

به نام خدا



فاز سوم پروژه

طراحی سیستم های دیجیتال

استاد فلاحتی

اعضای تیم:

سهیل نظری مندجین

علیرضا حقی

مهدی سعادت بخت

نیمسال دوم سال تحصیلی ۱۴۰۰-۱۴۰۱

دانشگاه صنعتی شریف

دانشکده مهندسی کامپیوتر

فهرست مطالب

3	توضیحات پروژه
3	چکیده
4	مقدمه
4	کاربردهای ضرب دو ماتریس
4	روش‌های ضرب دو عدد
4	تقسیم و غلبه
5	درخت والاس
6	ضرب‌کننده بران (آرایه‌ای)
7	FSM and ASM
9	الگوریتم ضرب ماتریس
10	مرحله اول و دوم (پارتیشن‌بندی ماتریس به ساب‌بلوک‌های $np * np$)
10	مرحله سوم (ضرب ساب‌بلوک‌ها در هم و سپس جمع آن‌ها با هم)
11	مرحله چهارم (شیفت دادن ماتریس a به راست و ماتریس b به پایین)
11	ضرب ماتریس
12	سنتز
12	RTL Schematic
12	Synthesis schematic

توضیحات پروژه

یک واحد پردازشی ضرب دو ماتریس را با استفاده از الگوریتم تقسیم و حل پیاده سازی کرده ایم. در این طراحی مولفه های ماتریس مطابق با استاندارد IEEE754 دریافت می شوند. این طراحی با استفاده از تقسیم و حل به گونه ای انجام شده که انجام عملیات ضرب ماتریس به صورت موازی و بهینه انجام شود.

چکیده

با توجه به محاسبات سنگین محاسباتی ماتریسی در حوزه های مختلف، پردازش موازی ضرب ماتریس ها در راستای کمینه کردن میزان انرژی و مساحت مورد استفاده اهمیت ویژه ای پیدا کرده است.

در این پروژه، بنابر دغدغه ی الزام ضرب موازی ماتریس ها، به طراحی یک مدل تقریباً بهینه با استفاده از زبان برنامه نویسی Verilog بر اساس مساحت و تعداد ماژول ها رسیده ایم. در مدل ما از $O(\frac{n^3}{p})$ واحدهای محاسباتی استفاده شده است که قادر است در $O(p)$ کلاک محاسبات موازی دو ماتریس را انجام دهد که در این جا p بیانگر مجذور تعداد واحدهای پردازش موازی است. کدهای ما کاملاً قابل سنتز است و این امر به وسیله نرم افزار سنتز Vivado صورت گرفته است. همچنین صحت کد و کارایی آن توسط آزمایش کننده ای توسط دو کد پایتون که یکی عدد رندوم تولید کرده و دیگری پاسخ آن را میابد صورت گرفته است. نکته ی حائز اهمیت دیگر مدل ما، پارامتریک بودن است که می توان به وسیله آن برای هر ابعاد دلخواه و هر تعداد دلخواه پردازنده آن را در نظر گرفت که در آن تعداد پردازنده مربع کامل است و ابعاد بر مجذور پردازنده ها بخش پذیر است.

مقدمه

کاربردهای ضرب دو ماتریس

ضرب دو ماتریس در علوم کامپیوتر کاربردهای گسترده‌ای دارد و از اساسی‌ترین دلایلی که سعی داریم تا این عمل را در پایین‌ترین سطح پردازشی، یعنی سطح ترانزیستور و چپ انجام دهیم و از انجام آن به صورت نرم‌افزاری خودداری کنیم، کاربردهای گسترده آن است. از گسترده‌ترین استفاده‌های 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 می‌پذیرد). هر خانه از این ماتریس را می‌توان حاوی اطلاعات یکی از پیکسل‌های این عکس در نظر گرفت که عدد آن‌ها نشان‌دهنده ترکیب رنگی آن پیکسل است. حال برای مثال، برای دیکود کردن یک ویدیو دیجیتالی، نیاز به ضرب ماتریس‌ها داریم. بسیار دیده‌ایم که برای مثال یک ویدیو را می‌توان از فرمت x254 به x255 دیکود کرد و این امر با توجه به بالا بودن نرخ فریم‌ها در یک ثانیه برای ویدیوهای جدید، اگر می‌خواست در سطح نرم‌افزار انجام می‌شد، زمان بسیار زیادی را از کاربر می‌گرفت. پژوهشگران MIT یکی از اولین چپ‌های مخصوص (ASIC) و بهینه برای کد کردن ویدیوها که در تلویزیون‌های با کیفیت بالا استفاده می‌شود را با استفاده از بهینه‌سازی ضرب ماتریس‌ها خلق کرده‌اند. گرافیک در کامپیوتر با استفاده از اصطلاح پردازش دیجیتال سیگنال بیان می‌شود. از دیگر کاربردهای ضرب ماتریس‌ها در این زمینه می‌توان به تصویربرداری دیجیتال، پردازش سیگنال‌ها و چندرسانه‌ای اشاره کرد. همچنین در کاربردهای جدیدتر برای reinforce کردن یک یادگیری در بینایی ماشینی برای شتابنده‌های سخت‌افزاری هوش مصنوعی، ضرب ماتریس در مقیاس بالا استفاده می‌شود.

