

«بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ»



دانشگاه صنعتی شریف

پروژه پایانی درس طراحی سیستم های دیجیتال

دکتر فرشاد بهاروند

تابستان ۱۴۰۰

# اعضاي تيم

## محمد علی خدابنده لو

پياده سازي ماژول ضرب دو ماترييس ، ممورى و جمع دو ماترييس

## محمد علی محمد خان

آماده سازی داک . تحقیق در مورد اعداد Floating Point . آماده سازی داک مربوط به آماده سازی نهایی ماتریس

## علي رضا ايجي

نوشتن تست بنج . خواندن ماتریس از فایل

## محمد ابوالنژادیان

فرمت داک و صفحه آرایی . پياده سازی مقدمه . آماده سازی داک ماژول ها . طراحی شماتیک سیستم

## سینا الهی منش

پياده سازی ماژول مموری، ضرب دو ماتریس خرد و جمع آن ها، آماده سازی داک نتیجه گیری و کاربردها

## محمدحسین حاجی سید سلیمان

تحقیق درباره ضرب ماتریس ها . پياده سازی گلدن مدل . سنتز

## کیان عمومی

تحقیق در مورد استانداردهای مختلف نمایش اعداد . تست بنج جمع گننده



# فهرست کنایین

8	بخش ۱ : مقدمه و توضیقات ابتدایی
8	۱.۱ : چکیده پروژه
9	۱.۲ : کاربرد های ضرب دو ماتریس
11	۱.۳ : اعداد شناور (Floating Point)
11	۱.۳.۱ : تاریخچه پیدایش نمایش اعداد شناور
12	۱.۳.۲ : کاربرد ها و مقایسه با نمایش ممیز ثابت
12	Floating Point مخصوص
13	Fixed Point مخصوص
13	۱.۳.۳ : استاندارد IEEE754 برای نمایش اعداد با ممیز شناور
13	single precision: .۱
14	double precision: -2
17	۱.۳.۴ : ضرب دو عدد اعشاری با ممیز شناور
17	محاسبه علامت:
18	محاسبه بخش توانی (Exponent):
18	محاسبه بخش عدد علمی (Mantissa):
20	۱.۳.۵ : روش های ضرب دو عدد
20	تقسیم و غلبه:
21	درخت والاس:
23	ضرب کنندۀ بران (آرایه ای):
24	۴.۱: برس الگوریتم های مختلف ضرب دو ماتریس در بکدیگر
24	۴.۱.۱: اهمیت ضرب ماتریس ها
24	۴.۱.۲: ضرب پذیر بودن دو ماتریس

25	ا.۴.۳ : چکونگی ضرب دو ماتریس
26	ا.۴.۴ : الگوریتم های ضرب دو ماتریس
26	الگوریتم پیمایشی (بديهی)
27	الگوریتم تقسیم و غلبه   Divide-and-conquer
28	الگوریتم استراسن
29	<b>بخش ۲ : توصیف معماری سیستم:</b>
29	۱.۱ : معرفی کلی الگوریتم
34	۱.۲ : اینترفیس های سیستم
34	۱.۲.۱ : ورودی ها
34	۱.۲.۲ : خروجی ها
35	۱.۲.۳ : کلای سیستم
36	۱.۲.۴ : مأژول ها
36	۱.۳.۱ : واحد ضرب کننده خرد
37	۱.۳.۲ : واحد جمع کننده خرد
37	۱.۳.۳ : مموری
39	ندوهه قرارگیری اطلاعات در واحد ذخیره اطلاعات مموری
40	۱.۳.۴ : واحد کنترل
41	۱.۳.۰ : واحد wrapper
41	۱.۳.۶ : خواندن از فایل
43	۱.۳.۷ : دیاگرام ساخت افزار
43	۱.۳.۸ : واحد مموری
44	۱.۳.۹ : واحد کنترل
44	۱.۳.۱۰ : سیگنال done
45	۱.۰ : سلسه مراتب طراحی و ساختار درختی

45	<b>۵.۰.۱: شهای کلی سیستم</b>
47	<b>۵.۰.۲: ساختار درختی</b>
48	<b>بخش ۳: شبیه سازی و نتایج حاصله</b>
48	۳.۱: توصیف test bench و سیکنال های آن
53	۳.۲: توصیف روند شبیه سازی ساخت افزار
54	۳.۳: پیاده سازی مدل طلایی و مقایسه نتایج
62	نتیجه حاصل شده از گذشت Golden Model
63	صدت سنجی توسط یک ماشین دسابت آنلاین
64	<b>بخش ۴: پیاده سازی و نتایج حاصله / implementation</b>
64	۴.۱: سنتر سیستم روی CAD TOOL با FPGA
69	۴.۲: گزارش پیاده سازی
69	۴.۲.۱: مسافت
69	۴.۲.۲: تعداد فلیپ فلادپ
70	۴.۲.۳: LUT
70	۴.۲.۴: داداکثر فرکانس قابل حصول
75	<b>بخش ۵: نتیجه گیری</b>
75	۵.۱: کاربرد های سیستم طراحی شده
75	۵.۱.۱: کاربرد در فیزیک و مهندسی Physics and Engineering)
75	۵.۱.۲: کاربرد در کامپیووتر Applications)
76	۵.۱.۳: کاربرد در کامپیووتر Computer Science and Engineering) Application)
77	۵.۱.۴: کاربرد در پزشکی Medical Applications))
77	۵.۱.۵: کاربرد در اقتصاد Economy Applications))

78	0.1.0: کاربرد در فناوری اطلاعات (IT Applications))
78	0.1.6: کاربرد در رباتیک (Robotics Applications))
79	0.1.0: کاربرد در جامعه شناسی (Sociology Applications))
79	0.1.0: کاربرد در زمین شناسی (Geology Applications))
79	0.1.9: کاربرد در رمزگاری (Encryption Applications))
79	0.1.1: کاربرد در بازی ۳بعدی (Games Applications))
81	0.۱: مزیت پیاده سازی سیستم طراحی شده به صورت سخت افزاری
82	0.۳: آموخته های این پروژه

## **بخش ۱ : مقدمه و توضیحات ابتدایی**

### **۱.۱ : چکیده پروژه**

پروژه ای که قرار است تا در ادامه توضیحات راجع به آن را مطالعه کنید،  
کلیتی مبنی بر ضرب دو ماتریس  $n \times n$  به روشن تقسیم ماتریس ها به زیر ماتریس  
هایی برای رسیدن به موازی سازی مناسب در طول پردازش دارد.

در بخش اول سعی داریم تا در ابتدا دلیل و کاربردهای این عمل ( ضرب دو  
ماتریس) را بیان، سپس با توجه به اینکه خانه های ماتریس ها می توانند از  
اعداد اعشاری تشکیل شده باشند، توضیحاتی راجع به نهایش و طریقه ضرب  
این اعداد می دهیم. سپس روش های موجود برای ضرب ماتریس ها را بررسی  
میکنیم و مزایا و معایب هر کدام را با هم بررسی میکنیم.

در بخش بعدی الگوریتمی که برای پیاده سازی این مهم انجام شده است  
را توضیح میدهیم و دلیل انتخاب آن را به تفصیل خواهیم دید. در این بخش سعی  
شده تا دلایل انتخاب و روند انجام الگوریتم کامل توضیح داده شود.

سپس در بخش بعدی نتایج عملکرد سیستم را بررسی میکنیم. طریقه  
سنتز و نتایج به دست آمده را با گلدن مدل مقایسه میکنیم.

در بخش نهایی نیز به نتیجه گیری خواهیم پرداخت و جمع بندی ای در این  
زمینه که گام بعدی چیست و این سیستم در کجا ها می تواند استفاده شود.

## ۱.۲: کاربردهای ضرب دو ماتریس

ضرب دو ماتریس در علوم کامپیوتر کاربردهای گستردگی دارد و از اساسی ترین دلایلی که سعی داریم تا این عمل را در پایین ترین سطح پردازشی، یعنی سطح ترانزیستور و چیپ انجام دهیم و از انجام آن به صورت نرم افزاری خودداری کنیم، کاربردهای گستردگی آن است.

از گستردگی ترین استفاده های ضرب دو ماتریس میتوان در گرافیک یاد کرد. هر تصویر دیجیتالی را میتوان با یک ماتریس مدل کرد (همانطور که در حال حاضر نیز این امر صورت می پذیرد). هر خانه از این ماتریس را می توان حاوی اطلاعات یکی از پیکسل های این عکس در نظر گرفت که عدد آن ها نشان دهنده ترکیب رنگی آن پیکسل است. حال برای مثال، برای دیکود کردن یک ویدیو دیجیتالی، نیاز به ضرب ماتریس ها داریم. بسیار دیده ایم که برای مثال یک ویدیو را می توان از فرمت  $254 \times 255$  به دیکود کرد و این امر با توجه به بالع بودن نرخ فریم ها در یک ثانیه برای ویدیو های جدید، اگر میخواست در سطح نرم افزار انجام میشد، زمان بسیار زیادی را از کاربر میگرفت.

پژوهشگران MIT یکی از اولین چیپ های مخصوص (ASIC) و بهینه برای کد کردن ویدیو ها که در تلویزیون های با کیفیت بالع استفاده میشود را با استفاده از بهینه سازی ضرب ماتریس ها خلق کرده اند. (1)

گرافیک در کامپیوتر با استفاده از اصطلاح digital signal processing (پردازش دیجیتال سیگنال) بیان میشود. از دیگر کاربردهای ضرب ماتریس ها در این زمینه می توان به تصویر برداری دیجیتال، پردازش سیگنال ها و چندرسانه ای اشاره کرد.

همچنین در کاربرد های جدیدتر برای reinforce کردن یک یادگیری در بینایی ماشینی برای شتابندۀ های سخت افزاری هوش مصنوعی، ضرب ماتریس در مقیاس بالا استفاده میشود.

## ○ ۳.۱: اعداد شناور (Floating Point) در کامپیووتر

تا به اینجا دیدیم که می خواهیم دو ماتریس را در هم ضرب کنیم و هدف بهینه کردن این عمل است. به دلیل اینکه هر خانه از این ماتریس ها می توانند یک عدد اعشاری با ممیز شناور باشند، لازم است تا در ابتدا با این مفهوم و ندوه ضرب آن ها و الگوریتم های متفاوت در این زمینه توضیحاتی ارائه شود.

## ○ ۳.۲: تاریخچه پیدایش نمایش اعداد شناور (Floating Point)

نمایش اعداد باینری ممیز شناور در سال ۱۹۷۰ به وسیله استاندارد IEEE 754 پیاده سازی شد. این نمایش مخصوص اعداد اعشاری است. چون نمایش Floating point نسبت به حالت قبلی نمایش اعداد اعشاری (fixed point)، سخت افزار پیچیده‌تر و کستردوه‌تری دارد، در گذشته تمام کامپیووتر ها نمی توانستند این نوع سخت افزار را در خود CPU اصلی داشته باشند. به همین دلیل معمولاً در گذشته، واحد پردازش FP توسط یک سخت افزار جانبی پیاده سازی می‌شد و در کنار CPU اصلی قرار می‌گرفت. اما اکنون با پیشرفت تکنولوژی، اکثر کامپیووتر ها در قالب یک کارت گرافیک جانبی و کاملاً مستقل از CPU اصلی، می‌توانند از امکانات پردازش اعداد اعشاری به شیوه FP استفاده کنند.

## ۱.۳.۲ : کاربرد ها و مقایسه با نمایش ممیز ثابت(fixed)

### (point

در نمایش fixed point، می‌دانیم که ۲۴ بیت به بیت‌های integer و ۰ بیت هم به بیت‌های fraction اختصاص داده شده است. در این نوع از نمایش، عملیات decode و برنامه‌ریزی مدارمان بسیار ساده است و به راحتی می‌توانیم قسمت‌های صحیح و اعشاری را از هم تفکیک کنیم و معادل دسیمال آن را به دست بیاوریم. در مقابل، در نمایش Floating Point، فهم و decode مدار بسیار سخت‌تر است. اگر بخواهیم به طور خلاصه به مزايا و معایب این دو روش نمایش اشاره کنیم، می‌توانیم به جدول‌های زیر استناد کنیم:

### Floating Point مخصوص

معایب	مزایا
عملیات‌های حسابی، سخت‌تر انجام می‌شوند.	کستره بیشتری از اعداد را شامل می‌شود
سخت افزار پیچیده‌تری دارد و زمان بر هم می‌باشد.	دققت بیشتری خواهیم داشت. (به دلیل وجود عنصر exponent)

## مخصوص Fixed Point

معایب	مزایا
دقیق کمتری دارد.	ساخت افزار ساده‌تری دارد.
کسری کمتری از اعداد را شامل می‌شود.	محاسبه و decode ساده‌تر است.

○ م.م.ا : استاندارد IEEE754 برای نمایش اعداد با ممیز

### شناور

استاندارد 754 IEEE برای نمایش floating point، به دو صورت می‌باشد که هر کدام را در ادامه توضیح می‌دهیم:

#### single precision .I

در این حالت، ۳۲ بیت در اختیار داریم که از این ۳۲ بیت، یک بیت مختص به بیت علامت (sign) است، ۸ بیت برای قسمت exponent و ۲۴ باقیمانده مخصوص قسمت fraction یا mantissa را تخصیص این ۳۲ بیت را نشان داد:

31	30	...	24	23	22	21	...	1	0
S	Exponent				Fraction				

در استاندارد IEEE 754، نهایش biased exponent به صورت exponent است. علت اینکه از این نوع از نهایش بیت‌ها استفاده می‌کنیم، این است که مقایسه اعداد در FP ساده‌تر شود. در اینجا چون ۰ بیت برای exponent داریم، از بایاس ۱۲۷ استفاده می‌کنیم. بایاس ۱۲۷ به این صورت است که عدد ۰- در واقعیت، با عدد ۱ مپ می‌شود و عدد ۱۲۷- با عدد ۰ مپ می‌شود و همینطور الی آخر. در قسمت exponent، مپ شده اعداد را به جای خود آنها قرار می‌دهیم. برای به دست آوردن معادل دسیمال عددمان، به طریق زیر عمل می‌کنیم:

$$N = (-1)^{\text{signbit}} \times (1 + \text{fraction}) \times 2^{\text{exponent\_real}}$$

که در فرمول صفحه قبل، exponent\_real برا بر حاصل عملیات بایاس معکوس روی ۰ بیت exponent می‌باشد و در واقع ما در اینجا، مقدار واقعی exponent را در نظر می‌گیریم (که این کار با ۱۲۷ واحد کم کردن از exponent هشت بیتی در عددمان انجام می‌شود).

### :double precision -2

در اینجا، به منظور افزایش دقت و همچنین افزایش گستره اعدادمان، از ۶۴ بیت استفاده می‌کنیم که تخصیص آن به صورت زیر خواهد شد:

- یک بیت علامت
  - ۱۱ بیت برای exponent
  - ۵۰ بیت برای fraction
- و نحوه کردن آن، به صورت زیر است:

31	30	...	21	20	19	18	...	1	0
S	Exponent				Fraction				
31	30	...	24	23	22	21	...	1	0
Fraction (continued)									

در اینجا، مقدار بایاس برابر  $1 \cdot 2^3$  خواهد شد چون || بیت برای exponent در نظر گرفته‌ایم. بقیه چیزها همانند حالت single precision است و تفاوتی وجود ندارد. نمایش دسیمال عددمان در این حالت، برابر خواهد شد با:

$$N = (-1)^{\text{signbit}} \times (1 + \text{fraction}) \times 2^{\text{exponent\_real}}$$

که در اینجا، exponent\_real مقدار واقعی توانمان، بدون بایاس است که از کم کردن عدد  $1 \cdot 2^3$  از || بیت موجود در قسمت exponent به دست می‌آید. حالا به بررسی یک سری حالت خاص می‌پردازیم و در هر کدام مشخص می‌کنیم که عددمان به چه صورت است:

Single precision		double precision		Number
exponent	fraction	exponent	fraction	-
0	0	0	0	0
0	NonZero	0	NonZero	denormalized
1-254	Anything	1-2046	Anything	FP number
255	0	2047	0	Infinity
255	NonZero	2047	NonZero	Not a Number

در انتهای، یک توضیح در مورد اعداد می‌دهیم. اگر مطابق جدول بالا، تمام بیت‌های exponent صفر باشند و قسمت fraction ناصفر باشد، با یک عدد denormalized موافقیم. این اعداد، به ما کمک می‌کنند که بتوانیم اعداد بسیار کوچک و نزدیک به صفر را هم نمایش دهیم. یعنی به عبارت دیگر، دقیقاً مان بالاتر می‌روند. این اعداد به این صورت عمل می‌کنند که در نمایش آنها به صورت دسیمال، دیگر عدد یک را به علاوه fraction نمی‌کنیم بلکه تنها خود exponent را وارد محاسبات می‌کنیم. بر اثر این کار، توان عدد ۲ یا همان مان قابل افزایش می‌شود به این علت که اگر بخواهیم عددمان را به صورت نماد علمی بنویسیم، باید در توانی از ۲ ضرب کنیم تا عدد یک، پشت اعشار بیفتد. به همین علت، exponent، مقدار قدرمطلق (absolute) توان، قابل افزایش است که در نتیجه آن، می‌توانیم اعداد کوچک با دقت بالا را هم نمایش دهیم. کوچکترین عدد مثبتی که در نمایش denormalized می‌توانیم نمایش دهیم، به صورت زیر است:

$$0.00000000000000000000000001 \times 2^{-126} = 1.0 \times 2^{-149}$$

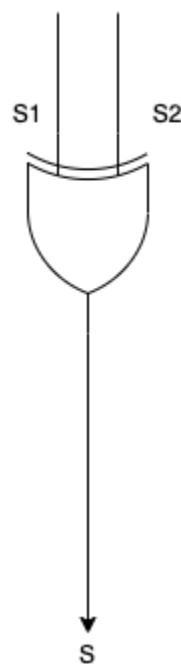
## ۱۴.۳.۱: ضرب دو عدد اعشاری با ممیز شناور ○

برای محاسبه ضرب دو ماتریس با خانه های شامل اعداد اعشاری با ممیز شناور، فارغ از روشی که قرار است برای ضرب ماتریس ها در نظر بگیریم، نیاز است تا دو به دو خانه هایی مشخص را در هم ضرب کنیم. پس نیاز است تا روش ضرب دو عدد اعشاری را بررسی کنیم.

به طور کلی این عمل در ۳ گام صورت می پذیرد:

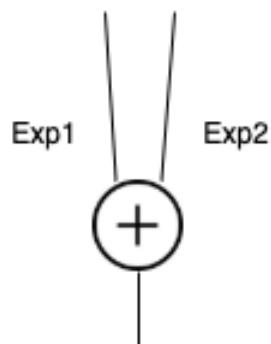
### ۱. محاسبه علامت:

برای این کار کافی است تا علامت دو عدد را با هم xor کنیم.



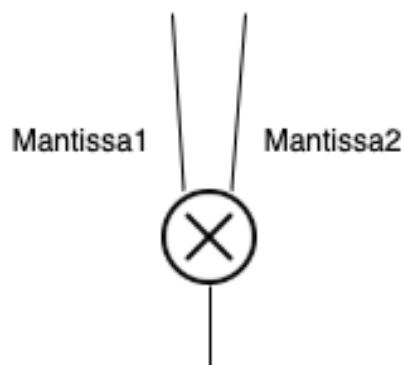
## 2. محاسبه بخش توانی (Exponent)

برای محاسبه بخش توانی در جواب، همانطور که در ضرب های عادی هم انجام میشود، این توان ها را با هم جمع میکنیم.

