بسمه تعالی

پروژه درس طراحی سیستم های دیجیتال

استاد: دکتر بهاروند

اعضای گروه:

علی شریفی

سید محمدمهدی حاتمی

امیرحسین باقری جبلی

امیرحسین ندایی پور اصل

1. مسئولیت اعضای گروه

- علی شریفی: نوشتن کد وریلاگ ماژول adder و واحد PE و آزمون واحد (testbench) آن ها – سنتز کد وریلاگ و برطرف کردن اخطار ها (warnings) در هنگام سنتز
- سید محمدمهدی حاتمی: نوشتن کد وریلاگ ضرب کننده با استفاده از ماژول PE – آزمون واحد (testbench) ماژول ضرب کننده با استفاده از ورودی های ذخیره شده در فایل و ذخیره خروجی testbench در فایل
- امیرحسین باقری جبلی: تکمیل گلدن مدل – تولید ورودی تصادفی برای تست ماژول ضرب کننده و مقایسه فایل خروجی مدار با خروجی گلدن مدل
- امیرحسین ندایی پور اصل: پیدا کردن مقاله الگوریتم بهینه و مقایسه الگوریتم ها، پیاده سازی نحوه عملکرد الگوریتم با زبان نرم افزاری (پایتون) – تولید خروجی معتبر جمع و ضرب با استفاده از پایتون برای استفاده در گلدن مدل – کمک در نوشتن کد وریلاگ واحد PE – نوشتن گزارش پروژه

2. الگوریتم های ضرب ماتریس

در این قسمت ماتریس های ضرب شونده را A و B و ماتریس حاصل ضرب را C نامیده ایم. ($A \times B = C$) ابعاد تمام این ماتریس ها را $n \times n$ در نظر گرفته ایم و مقدار خانه ی سطر i ام و ستون j ام از یک ماتریس (برای مثال A) را هم به صورت A_{ij} نشان داده ایم.

2.1. استفاده مستقیم از تعریف ضرب ماتریس ها

خروجی C_{ij} از ماتریس حاصل ضرب تعریف مشخصی دارد که با استفاده از مقادیر سطر i ام ماتریس A، ستون j ام ماتریس B و عملگر های ضرب و جمع تعریف می شود. اولین ایده هایی که برای انجام ضرب ماتریس به صورت سخت افزاری به ذهن می رسند، پیاده سازی مستقیم همین تعریف است.

ماژولی مثل 1xn_multiplier را در نظر بگیرید که با گرفتن سطر i ام از ماتریس A (A_i) و دریافت محتویات همه ی خانه های ماتریس B، حاصل ضرب $A_i \times B$ را تولید کند. همچنین فرض کنید تولید حاصل ضرب یک سطر از ماتریس A در یک ستون از ماتریس B، با استفاده از واحد های جمع کننده و ضرب کننده ی ترکیبی، تنها یک کلاک زمان ببرد. در اینصورت 1xn_multiplier می تواند $A_i \times B$ را بعد از ضرب A_i در هر یک از ستون های ماتریس B، در n کلاک تولید کند. این حاصل ضرب یک بردار سطری $1 \times n$ است و برابر مقادیر سطر i ام ماتریس C (C_i) در حاصل ضرب نهایی است.

در نهایت ضرب A در B می تواند با استفاده از n ماژول 1xn_multiplier که هر یک حاصل ضرب یک سطر از ماتریس A در کل ماتریس B را محاسبه می کنند، انجام شود. با توجه به کارکرد موازی این ماژول ها، حاصل ضرب نهایی ماتریس ها بعد از n کلاک به دست خواهد آمد. این درصورتی است که ورودی ها در ابتدای اجرای الگوریتم به صورت مستقیم و نه Serialized به هر ماژول داده شود.

دقت کنید که اگر ورودی ها به صورت Serialized به ماژول ها داده شوند باید n^2 کلاک هم برای دریافت مقادیر ماتریس های A و B صرف شود، چون در این الگوریتم در صورتی که ورودی های A_i و B از 1xn_multiplier مقدار دهی نشده باشند، این ماژول نمی تواند کار خود را شروع کند.

در پیاده سازی سخت افزاری این الگوریتم، هر یک از واحد های 1xn_multiplier شامل n ماژول ضرب کننده و n ماژول جمع کننده خواهند بود و خروجی نهایی بعد از n کلاک تولید خواهد شد.

2.2. الگوریتم Strassen

الگوریتم Strassen یک الگوریتم بازگشتی متداول برای ضرب ماتریس در پیاده سازی نرم افزاری است که پیچیدگی زمانی آن از انجام ضرب ماتریس به صورت مستقیم کمتر است.

در این الگوریتم هر کدام از ماتریس های A، B و C به 4 ماتریس با ابعاد $\frac{n}{2} \times \frac{n}{2}$ تقسیم می شوند و مقدار هر یک از 4 زیر ماتریس C به صورت بازگشتی با انجام تعدادی ضرب $\frac{n}{2} \times \frac{n}{2}$ محاسبه می شود. به این رویکرد محاسبه ی بازگشتی در نرم افزار Divide and Conquer می گویند.

پیاده سازی این الگوریتم به صورت سخت افزاری منجر به تولید مداری تماما ترکیبی می شود که دارای $\lg n$ سطح است. در هر یک از این سطوح تعدادی جمع کننده به کار می رود و در پایین ترین سطح محاسبات از n^3 ماژول ضرب کننده استفاده می شود. این ضرب کننده ها همه ی حاصل ضرب های لازم برای تولید ماتریس حاصل ضرب (C) را در سطح یک انجام می دهند و از آنجا به بعد با انجام تعدادی جمع در هر سطح نتیجه نهایی تولید می شود.

