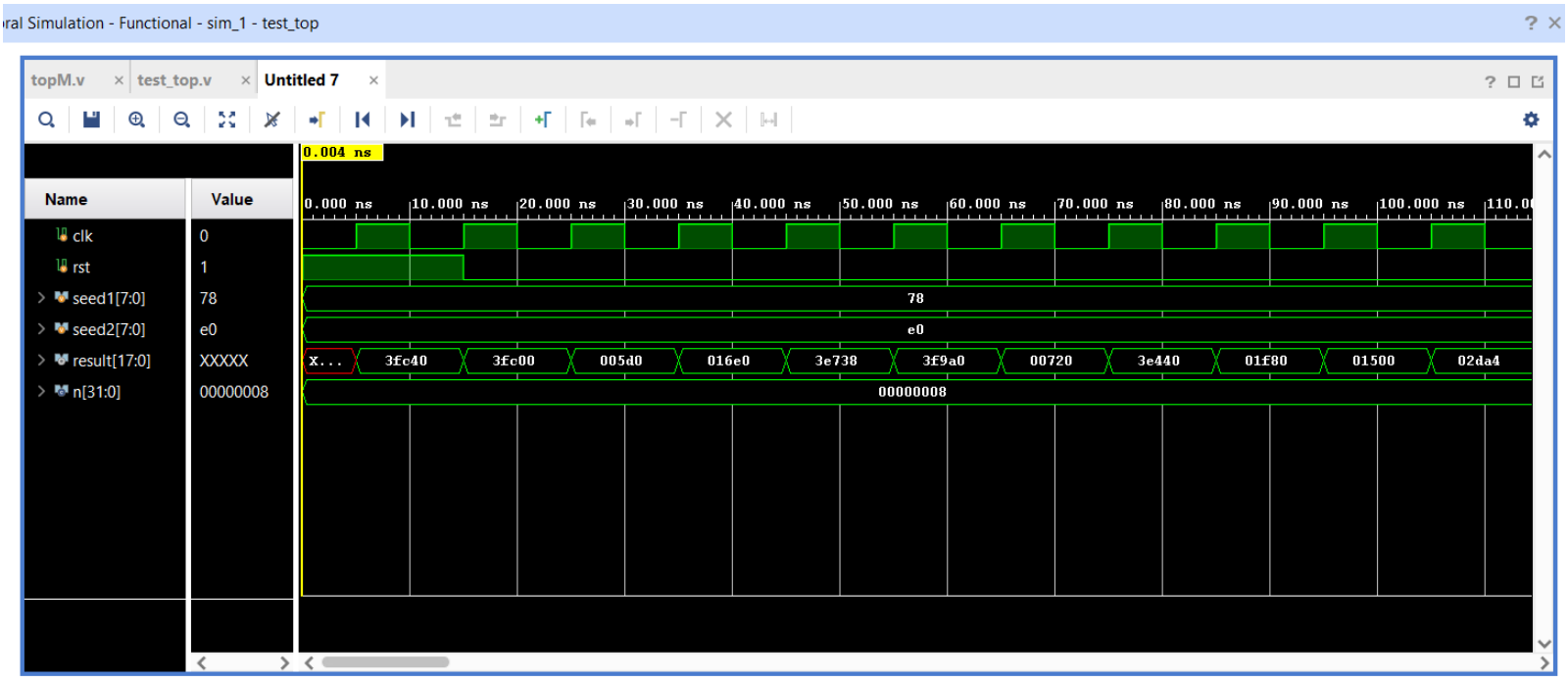
در این بخش قرار شد که یک مدار تولید کننده اعداد شبه تصادفی ایجاد کنم. برای این که بتوانم در بخش های بعدی پروژه از ان استفاده کنم، به صورت پارامتریک طراحی کردم. به این صورت که یک چند جمله ای اولیه(seed) در مرحله ریست اولیه در مدار وارد میشود. سپس فلیپ فلاپ شماره ۱ و ۷ با هم xor میشوند تا به عنوان بیت جدید در دنباله اعداد تصادفی قرار گیرد. بدین ترتیب در هر لبه کلاک به شرط انکه ریست نباشد یک عدد تصادفی جدید تولید میشود. چون با ورود بیت جدید، مقادیر قبلی یک واحد شیفت به چپ میخورند.

