به نام خدا

فاز سوم پروژه

طراحی سیستم های دیجیتال

استاد فلاحتی

اعضای تیم:

سهیل نظری

علیرضا حقی

مهدی سعادت بخت

نیمسال دوم سال تحصیلی ۱۴۰۰ـ۱۴۰۱

دانشگاه صنعتی شریف

دانشکده مهندسی کامپیوتر

فهرست مطالب

توضیحات پروژه

یک واحد پردازشی ضرب دو ماتریس را با استفاده از الگوریتم تقسیم و حل پیاده سازی کرده‌ایم. در این طراحی مولفه‌های ماتریس مطابق با استاندارد IEEE754 دریافت می‌شوند. این طراحی با استفاده از تقسیم و حل به گونه‌ای انجام شده که انجام عملیات ضرب ماتریس به صورت موازی و بهینه انجام شود.

چکیده

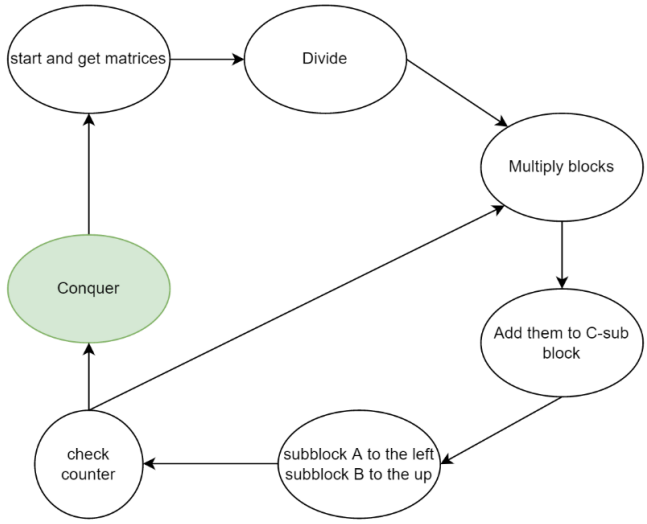
با توجه به محاسبات سنگین محاسباتی ماتریسی در حوزه‌های مختلف، پردازش موازی ضرب ماتریس‌ها در راستای کمینه کردن میزان انرژی و مساحت مورد استفاده اهمیت ویژه‌ای پیدا کرده است.

در این پروژه، بنابر دغدغه‌ی الزام ضرب موازی ماتریس‌ها، به طراحی یک مدل تقریبا بهینه با استفاده از زبان برنامه‌نویسی Verilog بر اساس مساحت و تعداد ماژول‌ها رسیده‌ایم. 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 کاملا قابل سنتز است و این امر به‌وسیلهٔ نرم‌افزار سنتز Vivado صورت گرفته است. همچنین صحت کد و کارایی آن توسط آزمایش‌کننده‌ای توسط Verilator صورت گرفته است. نکته‌ی حائز اهمیت دیگر مدل ما، پارامتریک بودن است که می‌توان به‌وسیلهٔ 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 پردازنده‌ها بخش‌پذیر است.