روش‌های ضرب دو عدد

برای ضرب دو عدد، روش‌های متعددی وجود دارد و انتخاب هر کدام از آن‌ها، مسئله انتخاب بین سرعت و استفاده از سخت‌افزار کمتر است. به طوری که برخی از روش‌ها با وجود سرعت بالا در محاسبه، نیازمند تعداد بالایی منابع سخت‌افزاری هستند، اما برخی دیگر می‌توانند این کار را در زمانی طولانی‌تر اما با منابع سخت‌افزاری محدودتر انجام دهند. ۳ روش معروف برای این کار که به بررسی آن‌ها خواهیم پرداخت عبارتند از:

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

تقسیم و غلبه

این روش بیشتر در FPGAها به کار می‌رود که همین امر باعث سرعت بیشتر می‌شود. به دلیل موازی‌سازی‌های که در این روش انجام می‌شود، 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 کرده. بر اساس تقسیم‌بندی‌ای که کردیم، عبارت نهایی پس از ضرب باید به شکل زیر باشد:

$$Ah.Bh.10^n + AlBl + AhBl10^{\frac{n}{2}} + AlBh10^{\frac{n}{2}}$$

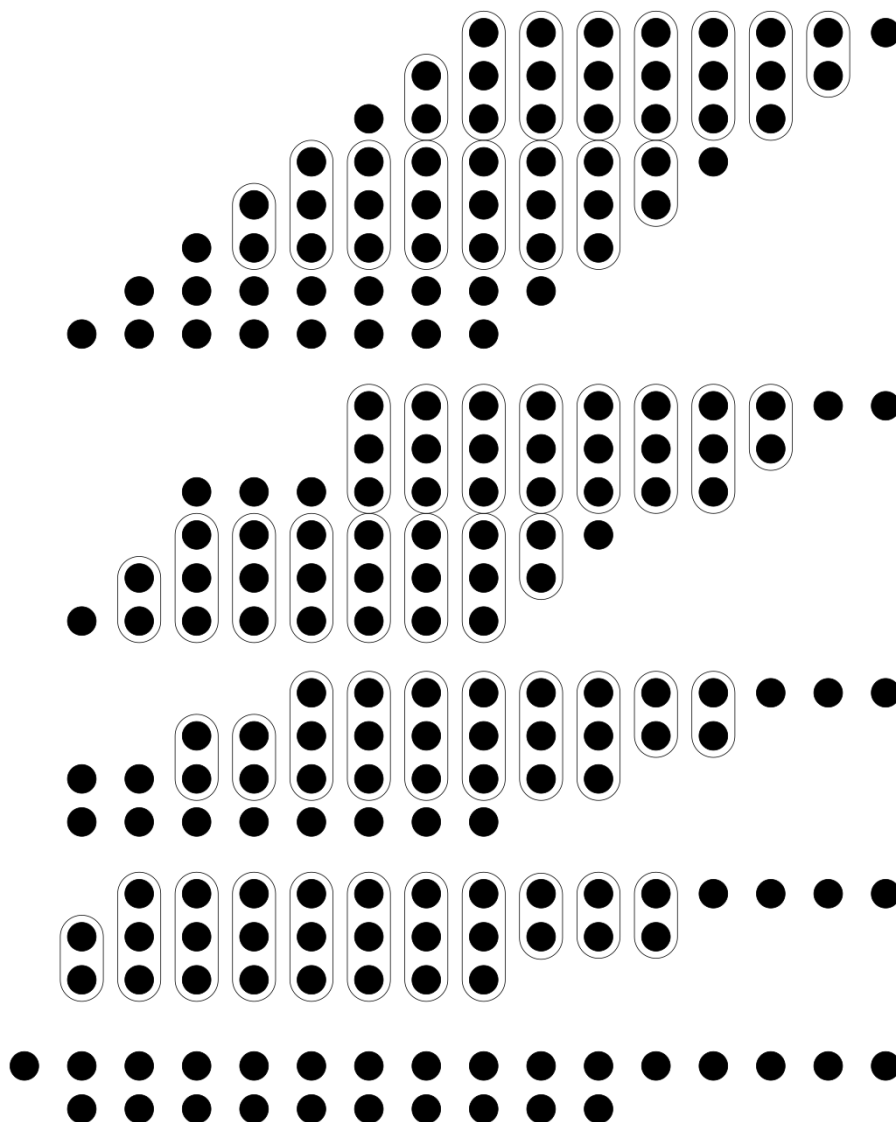
سپس ۳ عبارت زیر را محاسبه می‌کنیم:

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

بعد از محاسبه این ۳ عبارت، می‌توانیم 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 والاس

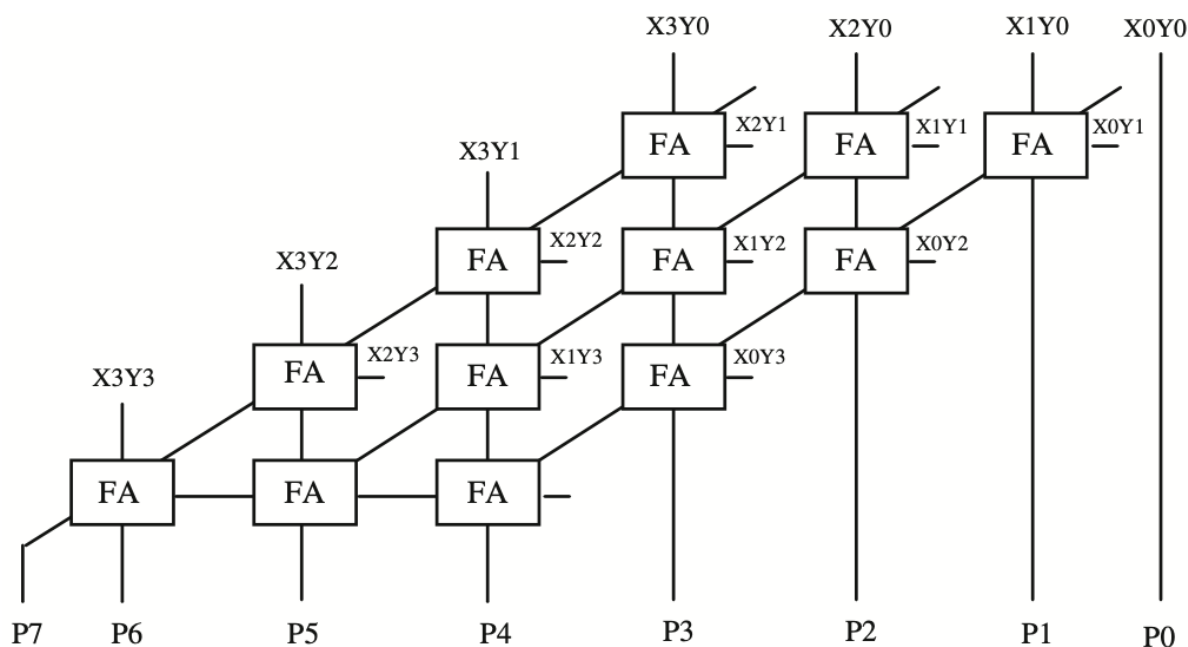
درخت والاس در هر سطح از ضرب، تعداد بیت‌های کمتری دارد ولی تاخیر به صورت موازی تقسیم نمی‌شود. با توجه به ارتباطات میانی بسیاری که درخت والاس دالاس دارد، مساحت این روش زیاد می‌شود. صورت کلی انجام این روش برمی‌گردد به جمع ریز-ضرب‌هایی که انجام می‌دهیم. شکل کلی آن در زیر قابل مشاهده است:



تصویر بالا مربوط به یک درخت والاس 8×8 است که ۴ لایه از آن کاهش یافته است. هر کدام از نقطه‌ها نمایانگر یک بیت با ارزش وزن یکسان هستند. این عمل کاهش با استفاده از ۱۴ adder-half و ۳۸ adder-full انجام شده که adder-half ها با پکیج کردن دو نقطه نشان داده شده و adder-full ها با پکیج کردن سه نقطه. همان‌طور که در ابتدای این بخش نیز بیان شد، کاهش زمانی انجام یک عملیات (که در اینجا با کاهش لایه‌ها قابل مشاهده است)، به قیمت افزایش استفاده از منابع سخت‌افزاری است.