در تست بنچ هم لبه کلاک هر ۵ واحد زمانی ظاهر میشود. بعد از ریست ابتدا seed وارد رجیستر میشود و بعد از غیر فعال شدن ریست بعد از هر لبه کلاک داده جدید تولید میشود.

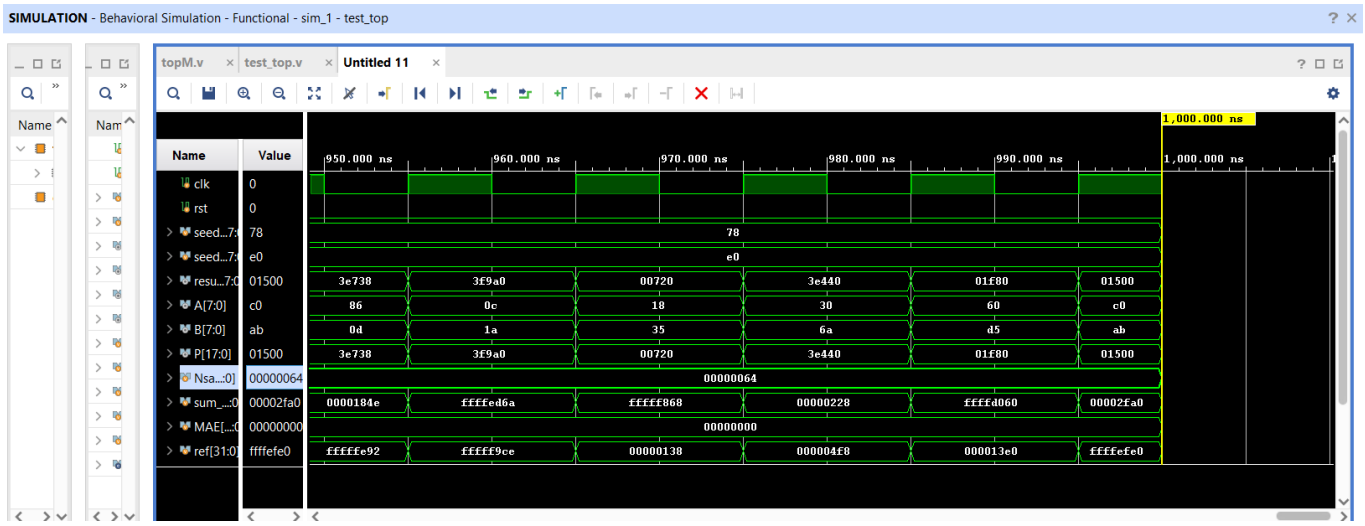
در ادامه هم تصویری از خروجی ویوفرم میبینیم:



در ادامه این بخش خواسته شده که از تابع ضرب کننده در بخش اول یک اینستنس در یک تاپ ماژول بگیریم که اعداد ورودی اش همان اعداد تولید شده توسط lfsr است.برای این کار من از ساختار generate استفاده کردم تا بتوانم از ماژول ها اینستنس بگیرم. در ادامه اتصالات لازم و ورودی های اصلی را به تاپ ماژول انجام دادم. در تست بنچ هم دو عدد ssed مختلف تعریف کردم تا دو عدد شبه تصادفی مختلف حاصل شود. در اینجا من دستی وارد کردم ولی میشود از سیستم تسک $random هم استفاده کرد. در ادامه نتیجه خروجی را میبینم:



برای بخش اخر هم که بررسی دقت در حالت ۱۶ بیت و ۸ بیت هست، من از یک تعداد عدد نمونه برداری کردم(۱۰۰) که در ازای آن ۱۰۰ عدد حاصل ضرب واقعی را که از همان عملگر * استفاده میکند با نتیجه حاصل از ضرب بوث مقایسه میکند. هم برای ۸ بیت و هم برای ۱۶ بیت. چون الگوریتم بوث هم بسیار دقیق است نتیجه ان در هر ۱۰۰ مرحله از محاسبات با نتیجه مرجع یعنی همان استفاده مستقیم از عملگر یکی است. بنابراین مقدار میانگین تفاوت یا همان دیفریانسل همواره صفر خواهد بود. مطابق شکل زیر:



در این شکل سطر P نمایانگر ضرب با عملگر مستقیم است و result نمایانگر ضرب با استفاده از الگوریتم بوث است. بنابراین مقدار دیفرانسیل در همه مراحل صفر خواهد بود. MAE همان مقدار دیفرانسیل را نشان میدهد.