در نتیجه در پیاده سازی سخت افزاری این الگوریتم از n^3 ماژول ضرب کننده و $O(n)$ ماژول جمع کننده استفاده شده است و نتیجه نهایی بعد از تاخیر $\lg n$ جمع کننده، در خروجی آماده خواهد بود. البته اگر ورودی های ماژول به صورت Serialized دریافت شوند باید n^2 کلاک برای دریافت کامل ورودی ها صرف شود چرا که اگر در ابتدای اجرای الگوریتم ورودی ها به صورت کامل دریافت نشده باشند، خروجی تولید شده هم نامعتبر خواهد بود.

2.3. الگوریتم شرح داده شده در منبع 5.3

در این الگوریتم محاسبه مقادیر ماتریس C به صورت ترتیبی و به کمک تعدادی واحد به نام PE انجام می شود. هر PE مسئول تولید یک ستون از ماتریس نهایی است. در این روش مقادیر ماتریس های A و B به صورت Serialized و به ترتیبی خاص وارد واحد های PE می شوند. هر PE شامل یک رجیستر به نام A، 3 رجیستر به نام های BU، BM و BL، تعدادی MUX 2x1، دو شمارنده به نام های Counter1 و Counter2، یک ضرب کننده، یک جمع کننده و n رجیستر جهت نگه داری نتیجه نهایی (ستون i ام از ماتریس C) است.

در ابتدای اجرای الگوریتم، مقدار یکی از خانه های ماتریس A و یکی از خانه های B به واحد PE_1 داده می شود. این واحد پس از انجام محاسبات و ذخیره نتایج، این دو عدد را به واحد بعدی (PE_2) می دهد و به همین شکل انجام محاسبات تا PE_n ادامه می یابد.

رجیستر های A و BU در هر واحد PE، مسئول نگه داری مقدار خانه ای از ماتریس A و B هستند که به عنوان ورودی از واحد PE قبلی دریافت شده اند. از رجیستر های BM و BU برای نگه داشتن احتمالی مقدار رجیستر BU برای استفاده در محاسبات کلاک های بعدی استفاده می شود.

دقت کنید که باید ورودی های ماتریس A به صورت ستونی (یعنی با ترتیب $A_{11}, A_{21}, A_{31}, \ldots$) و ورودی های ماتریس *B* به صورت سطری (یعنی با ترتیب $B_{11}, B_{12}, B_{13}, \ldots$) سیستم را تغذیه کنند و اولین ورودی از ماتریس *A* دقیقا *n* کلاک بعد از اولین ورودی از ماتریس *B* به سیستم داده شود. درون هر *PE* مثل PE_k، تنها B_{ij} هایی در *BM* یا *BL* نگه داشته می شوند که در آن ها *j = k*. در اینصورت در یک *PE* مقادیر رجیستر *BU* از حداکثر *2n* کلاک قبل نگه داشته می شوند و در طی این مدت همه حاصل ضرب هایی که B_{ij} در آن ها حضور دارد محاسبه می شوند و در نهایت این مقدار دور انداخته می شود. تصمیم گیری در مورد اینکه دقیقا در چه زمان هایی مقدار *BU* در *BM* یا *BL* ذخیره شود هم با استفاده از مقادیر *Counter1* و *Counter2* صورت می گیرد. بنابراین اجرای کامل محاسبات و تولید خروجی نهایی در صورت ترکیبی بودن مدار های جمع و ضرب کننده، مجموعا $n^2 + n$ کلاک زمان خواهد برد. و این در حالی است که دریافت ورودی ها از ماتریس ها به صورت *Serialized* صورت می گیرد. دقت کنید که برای پیاده سازی سخت افزاری این الگوریتم از *n* جمع کننده و *n* ضرب کننده استفاده شده است.

برای نشان دادن واضح تر نحوه عملکرد الگوریتم، با استفاده از کتابخانه *openpyxl* زبان پایتون یک فایل اکسل تولید کرده ایم که مقدار دهی واحد های *PE* و محاسبات را بعد از گذشت هر کلاک نشان می دهد. کد زبان پایتون و فایل اکسل یاد شده پیوست شده اند. همچنین توضیح دقیق تری از الگوریتم با استفاده از نماد های ریاضی در منبع 5.3 آورده شده است که می توانید به آن مراجعه کنید.

2.4. مقایسه الگوریتم ها و انتخاب الگوریتم بهینه جهت پیاده سازی

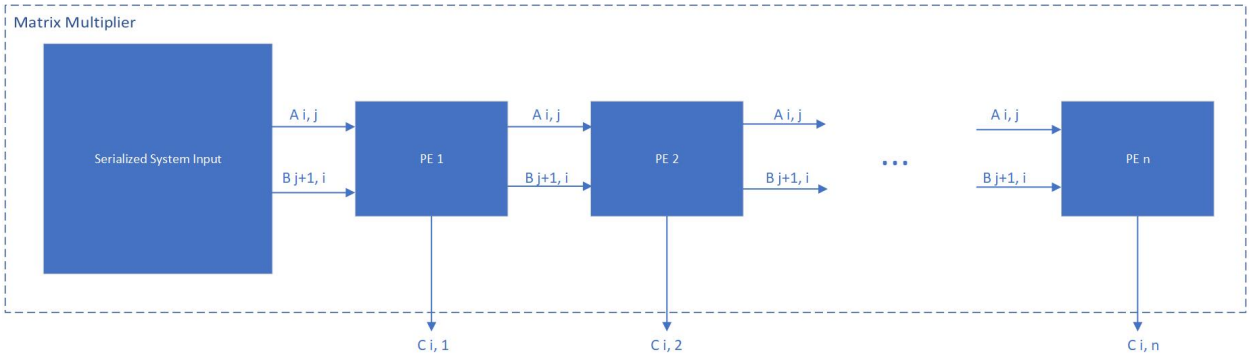
با توجه به توضیحات داده شده ما الگوریتم 2.3 را برای پیاده سازی انتخاب کردیم. مهم ترین مزیت این الگوریتم این است که دریافت ورودی در آن به صورت Serialized انجام می شود و انجام محاسبات برای تولید ماتریس نهایی هم در حین دریافت ورودی انجام می شود. الگوریتم های 2.1 و 2.2 این ویژگی را ندارند و ورودی های مورد استفاده آن ها باید قبل از شروع الگوریتم محاسبه حاصل ضرب دریافت شده باشد. اگر این دریافت ورودی به صورت مستقیم (یک ورودی به ازای هر خانه از ماتریس A یا B) صورت بگیرد تعداد پورت های ورودی و خروجی ماژول بسیار زیاد خواهد بود و اگر به صورت Serialized انجام شود، باید تا قبل از آماده شده ورودی ها برای محاسبه، n^2 کلاک صبر کنیم. این باعث برتری زمانی این الگوریتم نسبت به الگوریتم های دیگر می شود.