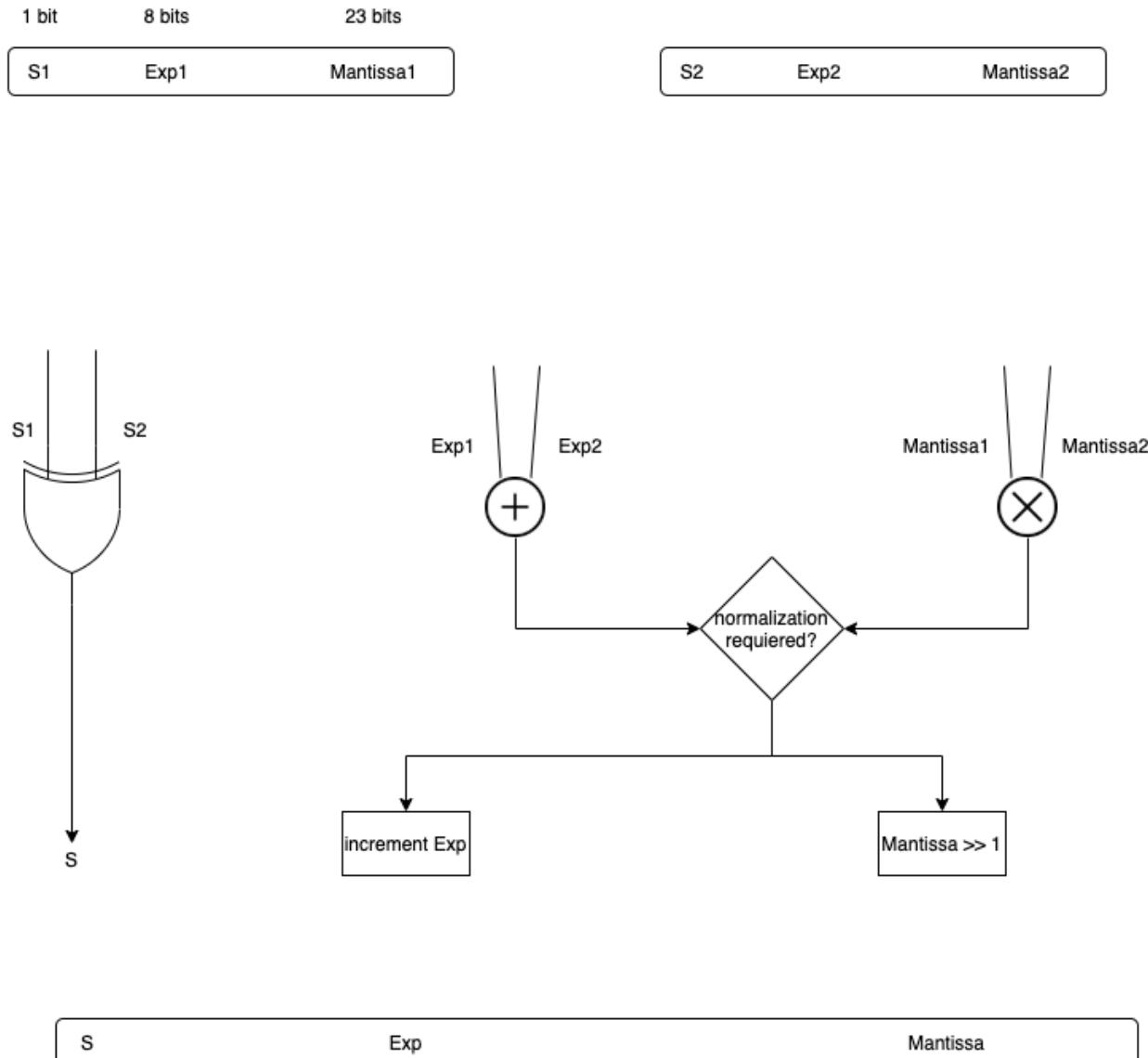


## 3. محاسبه بخش عدد علمی (Mantissa)

این دو عدد را با هم ضرب میکنیم.



## شکل کلی مدار ضرب دو عدد اعشاری با ممیز شناور:



حال همانطور که مشهود است، باید بخش mantissa در این دو عدد با هم ضرب شوند که به دلیل طول بیشتر این بخش نسبت به دو بخش دیگر (۲۳ بیت)،

این کار کندتر رخ می دهد. برای همین مرحله بعدی بهینه سازی ضرب این دو بخش خواهد بود.

### ۳.۴.۱: روش های ضرب دو عدد

بعد از بررسی شیوه ضرب دو عدد اعشاری با ممیز شناور، دریافتیم که تنها جایی که نیاز به بهینه سازی دارد، ضرب دو عدد در بخش mantissa هر کدام از اعداد اعشاری است.

برای ضرب دو عدد، روش های متعددی وجود دارد و انتخاب هر کدام از آنها، مسئله انتخاب بین سرعت و استفاده از سخت افزار کمتر است. به طوری که برخی از روش ها با وجود سرعت بالا در محاسبه، نیازمند تعداد بالایی منابع سخت افزاری هستند، اما برخی دیگر می توانند این کار را در زمانی طولانی تر اما با منابع سخت افزاری محدود تر انجام دهند.

۳ روش معروف برای این کار که به بررسی آن ها خواهیم پرداخت عبارتند

:j|

- تقسیم و غلبه (Divide & Conquer)
- درخت والاس (Wallace Tree)
- ضرب کننده براون (Braun Multiplier)

#### تقسیم و غلبه:

این روش بیشتر در FPGA ها به کار میروند که همین امر باعث سرعت بیشتر میشود. به دلیل موازی سازی های که در روش انجام می شود، مساحت به کار رفته در آن و همچنین قطعات مورد استفاده میتوانند بیشتر شوند. این روش به روش کاراتسوبا نیز معروف است. طرز کار این الگوریتم به این صورت است که ابتدا هر کدام از دو عدد را به نیم تقسیم کرد.

Ah

Al

Bh

Bl

بر اساس تقسیم بندی ای که کردیم، عبارت نهایی پس از ضرب باید به شکل زیر باشد:

$$Ah \cdot Bh \cdot 10^n + AlBl + AhBl10^{n/2} + AlBh10^{n/2}$$

سپس ۳ عبارت زیر را محاسبه می کنیم:

1.  $Ah \cdot Bh$
2.  $Al \cdot Bl$
3.  $(Ah + Al) \cdot (Bh + Bl)$

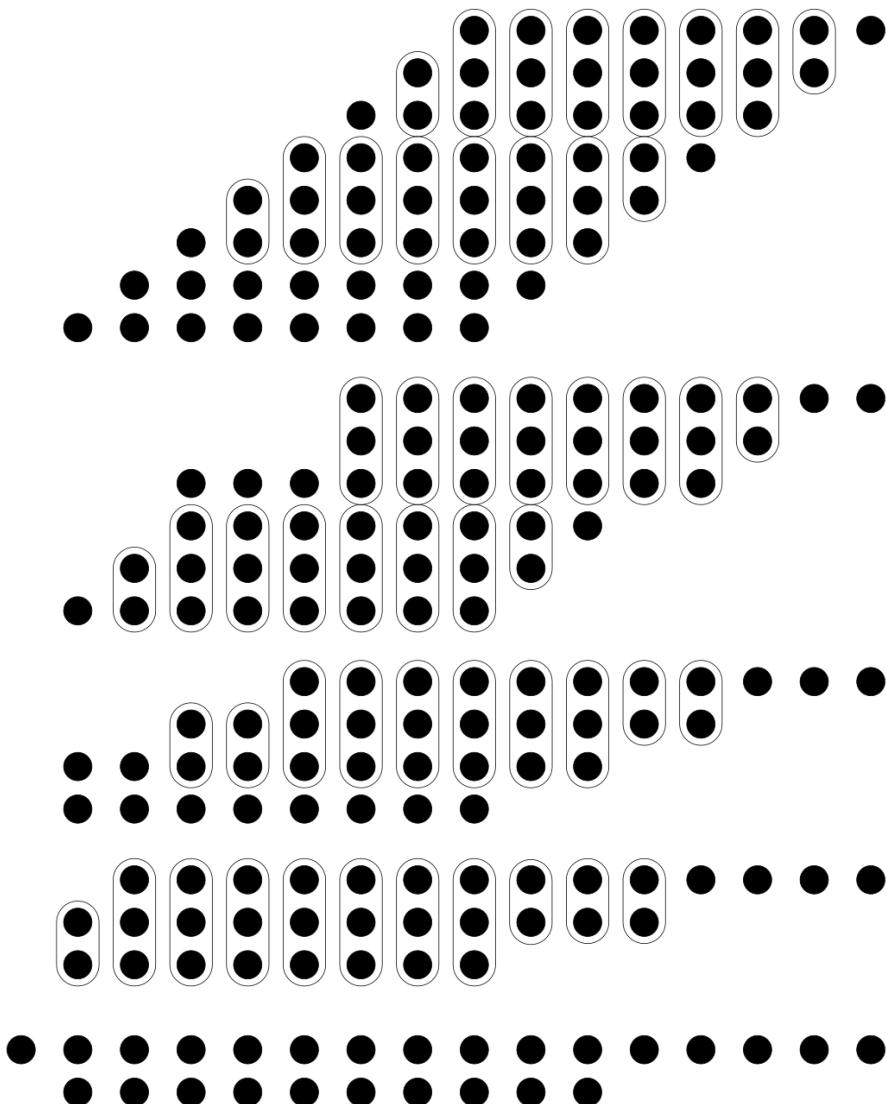
بعد از محاسبه این ۳ عبارت، می توانیم به سادگی عبارت نهایی را محاسبه کنیم. پیچیدگی زمانی نهایی این محاسبات برای رسیدن به جواب نهایی، از مرتبه زمانی  $O(n^{\log(3, 2)})$  خواهد بود.

### درخت والس:

درخت والس در هر سطح از ضرب، تعداد بیت‌های کمتری دارد ولی تأخیر به صورت موازی تقسیم نمی‌شود. با توجه به ارتباطات میانی بسیاری که درخت والس دارد، مساحت این روش زیاد می‌شود.

صورت کلی انجام این روش برمیگردد به جمع ریز-ضرب هایی که انجام میدهیم.

شکل کلی آن در زیر قابل مشاهده است:

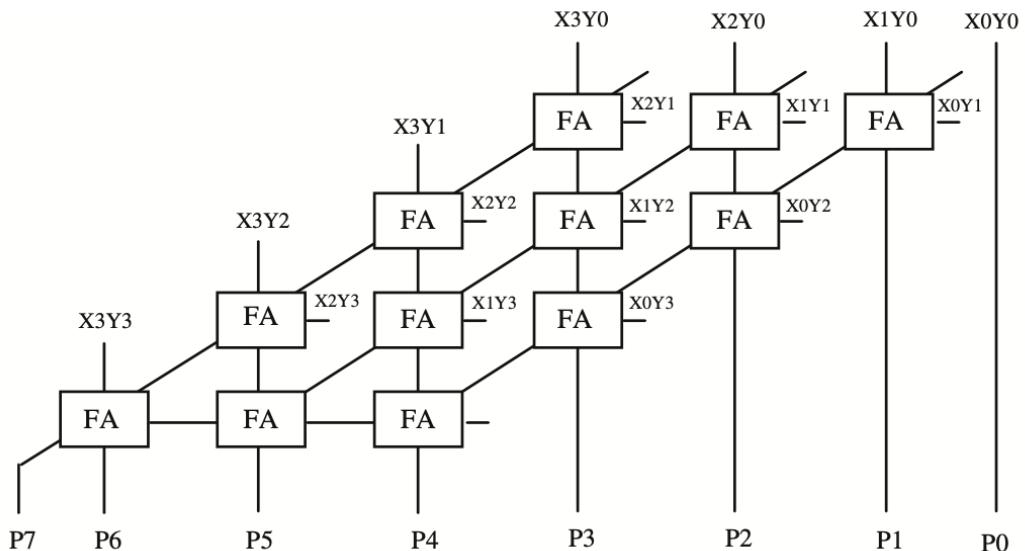


تصویر بال مربوط به یک درخت والدس  $0^*0$  است که ۴ لایه از آن کاہش یافته است. هر کدام از نقطه ها نمایانگر یک بیت با ارزش وزن یکسان هستند. این عمل کاہشی با استفاده از  $2^0$  و  $half-adder$  انجام شده که

همانطور که در ابتدای این بخش نیز بیان شد، کاهش زمانی انجام یک عملیات (که در اینجا با کاهش لایه ها قابل مشاهده است)، به قیمت افزایش استفاده از منابع سخت افزاری است.

### ضرب کننده بران (آرایه ای):

این روش برخلاف روش قبل، توزیع تاخیر یکسانی برای تمام سطوح دارد. این روش در اصل همان ضرب آرایه ای است که از دوران مدرسه آن را آموخته ایم. شکل آن به صورت زیر است:



این روش از بدترین پیچیدگی زمانی با مرتبه زمانی  $O(n^3)$  برخوردار است. اما مزیت استفاده از این روش، کاهش پیچیدگی پیاده سازی و استفاده از منابع سخت افزاری خواهد بود.

## ۴:۱. بررسی الگوریتم های مختلف ضرب دو ماتریس در یکدیگر

### ۱.۱.۱: اهمیت ضرب ماتریس ها

ضرب ماتریس ها در یکدیگر، یکی از عملیات های پرکاربرد در بسیاری از الگوریتم های آنالیز عددی است. به همین دلیل تلاش های فراوانی برای بهبود و بهینه سازی آن صورت گرفته است.

### ۱.۱.۲: ضرب پذیر بودن دو ماتریس

برای ضرب دو ماتریس باید ابتدا توجه داشت که حتی این ماتریس ها قابل ضرب شدن در یکدیگر باشند. شرط لازم و کافی برای معتبر بودن عملیات ضرب دو ماتریس A و B به شکل  $A^*B$  این است که تعداد ستون های ماتریس A برابر تعداد سطر های ماتریس B باشد. به عبارت دیگر اگر ماتریس A، اندازه  $n^m$  داشته باشد؛ ماتریس B باید  $q^*n$  باشد. در نهایت خواهیم دید که ماتریس حاصل  $m^*q$  خواهد بود.

با توجه به تعریف فوق، اگر عبارت  $A^*B$  از نظر ریاضی معتبر باشد؛ لزوماً  $B^*A$  معتبر نخواهد بود. (عدم خاصیت جابجا پذیری ضرب ماتریسی)

## • ۳.۴.۱: چکونگی ضرب دو ماتریس

حال فرض می کنیم ضرب ماتریسی  $A^*$  $B$  از نظر ریاضی مععتبر باشد. برای انجام این عملیات باید از سطر ابتدایی ماتریس  $A$  شروع کرد و درایه های آن را یکی پس از دیگری در درایه های ستون اول ماتریس  $B$  ضرب می کنیم. در نهایت تمام این حاصل ضرب ها را با یکدیگر جمع کرد و در درایه اول ستون اول ماتریس حاصل قرار می دهیم. شکل های زیر نحوه ضرب درایه ها در یکدیگر و تشکیل اولین درایه ماتریس حاصل ضرب را نشان می دهد.

$$\begin{aligned}
 A \times B &= [-1 \quad 2 \quad 0 \quad 3 \quad -5] \times \begin{bmatrix} -2 \\ 3 \\ 7 \\ -1 \\ -2 \end{bmatrix} \\
 &= [(-1) \times (-2) + 2 \times 3 + 0 \times 7 + 3 \times (-1) + (-5) \times (-2)] \\
 &= [2 + 6 + 0 + (-3) + 10] = [15] = 15
 \end{aligned}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 58 \end{bmatrix}$$

"Dot Product"

به طریق مشابه تمام سطر های ماتریس  $A$  را در ستون متناظر با آن در ماتریس  $B$  ضرب کرد و در نهایت جواب حاصل را در درایه با شماره همان سطر و ستون در ماتریس نهایی قرار می دهیم.

## ۱۴.۱۴.۱: الگوریتم های ضرب دو ماتریس

همان طور که بیان شد؛ تلاش های بسیاری برای یافتن روش های سریع برای ضرب دو ماتریس انجام شده است. در ادامه به برخی از الگوریتم های مشهور در این زمینه می پردازیم.

### الگوریتم پیمایشی (بدیره)

$c_{ij} = \sum_{k=1}^m a_{ik} b_{kj}$ . همان طور که گفته شد هر درایه ماتریس حاصل ضرب به کمک رابطه زیر محاسبه می شود. بنابراین کافی است با پیمایش روی سطر و ستون ها این درایه را محاسبه کنیم. شبیه کد این الگوریتم در ادامه آمده است.

- Input: matrices  $A$  and  $B$
- Let  $C$  be a new matrix of the appropriate size
- For  $i$  from 1 to  $n$ :
  - For  $j$  from 1 to  $p$ :
    - Let sum = 0
    - For  $k$  from 1 to  $m$ :
      - Set sum  $\leftarrow$  sum +  $A_{ik} \times B_{kj}$
      - Set  $C_{ij} \leftarrow$  sum
  - Return  $C$

فرض کنیم اندازه ماتریس های  $A$  و  $B$  به ترتیب  $n \times q$  و  $m \times n$  باشد. از آنجا که ما روی تمام سطر های ماتریس اول پیمایش کردیم و در هر پیمایش روی تمام ستون های ماتریس  $B$  پیمایش می کنیم،  $O(mq)$  بار نیاز به عملیات ضرب داریم. اما هر عملیات ضرب نیز  $O(n)$  زمان نیاز دارد. بنابراین می توان گفت زمان

اجرای این الگوریتم  $O(mnq)$  می باشد. اگر همه ابعاد ماتریس شا در یک  $n$  باشد میتوان گفت ضرب ماتریس به کمک این روش در زمان  $O(n^3)$  انجام می شود.

### الگوریتم تقسیم و غلبه | Divide-and-conquer

در این روش ماتریس را به بلوک های کوچک تقسیم کرد و در هم ضرب می کنیم. در نهایت به کمک این بلوک های کوچکتر که در هم ضرب شده اند؛ ماتریس اصلی را تشکیل می دهیم. در واقع با هر بلوک از ماتریس ها مانند یک درایه بروخورد می کنیم. سپس این کار را به شکل بازگشتی انجام می دهیم تا به ماتریس های  $1 \times 1$  برسیم. از در این الگوریتم به شکل زیر محاسبه می شود و برا بر  $O(n^2)$  می باشد.

$$T(n) = 7T(n/2) + O(n^2)$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}_A \times \begin{bmatrix} e & f \\ g & h \end{bmatrix}_B = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}_C$$

A, B and C are square metrices of size  $N \times N$

a, b, c and d are submatrices of A, of size  $N/2 \times N/2$

e, f, g and h are submatrices of B, of size  $N/2 \times N/2$

## الگوریتم استراسن

این الگوریتم نیز مشابه الگوریتم تقسیم و غلبه است. با این تفاوت که برای محاسبه ماتریس اصلی کافی است به جای 8 عملیات، 7 عملیات ضرب ماتریسی انجام دهیم. ندوه انجام این الگوریتم و پیچیدگی زمانی آن در ادامه آمده است.

$$\begin{array}{ll}
 p1 = a(f - h) & p2 = (a + b)h \\
 p3 = (c + d)e & p4 = d(g - e) \\
 p5 = (a + d)(e + h) & p6 = (b - d)(g + h) \\
 p7 = (a - c)(e + f) &
 \end{array}$$

The  $A \times B$  can be calculated using above seven multiplications.

Following are values of four sub-matrices of result C

$$\left[ \begin{array}{c|c} a & b \\ \hline c & d \end{array} \right] \times \left[ \begin{array}{c|c} e & f \\ \hline g & h \end{array} \right] = \left[ \begin{array}{c|c} p5 + p4 - p2 + p6 & p1 + p2 \\ \hline p3 + p4 & p1 + p5 - p3 - p7 \end{array} \right]$$

A                      B                      C

A, B and C are square metrices of size  $N \times N$

a, b, c and d are submatrices of A, of size  $N/2 \times N/2$

e, f, g and h are submatrices of B, of size  $N/2 \times N/2$

p1, p2, p3, p4, p5, p6 and p7 are submatrices of size  $N/2 \times N/2$

پیچیدگی زمانی این الگوریتم به کمک قضیه اصلی قابل محاسبه است:

$$\begin{aligned}
 T(n) &= 7T(n/2) + O(n^2) \\
 &= O(n \log(7, 2)) \approx O(n^{2.807})
 \end{aligned}$$

## بخش ۲ : توصیف معماری سیستم:

### ۲.۱ : معرفی کلی الگوریتم

ابتدا الگوریتم را به صورت اجمالی توضیح می‌دهیم. در مازول های طراحی شده‌مان، قابلیت این را داریم که دو بلک ماتریسی  $\Gamma^*$   $\Gamma$  را در هم ضرب کنیم. پس در ابتدای امر، ماتریس های اصلی که مد نظرمان است و می‌خواهیم آنها را در هم ضرب کنیم را به بلک های  $\Gamma^*$   $\Gamma$  تقسیم‌بندی می‌کنیم. (اگر احیاناً ابعاد ماتریس هایمان، فرد بودند، مسئولیت واحد حافظه است که با قرار دادن سطر و ستون ها به تعداد لازم و با محتویات و درایه های صفر، ابعاد ماتریس ها را زوچ کند تا قابل تقسیم به  $\Gamma$  باشند). حال از روش iterative (پیمایشی)، یکی یکی، بلک های  $\Gamma^*$   $\Gamma$  از هر دو ماتریس را از حافظه اصلی مان می‌خواهیم و در هم ضرب می‌کنیم (ضرب بلک ها، مانند ضرب ماتریس های معمولی است). برای راحتی و سرعت بخشیدن به الگوریتم، دو تا واحد محاسبه (core) داریم که مستقل از هم، ولی نه همزمان، محاسبات را انجام دهند و هر یکی از core ها، مشغول دو بلک متفاوت از ماتریس هایمان باشد. علت اینکه نباید این دو core با هم، همزمان عمل کنند، این است که معکن است در استفاده مشترک از منابع، به مشکل و برخورد conflict می‌رسد. پس از اینکه کار هر کدام از core ها تمام شد، نتیجه ضرب در بلک مربوطه از ماتریس اصلیمان، در حافظه اصلی نوشته خواهد شد و عملیاتی طبق الگوریتمی که در ادامه به آن خواهیم پرداخت، در مموری رخ می‌دهد.