ضرب‌کننده بران (آرایه‌ای)

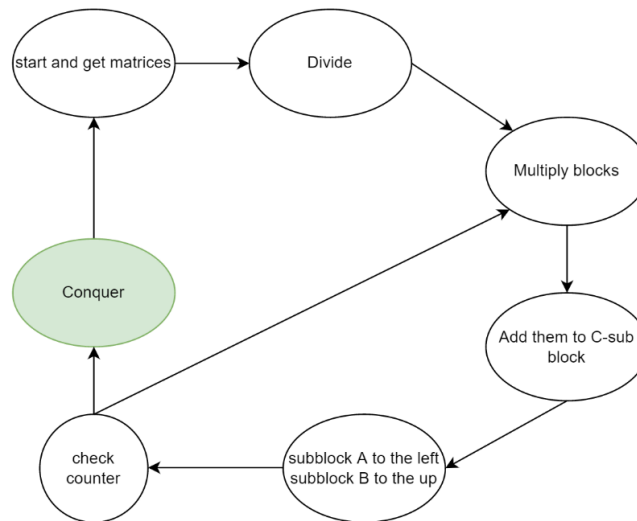
این روش برخلاف روش قبل، توزیع تاخیر یکسانی برای تمام سطوح دارد. این روش در اصل همان ضرب آرایه‌ای است که از دوران مدرسه آن را آموخته‌ایم. شکل آن به صورت زیر است:



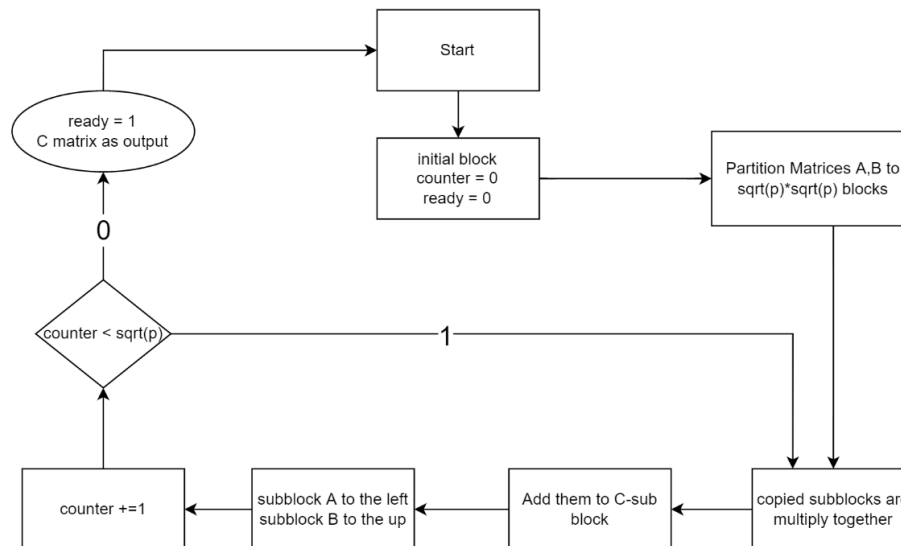
این روش از بدترین پیچیدگی زمانی با مرتبه زمانی $O(n^3)$ برخوردار است. اما مزیت استفاده از این روش، کاهش پیچیدگی پیاده‌سازی و استفاده از منابع سخت‌افزاری خواهد بود.

FSM and ASM

در این بخش ما شمایی کلی از حالت و وضعیت‌های محاسباتی ممکن برای ماژول ضرب‌کننده نشان می‌دهیم که از ۷ حالت تشکیل شده است:



ASM



با استفاده از طراحی FSM انجام شده یک ماژول کنترلر تعریف کرده‌ایم که به وسیله آن استیت‌های مختلف ماشین ما تغییر کرده و سه بیت کنترلی برای خواندن، جمع کردن و شیفت دادن در اختیار داریم که میان استیت‌ها مقادیرشان را تنظیم می‌کنیم. نکته دیگر این است که ماشین کنترلی مد نظر از نوع Moore است. بنابراین به ورودی برای حرکت بین حالات حساس نیست.

تکه کد ریست و استارت در استیت صفر:

```
always @(posedge clk, posedge reset)
begin
    if (state == 0)
    begin
        nx_state = 1;
        enable_read = 1;
    end
    if(reset == 1)
        state <= 0;
    else
        state <= nx_state;
    end
end
```

تکه کد انتقال میان استیت‌ها و ست کردن بیت‌های کنترلی:

```
always @(state)
begin
    if(!reset)
    begin
        if(enable)
        case(state)
            0:
                begin
                    nx_state = 1;
                    enable_read = 1;
                end
            1:
                begin
                    enable_read = 1;
                    nx_state = 2;
                end
            2:
                begin
                    enable_read = 0;
                    enable_sum = 1;
                    nx_state = 3;
                end
            3:
                begin
                    enable_sum = 0;
                    enable_shift = 1;
                    nx_state = 4;
                end
            4:
                begin
                    enable_shift = 0;
                    if (shifted < sqrt_p - 1)
                        nx_state = 2;
                    else
                        nx_state = 5;
                    shifted = shifted + 1;
                end
            5:
                begin
                    out_ready = 1;
                    nx_state = 6;
                end
            6:
                begin
                    out_ready = 0;
                    nx_state = 0;
                end
            default:
                begin
                    out_ready = 0;
                    enable_shift = 0;
                    enable_sum = 0;
                    enable_read = 0;
                    nx_state = 0;
                end
        endcase
    else
        nx_state = state;
    end
end
```