همچنین الگوریتم 2.3 در زمینه مساحت نسبت به الگوریتم های 2.1 و 2.2 برتری دارد. چرا که در آن تنها از n ضرب کننده و n جمع کننده استفاده شده است در حالی که در الگوریتم 2.1 مجموعا از n^2 ضرب کننده و n^2 جمع کننده و در الگوریتم 2.2 از n^3 ضرب کننده و $O(n)$ جمع کننده استفاده شده است.

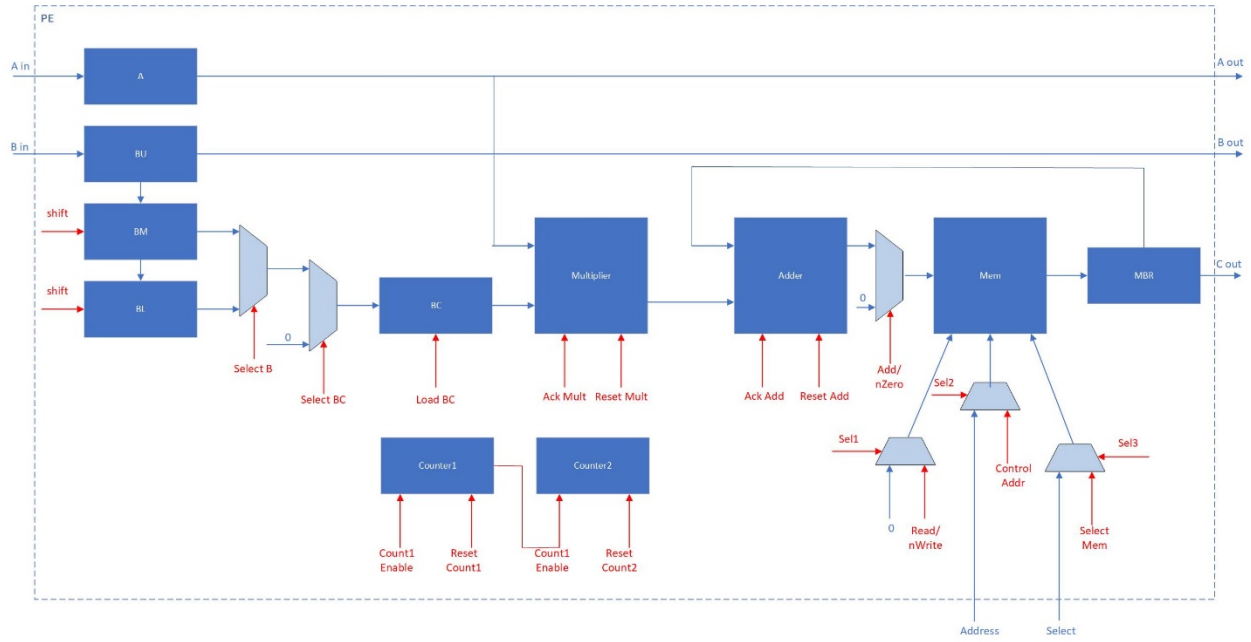
بنابراین الگوریتم 2.3 هم از نظر مساحت و هم از نظر تایمینگ بهتر از سایر الگوریتم های پیشنهادی عمل می کند.

3. معماری سیستم

3.1. ضرب کننده



3.2. واحد PE



ورودی ها و خروجی های سیستم با رنگ آبی و سیگنال های کنترلی با رنگ قرمز مشخص شده اند.

3.3. ورودی و خروجی های واحد PE

با توجه به اینکه در طراحی ما واحد های جمع و ضرب کننده مدار هایی ترتیبی هستند، اجرای عملیات محاسبه در هر واحد PE بیش از یک کلاک طول خواهد کشید و با توجه به پیاده سازی واحد های جمع و ضرب کننده، تعداد کلاک های انجام عملیات در هر PE هم می تواند بسته به شرایط متفاوت باشد. بنابراین باید برای ارتباط صحیح بین واحد های PE از چند سیگنال اضافی استفاده شده است.

```
1   module PE #(parameter log_size, parameter index)
2   (
3           input [31:0] a,
4           input [31:0] b,
5           input stb,
6           input rst,
7           input clk,
8           input next_PE_ack,
9           output input_ack,
10          input input_b_valid,
11          output output_b_valid,
12          input [log_size-1:0] addr,
13          input mem_select,
14          output [31:0] c,
15          output [31:0] output_b,
16          output [31:0] output_a,
17          output output_stb
18  );
```