نحوه تخصیص core ها به این صورت است که عملیات مربوط به جفت سطر های با شروع فرد (مثل سطر های  $\Gamma$  و  $\Gamma^*$ )، با اول انجام می‌شود و عملیات

های مربوط به جفت سطر های با شروع زوچ، با core دوم انجام می‌شود. روند کلی اجرای الگوریتم توسط یک ماشین حالت کنترل می‌شود که شرح آن در ادامه آورده شده است:

برای هر Core، ماشین حالت جداگانه داریم. حالت متفاوتی که هر core میتواند، داشته باشد، حالت زیر هستند:

1. آدرس بلای خواسته شده توسط core، خوانده است یا نه. توسط این آدرس، درایه ها خوانده خواهند شد. هندل کردن اینکه چه مقدار باید در حافظه، با توجه به ابعاد ماتریس، پیش روی کنیم، در اینجا انجام می‌شود.

2. مقادیر درایه هایی که در آدرس های داده شده از حافظه موجود بودند، خوانده شده اند یا نه. پس از خوانده شدن، آنها را در متغیر های always block کمکی شان که از نوع reg هستند تا بتوانیم به راحتی در هایمان با آنها کار کنیم، ریخته می‌شوند. باید توجه شود که در اینجا، مثل آنکه برای core اول ماشین حالت را تشکیل می‌دهیم، باید در این state، دو اسماع باشد که core دوم هم در همین state نباشد تا به مشکل بخوریم. این موضوع با قرار دادن یک ساختار شرطی if-else هندل می‌شود.

3. ضرب بلای ها و اعداد موردنیاز تا زمانی که عملیات ضرب تمام شود. برای بلای های  $\Gamma * \Gamma$  که در اینجا داریم، ۰ تا نمونه (instance) از ماژول multiplier نیاز داریم تا ضرب ها را انجام دهد.

4. عملیات جمع در ضرب ماتریسی که حاصل ضرب ها را با هم جمع می‌کنیم، انجام شده است یا نه. برای بلای های  $\Gamma * \Gamma$  که در اینجا داریم، به ۴ تا نمونه از ماژول adder نیاز داریم.

5. حاصل عملیات  $\phi$ , در حافظه ذخیره شده است یا نه. (در واقع منظور عملیات نوشتن در حافظه است که با سیگنال write enable کنترل می‌شود).

6. در این که state نام دارد، چک می‌کنیم که آیا عملیات نوشتن با موفقیت و با اندکی تأخیر (delay) برای جلوگیری از glitch، انجام شده است یا نه.

7. آدرس های داده شده پس از انجام عملیات، یک واحد زیاد شده اند یا نه. (البته به شرط اینکه تعداد عملیات های افزایش، از ابعاد ماتریس بزرگ‌تر نشوند)

8. و در نهایت، یک سیگنال end برای هر core می‌دهیم تا در صورت اتمام کار، متوجه شویم.

تصویری از نام  $\phi$  را در ادامه مشاهده می‌کنیم:

```
parameter s_SET_ADDRESS_FIRST_CORE = 3'b000;
parameter s_SET_ADDRESS_SECOND_CORE = 3'b000;
parameter s_READ_FIRST_CORE_INPUTS = 3'b001;
parameter s_READ_SECOND_CORE_INPUTS = 3'b001;
parameter s_MULTIPLY_FIRST_CORE    = 3'b010;
parameter s_MULTIPLY_SECOND_CORE   = 3'b010;
parameter s_ADD_FIRST_CORE        = 3'b011;
parameter s_ADD_SECOND_CORE       = 3'b011;
parameter s_PUT_FIRST_CORE       = 3'b100;
parameter s_PUT_SECOND_CORE      = 3'b100;
parameter s_FIRST_WAIT           = 3'b101;
parameter s_SECOND_WAIT          = 3'b101;
parameter s_FIRST_INCREASE_COUNTER = 3'b110;
parameter s_SECOND_INCREASE_COUNTER = 3'b110;
parameter s_END_FIRST_CORE       = 3'b111;
parameter s_END_SECOND_CORE      = 3'b111;
```

ابتدا در always block به ماشین حالت است، پیش از هر چیزی، سیگنال reset را چک می‌کنیم که negedge و active low است. اگر قرار نبود که مدار ریست شود، ادامه می‌دهیم و برای هر core عملیات‌هایی را که گفته شد، اجرام می‌دهیم (به وسیله switch case امکان تحقق ماشین حالت را فراهم می‌کنیم)

برای برخی سیگنال‌ها در چند تا استیت مختلف به آنها نیاز داریم، متغیر جدیدی به نام ack تعریف شده که نمایانگر این است که آن متغیر، مقدار جدید را دریافت کرده است یا نه تا در state‌های بعدی، وضعیت آن متغیر‌ها را بدانیم. با یک مثال این کلیت را بیشتر گسترش می‌دهیم. فرض کنیم که دو ماتریس  $3 \times 3$  داریم.

$$\begin{array}{|c|c|c|c|} \hline & \text{A} & & \\ \hline & \text{B} & & \\ \hline \end{array}$$

$$\begin{matrix}
 \begin{bmatrix}
 \text{○} & \text{○} & \text{○} \\
 \text{○} & \text{○} & \text{○} \\
 \text{○} & \text{○} & \text{○} \\
 \text{○} & \text{○} & \text{○}
 \end{bmatrix} & \times & \begin{bmatrix}
 \text{○} & \text{○} & \text{○} \\
 \text{○} & \text{○} & \text{○} \\
 \text{○} & \text{○} & \text{○} \\
 \text{○} & \text{○} & \text{○}
 \end{bmatrix} & = \begin{bmatrix}
 \text{○} & \text{○} & \text{○} \\
 \text{○} & \text{○} & \text{○} \\
 \text{○} & \text{○} & \text{○} \\
 \text{○} & \text{○} & \text{○}
 \end{bmatrix}
 \end{matrix}$$

در ابتدای سیکل، هسته شماره ۱، ۰ عدد زرد رنگ را می‌گیرد و هسته شماره ۲، ۰ عدد آبی رنگ. در ابتدا ماتریس جواب، یک ماتریس  $3 \times 3$  است که تمامی مقادیر آن صفر است. حاصل ضرب این دو ماتریس خرد  $2^*2^*2$ ، جواب ماتریس خرد  $2^*2^*2$  قرمز در ماتریس جواب را محاسبه می‌کند.


Result

### روش پیاده سازی:

برای پیاده سازی این سیستم از زبان verilog استفاده شده است. همانطور که در پژوهه نیز مشهود است، ماتریس ها در اصل آرایه های دو بعدی ای هستند که هر خانه آن ها یک عدد ۳۲ بیتی اعشاری با ممیز شناور با استاندارد IEEE754 هستند. زبان verilog دارای محدودیت برای آرایه های دو بعدی است، برای همین در ادامه توضیحات به این موضوع توجه کنید که برای ارتباط میان مموری و واحد ضرب کننده و جمع کننده، مجبور به استفاده از وکتور شده ایم و مقادیر و آدرس ها به صورت جدا منتقل می شوند لازم به ذکر است که این محدودیت را از بین برد است.

## ۲.۲: اینترفیس های سیستم

در این قسمت، به صورت کلی، واسطه های سیستم را بررسی و تشریح می‌کنیم.

### ۲.۲.۱: ورودی ها

ورودی اصلی ها، دو آدرس و ابعاد ماتریس هاست تا بتوانیم درایه های هر ماتریس را به وسیله آدرس ها به دست بیاوریم. به این صورت عمل می‌کنیم که چون حافظه ها عملد یک بعد دارد (به این معنا که فقط قابلیت گسترش از یک سمعت را دارد چون سمعت دیگر ثابت و ۳۲ بیتی است)، به اندازه ابعاد ماتریس در حافظه‌مان پیش روی می‌کنیم تا درایه های هر سطر و ستون مشخص شوند. به جز از این ورودی های اصلی، یک سری ورودی جانبی نظیر پورت ریست، کلک و write enable داریم. در ورودی دیگر ماژول هایهان، مثل ماژول ضرب دو بلای  $2 \times 2$  در یکدیگر، به تبع ورودی‌ها کمی تغییر خواهد کرد که جلوتر راجع به آن بحث خواهیم کرد.

### ۲.۲.۲: خروجی ها

خروجی اصلی ها، حاصل ضرب دو ماتریس داده شده است که در اولین مکان ممکن در حافظه نوشته خواهد شد. (میتوانیم از ابعاد ماتریس های ورودی، ابعاد ماتریس خروجی را بیابیم. مجدداً به علت تکبعده بودن حافظه‌مان به معنایی که در بالا گفته شد، باید به اندازه ابعاد ماتریس، در حافظه پیش برویم و درایه های ماتریس خروجی را در حافظه ذخیره کنیم. یک

سیگنال خروجی done مه داریم که در صورت اتمام عملیات، تحریک و برابر ا می شود.

### ○ ۳.۲.۳: کلای سیستم

کلای سیستم در حال حاضر روی ۲ نانوثانیه تنظیم شده است. طول آن را نسبتاً زیاد در نظر گرفته ایم تا مشکلی در اجرای دستورات در هر کلای رخ ندهد.

## ۲.۳: مازول ها

در این پروژه از ۳ مازول اصلی سنتز پذیر، دو زیر مازول برای جمع و ضرب اعداد اعشاری با ممیز شناور، یک مازول برای خواندن ماتریس ها از فایل که سنتز پذیر نخواهد بود و یک مازول جهت تست بنچ استفاده شده است.

۳ مازول اصلی شامل مموری، که در خود یک واحد جمع کننده به عنوان واحد عملیاتی دارد، یک واحد کنترل برای کنترل ورودی و خروجی های مموری به واحد جمع کننده و ضرب کننده برای محاسبه ماتریس های خرد، و همچنین یک واحد wrapper است تا بتواند اینترفیس های سیستم را دارا باشد و تمامی واحد ها را در خود پوشش دهد. در ادامه به تفصیل این مازول ها را بررسی می کنیم.

### ۱.۳.۱: واحد ضرب کننده خرد

ضرب دو عدد اعشاری با ممیز شناور در بخش A به تفصیل توضیح داده شد. اگر خانه های ماتریس از اعداد صحیح تشکیل شده بود، نیازی به وجود این مازول نبود و اعداد را به سادگی می توانستیم در یکدیگر با استفاده از عملوند ضرب در وریلانگ ضرب کنیم.

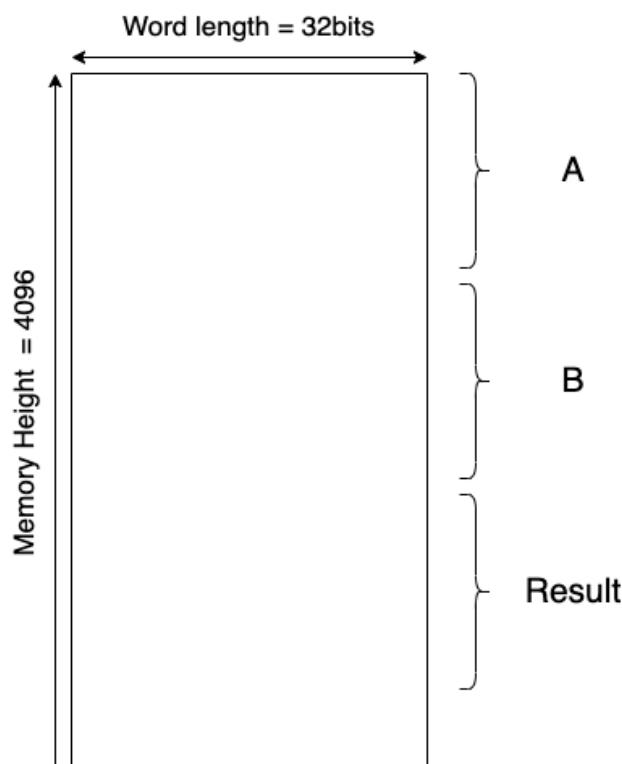
در این مازول با توجه به عددي که داریم، تعداد کلای های مورد نیاز در هر سیکل می تواند متفاوت باشد. این ضرب با جمع کردن دو نما با یکدیگر و xor کردن علامت ها همراه است. بخش چالشی این مازول پس از ضرب کردن بخش کار را با استفاده از روش هایی که در بخش A گفتیم انجام دهیم. برای ضرب دو بلای  $G^*$ ، نیاز به ۰ تا واحد ضرب کننده خرد داریم چون ۰ بار عملیات ضرب داریم.

## ○ م.م.م : واحد جمع کننده خرد

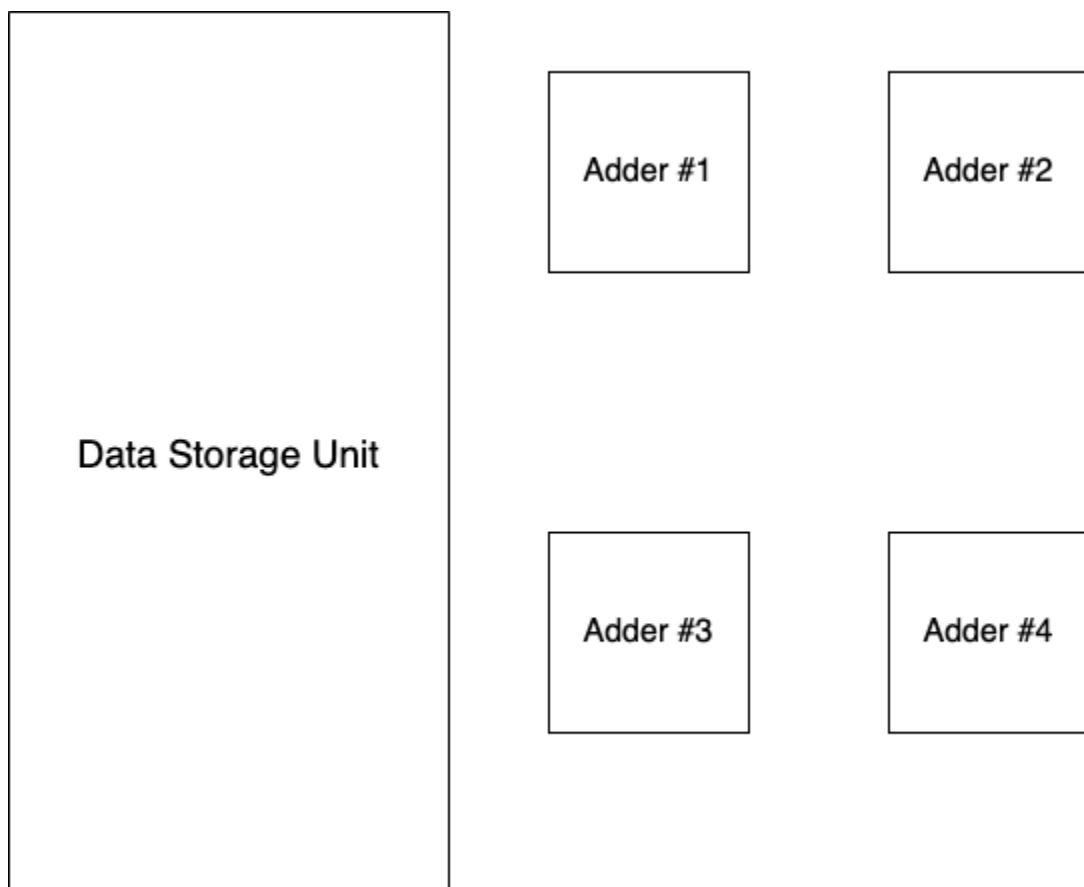
این بخش نیز به دلیل استفاده از اعداد اعشاری با ممیز شناور در خانه های ماتریس ها به وجود آمده است. جمع کننده خرد با اعداد اعشاری با ممیز شناور در فرمت استاندارد IEEE754 کار می کند.

## ○ م.م.م : مصوری

این بخش وظیفه ذخیره سازی دو ماتریس ورودی و ماتریس خروجی را برعهده دارد. هر خانه این مازول، یک رجیستر ۳۲ بیتی است. برای ساخت مموری از یک آرایه به سایز Memory Height کردیم که هر خانه آن یک وکتور ۳۲ بیتی است. دلیل اینکه هر خانه ۳۲ بیتی است، استاندارد IEEE754 و ندوه ذخیره سازی اعداد اعشاری با ممیز شناور است.



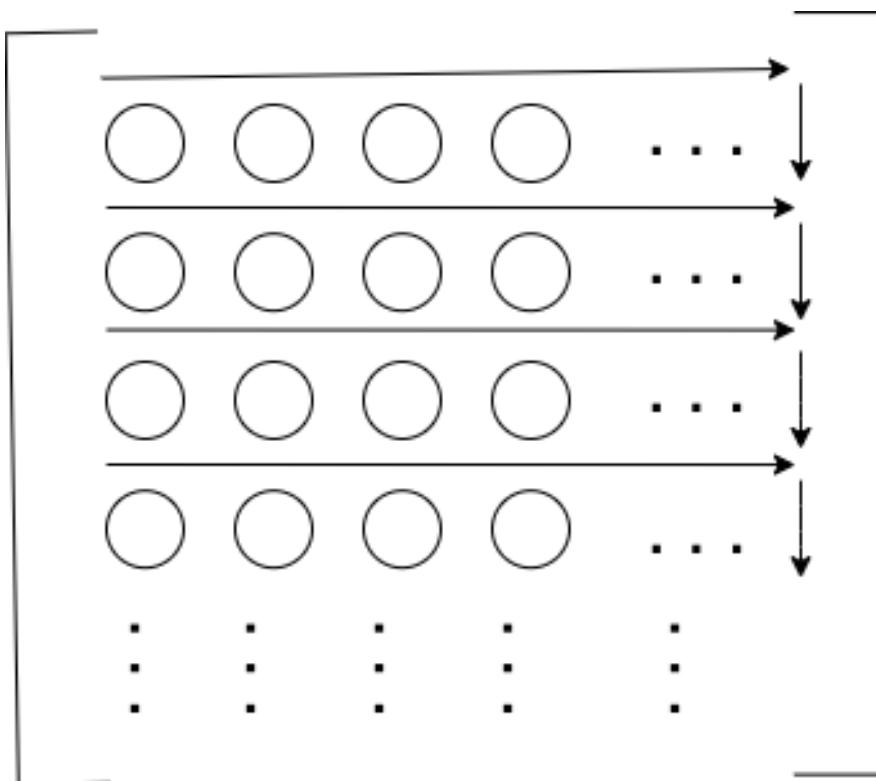
ماژول مموری خود از چندین بخش تشکیل شده است و علاوه بر ذخیره اطلاعات، شامل ۴ ماژول جمع کننده خرد نیز هست تا بعد از دریافت ماتریس جواب خرد، بتواند آن را با جواب های پیشین ادغام کند و ماتریس نهایی را بسازد. پس همانطور که مشاهده می شود واحد کنترلی با رهبری مموری و ضرب کننده و جمع کننده خرد، خانه های دو ماتریس خرد  $G^*G$  را از مموری به ضرب کننده می فرساند و ضرب کننده با کمک جمع کننده خرد، جواب این ضرب را به روش عادی که در بخش ا توضیح داده شد، محاسبه می کند و سپس پاسخ را به مموری باز می گرداند. حال مموری برای محاسبه ماتریس نهایی، نیاز به تعداد جمع کننده دارد. به دلیل این که در هر سیکل تنها ۴ خانه از ماتریس نهایی مستخوش تغییر می شوند، نیاز به ۴ عدد جمع کننده خرد داریم.



## ندوه قرارگیری اطلاعات در واحد ذخیره اطلاعات مموری:

این مسئله که واحد مموری یک داده ساختار تک بعدی مدسوب می شود اما ماتریس های ما (دو ماتریس ورودی و یک ماتریس خروجی)، داده ساختار هایی دو بعدی هستند که هر کدام از آنها دارای سطر ها و ستون هایی هستند. پس لازم است تا برای قراردادن این خانه های این ماتریس ها در واحد ذخیره سازی مموری، یک قرارداد تنظیم کنیم.

همانطور که در بخش ۱.۲.۱ ورودی ها توضیح داده شدند، برای گرفتن ماتریس های ورودی از محیط خارج از سیستم، ابتدا ابعاد ماتریس را گرفته و سپس از چپ به راست خانه های ۳۲ بیتی ماتریس را تک به تک دریافت میکند.