الگوریتم ضرب ماتریس

الگوریتم کلی بدین شکل است که هر دو ماتریس $n \times n$ را به p^2 ماتریس با ابعاد $\frac{n}{p} \times \frac{n}{p}$ تقسیم می‌کنیم. سپس در طی p مرحله با استفاده از شیفت و ضرب اجزای متناظر ماتریس‌های به‌دست آمده و در نهایت جمع آن‌ها جواب نهایی حاصل می‌شود. در عکس پایین نیز شبیه‌کدی برای الگوریتم ارائه شده است (p^2 را تعداد پردازنده‌ها است).

عملیات‌های ضرب دو ماتریس به شکل تقسیم و حل به \sqrt{p} مرحله تقسیم می‌شود که به شکل زیر است.



Implementation

➤ Consider two square matrices A and B of size n that have to be multiplied:

1. Partition these matrices in square blocks p , where p is the number of processes available.
2. Create a matrix of processes of size $p^{1/2} \times p^{1/2}$ so that each process can maintain a block of A matrix and a block of B matrix.
3. Each block is sent to each process, and the copied sub blocks are multiplied together and the results added to the partial results in the C sub-blocks.
4. The A sub-blocks are rolled one step to the left and the B sub-blocks are rolled one step upward.
5. Repeat steps 3 y 4 \sqrt{p} times.

ما در یک ماژول به اسم array-divider این مراحل را با استفاده از جنریتورها پیاده‌سازی کرده‌ایم. تکه‌کد مراحل مختلف به ترتیب آمده‌اند. در این بخش به‌واسطه جنریتورها توانستیم برنامه کاملاً کارا و پارامتریک که هیچ وابستگی به متغیر و عدد ثابتی ندارد را پیاده‌سازی کنیم که هر چند منجر به پیچیدگی‌های قابل ملاحظه‌ای در پیاده‌سازی کد شد.

مرحله اول و دوم (پارتیشن‌بندی ماتریس به ساب‌بلوک‌های سایز $\frac{n}{p} * \frac{n}{p}$)

```

genvar i, j, k, v;
generate
  for (i = 0; i < sqrt_p; i = i + 1)
    for (j = 0; j < sqrt_p; j = j + 1)
      for (k = 0; k < n_divide_ps; k = k + 1)
        for (v = 0; v < n_divide_ps; v = v + 1)
          always @(*)
            begin
              if(enable_read)
                begin
                  tmp_A[i][j][(k * n_divide_ps + v) * 32 + 31: (k * n_divide_ps + v)*32] = matrix_A[((i * n_divide_ps + k) + n * (j * n_divide_ps + v)) * 32 + 31: ((i * n_divide_ps + k) + n * (j * n_divide_ps + v)) * 32];
                  tmp_B[i][j][(k * n_divide_ps + v) * 32 + 31: (k * n_divide_ps + v)*32] = matrix_B[((i * n_divide_ps + k) + n * (j * n_divide_ps + v)) * 32 + 31: ((i * n_divide_ps + k) + n * (j * n_divide_ps + v)) * 32];
                end
              end
            end
        end
      end
    end
  endgenerate

generate
  for (i = 0; i < sqrt_p; i = i + 1)
    for (j = 0; j < sqrt_p; j = j + 1)
      for (k = 0; k < n_divide_ps; k = k + 1)
        for (v = 0; v < n_divide_ps; v = v + 1)
          assign multiple_result[((i * n_divide_ps + k) * n + (j * n_divide_ps + v)) * 32 + 31: ((i * n_divide_ps + k) * n + (j * n_divide_ps + v)) * 32] = out_sum[i][j][(k * n_divide_ps + v)*32 + 31: (k * n_divide_ps + v)*32];
        end
      for (i = 0; i < sqrt_p; i = i + 1)
        for (j = 0; j < sqrt_p; j = j + 1)
          for (k = 0; k < n_divide_ps; k = k + 1)
            for (v = 0; v < n_divide_ps; v = v + 1)
              always @(reset)
                if(reset)
                  out_sum[i][j][(k * n_divide_ps + v)*32 + 31: (k * n_divide_ps + v)*32] = 0;
            end
          end
        end
      endgenerate