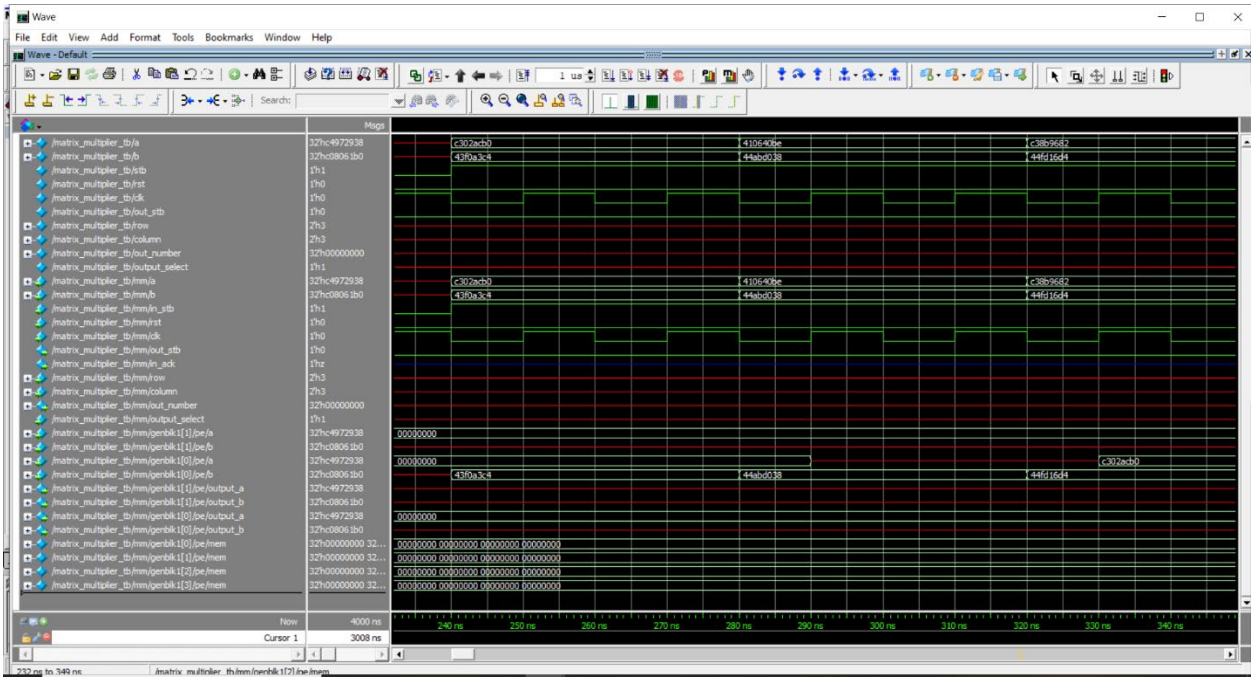
سیگنال ورودی stb نشان دهنده معتبر بودن ورودی های a و b است. بعد از دریافت ورودی های a و b از واحد قبلی، سیگنال خروجی input_ack برابر 1 می شود تا به واحد قبلی اطلاع داده شود که انتقال مقادیر a و b به درستی انجام شده است. ورودی input_b_valid و خروجی output_b_valid نشان می دهند که آیا در n کلاک اول دریافت ورودی توسط سیستم هستیم یا نه. دقت کنید که در n کلاک نخست دریافت ورودی، مقدار هیچ خانه ای از ماتریس A وارد سیستم نمی شود و این تنها مقادیر ماتریس B هستند که وارد می شوند. در واقع سیگنال های b_valid همزمان با دریافت اولین ورودی از سیستم برابر 1 قرار داده می شوند تا هر واحد PE از شروع عملیات ضرب ماتریس مطلع شود.

ورودی های addr (آدرس) و mem_select برای خواندن یک خانه از حافظه ی n تایی هر واحد PE استفاده می شوند. هنگامی که محاسبات ضرب ماتریس به طور کامل به پایان برسد هم سیگنال خروجی output_stb برابر 1 قرار داده می شود. این به معنی است که می توان با دادن آدرس به واحد PE، مقدار یک خانه از ماتریس حاصل ضرب نهایی را دریافت کرد.