## شیوه ورودی گرفتن ماتریس. از چپ به راست هر سطر کامل می‌شود و سپس سطر بعدی

همانطور که در شکل (بخش ذخیره سازی مموری) نیز مشهود است، خانه‌ای ابتدایی مموری به ذخیره ماتریس اول، خانه‌های بعدی به ذخیره ماتریس دوم و بعد از آن‌ها برای ذخیره ماتریس جواب استفاده می‌شود. ماتریس جواب در ابتدا همگی با مقدار صفر به آنها مقدار اولیه داده می‌شود. دلیل این موضوع تشکیل ماتریس جواب نهایی از حاصل ضرب ماتریس‌های خرد است. این عملیات به این صورت شکل می‌گیرد. برای درک بهتر این موضوع به بخش ۱.۲ مراجعه شود.

### ۱۴.۳.۲: واحد کنترل

وظیفه این واحد، مدیریت ورودی‌ها و خروجی‌های هسته‌های محاسباتی و همچنین ترافیک بین مموری و این هسته‌ها است. در نگاهی کلی، محاسبات یک ماتریس از محاسبه چندین ماتریس خرد تشکیل می‌شود. به همین دلیل برای محاسبه ماتریس‌های خرد و ارتباط آن با ماتریس نهایی نیاز به یک سیستم پردازشی داریم تا علاوه بر جلوگیری از conflict، محاسبات به درستی در سیکل مناسب انجام شوند. واحد کنترل از یک واحد پردازشی برای انجام این مهم و دو هسته محاسباتی که هر هسته شامل ۰ ضرب کننده و ۳ جمع کننده است تشکیل شده است.

هر هسته در این واحد مسئولیت محاسبه ضرب دو ماتریس خرد  $G^*$  را دارد. به صورتی که در ابتدای سیکل، واحد پردازشی ۰ عدد که مربوط به خانه‌های دو ماتریس  $G^*$  است را وارد واحد کنترل می‌کند و به هسته مربوط ارجاع می-

نهاد. حال باید ۰ ضرب به صورت موازی در این هسته انجام شود و سپس حاصل این ۰ ضرب دو به دو با یکدیگر جمع شوند. برای مطالعه بیشتر راجع به چکونگی ضرب ماتریس ها به بخش [۱.۴.۳ مراجعت](#) شود.

پس از انجام ضرب در هر کدام از هسته ها، پاسخ حاصل به همراه آدرسی که باید این ماتریس جواب خرد در آن قرار بگیرد به ممoria بازگردانده می شود.

#### م.۴.۴: واحد wrapper ○

این بخش وظیفه پوشش کلی مازول های دیگر را دارد. همچنین وروگی های اولیه که قرار است از محیط خارج سیستم گرفته شود از طریق این مازول گرفته می شود و پس از آن خروجی های معین از طریق این مازول به محیط خارج از سیستم داده می شود.

#### م.۴.۵: خواندن از فایل ○

این مازول، صرفا جهت نمونه گیری نرم افزاری نوشته شده است و سنتز نمی شود. برای استفاده از این مازول در خط اول سه وروگی به شکل زیر داده می شود:

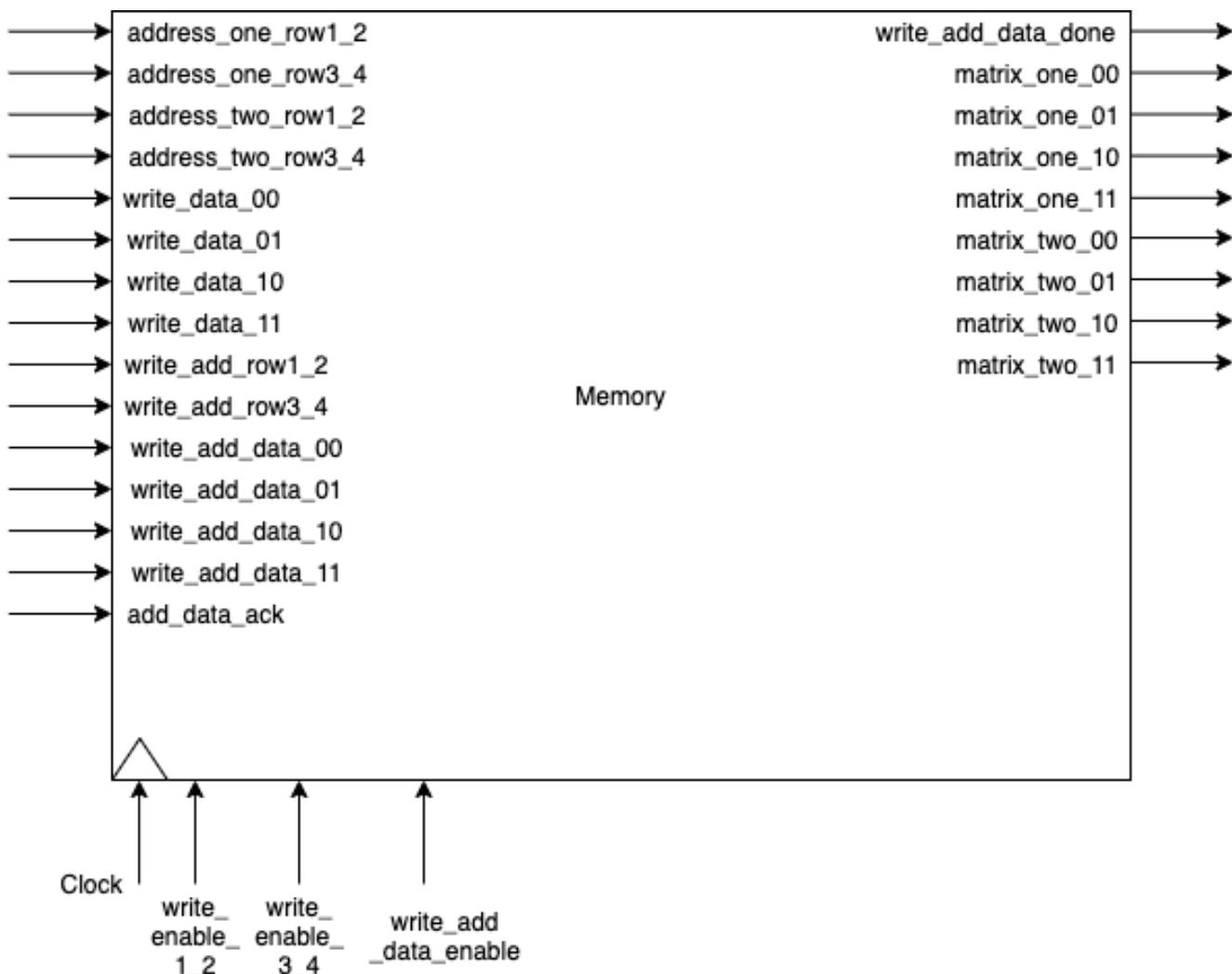
```
"#Row_First_Matrix #Column_First_Matrix #Column_Second_Matrix"
```

عدد میانی در اصل تعداد سطر های ماتریس دوم نیز هست، زیرا میدانیم شرط ضرب پذیری دو ماتریس برابر بودن تعداد سطر های ماتریس دوم با تعداد ستون های ماتریس اول است.

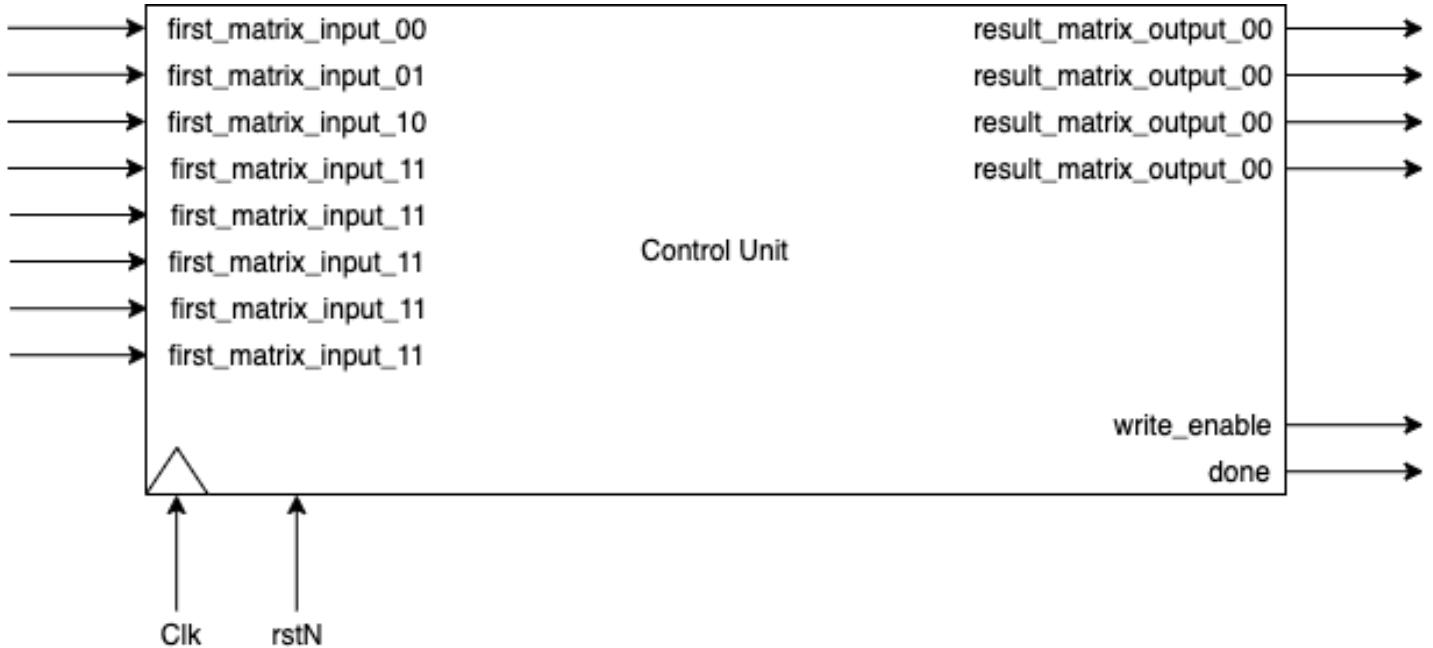
بعد از به همان نحوی که [درایه های ماتریس ها از ممoria به واحد کنترل پاس داده می شوند](#)، درایه های ماتریس را به این مازول وروگی می دهیم.

## ۲.۴ : دیاگرام ساخت افزار •

### ۲.۴.۱ : واحد معمولی ○



## ○ م.م.م : واحد کنترل



در این مازول، تنها ۸ سیگنال پرودی دارد که این به معنی این است که فقط پرودی لازم برای یک هسته که دو ماتریس  $2 \times 2$  را در هم ضرب می‌کند را داریم. برای حل این مشکل، بعد از چند کلادی که تمام پرودی های یک هسته فراهم شد، پرودی هسته دوم نیز گرفته می شود تا هسته دوم به صورت موازی با هسته اول کار کند.

### سیگنال done

این سیگنال بعد از اینکه محاسبات هر دو هسته به پایان رسید فعال میشود تا معموری بعد از دریافت این سیگنال بتواند ماتریس خروجی این مازول را به عنوان پرودی دریافت کند.

## • ۱۲.۵: سلسله مراتب طراحی و ساختار درختی

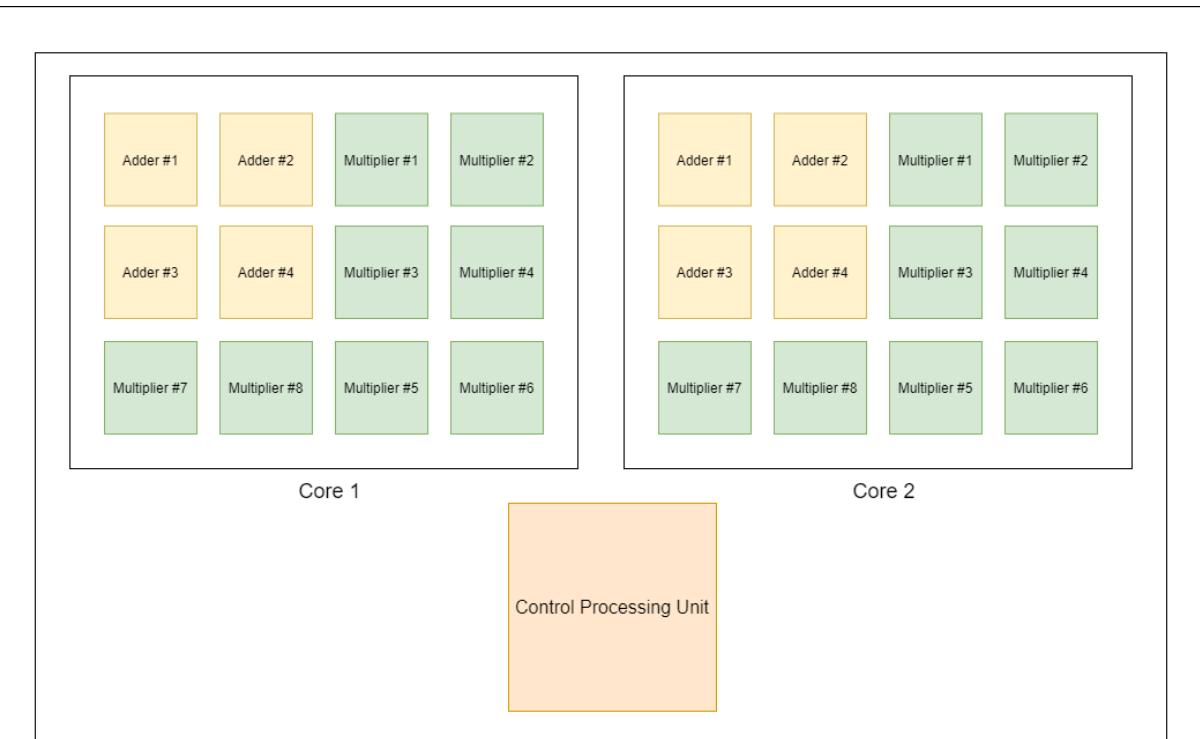
### ○ ۱.۴.۳: شعای کلی سیستم:

این سیستم بر اساس مدلی که در تصویر زیر مشاهده میکنید طراحی شده است.

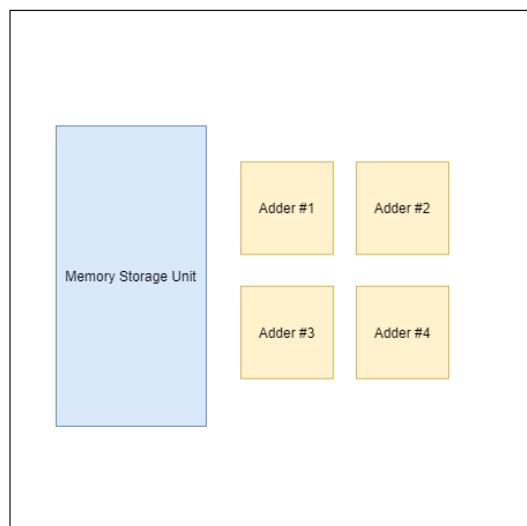
طراحی این سیستم در بالاترین سطح دارای یک ماژول اصلی است که به عنوان wrapper تمامی اجزا را در بر می‌گیرد و مسئولیت دریافت ورودی های سیستم و توزیع آن بین منابع درست و سپس گرفتن خروجی های سیستم و دادن آن به کاربر نهایی است.

در سطح بعدی سیستم دارای یک ماژول مموری و یک ماژول کنترلر است. وظیفه ماژول کنترلر، گرفتن و قرار دادن به موقع اعداد از مموری و محاسبه ماتریس خرد است تا سپس بتواند آن را به مموری پاس بدهد. ماژول مموری نیز پس از دریافت ماتریس های خرد، با استفاده از جمع کننده هایی که دارد، ماتریس نهایی را محاسبه میکند.

در قلب بخش کنترلر، یک واحد پردازشی نیز قرار دارد که وظیفه توزیع درست منابع را دارد. به طوری که هسته ا تنها بتواند ماتریس های  $G^*$  فرد را دریافت کند و محاسبه کند و هسته  $G$  ماتریس های  $G^*$  زوج. پس از اتمام عملیات های نیز این واحد پردازشی خروجی ها را به مموری باز می‌گرداند و با سیگنالی آماده بودن خروجی را اعلام می‌کند.



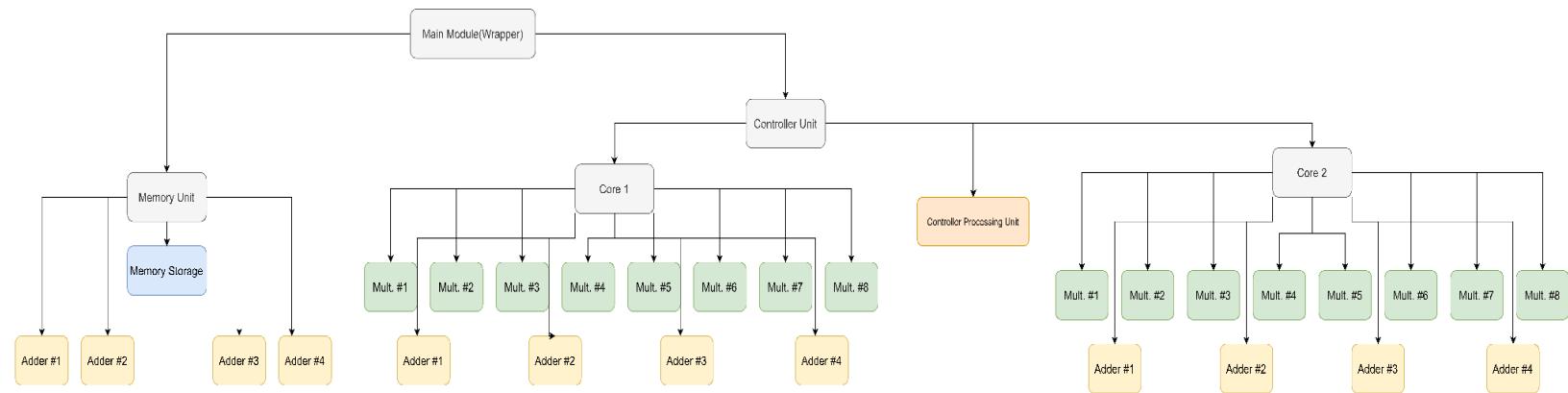
Control Unit



Memory Unit

Main Module

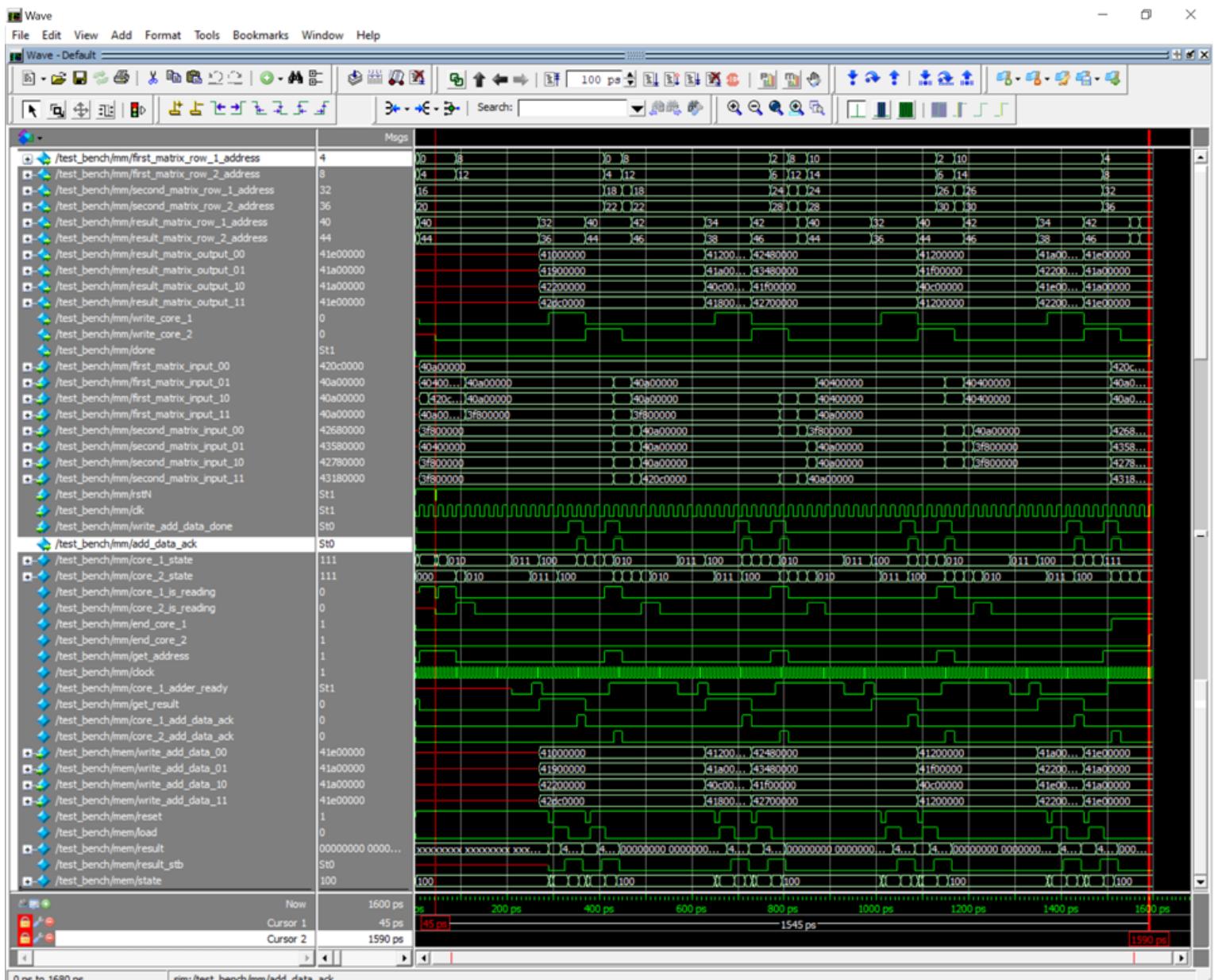
## ۴.۵.۲: ساختار درختی:



## بخش ۳: شبیه سازی و نتایج حاصله

### ۳.۱ • توصیف test bench و سیگنال های آن

ابتدا waveform حاصل از شبیه سازی در Modelsim را نمایش می دهیم.