```

مرحله سوم (ضرب ساب‌بلوک‌ها در هم و سپس جمع آن‌ها با هم)

```

//summing procedure
generate
  for (i = 0; i < sqrt_p; i = i + 1)
    for (j = 0; j < sqrt_p; j = j + 1)
      for (k = 0; k < n_divide_ps; k = k + 1)
        for (v = 0; v < n_divide_ps; v = v + 1)
          adder_new_sum_temp[out_sum[i][j][(k * n_divide_ps + v)*32 + 31: (k * n_divide_ps + v)*32], out_sum_temp[i][j][(k * n_divide_ps + v)*32 + 31: (k * n_divide_ps + v)*32], out_sum_new_temp[i][j][(k * n_divide_ps + v)*32 + 31: (k * n_divide_ps + v)*32]];
        end
      for (i = 0; i < sqrt_p; i = i + 1)
        for (j = 0; j < sqrt_p; j = j + 1)
          for (k = 0; k < n_divide_ps; k = k + 1)
            for (v = 0; v < n_divide_ps; v = v + 1)
              always @(posedge enable_sum)
                if (enable_sum)
                  begin
                    out_sum[i][j][(k * n_divide_ps + v)*32 + 31: (k * n_divide_ps + v)*32] = out_sum[i][j][(k * n_divide_ps + v)*32 + 31: (k * n_divide_ps + v)*32] + out_sum_temp[i][j][(k * n_divide_ps + v)*32 + 31: (k * n_divide_ps + v)*32];
                  end
                end
            end
          end
        end
      endgenerate

//multipling procedure
generate
  for (i = 0; i < sqrt_p; i = i + 1)
    for (j = 0; j < sqrt_p; j = j + 1)
      begin
        mul_matrix #(n_divide_ps) i_j_AB(tmp_A[i][j], tmp_B[i][j], out_sum_temp[i][j]);
      end
    end
  endgenerate

genvar ii, jj, kk, ll;

```

مرحله چهارم (شيفت دادن ماتريس a به راست و ماتريس b به پايين)

```
generate
    //shift tmp_A to right
    //changes don't effect at all
    for (ii = 0; ii < sqrt_p; ii = ii + 1)
        for (jj = 0; jj < sqrt_p; jj = jj + 1)
            for (kk = 0; kk < n_divide_ps; kk = kk + 1)
                for (ll = 0; ll < n_divide_ps; ll = ll + 1)
                    if (ii == sqrt_p - 1)
                        always @(posedge clk)
                            if (enable_shift)
                                tmp_A[ii][jj][(kk * n_divide_ps + 11) * 32 + 31 : (kk * n_divide_ps + 11) * 32] <= tmp_A[ii][jj][(kk * n_divide_ps + 11) * 32 + 31 : (kk * n_divide_ps + 11) * 32];

    for (ii = 0; ii < sqrt_p; ii = ii + 1)
        for (jj = 0; jj < sqrt_p; jj = jj + 1)
            for (kk = 0; kk < n_divide_ps; kk = kk + 1)
                for (ll = 0; ll < n_divide_ps; ll = ll + 1)
                    if (ii != sqrt_p - 1)
                        always @(posedge clk)
                            if (enable_shift)
                                tmp_A[ii + 1][jj][(kk * n_divide_ps + 11) * 32 + 31 : (kk * n_divide_ps + 11) * 32] <= tmp_A[ii][jj][(kk * n_divide_ps + 11) * 32 + 31 : (kk * n_divide_ps + 11) * 32];

    //shift B to down
    for (ii = 0; ii < sqrt_p; ii = ii + 1)
        for (jj = 0; jj < sqrt_p; jj = jj + 1)
            for (kk = 0; kk < n_divide_ps; kk = kk + 1)
                for (ll = 0; ll < n_divide_ps; ll = ll + 1)
                    if (jj == sqrt_p - 1)
                        always @(posedge clk)
                            if (enable_shift)
                                tmp_B[ii][0][(kk * n_divide_ps + 11) * 32 + 31 : (kk * n_divide_ps + 11) * 32] = tmp_B[ii][jj][(kk * n_divide_ps + 11) * 32 + 31 : (kk * n_divide_ps + 11) * 32];

    for (ii = 0; ii < sqrt_p; ii = ii + 1)
        for (jj = 0; jj < sqrt_p; jj = jj + 1)
            for (kk = 0; kk < n_divide_ps; kk = kk + 1)
                for (ll = 0; ll < n_divide_ps; ll = ll + 1)
                    if (jj != sqrt_p - 1)
                        always @(posedge clk)
                            if (enable_shift)
                                tmp_B[ii][jj + 1][(kk * n_divide_ps + 11) * 32 + 31 : (kk * n_divide_ps + 11) * 32] = tmp_B[ii][jj][(kk * n_divide_ps + 11) * 32 + 31 : (kk * n_divide_ps + 11) * 32];
endgenerate
```

مرحله پنجم نیز در تصاویر نشان داده است. همان‌طور که می‌بینید داخل یک حلقه به اندازه \sqrt{p} قرار دارند.