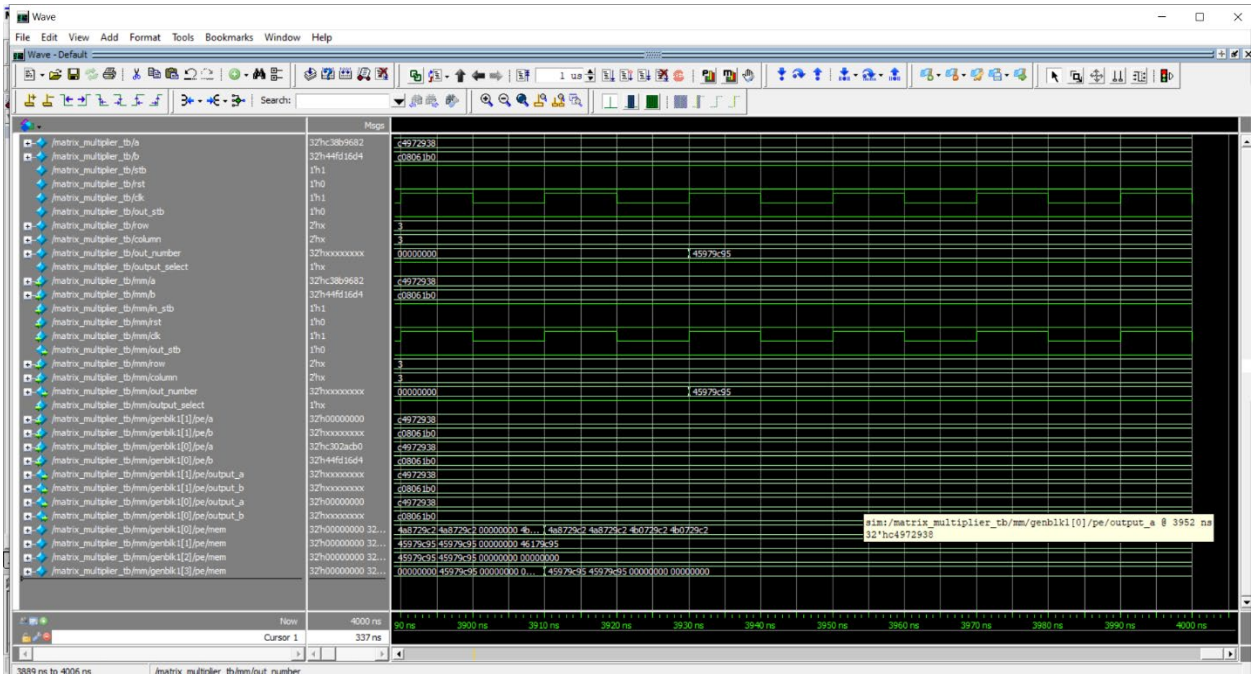
4. تست

4.1. امواج تولید شده در هنگام شبیه سازی ضرب کننده در ModelSim

در هنگام اعمال ورودی ها



در هنگام دریافت خروجی ها

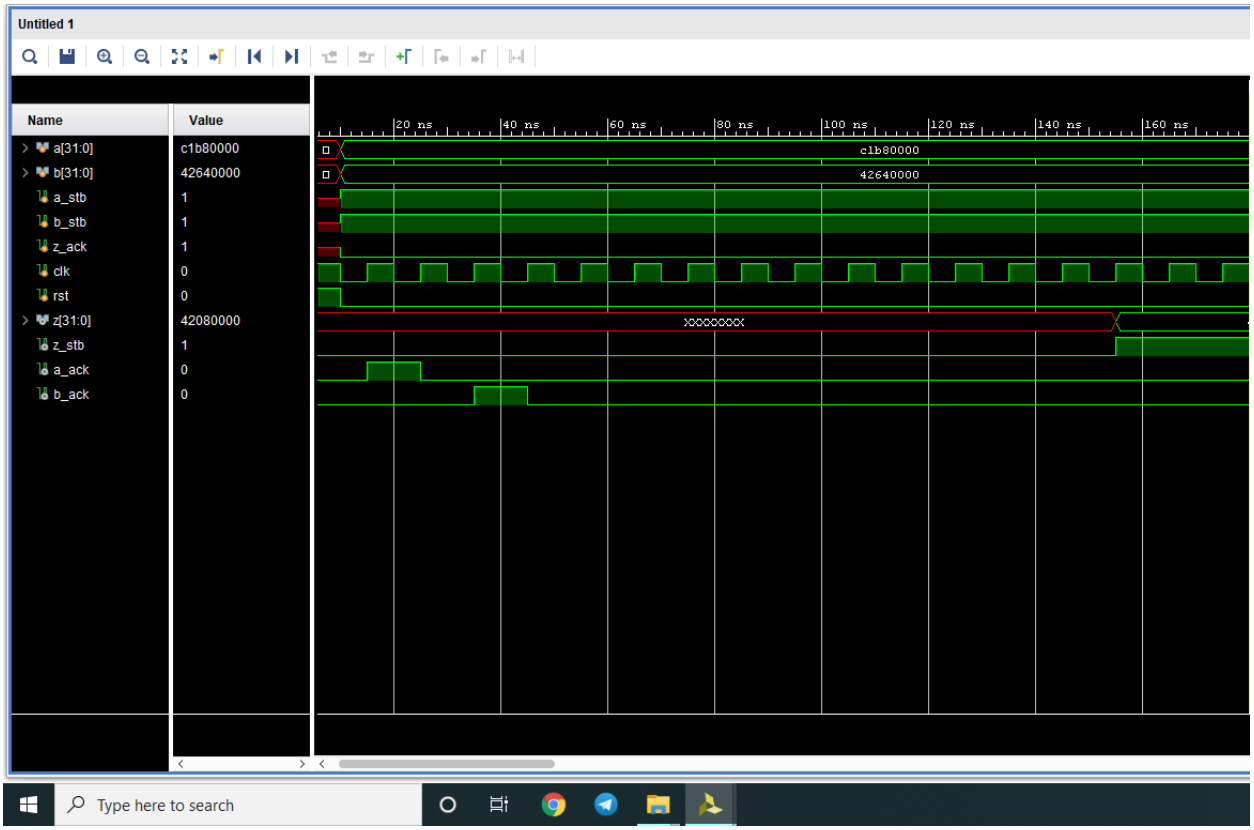


4.2. آزمون واحد

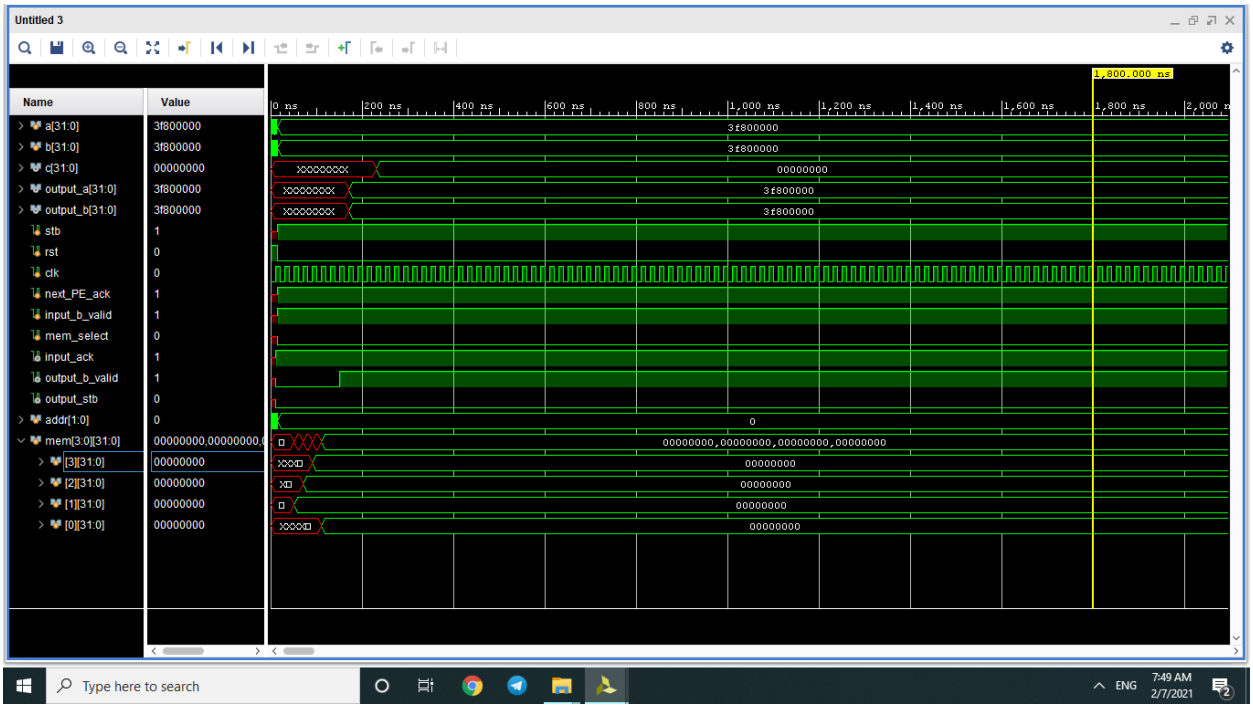
برای تمامی ماژول های طراحی شده، testbench نوشته شده است. برای تست ماژول های adder، PE و matrix_multiplier از adder_tb، PE_tb و matrix_multiplier_tb استفاده شده است که در فایل پروژه پیوست شده اند. تصاویر بخش 4.1 نتیجه اجرای testbench برای ماژول های یاد شده هستند.