در ادامه به شرح سیگنال های موجود در waveform بالع خواهیم پرداخت.

در ابتدا باید این نکته را بیان کنیم که ندوه ذخیره سازی ماتریس در حافظه به شکل سطري می باشد. یعنی در یک ماتریس  $3 \times 4$ ، درایه های سطر اول در خانه اول تا چهارم حافظه، درایه های سطر دوم در خانه پنجم تا هشتم حافظه و درایه های سطر سوم در خانه های نهم تا دوازدهم حافظه ذخیره شده اند. آدرس شروع ماتریس دوم نیز بلافاصله بعد از ماتریس اول بوده و ماتریس پاسخ نیز پس از ماتریس دوم در حافظه ذخیره خواهد شد.

### سیگنال ها:

- سیگنال first\_matrix\_row\_i\_address : آدرس درایه های ستون اول سطر ام مربوط به بلک اول.
- منظور از بلک ماتریس های دو در دو می باشد که در الگوریتم شرح داده شده است.
- این نوع از آدرس دهنی به دلیل ساختار موجود در مآژول حافظه ما می باشد.
- سیگنال second\_matrix\_row\_i\_address : آدرس درایه های ستون اول سطر ام مربوط به بلک دوم.
- سیگنال result\_matrix\_row\_i\_address : آدرس درایه های ستون اول سطر ام مربوط به بلک حاصل ضرب دو بلک قبلی.
- سیگنال result\_matrix\_output\_i : بلک حاصل از ضرب بلک های اول و دوم، چهار عدد را در خود ذخیره می کند. این چهار سیگنال هر کدام یکی از مقادیر آن ماتریس را در خود دارند.
- مثلاً Memory[result\_matrix\_row\_0\_address] مبابر خواهد بود با result\_matrix\_output\_00

- در طراحی پروژه ما دو موجود می باشد. سیگنال های write\_core\_1 و write\_core\_2 نشان می دهد که کدام حسنه در حال نوشتن بر روی حافظه می باشد.
- سیگنال done: در پایان برنامه و با اتمام عملیات ضرب، این سیگنال فعال می شود.
- سیگنال first\_matrix\_input\_i: محتوای بلک اول که در حافظه ذخیره شده است را در خود دارد. همان طور که پیش تر گفته شد، آدرس این خانه حافظه به کمک سیگنال first\_matrix\_row\_i\_address دست می آید.
- سیگنال second\_matrix\_input\_i: محتوای بلک اول که در حافظه ذخیره شده است را در خود دارد. همان طور که پیش تر گفته شد، آدرس این خانه حافظه به کمک سیگنال second\_matrix\_row\_i\_address دست می آید.
- سیگنال reset: برای کردن مدار استفاده می شود. این سیگنال ACTIVE LOW می باشد.
- سیگنال clk: برای clock اصلی مدار استفاده می شود.
- سیگنال write\_add\_date\_done: فعال شدن این سیگنال نشان می دهد که عملیات جمع بلک حاصل ضرب با محتوای قبلی حافظه تمام شده و نتیجه در آن خانه overwrite شده است.
- سیگنال add\_data\_ack: این سیگنال برای ارتباط بین حافظه و ضرب کنندو و به عنوان acknowledge استفاده می شود.
- سیگنال core\_i\_state: این سیگنال برای مشخص کردن وضعیت هر state استفاده می شود. در ادامه بیان خواهیم کرد که هر یک ای state بین 000 تا 111 نشان دهنده چه چیزی می باشند.

- 000: تعیین آدرس هایی که ماتریس از روی آنها خوانده می شود در این state صورت می گیرد.
- 001: خواندن محتوای ماتریس در این وضعیت اتفاق می افتد.
- 010: ضرب کردن درایه های بلکهای دو در دو، در این state اتفاق می افتد. بنابراین پروژه ما هشت مأذول ضرب کننده نیاز دارد.
- 011: اعداد حاصله از ضرب بخش قبلی در این state با هم جمع می شوند.
- 100: نتیجه به دست آمده (بلک دو در دو ناشی از ضرب بلک های تشکیل شده)، در حافظه نوشته می شود. (با محتوای موجود جمع می گردد.)
- 101: این وضعیت برای انتظار و عدم ایجاد تأخیر در حافظه استفاده می شود.
- 110: افزایش counter مخصوص به حافظه.
- 111: تمام شدن عملیات.
- سیگنال core\_is\_reading: این سیگنال نشان می دهد که هر یک از هسته ها آیا در حال خواندن از روی حافظه می باشند یا خیر. به کمک آن می توانیم خواندن از روی حافظه را مدیریت کنیم تا هم زمان بیش از یک هسته از حافظه اطلاعات نخواهند. رعایت این مسئله به دلیل ساختار داخلی حافظه ضروری می باشد.
- سیگنال end\_core\_i: این سیگنال نشان می دهد که آیا کار هر هسته در عملیات ضرب به پایان رسیده است یا خیر. هسته اول بلک سطر های فرد و هسته دوم بلک سطر های زوج را ضرب می کند. اگر این سیگنال برای هر دو هسته فعال باشد، سیگنال done روشن شده و برنامه پایان می پذیرد.

- سیگنال get\_address عمل multiplexer یک مانند: این سیگنال علاوه بر مانند می کند و بین آدرس مربوط به بلک اول و دوم انتخاب می نماید. آدری منتخب به حافظه داده خواهد شد تا محتوای آن خوانده شود.
- سیگنال adder: این سیگنال inner clock مربوط به داخلي حافظه است.
- سیگنال get\_result: این سیگنال مشابه سیگنال get\_address عمل می کند با این تفاوت که انتخابی که انجام می دهد بین محتوای دو بلک دو در دو می باشد تا آدرس آنها.
- سیگنال core\_i\_add\_data\_ack: عملیات acknowledg در فرایند جمع با حافظه برای هر هسته به کمک این سیگنال انجام می شود.
- سیگنال write\_add\_data\_i: در این سیگنال محتوایی که باید با خانه های حافظه جمع شود قرار دارد.
- سیگنال های reset, load, result به شکل مشابه برای adder داخلي حافظه موردن استفاده قرار می گیرند.

**نکته:**

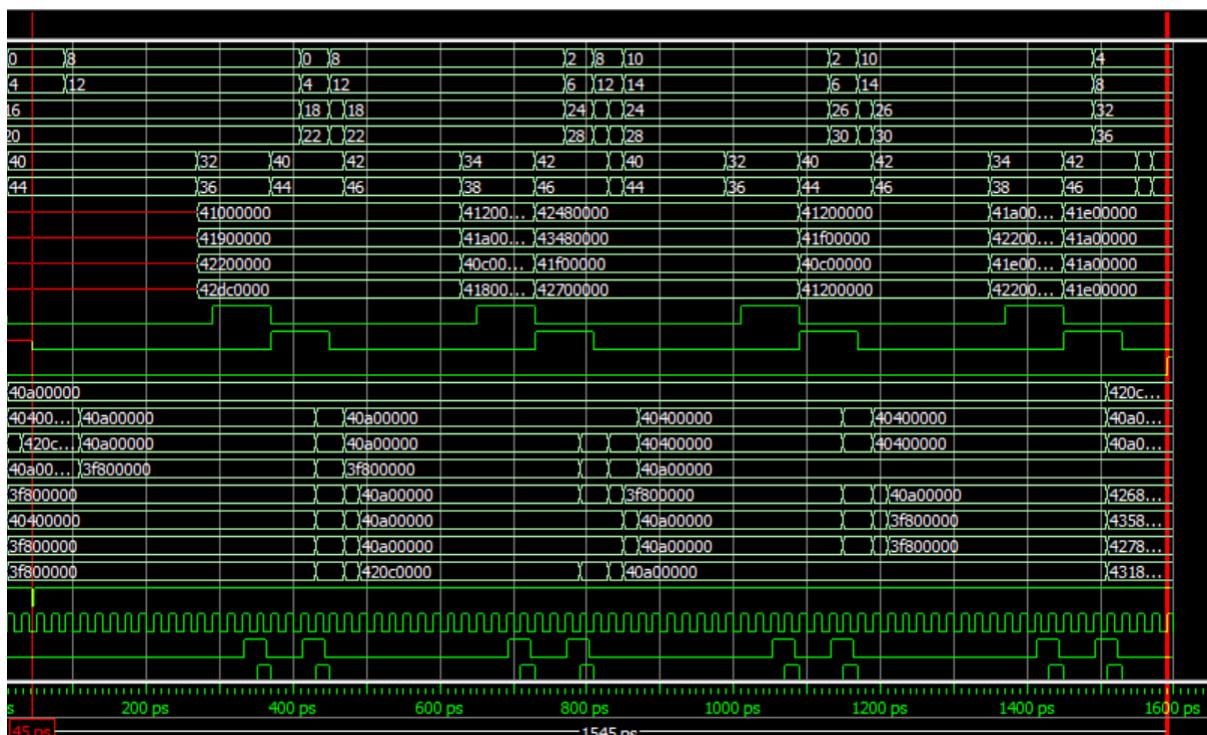
1. برخی از سیگنال ها در شبیه سازی دقیقا مشابه هم می باشند. این مسئله به دلیل ورودی و خروجی های متعدد برای مازول های مختلف است. در بررسی testbench سعی کردیم سیگنال های متفاوت آورده شود تا از بروز تکرار در گزارش پرهیز کرده باشیم.
2. سیگنال های بالا به ترتیب موجود در عکس فوق بررسی شده اند.

## ۳.۲: توصیف روند شبیه سازی سخت افزار

یکی از نکات مهم برای شبیه سازی این است که بعد از نوشتن هر ماژول، حتماً تست بنچ مربوط به آن نوشته شود تا عملکرد آن مستقل از دیگر ماژول ها و بخش های سخت افزار سنجیده شود.

حال کاری که برای شبیه سازی هر ماژول انجام شده به این صورت بود که یک ماژول به نام ModuleName\_TB ساخته شده که این ماژول در اصل در بالاترین سطح از لاحظ سلسله مراتبی قرار می‌گیرد. سپس در این ماژول با توجه به پروتکل ها و خروجی های ماژول مورد نظر، پروتکل های متفاوت به آنها داده شده و نتایج آن با نتایج مدل طلایی مقایسه شده تا از صحت آن ها اطمینان حاصل شود.

برای دیدن نتایج حاصله از برنامه ModelSim استفاده شده که در آن با استفاده از قابلیت Simulation، این تست بنچ مورد ارزیابی قرار می‌گرفت و خروجی ها به صورت waveform نمایش داده می‌شدند.



### • ۳.۳: پیاده سازی مدل طلایی و مقایسه نتایج

در بخش Golden Model پروژه را به کمک زبان java پیاده سازی کردیم که بررسی منطق و کد آن در ادامه خواهد آمد.

**معماری کلی کد:**

کد پروژه متشکل از یک کلاس اصلی `GoldenModel` و سایر قسمت ها داخل آن طراحی شده است.

```
5 //DSD Project Golden Model
6 ► public class GoldenModel {
7
8     // Matrix class for store the matrix values
9     private static class Matrix {...}
103
104     // FileReader class for read matrix elements from file
105     private static class FileReader {...}
128
129     // Controller class for running the program
130     private static class Controller {...}
163
164 ►     public static void main(String[] args) {...}
168 }
```

**کلاس ها:**

کلاس Matrix برای ذخیره ماتریس ها و کار با آنها طراحی شده است.  
کلاس FileReader برای خواندن ماتریس ها از روی فایل استفاده می شود.  
کلاس Controller مسئولیت مدیریت اجرای برنامه را بر عهده دارد.  
در ادامه، روند اجرای کد را پیگیری کرد و در طی آن با بخش های داخلی کد آشنا می شویم.

۲۵ ما اجرای خود را از نقطه زیر آغاز می کند:

```
164 ► ┌ public static void main(String[] args) {  
165   └─ Controller controller = new Controller();  
166   controller.run();  
167 ┌ }  
168 }
```

یک نمونه از کلاس Controller ساخته شده و در ادامه تابع run که در همین کلاس نوشته شده، صدای ۵۰ می شود.

```
129 // Controller class for running the program  
130 ┌ private static class Controller {  
131  
132   // Run the program  
133   ┌ public void run() {  
134     // Get user inputs  
135     Scanner scanner = new Scanner(System.in);  
136     System.out.println("Please enter first matrix row number:");  
137     int rowA = scanner.nextInt();  
138     System.out.println("Please enter first matrix column number:");  
139     int columnA = scanner.nextInt();  
140     System.out.println("Please enter second matrix row number:");  
141     int rowB = scanner.nextInt();  
142     System.out.println("Please enter second matrix column number:");  
143     int columnB = scanner.nextInt();  
144  
145     // Validation the multiply  
146     if (rowB != columnA) {  
147       System.err.println("This multiplication is not valid.\n" +  
148       "Because first matrix's column number and second matrix's row number must be the same.");  
149     }  
150   }
```

در این تابع ابتدا ابعاد ماتریس های مورخ نظر گرفته می شود. در ادامه شرط ضرب پذیر بودن این دو ماتریس چک شده و اگر این عملیات ضرب قابل انجام نبود؛ به کاربر پیام خطای داده و از برنامه خارج می شویم.

```

152     // Create matrix and fill them
153     Matrix matrixA = new Matrix(rowA, columnA);
154     Matrix matrixB = new Matrix(rowB, columnB);
155     matrixA.fillMatrixValue( matrixFilePath: "src/matrixA.txt");
156     matrixB.fillMatrixValue( matrixFilePath: "src/matrixB.txt");
157
158     // Multiply tow matrices
159     Matrix answer = matrixA.multiplyMatrix(matrixB);
160     answer.printMatrix();
161 }
162 }
```

سپس در این تابع، نمونه های (instance) این دو ماتریس ساخته شده به کمک multiplyMatrix موجود در تابع، مقدار دهنده می شوند. در نهایت تابع که در کلاس Matrix موجود است؛ صدای زده شده و جواب نهایی نمایش داده می شود.

اکنون به سراغ بررسی کلاس FileReader می رویم.

```

104     // FileReader class for read matrix elements from file
105     private static class FileReader {
106
107         // public method to read matrix from file
108         public void readMatrix(String filePath, float[][] values) {
109             try {
110                 File file = new File(filePath);
111                 Scanner scanner = new Scanner(file);
112                 int count = 0;
113                 while (scanner.hasNextLine()) {
114                     String data = scanner.nextLine();
115                     String[] row = data.split( regex: "[ ]" );
116                     for (int i = 0; i < row.length; i++) {
117                         values[count][i] = Float.parseFloat(row[i]);
118                     }
119                     count++;
120                 }
121                 scanner.close();
122             } catch (FileNotFoundException e) {
123                 System.out.println("An Error Occurred!");
124                 e.printStackTrace();
125             }
126         }
127     }
```

تنها تابع موجود در `readMatrix` که در پروگرام، آدرس حافظه ای که ماتریس در آن ذخیره شده است را گرفته و مقادیر ذخیره شده در آن آدرس را در یک آرایه ذخیره می کند. این آرایه نیز در پروگرام به عنوان پارامتر داده می شود.

فرض شده است که ماتریس به شکل csv یا

حافظه ذخیره شده است.

حال به بررسی آخرین کلاس، یعنی کلاس Matrix خواهیم پرداخت.

```

8   // Matrix class for store the matrix values
9   private static class Matrix {
10
11     // Fields
12     private final float row;
13     private final float column;
14     private final float[][] values;
15
16     // Constructor
17     public Matrix(float row, float column) {
18       this.row = row;
19       this.column = column;
20       this.values = new float[(int) (row % 2 == 0 ? row : row + 1)][(int) (column % 2 == 0 ? column : column + 1)];
21     }
22
23     // Fill matrix value
24     public void fillMatrixValue(String matrixFilePath) {
25       FileReader fileReader = new FileReader();
26       fileReader.readMatrix(matrixFilePath, values);
27     }

```

در این کلاس سه عدد field داریم.

- فیلد `row` برای مشخص کردن تعداد سطرها
- فیلد `column` برای مشخص کردن تعداد ستون ها
- فیلد `values` برای ذخیره مقادیر موجود در خانه های ماتریس

طبق الگوریتم مان که پیش از این توضیح داده شد؛ لازم است اکثر تعداد سطر و یا ستون فرد بود، به کمک درایه های 0 تعداد را اصلاح کنیم تا هم سطر و هم ستون زوج شوند. این مطلب در constructors مشخص شده است. (هنگام `new`

کردن فیلد `values`، زوج و فرد بودن بررسی می شود.)

تابع `fillMatrixValue` نیز که پیش از این صدای ۰۰ شده بود، برای پر کردن خانه های ماتریس به کمک مقادیر موجود در حافظه استفاده می شود. همان طور که مشخص است ارتباط با حافظه نیز به کمک `FileReader` صورت پذیرفته است.

```

28     private void addMatrix(Matrix otherMatrix) {
29         for (int i = 0; i < row; i++) {
30             for (int j = 0; j < column; j++) {
31                 values[i][j] += otherMatrix.values[i][j];
32             }
33         }
34     }

```

تابع فوق در کلاس `Matrix`، برای عملیات جمع ماتریسی استفاده می شود. این تابع در الگوریتم ضرب ماتریس به کمک ما خواهد آمد. اکنون به بررسی تابع ضرب ماتریسی خواهیم پرداخت.

```

36     // Divide & Conquer algorithm for multiply two matrices
37     public Matrix multiplyMatrix(Matrix otherMatrix) {
38
39         // Multiplied matrix
40         Matrix multiplied = new Matrix(this.row, otherMatrix.column);
41         for (int i = 0; i < Math.ceil(this.row / 2); i++) {
42             for (int j = 0; j < Math.ceil(otherMatrix.column / 2); j++) {
43                 Matrix tempA = new Matrix( row: 2, column: 2);
44                 Matrix tempB = new Matrix( row: 2, column: 2);
45                 Matrix tempMultiplied = new Matrix( row: 2, column: 2);
46
47                 // loop for multiply small blocks
48                 for (int k = 0; k < Math.ceil(this.column / 2); k++) {
49                     // Set tempA values
50                     tempA.values[0][0] = this.values[2 * i][2 * k];
51                     tempA.values[0][1] = this.values[2 * i][2 * k + 1];
52                     tempA.values[1][0] = this.values[2 * i + 1][2 * k];
53                     tempA.values[1][1] = this.values[2 * i + 1][2 * k + 1];
54                     // Set tempB values
55                     tempB.values[0][0] = otherMatrix.values[2 * k][2 * j];
56                     tempB.values[0][1] = otherMatrix.values[2 * k][2 * j + 1];
57                     tempB.values[1][0] = otherMatrix.values[2 * k + 1][2 * j];
58                     tempB.values[1][1] = otherMatrix.values[2 * k + 1][2 * j + 1];
59
60                     // Set tempMultiplied values
61                     tempMultiplied.addMatrix(multiplySmallBlock(tempA, tempB));
62                 }

```

همان طور که در توضیح الگوریتم بیان شد، ماتریس به بلکه های دو در دو تقسیم شده و به کمک الگوریتم divide and conquer حل می شود. تکه که بالع همین مفهوم را در الگوریتم ما نشان می دهد، بلکه های انتخاب شده در دو ماتریس با نام های tempB و tempA مشخص شده اند و حاصل ضرب آنها در یک ماتریس به نام tempMultiplied گشته است.

نکته اول: تابع addMatrix در خط انتهای تصویر بالا، در بخش قبلی تشریح شده است.