ضرب ماتريس

يکي ديگر از بخش‌های مورد نیاز برای پیاده‌سازی ضرب موازی ماتریس‌ها، ضرب دو ماتریس با ابعاد دلخواه است که در آن برای ضرب دو ماتریس $n * n$ از $O(n^3)$ واحد ضرب و واحد جمع استفاده شده است که در این عملیات وجود دارد برابر است با n برابر delay یک واحد جمع‌کننده به علاوه یک واحد ضرب‌کننده. هر چند اگر بهینه‌تر پیاده‌سازی کرد، می‌توان این delay را به $O(\log(n))$ بار delay جمع‌کننده و یک delay ضرب‌کننده رساند که با توجه به بزرگتر شدن n می‌توان به شکل قابل توجهی سرعت کلاک را کمتر کرد و در غیر این‌صورت می‌توان سرعت کلاک را پایین نگه داشت ولی تعداد دورهایی که کلاک در استییت ضرب کردن است بنابر آزمایشات زمانی به‌دست آورد.

```
module mul_matrix
#(parameter n = 2)
(input [32 * n * n - 1: 0] mat_A, input [32 * n * n - 1: 0] mat_B, output [32 * n * n - 1: 0] mat_out);

wire [31: 0] mul_res [n - 1: 0][n - 1: 0][n - 1: 0];
wire [31: 0] acc_res [n - 1: 0][n - 1: 0][n - 1: 0];
genvar i, j, k, l;

generate
    for (i = 0; i < n; i = i + 1)
        for (j = 0; j < n; j = j + 1)
            for (k = 0; k < n; k = k + 1)
                begin
                    mul i_j_k(mat_A[(i + n * j) * 32 + 31 : (i + n * j) * 32], mat_B[(j + n * k) * 32 + 31 : (j + n * k) * 32], mul_res[i][k][j][31: 0]);
                end

    for (i = 0; i < n; i = i + 1)
        for (j = n - 1; j < n; j = j + 1)
            for (k = 0; k < n; k = k + 1)
                assign acc_res[i][k][j][31: 0] = mul_res[i][k][j][31: 0];

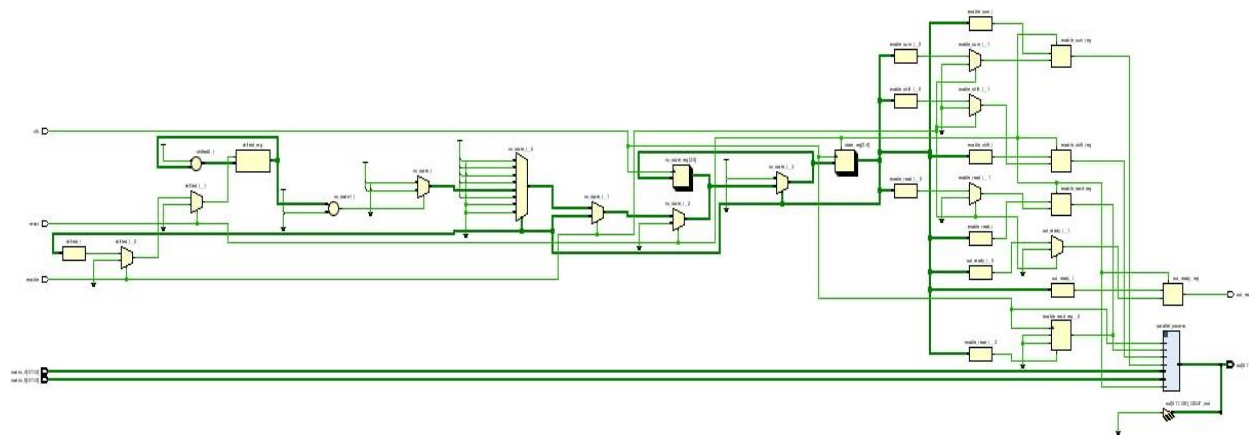
    for (i = 0; i < n; i = i + 1)
        for (j = 1; j < n; j = j + 1)
            for (k = 0; k < n; k = k + 1)
                adder i_j_k(acc_res[i][k][j][31: 0], mul_res[i][k][j - 1][31: 0], acc_res[i][k][j - 1][31: 0]);

    for (i = 0; i < n; i = i + 1)
        for (k = 0; k < n; k = k + 1)
            if (n > 1)
                assign mat_out[(i + n * k) * 32 + 31 : (i + n * k) * 32] = acc_res[i][k][0][31: 0];
            else
                assign mat_out[(i + n * k) * 32 + 31 : (i + n * k) * 32] = mul_res[i][k][0][31: 0];
endgenerate
endmodule
```