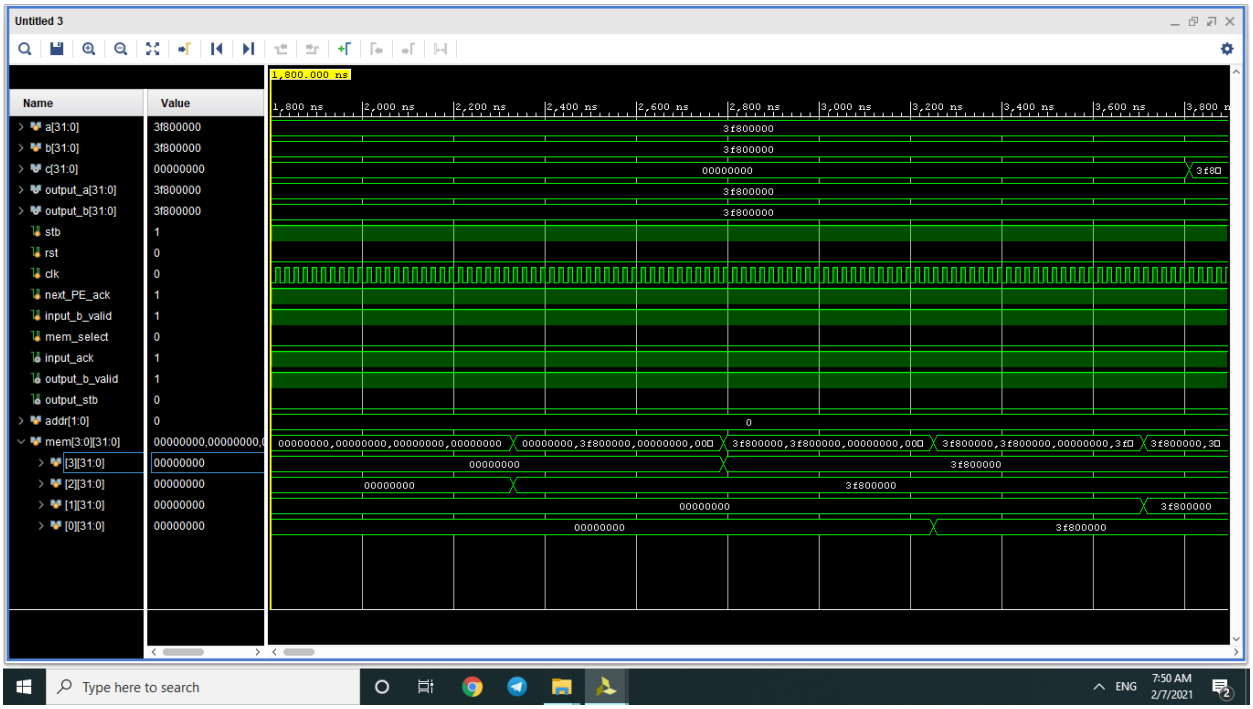
4.3. سنتز

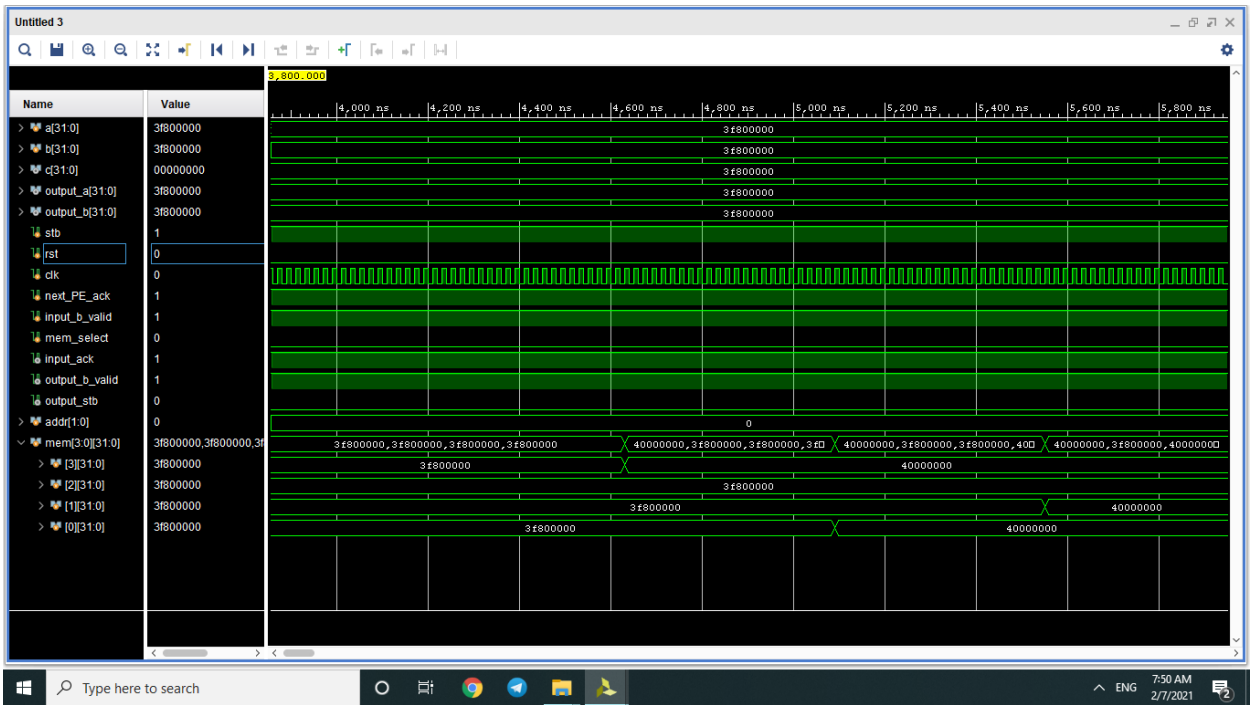
نتیجه سنتز ماژول adder

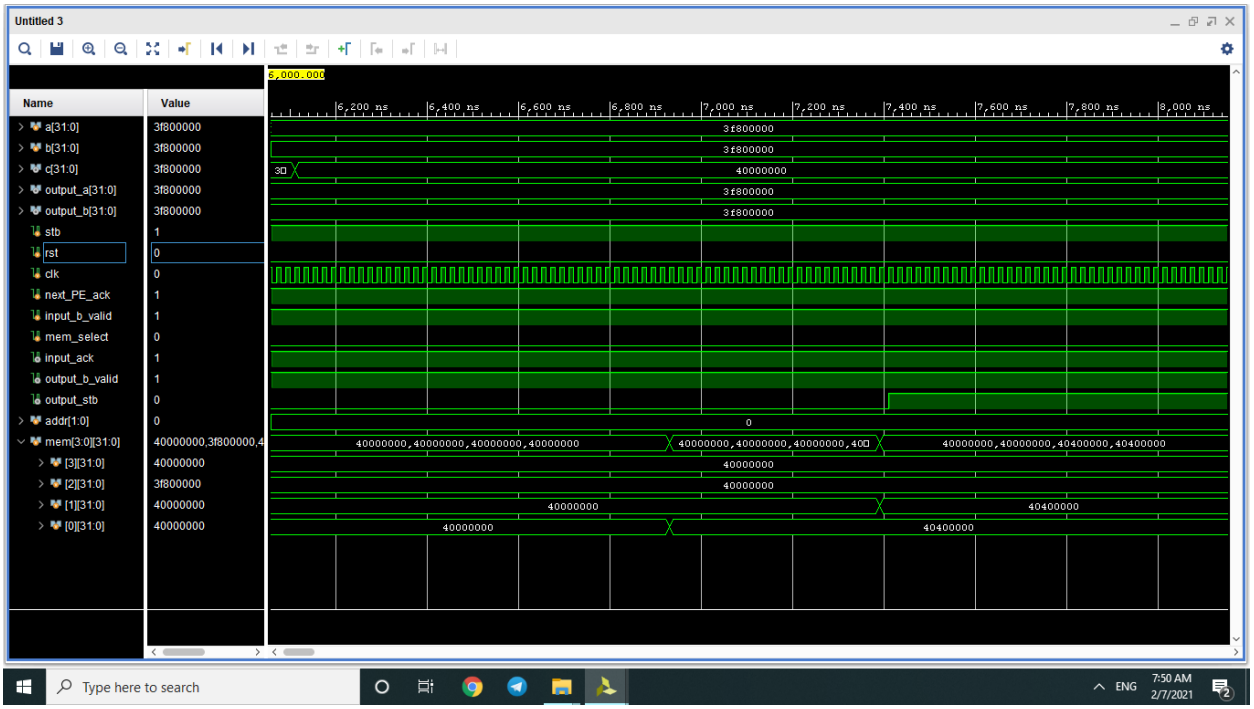


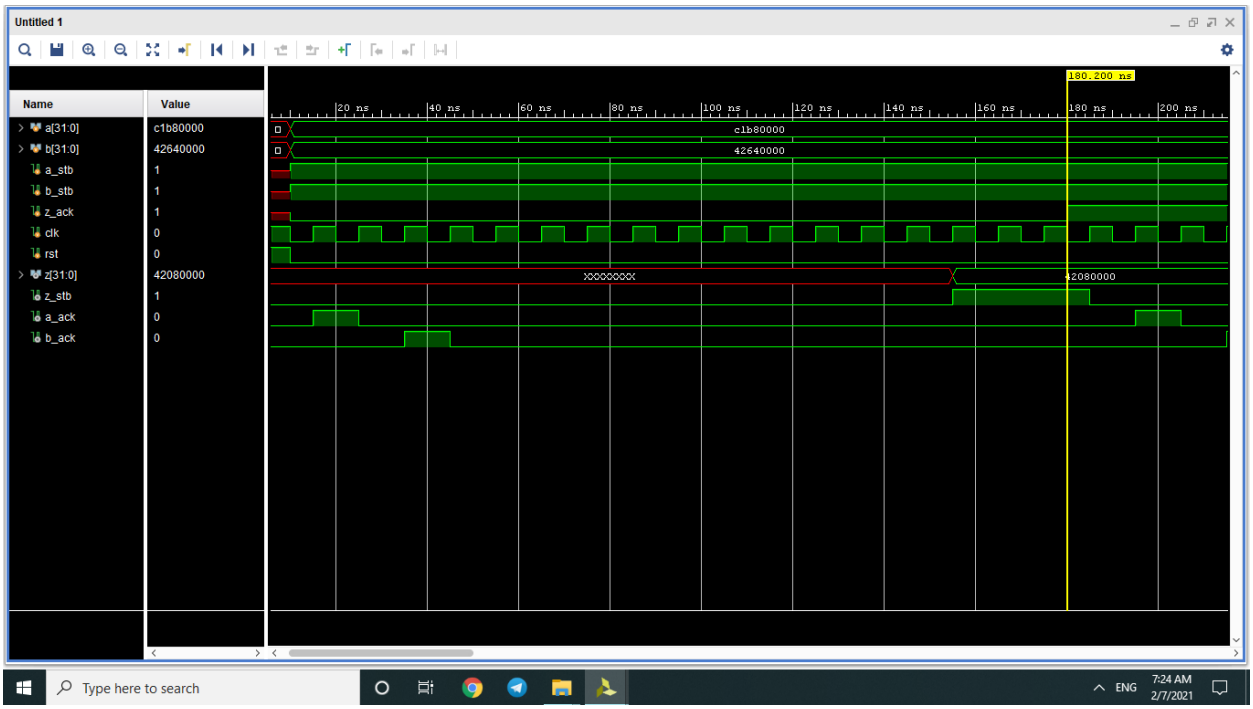
نتیجه سنتز ماژول PE



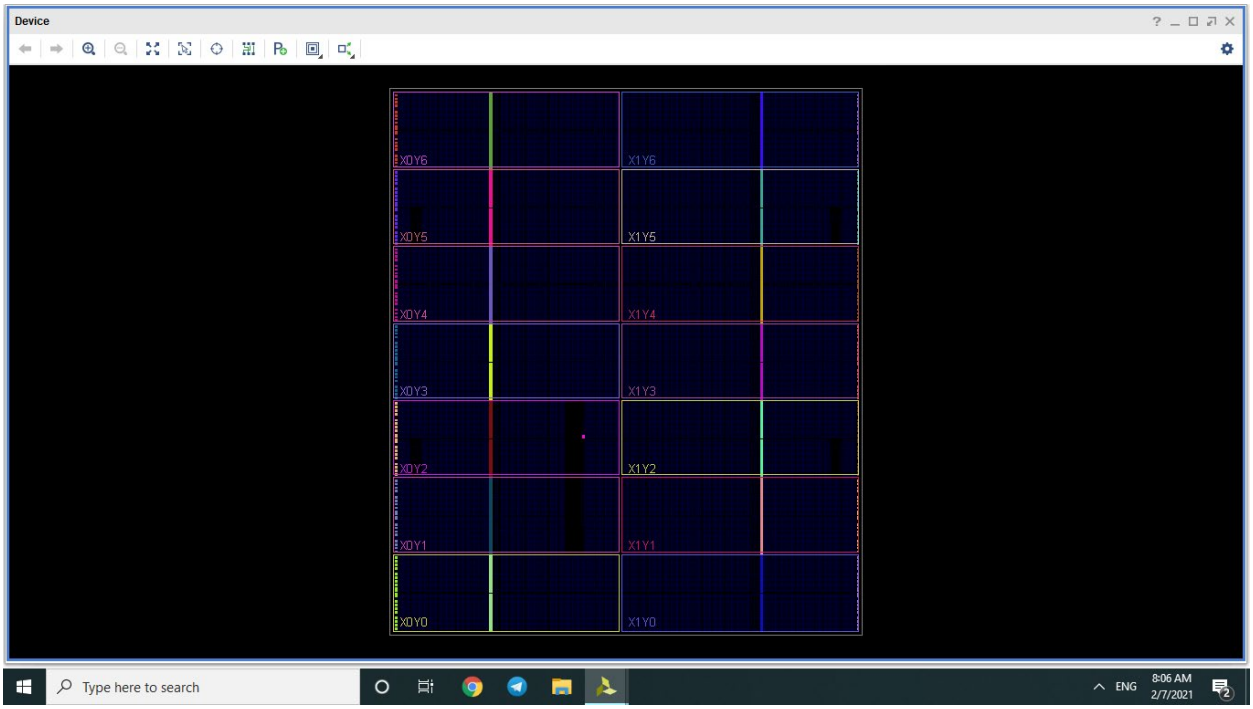



Untitled 3
Name
Value
a[31:0]
b[31:0]
c[31:0]
output_a[31:0]
output_b[31:0]
stb
rst
clk
next_PE_ack
input_b_valid
mem_select
input_ack
output_b_valid
output_stb
addr[1:0]
mem[3:0][31:0]
[3][31:0]
[2][31:0]
[1][31:0]
[0][31:0]
3f800000
00000000
40000000
Type here to search
ENG
7:50 AM
2/7/2021


Untitled 3
Name
Value
a[31:0]
b[31:0]