نکته دوم: تابع multiplySmallBlock نیز برای ضرب ماتریسی بلکه های دو در دو استفاده می شود. هر چند الگوریتم آن واضح است اما برای کامل بودن گزارش، پس از اتفاق بزرگی این تابع، که آن نیز بزرگی خواهد شد.  
حال به ادامه method اصلی مربوط به ضرب ماتریسی پردازیم.

```

64
65
66
67
68
69
70
71
72
73
74
75
76
    // Set multiplied matrix values
    for (int rowCount = 0; rowCount < 2; rowCount++) {
        for (int columnCount = 0; columnCount < 2; columnCount++) {
            multiplied.values[2 * i][2 * j] = tempMultiplied.values[0][0];
            multiplied.values[2 * i][2 * j + 1] = tempMultiplied.values[0][1];
            multiplied.values[2 * i + 1][2 * j] = tempMultiplied.values[1][0];
            multiplied.values[2 * i + 1][2 * j + 1] = tempMultiplied.values[1][1];
        }
    }
    return multiplied;
}

```

در انتهای این method، بلکه حاصل از ضرب ماتریس های کوچک، در جای صحیح خود در ماتریس نهایی جواب قرار خواهد گرفت. این عملیات در تکه که بالع مشخص شده است.

```

78     // Method for multiply 2 * 2 matrix
79     @
80
81         // Matrix named out is the answer of multiply
82         Matrix out = new Matrix( row: 2, column: 2);
83         for (int i = 0; i < 2; i++) {
84             for (int j = 0; j < 2; j++) {
85                 out.values[i][j] =
86                     firstBlock.values[i][0] * secondBlock.values[0][j]
87                     + firstBlock.values[i][1] * secondBlock.values[1][j];
88             }
89         }
90         return out;
91     }

```

تابع ضرب بلند های و چرخه ماتریسی در تصویر تکه کد بالا آورده شده است.

```

91     public void printMatrix() {
92         for (int i = 0; i < this.row; i++) {
93             for (int j = 0; j < this.column; j++) {
94                 System.out.printf("%10.2f | ", values[i][j]);
95             }
96             System.out.println();
97         }
98     }
99 }

```

برای نمایش ماتریس، (برای ماتریس نهایی حاصل ضرب) از تابع موجود در تکه کد بال استفاده می شود.

بخش import های این پروژه نیز به شکل زیر است.

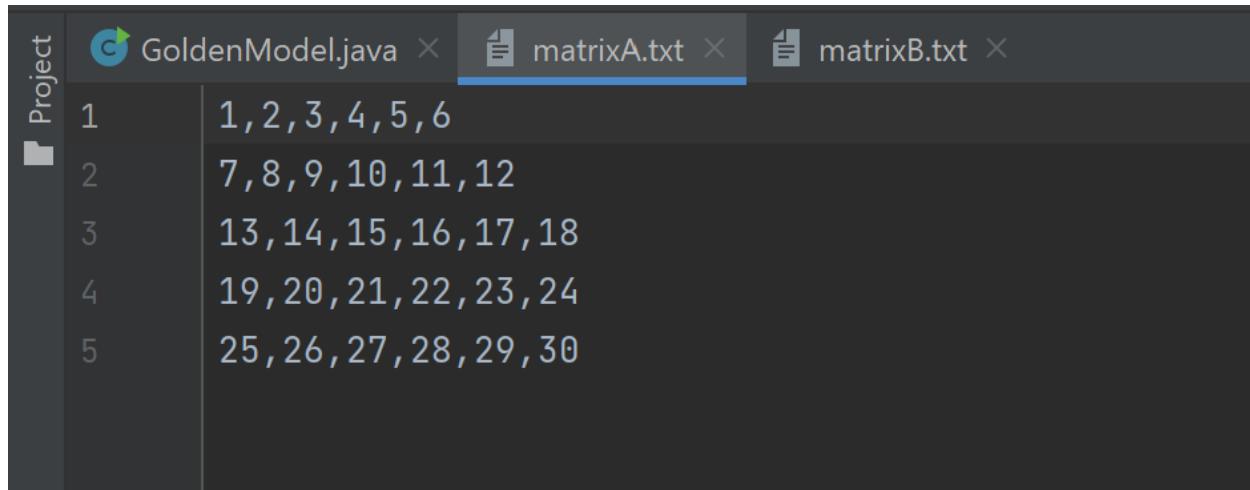
```

1 import java.io.File;
2 import java.io.FileNotFoundException;
3 import java.util.Scanner;

```

تمام بخش های کد در قسمت های بالا شرح داده شد. ممکن است بررسی یک مثال از عملکرد این کد خواهیم پرداخت. صحت این عملیات ضرب نیز توسط یک ماشین حساب آنلاین تایید شده است.

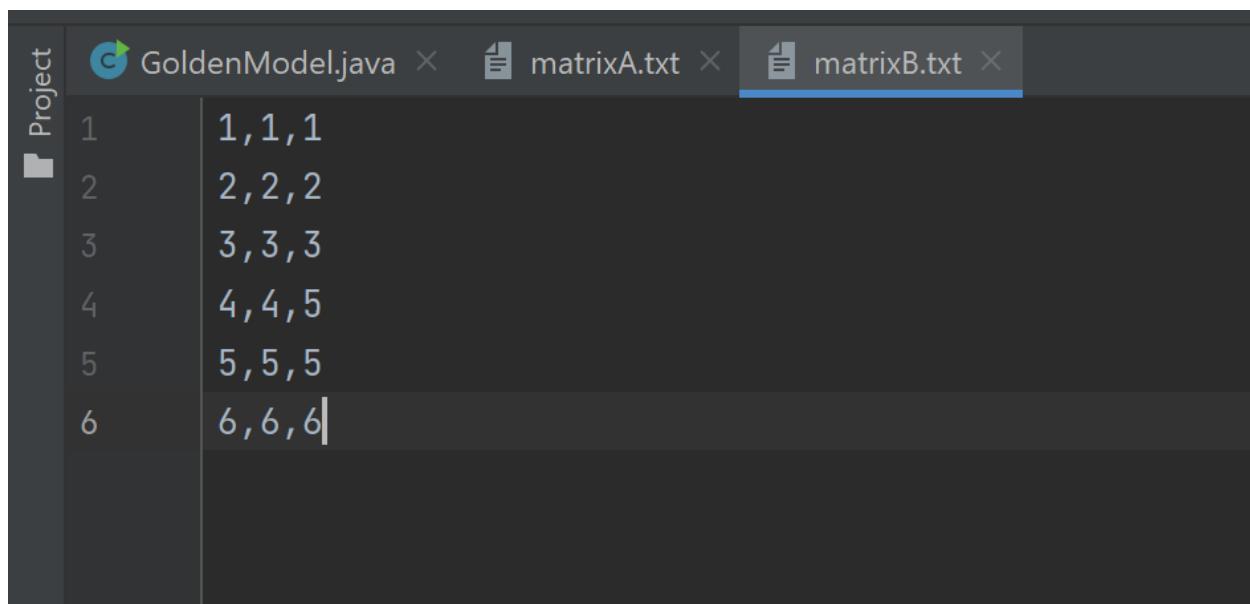
ماتریس اول:



The screenshot shows a Java IDE interface with three tabs at the top: 'GoldenModel.java', 'matrixA.txt', and 'matrixB.txt'. The 'matrixA.txt' tab is active, displaying the following matrix data:

	1,2,3,4,5,6	7,8,9,10,11,12	13,14,15,16,17,18	19,20,21,22,23,24	25,26,27,28,29,30
1					
2					
3					
4					
5					

ماتریس دوم:



The screenshot shows a Java IDE interface with three tabs at the top: 'GoldenModel.java', 'matrixA.txt', and 'matrixB.txt'. The 'matrixB.txt' tab is active, displaying the following matrix data:

	1,1,1	2,2,2	3,3,3	4,4,5	5,5,5	6,6,6
1						
2						
3						
4						
5						
6						

## نتیجه حاصل شده ای Golden Model

```
Run: Main ×
" C:\Program Files\Java\jdk-15.0.2\bin\java.exe" "-jar"
Please enter first matrix row number:
5
Please enter first matrix column number:
6
Please enter second matrix row number:
6
Please enter second matrix column number:
3
      91.00 |     91.00 |    95.00 |
      217.00 |    217.00 |   227.00 |
      343.00 |    343.00 |   359.00 |
      469.00 |    469.00 |   491.00 |
      595.00 |    595.00 |   623.00 |

Process finished with exit code 0
```

## محلت سنجی توسط یک ماشین حساب آنلاین:

- Gauss-Jordan Elimination
- Cramer's Rule
- Inverse Matrix Method
- Matrix Rank
- Determinant
- Inverse Matrix
- Matrix Power
- Matrix Transpose
- Matrix Multiplication**
- Matrix Addition/Subtraction

**Matrix A input**

Insert matrix    Restore matrix

	A <sub>1</sub>	A <sub>2</sub>	A <sub>3</sub>	A <sub>4</sub>	A <sub>5</sub>	A <sub>6</sub>
1	1	2	3	4	5	6
2	7	8	9	10	11	12
3	13	14	15	16	17	18
4	19	20	21	22	23	24
5	25	26	27	28	29	30

Clear    Fill empty cells with zero

**Matrix B input**

Insert matrix    Restore matrix

Complex numbers (more)

Fractional

	B <sub>1</sub>	B <sub>2</sub>	B <sub>3</sub>
1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	5
5	5	5	5
6	6	6	6

Clear    Fill empty cells with zero

**Calculate**

**About the method**

1. The main condition of matrix multiplication is that the number of columns of the 1st matrix must equal to the number of rows of the 2nd one.
2. As a result of multiplication you will get a new matrix that has the same quantity of rows as the 1st one has and the same quantity of columns as the 2nd one.
3. For example if you multiply a matrix of 'n' x 'k' by 'k' x 'm' size you'll get a new one of 'n' x 'm' dimension.

To understand matrix multiplication better input any example and examine the solution.

- Gauss-Jordan Elimination
- Cramer's Rule
- Inverse Matrix Method
- Matrix Rank
- Determinant
- Inverse Matrix
- Matrix Power
- Matrix Transpose
- Matrix Multiplication
- Matrix Addition/Subtraction

### Result of matrix multiplication

Show solution    Continue calculation

Result:

	C <sub>1</sub>	C <sub>2</sub>	C <sub>3</sub>
1	91	91	95
2	217	217	227
3	343	343	359
4	469	469	491
5	595	595	623

Computation time: 0.037 sec.

▲ Up

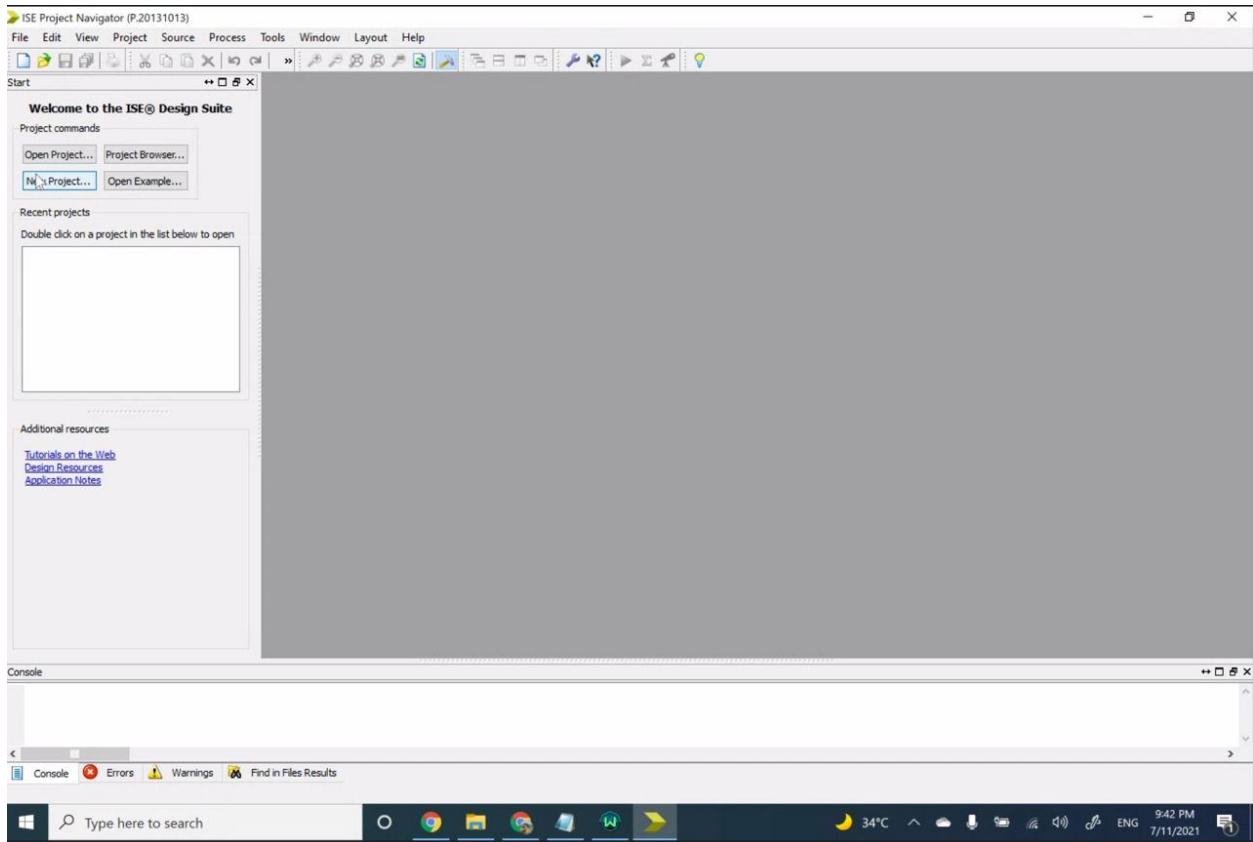
© reshish.com 2011 - 2021    Mobile version    Privacy Policy

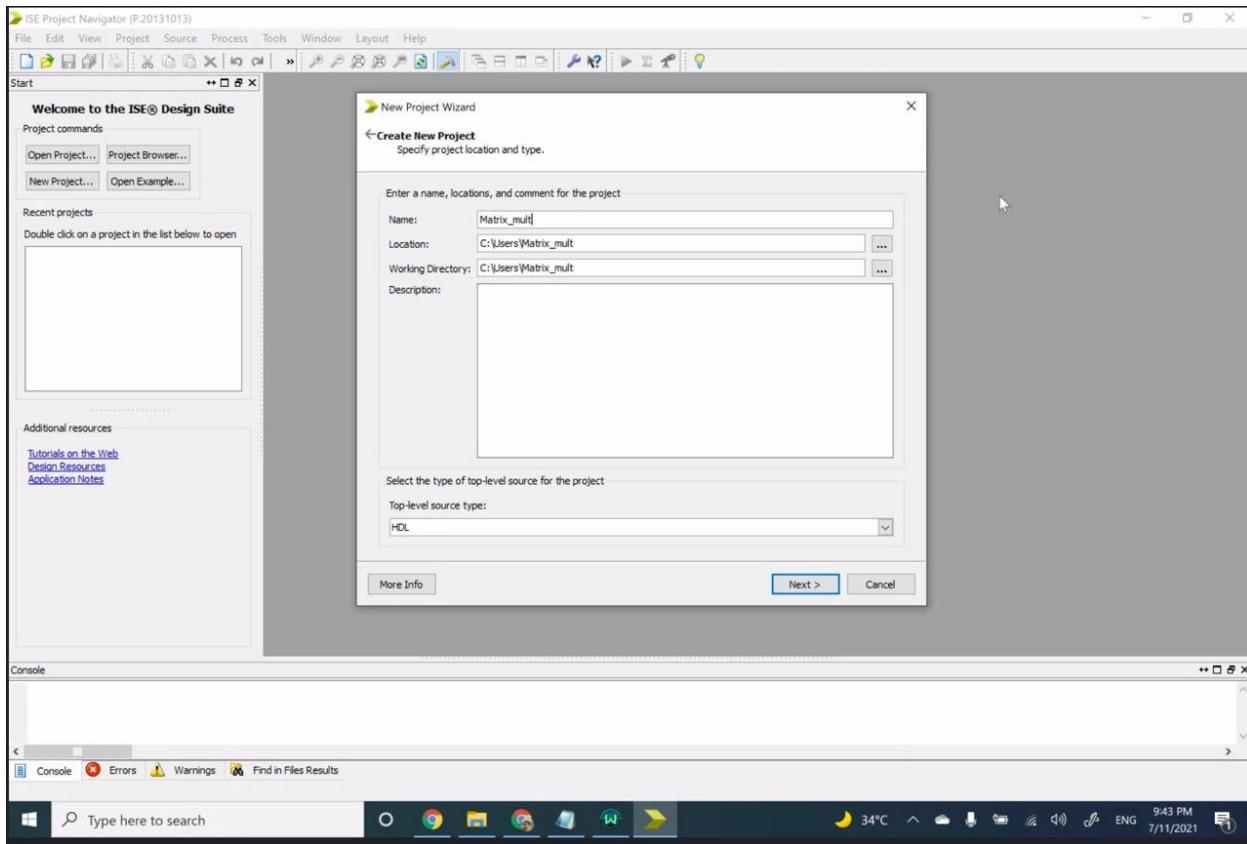
## بخش F: پیاده سازی و نتایج حاصله (implementation)

### 4.1 • CAD TOOL با FPGA

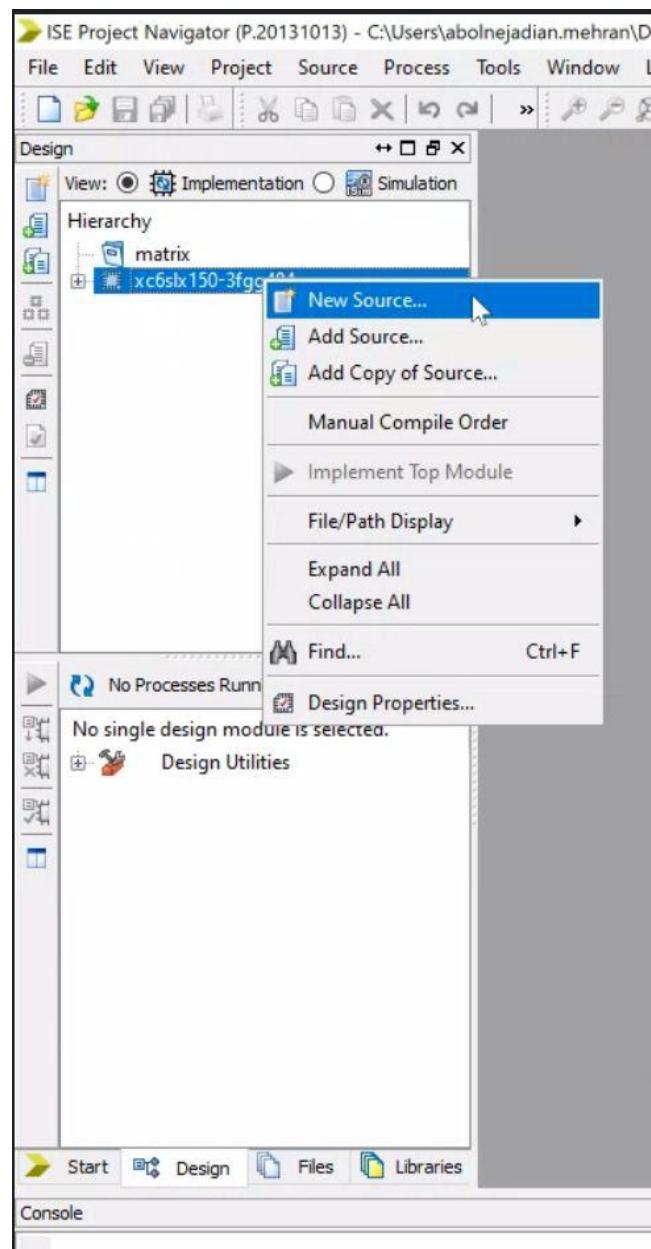
در این قسمت با استفاده از ابزار ISE Xilinx سنتر را انجام داده ایم. به این ترتیب که یک پروژه ایجاد کرده ایم و سپس با قرار دادن مأذول اصلی در top level کار سنتر را آغاز نموده ایم.