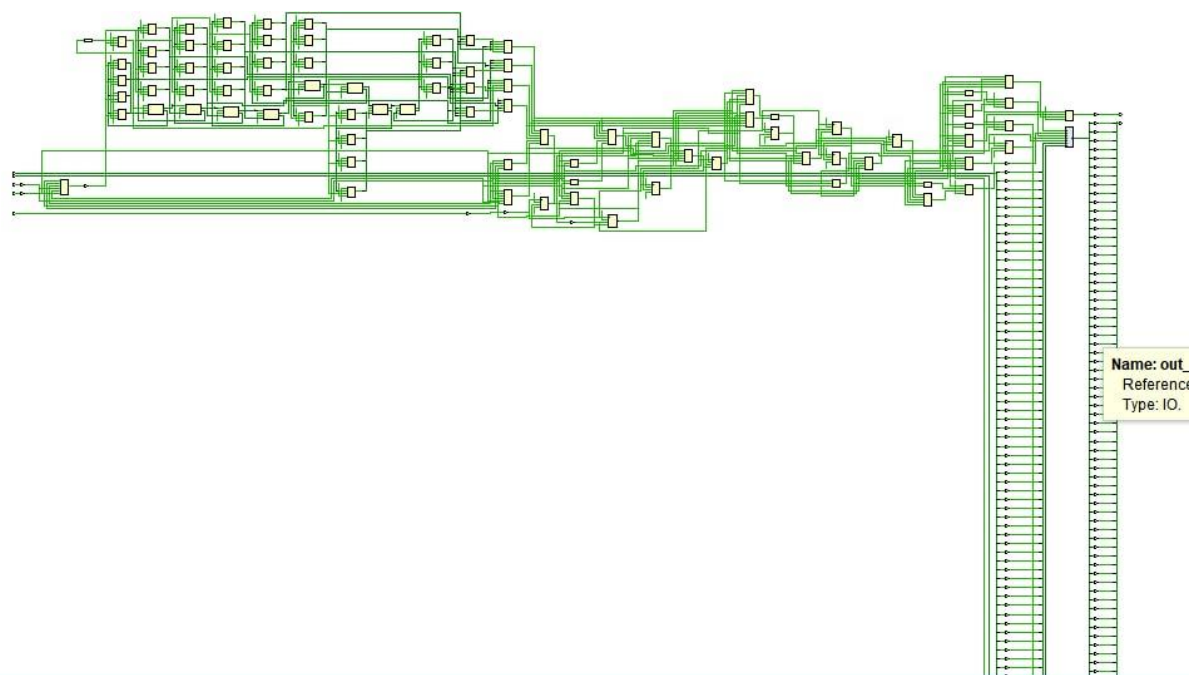
سنتز

ما با استفاده از ابزار سنتز Vivado طراحی خود در وریلاگ را سنتز کرده و کدهای ما کاملاً سنتز پذیر بوده و خروجی‌های آن به شکل زیر می‌باشد.

RTL Schematic



Synthesis schematic



ما با استفاده از روش تقسیم و حل و همچنین موازی‌سازی عملیات‌های پیدا کردن هر یک از درایه‌های ماتریس خروجی در توان مصرفی نهایی قطعه خود صرفه جویی بسیاری انجام داده‌ایم. طبق آزمایش‌ها و محاسباتی که بر روی این قضیه انجام شده است، جدول‌های زیر برای سه الگوریتم مختلف انجام این کار نشان داده شده است.

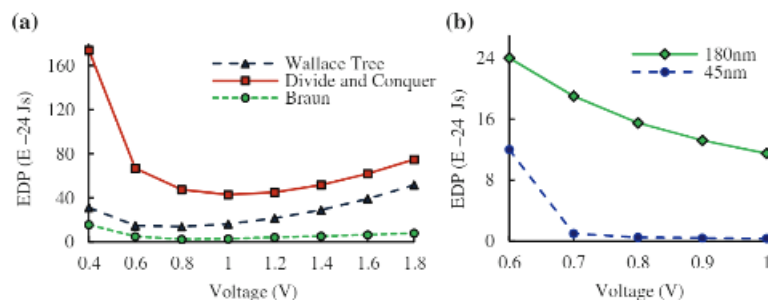


Fig. 4 a EDP of different multipliers at 180 nm technology node. b EDP comparison of Barun multiplier at 180 and 45 nm technology nodes

EDP

The proposed work uses energy-delay product (EDP), where energy the total energy consumption of cores and delay is the amount of time for executing applications.

نتیجه یکی از تست های امتحان شده با استفاده از طراحی ما:

```
[2022-07-29 05:29:40 EDT] iverilog '-wall' '-g2012' design.sv testbench.sv && unbuffer vvp a.out
```

Matrix A:

1	4
2	4

Matrix B:

0	1
1	1

Mult Result:

4	5
4	6

Done

صحت سنجی به کمک گلدن مدل:

	C_1	C_2
1	4	5
2	4	6