c[31:0]
output_a[31:0]
output_b[31:0]
stb
rst
clk
next_PE_ack
input_b_valid
mem_select
input_ack
output_b_valid
output_stb
addr[1:0]
mem[3:0][31:0]
[3][31:0]
[2][31:0]
[1][31:0]
[0][31:0]
3f800000
40000000
40400000
Type here to search
ENG
7:50 AM
2/7/2021



نتیجه سنتز واحد PE



خروجی سنتز ماژول های بالا هم به همراه پروژه پیوست شده است.

4.4. گلدن مدل

کتابخانه Selenium زبان پایتون یک کتابخانه متداول برای جمع آوری اطلاعات از وبسایت ها به صورت سیستماتیک است. در قسمت گلدن مدل با استفاده از این کتابخانه اعداد ورودی را به http://weitz.de/ieee/ می دهیم و خروجی تولید شده توسط سایت را برای جمع یا ضرب با استاندارد single precision دریافت می کنیم. پیاده سازی گلدن مدل ضرب ماتریس هم به زبان پایتون انجام شده است. ورودی و خروجی های استفاده شده در این گلدن مدل در فایل هایی ذخیره شده اند تا با خروجی ضرب ماتریس ماژول matrix_multiplier مقایسه شوند. تمامی این فایل ها به همراه پروژه پیوست شده اند.

5. منابع

5.1. https://en.wikipedia.org/wiki/Matrix_multiplication_algorithm

5.2. https://en.wikipedia.org/wiki/Strassen_algorithm

5.3. Area and Time Efficient Implementations of Matrix Multiplication on FPGAs, Ju-wook Jang, Seonil Choi and Viktor K. Prasanna

منبع آخر به همراه فایل پروژه پیوست شده است.