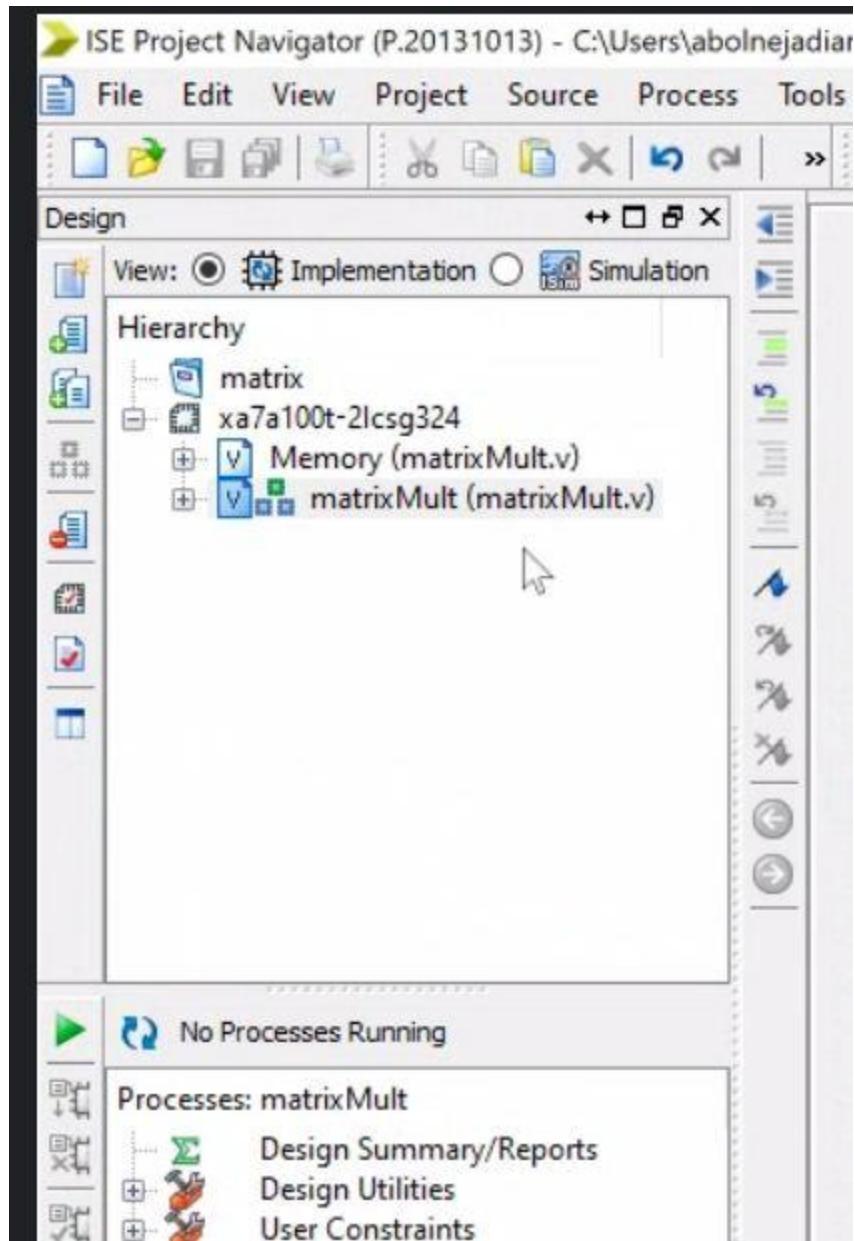
طبق ویدیویی درس می توان با زیر و انتخاب خانواده FPGA نوع آن به صورت تصویر زیر پروژه را سنتر کرد.



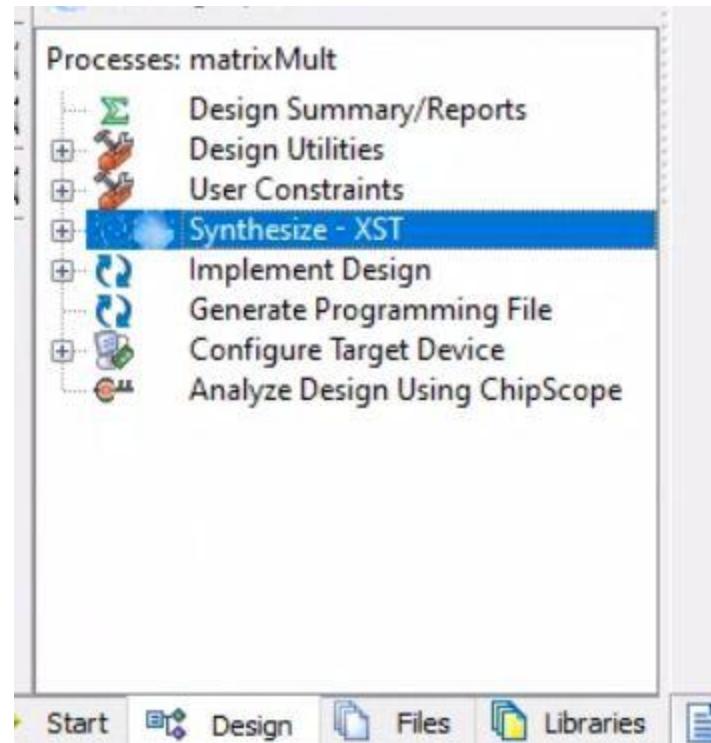


در مرحله ي بعد فايل هاي وريلدگ را به عنوان source file اضافه مي  
کنيم تا بتوان آنها را سنتز کرد.





به این ترتیب top module را انتخاب می کنیم.  
در نهایت کافیست سنتز را اجام دهیم و خروجی را مشاهده نماییم.



حال اینکه در خروجی سنتز می توان موارد بسیار مختلفی را دید که از جمله این موارد می توان به چیزهایی که در ادامه به آنها می پردازیم، اشاره نمود.

در توضیح این خروجی می توان گفت که در واقع باید به ساختار FPGA توجه نمود. از تعدادی Register به عنوان حافظه، تعدادی LUT به عنوان Look Up Table و I/O ها برای پین های ورودی/خروجی تشکیل شده است. به این ترتیب در گزارش خروجی سنتز می توان دید که در این FPGA چقدر از این منابع تا چه حد استفاده شده است. بنابراین موارد زیر را به کمک این گزارش می توان در حاصل سنتز مشاهده نمود.

## ۴.۲: گزارش پیاده سازی

### ۴.۲.۱: مساحت:

The screenshot shows two tables from the Xilinx ISE software interface.

**matrixMult Project Status (07/11/2021 - 23:02:22)**

<b>Project File:</b>	matrix.xise	<b>Parser Errors:</b>	No Errors
<b>Module Name:</b>	matrixMult	<b>Implementation State:</b>	Placed and Routed
<b>Target Device:</b>	xa7a100t-2icsg324	<b>• Errors:</b>	No Errors
<b>Product Version:</b>	ISE 14.7	<b>• Warnings:</b>	12590 Warnings (12531 new)
<b>Design Goal:</b>	Balanced	<b>• Routing Results:</b>	All Signals Completely Routed
<b>Design Strategy:</b>	Xilinx Default (unlocked)	<b>• Timing Constraints:</b>	All Constraints Met
<b>Environment:</b>	System Settings	<b>• Final Timing Score:</b>	0 ( <a href="#">Timing Report</a> )

**Device Utilization Summary**

Slice Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slice Registers	4	126,800	1%	
Number used as Flip Flops	4			
Number used as Latches	0			
Number used as Latch-thrus	0			
Number used as AND/OR logics	0			
Number of Slice LUTs	7	63,400	1%	
Number used as logic	7	63,400	1%	
Number using O6 output only	7			
Number using O5 output only	0			
Number using O5 and O6	0			
Number used as ROM	0			
Number used as Memory	0	19,000	0%	
Number used exclusively as route-thrus	0			
Number of occupied Slices	6	15,850	1%	
Number of LUT Flip Flop pairs used	8			
Number with an unused Flip Flop	4	8	50%	

### ۴.۲.۲: تعداد فلیپ فلادپ ها:

طبق تصویر زیر تعداد رجیستر ها ۴ تا و تعداد فلیپ فلادپ ها هم ۴ تا گزارش شده است.

```

Mapping all equations...
Building and optimizing final netlist ...
Found area constraint ratio of 100 (+ 5) on block matrixMult, actual ratio is 0.

Final Macro Processing ...

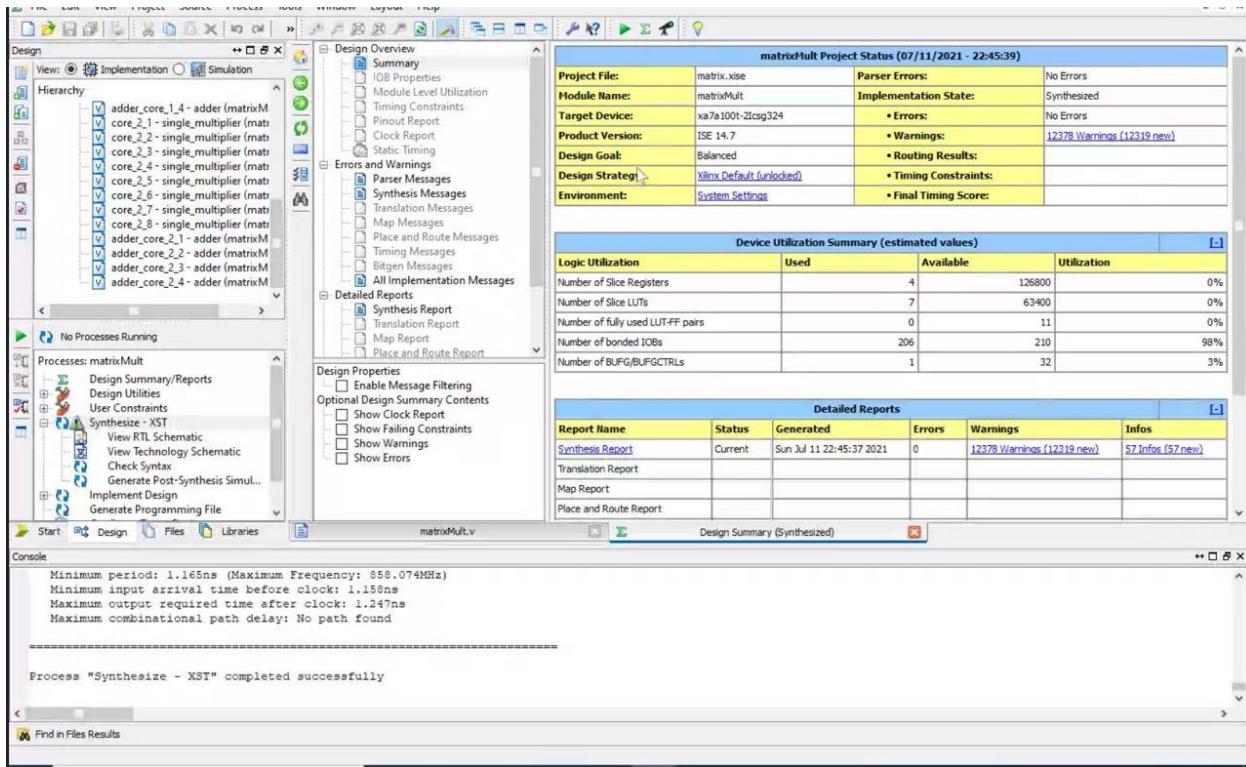
=====
Final Register Report

Macro Statistics
# Registers : 4
# Flip-Flops : 4

=====
* Partition Report *

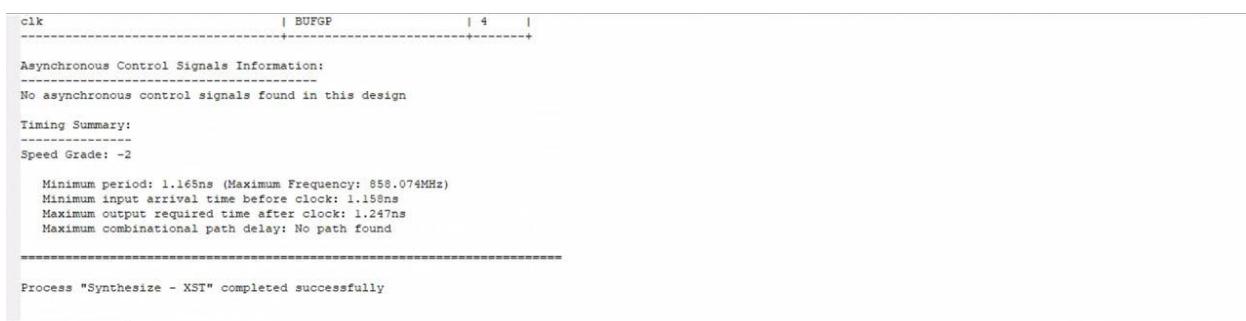
```

## ۱۰.۲.۳: LUT: م.م.ا



طبق این تصویر تعداد ۷ واحد کزارش شده است.

## ۱۰.۴: حداقل فرکانس قابل حصول:



طبق تصویر بالا حداقل فرکانس قابل حصول 858.07 مگاهرتز می باشد. که فرکانس بسیار خوبی است و نزدیک به یک گیگاهرتز است. از طرفی critical path در تصویر بالا آورده شده است.

در این قطعه بین state transition و تعداد قطعه input/output و state می‌توان تشخیص داده شده در طراحی را می‌بینیم.

```
Found finite state machine <FSM_1> for signal <core_1_state>.
```

States	8
Transitions	27
Inputs	19
Outputs	20
Clock	clk (rising_edge)
Reset	rstN (negative)
Reset type	asynchronous
Reset State	000
Power Up State	000
Encoding	auto
Implementation	LUT

و کلک سیستم هم در تصویر زیر تشخیص داده شده است و تعداد جاهایی که به کلک وصل هستند به عنوان load آورده شده است.

```
=====
*          Design Summary
=====
Clock Information:
-----
Clock Signal           | Clock buffer(FF name) | Load |
-----+-----+-----+
clk      | BUFGP            | 4   |
-----+-----+-----+
Asynchronous Control Signals Information:
No asynchronous control signals found in this design
```

همچنین مراحل سنتز هم در تصاویر زیر آورده شده اند.

```
=====
*          HDL Parsing
=====
Analyzing Verilog file "C:\Users\abolnejadian.mehran\Desktop\DSP\matrix\matrixMult.v" into library work
Parsing module <matrixMult>.
Parsing module <Memory>.
INFO:HDLCompiler:693 - "C:\Users\abolnejadian.mehran\Desktop\DSP\matrix\matrixMult.v" Line 1016. parameter declaration becomes local in Memory with formal parameter d
INFO:HDLCompiler:693 - "C:\Users\abolnejadian.mehran\Desktop\DSP\matrix\matrixMult.v" Line 1017. parameter declaration becomes local in Memory with formal parameter d
INFO:HDLCompiler:693 - "C:\Users\abolnejadian.mehran\Desktop\DSP\matrix\matrixMult.v" Line 1018. parameter declaration becomes local in Memory with formal parameter d
INFO:HDLCompiler:693 - "C:\Users\abolnejadian.mehran\Desktop\DSP\matrix\matrixMult.v" Line 1019. parameter declaration becomes local in Memory with formal parameter d
INFO:HDLCompiler:693 - "C:\Users\abolnejadian.mehran\Desktop\DSP\matrix\matrixMult.v" Line 1020. parameter declaration becomes local in Memory with formal parameter d
Parsing module <adder>.
Parsing module <single_multiplier>.
```

```

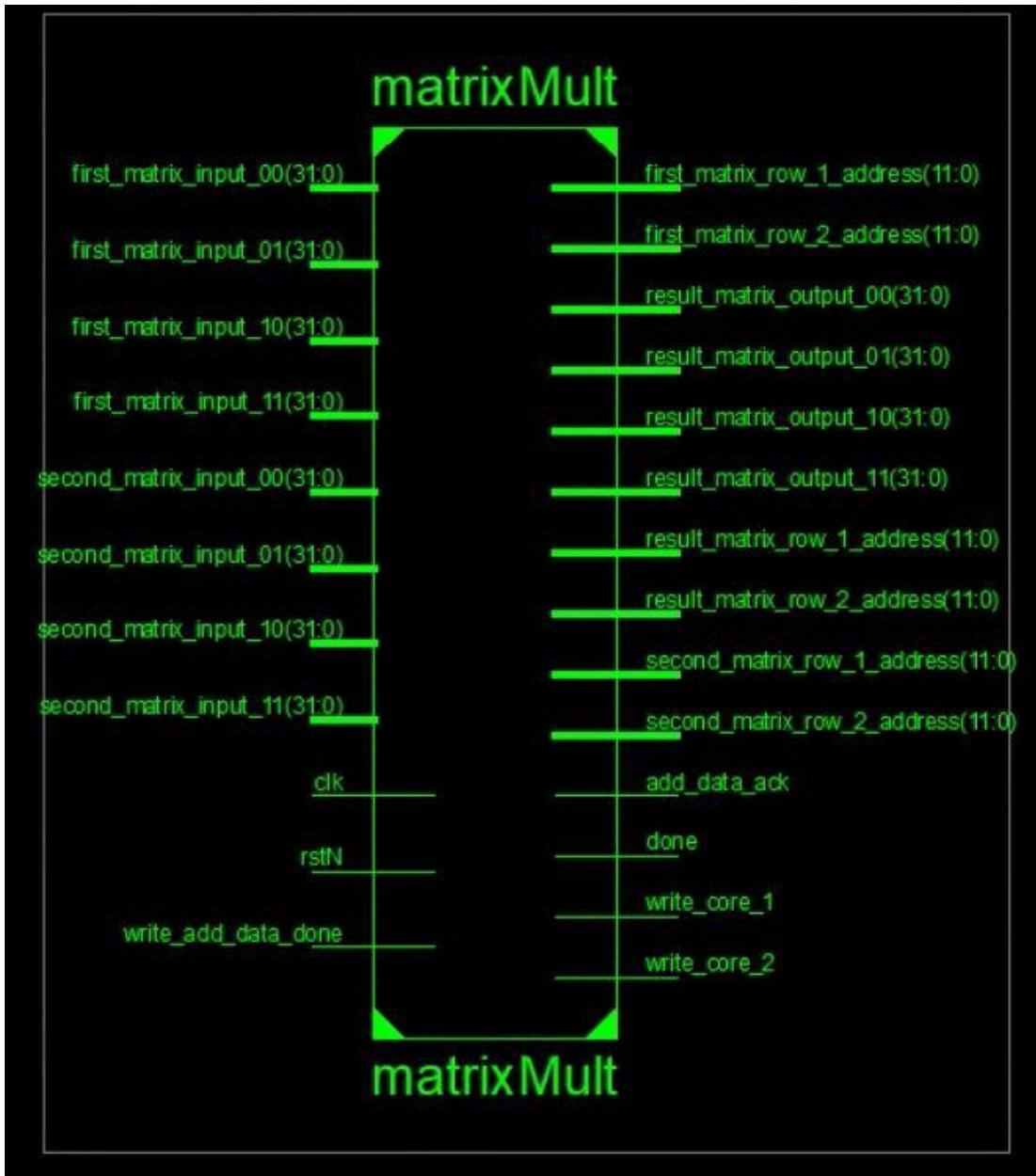
*-----*
          HDL Elaboration
*-----*

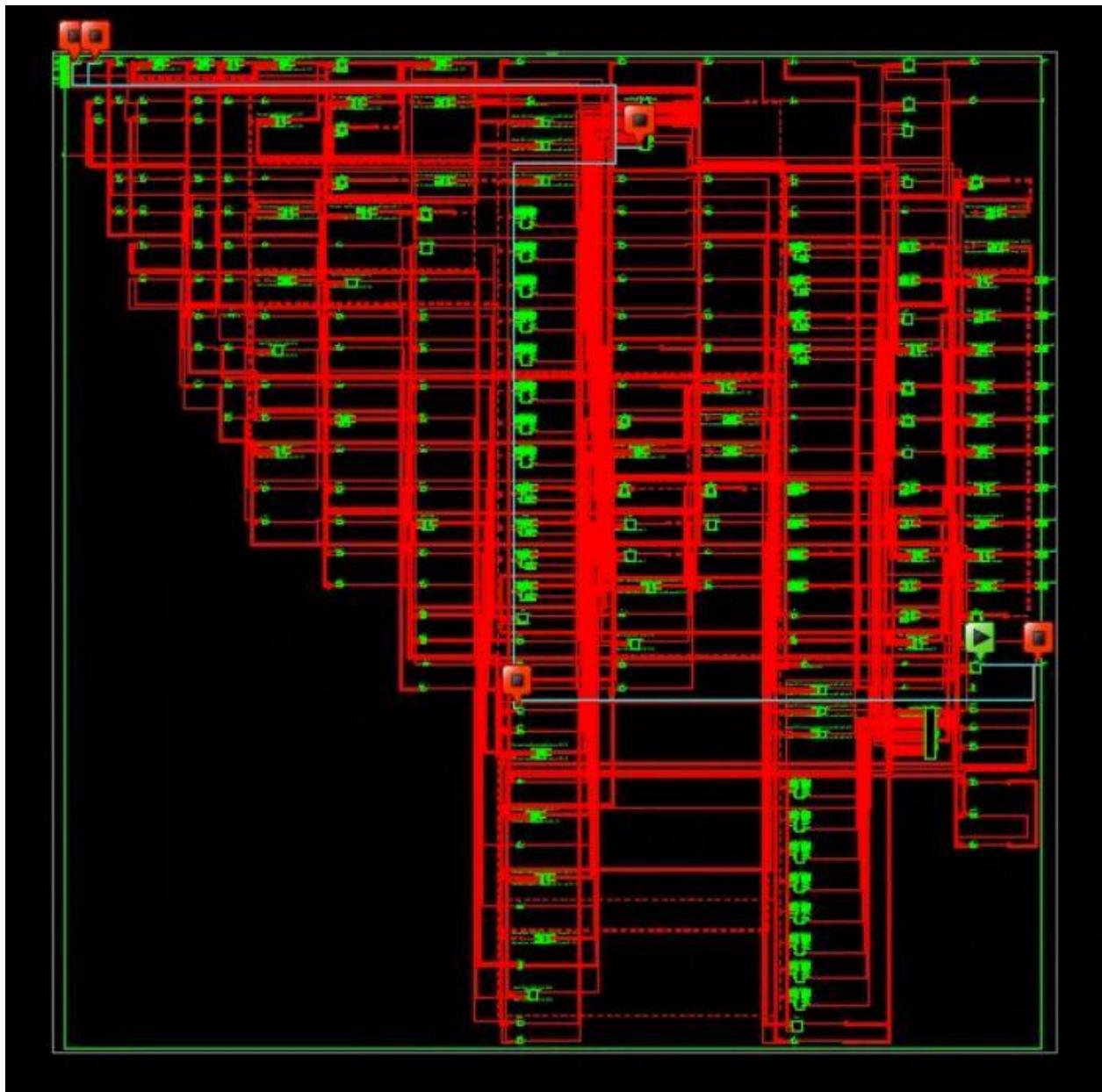
Elaborating module <matrixMult>.
# WARNING:HDLCompiler:413 - "C:\Users\abolinejadian.mehran\Desktop\DSO\matrix\matrixMult.v" Line 191: Result of 13-bit expression is truncated to fit in 12-bit target.
# WARNING:HDLCompiler:413 - "C:\Users\abolinejadian.mehran\Desktop\DSO\matrix\matrixMult.v" Line 199: Result of 13-bit expression is truncated to fit in 12-bit target.
# WARNING:HDLCompiler:413 - "C:\Users\abolinejadian.mehran\Desktop\DSO\matrix\matrixMult.v" Line 206: Result of 13-bit expression is truncated to fit in 12-bit target.
# WARNING:HDLCompiler:413 - "C:\Users\abolinejadian.mehran\Desktop\DSO\matrix\matrixMult.v" Line 214: Result of 13-bit expression is truncated to fit in 12-bit target.

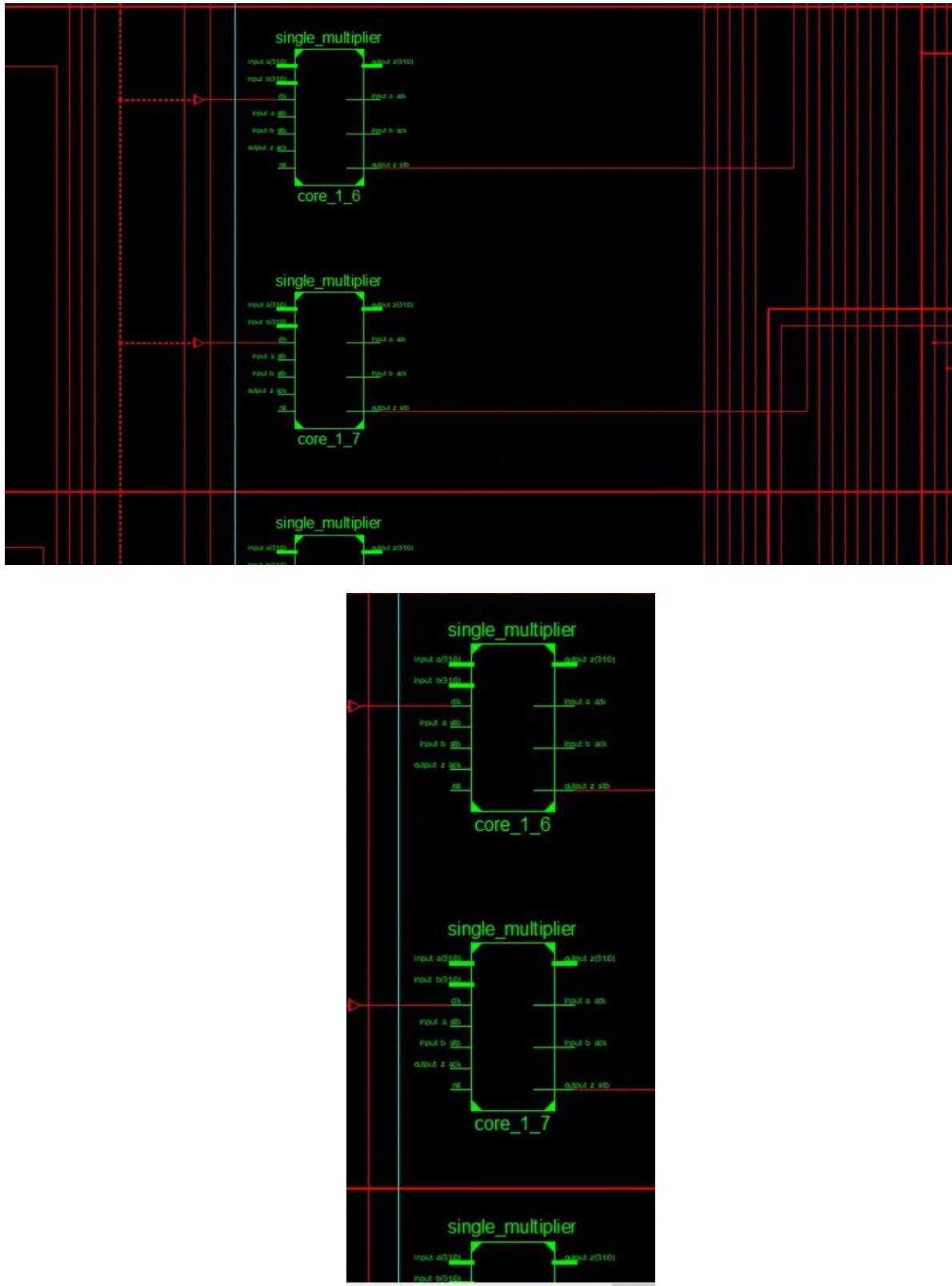
Elaborating module <single_multiplier>.
# WARNING:HDLCompiler:413 - "C:\Users\abolinejadian.mehran\Desktop\DSO\matrix\matrixMult.v" Line 1482: Result of 32-bit expression is truncated to fit in 10-bit target.
# WARNING:HDLCompiler:413 - "C:\Users\abolinejadian.mehran\Desktop\DSO\matrix\matrixMult.v" Line 1483: Result of 32-bit expression is truncated to fit in 10-bit target.
# WARNING:HDLCompiler:413 - "C:\Users\abolinejadian.mehran\Desktop\DSO\matrix\matrixMult.v" Line 1559: Result of 32-bit expression is truncated to fit in 10-bit target.
# WARNING:HDLCompiler:413 - "C:\Users\abolinejadian.mehran\Desktop\DSO\matrix\matrixMult.v" Line 1569: Result of 32-bit expression is truncated to fit in 10-bit target.
# WARNING:HDLCompiler:413 - "C:\Users\abolinejadian.mehran\Desktop\DSO\matrix\matrixMult.v" Line 1576: Result of 12-bit expression is truncated to fit in 10-bit target.
# WARNING:HDLCompiler:413 - "C:\Users\abolinejadian.mehran\Desktop\DSO\matrix\matrixMult.v" Line 1593: Result of 32-bit expression is truncated to fit in 10-bit target.
# WARNING:HDLCompiler:413 - "C:\Users\abolinejadian.mehran\Desktop\DSO\matrix\matrixMult.v" Line 1606: Result of 11-bit expression is truncated to fit in 10-bit target.
# WARNING:HDLCompiler:413 - "C:\Users\abolinejadian.mehran\Desktop\DSO\matrix\matrixMult.v" Line 1619: Result of 25-bit expression is truncated to fit in 24-bit target.
# WARNING:HDLCompiler:413 - "C:\Users\abolinejadian.mehran\Desktop\DSO\matrix\matrixMult.v" Line 1621: Result of 11-bit expression is truncated to fit in 10-bit target.

```

چراکه تصاویری را از شماتیک این RTL می بینیم.







## بخش ۵: نتیجه گیری

در آنرا به نتیجه گیری می پردازیم.  
بعدین ترتیب طی مرادل قبل طراحی این مدار ضرب کننده ماتریسی انجام  
شد. حال در نهایت بهتر است به این پردازیم که کاربرد این ضرب کننده در عمل  
برای چه است؟

می توان در زمینه های مختلفی کاربرد ضرب ماتریسی را بررسی کرد که در  
ادامه هر یک از آنها را بررسی می کنیم. در هر یک از این موارد از ماتریس استفاده  
های خاصی می شود و به این ترتیب ضرب ماتریسی هم به صورت متفاوت معنا  
می دهد و مفهوم حاصل ضرب به این ترتیب متفاوت است.

### ۵.۱: کاربرد های سیستم طراحی شده

۵.۱.۱: کاربرد در فیزیک و مهندسی ○

#### (Engineering Applications)

در فیزیک از ماتریس ها برای مطالعه مدارهای الکتریکی، مکانیک  
کوانتومی و اپتیک استفاده می شود.

مفهوم ماتریس در محاسبه خروجی قدرت باتری و تبدیل مقاومت انرژی  
الکتریکی به انرژی مفید دیگر نیز نقش مهمی ایفا می کند و به خصوص برای  
حل مسائل در استفاده از قوانین کیوشف ولتاژ و جریان ضروری می باشد.

فیزیکدانان و مهندسان سیستم های فیزیکی را مدل می کنند و  
محاسبات دقیق مورد نیاز برای کار کردن ماشین های پیچیده را انجام می دهند.

شبکه ها، پایه های فضایی و ساخت مواد شیمیایی همگی به محاسبات دقیق نیاز دارند که در گروی تبدیلات ماتریس می باشد.

## Computer Science and کامپیووتر(Engineer Application)

در برنامه های کاربردی مبتنی بر کامپیووتر ماتریس ها نقش مهمی در انتقال تصاویر سه بعدی به یک صفحه نمایش دو بعدی و نمایش آن ها و خلق درکات به ظاهر واقعی دارند.

تصاویر دیجیتالی مجموعه ای از پیکسل ها یا عناصر تصویری هستند که در فضای دو بعدی مرتب شده اند و در کامپیووتر به صورت ماتریس نمایش داده می شوند، این ماتریس ها حاوی مقادیر متفاوتی از اعداد صحیح هستند که نشان دهنده درخشندگی، شدت یا ویژگی های مشابه هستند.

نرم افزارهای گرافیکی همچون فتوشاپ از ماتریس ها برای پردازش تبدیل های خطی در زدن کردن تصاویر استفاده می کنند. با کمک یک ماتریس مربعی می توان تبدیل خطی یک جسم هندسی را نشان داد.

$$\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$$

برای مثال در صفحه دکارتی  $Y-X$  ماتریس رو ب رو:

یک جسم را در محور عمودی  $Y$  منعکس می کند. به عنوان مثال در یک بازی ویدیویی این باعث می شود بازتاب وارونه تصویر یک قاتل در استخر خون بیفتد!

اگر بازی ویدیویی دارای سطوح بازتاب کننده منحنی شکل مانند سپرهای فلزی براق باشد ماتریس مورد استفاده پیچیده تر خواهد بود زیرا باید تصویر بازتاب شده را کشیده یا کوچک کند.

### ○ ۳.۱.۵: کاربرد در پزشکی (Medical Applications)

از ماتریس ها در عملکرد تجهیزات مختلف پزشکی از جمله کت اسکن و ام آر آی استفاده می شود.

### ○ ۴.۱.۵: کاربرد در اقتصاد (Economy Applications)

از جمله کاربردهای ماتریس ها در اقتصاد می توان به محاسبه تولید ناخالص داخلی اشاره کرد که در نهایت به محاسبه کارآمد تولید کالاها کمک خواهد کرد.

در اقتصاد از ماتریس های بسیار بزرگ برای بهینه سازی مسائل استفاده می شود، مثل استفاده بهینه از دارایی ها - خواه نیروی کار یا سرمایه - و تولید یک محصول و مدیریت زنجیره های عرضه بسیار بزرگ همکنی به ماتریس ها نیاز دارد.

### ○ ۵.۱.۵: کاربرد در فناوری اطلاعات (IT Applications)

ماتریس و ماتریس معکوس توسط برنامه نویسان برای کد نویسی یا رمزگذاری پیام استفاده می شود.

یک پیام از دنباله ای اعداد در قالب باینری برای برقراری ارتباط ساخته شده و برای کدگشایی آن از تئوری کد استفاده می شود که برای حل معادلات آن به ماتریس ها نیاز است.

در دنیای امنیت اطلاعات بسیاری از سیستم ها برای کار با ماتریس ها طراحی شده اند، ماتریس ها در فشرده سازی اطلاعات الکترونیکی کاربرد داشته و نقش مهمی در ذخیره سازی اطلاعات اثر انگشت و داده های بیومتریک در برخی کارت های شناسایی جدید دارند.

در بسیاری از شرکت های فناوری اطلاعات از ماتریس به عنوان ساختار داده ای برای ردیابی اطلاعات کاربران، جستجوی پرس و جوها و مدیریت پایگاه داده استفاده می شود.

ماتریس پاد متقارن و بردار ویژه نیز در الگوریتم های رتبه بندی صفحات در جستجوکر گوکل به کار می روند.

### ○ ۵.۱.۶: کاربرد در رباتیک (Robotics Applications)

در علم رباتیک حرکت ربات ها با محاسبه ردیف ها و ستون های ماتریس برنامه ریزی می شود و ورودی های کنترل ربات براساس محاسبات حاصل از ماتریس ها می باشد.

#### ○ ۵.۱.۷: کاربرد در جامعه شناسی (Sociology Applications)

از ماتریس می توان برای نشان دادن داده های دنیای واقعی درباره یک جمعیت خاص - مثلاً تعداد افرادی که یک ویژگی خاص را دارا می باشند - یا میزان مرگ و میر نوزادان و غیره استفاده کرد.

ماتریس ها برای مدل سازی پیش بینی ها در رشد جمعیت نیز کاربرد دارند.

#### ○ ۵.۱.۸: کاربرد در زمین شناسی (Geology Applications)

در زمین شناسی ماتریس ها در انجام تحقیقات لزوه ای نقش مهمی بر عهده دارند.

#### ○ ۵.۱.۹: کاربرد در رمزگاری (Encryption Applications)

در علم رمزگاری هم از ماتریس ها برای مبادث کدگذاری کردن (Encode) و از حالت کدگذاری شده خارج کردن (Decode) استفاده می کنند. در واقع یک کلید برای رمز گشایی کردن عبارات رمزگاری شده وجود دارد که این کلید از جنس ماتریس است و با ضرب ماتریس ها این فرآیند انجام می شود.

#### ○ ۵.۱.۱۰: کاربرد در بازی سبکی (3D Games Applications)

برای اینکه بتوان اشیا را در دنیای بازی ها دستکاری کرد، جا به جا کرد و یا اندازه ی آن ها را کوچک و بزرگ کرد، از ماتریس ها استفاده می شود که بدین ترتیب با استفاده از ضرب ماتریسی می توان اشیا را با استفاده از تبدیل های خطی بدین صورت جا به جا کرد.

بنابراین؛ بدین ترتیب می‌توان توجه کرد که ماتریس‌ها یکی از عناصر مهم در بسیاری از علوم پایه و مهندسی مطرح می‌شوند و به همین صورت بسیار زیاد مورد استفاده قرار می‌گیرند. بر همین اساس؛ ضرب ماتریسی به عنوان مهم ترین عمل تعریف شده روی ماتریس‌ها کاربرد زیادی در هر یک از این علوم دارد و به این ترتیب می‌توان از آن در این راستا‌ها استفاده نمود.

## • ۵.۲: مزیت پیاده سازی سیستم طراحی شده به صورت سخت افزاری

در نهایت کافیست اشاره کنیم که پیاده سازی ضرب کننده‌ی ماتریسی به صورت سخت افزاری و با استفاده از HDL‌ها چه مزیتی نسبت به استفاده از واحد های پردازشی مثل CPU برای ضرب ماتریسی و نوشتن برنامه‌های نرم افزاری دارد؟!

در واقع در جواب این سوال می‌توان گفت که درست است که ما همواره برای برنامه‌هایی که میخواهیم بسازیم می‌توانیم از دو دیدگاه نرم افزاری و سخت افزاری عمل کنیم؛ اما تفاوت در اینجاست که پیاده سازی به صورت نرم افزاری و اجرا روی واحد های پردازشی مانند CPU کارایی کمتری دارند. زیرا؛ واحد های پردازشی برای کاربرد های گوناگونی طراحی شده‌اند و به این ترتیب چون به صورت خاص برای این کار (به عنوان نمونه ضرب ماتریسی) تعریف نشده‌اند، به این ترتیب کارایی مطلوب را ندارند.

در حالی که وقتی این مدار را به صورت سخت افزاری پیاده سازی می‌کنیم، در واقع یک واحد سخت افزاری به صورت اختصاصی برای کار ضرب ماتریسی طراحی کرده‌ایم و به این ترتیب چون کار این واحد فقط ضرب ماتریسی است، بسیار سریعتر و با کارایی بالاتری می‌تواند این عمل را انجام دهد.

به همین دلیل هم از این واحد ها همانند واحد ضرب ماتریسی ساخته شده در این پروژه می‌توان برای در «**شتابدهنده های سخت افزاری**» (Hardware Accelerators) برای انجام عملیات ضرب ماتریسی استفاده نمود. زیرا کارایی بسیار بالاتری نسبت به پیاده سازی نرم افزاری این موضوع دارد و به این ترتیب می‌تواند در راستای بالا بردن سرعت بسیار موثر عمل کند.

## • ۵.۳: آموخته های این پژوهه

- آموختیم که چکونه باید به صورت تیمی کار کرد و تقسیم کار و مسئولیت پذیری تیمی چکونه است.
- یاد گرفتیم که چکونه می توان برای دل یک مسئله‌ی واقعی همفکری کرد و با خود جمعی به بهترین راهکار برای دل مسئله رسید.
- یاد گرفتیم که الگوریتم‌های متفاوت ضرب ماتریسی چکونه اند و هرکدام چه مزایا و معایبی دارند.
- آموختیم که در پیاده سازی‌های سخت افزاری همه‌ی کار‌ها فرم «مصالحه» دارد و باید با استفاده از در نظر گیری جوانب مختلف هر راهکار بنا بر مسئله و کاربردی که از آن انتظار داریم باید بتوان بهترین راهکار را به صورت مصالحه کردن بین مزایا و معایب انتخاب نمود.
- در مورد کار کردن با زبان توصیف سخت افزاری Verilog بیشتر یاد گرفتیم و تجربه‌های جدیدی از موارد سنتز پذیری و مثال‌های آن یاد گرفتیم.
- آموختیم که چکونه باید برای هر عملی که انجام می‌دهیم، در کنار آن مستندسازی انجام داد و هر فرآیندی را در قالب و جایگاه مناسب در مستند به صورت مکتوب برای بقیه‌ی اعضای تیم و دیگران به جا گذاشت.
- و از همه مهم‌تر تجربه‌ی کار گروهی و مهارت ورزی تیمی را آموختیم! امید است توانسته باشیم به عنوان یک تیم بهترین عملکرد را در ده توان از خود به جا گذاشته باشیم!

Asadi, Dr. Hossein. *Floating Point Representation slides.*

“Floating Point.” *From Wikipedia,*

[https://en.wikipedia.org/wiki/Floating-point\\_arithmetic.](https://en.wikipedia.org/wiki/Floating-point_arithmetic)

Gargave, Soumya, et al. “Single-Precision Floating Point Matrix Multiplier Using Low-Power Arithmetic Circuits.”

IEEE Computer Society. “IEEE754.”

“Explained: Matrices Concepts familiar from grade-school algebra have broad ramifications in computer science.” *MIT news,*

[https://news.mit.edu/2013/explained-matrices-1206.](https://news.mit.edu/2013/explained-matrices-1206)

“Wallace Tree.” [https://en.wikipedia.org/wiki/Wallace\\_tree](https://en.wikipedia.org/wiki/Wallace_tree).

“Applications of Matrices and Matrix Multiplication”

<https://www.embibe.com/exams/where-are-matrices-used-in-daily-life/>

“Applications of Matrices”

<http://www.coca.ir/%DA%A9%D8%A7%D8%B1%D8%A8%D8%B1%D8%AF-%D9%85%D8%A7%D8%AA%D8%B1%DB%8C%D8%B3/>

“Matrix Multiplication Algorithms”

[https://en.wikipedia.org/wiki/Matrix\\_multiplication\\_algorithm](https://en.wikipedia.org/wiki/Matrix_multiplication_algorithm)