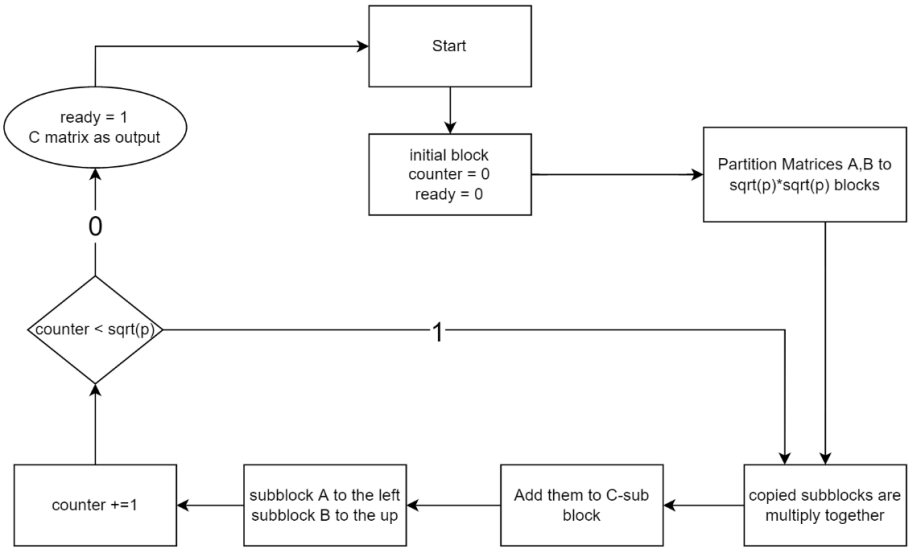
مقدمه

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. نکتهٔ 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
```

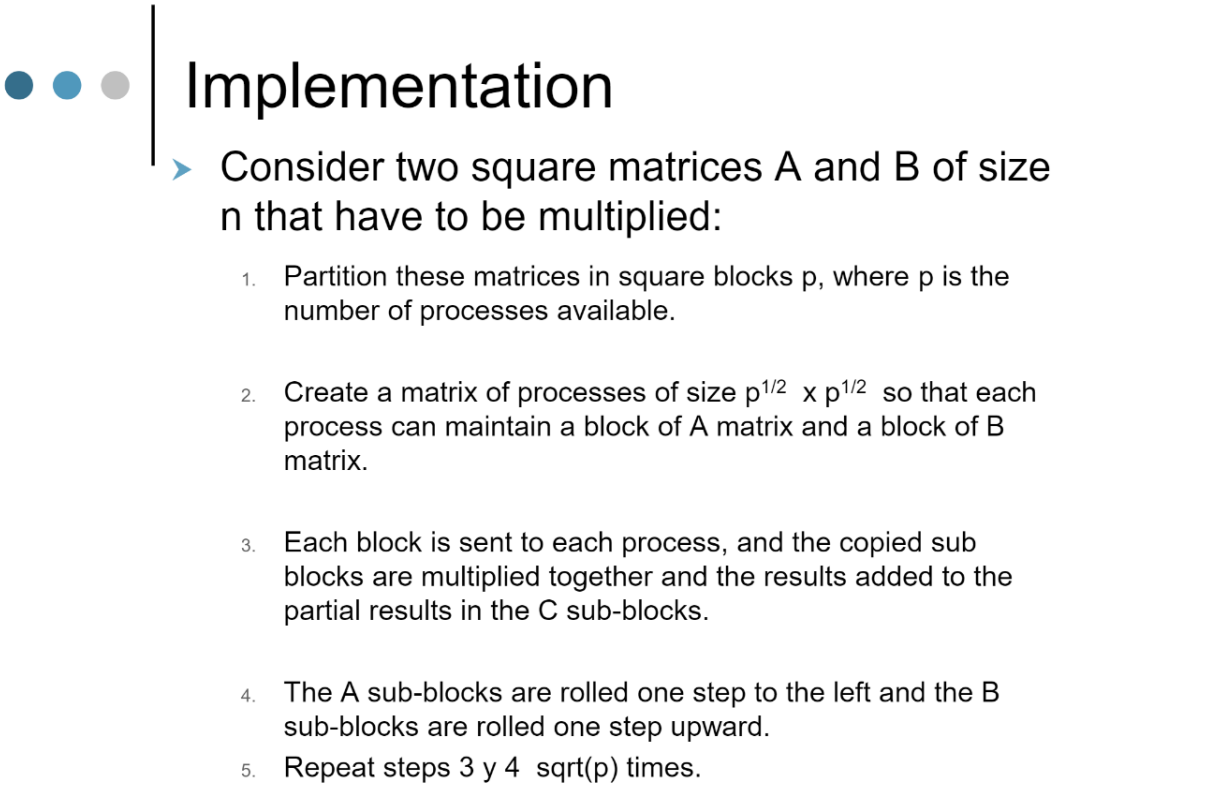
تکه کد انتقال میان استیت‌ها و ست کردن بیت‌های کنترلی:

```
always @(state)
begin
    if(!reset)
    begin
        if(enable)
            case(state)
                0:      alirezahaqi, 5 hours ago • C
                    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
```

الگوریتم ضرب ماتریس

الگوریتم کلی بدین شکل است که هر دو ماتریس n * n را به p^2 ماتریس با ابعاد $\frac{n}{p} * \frac{n}{p}$ 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 شبه‌کدی برای الگوریتم ارائه شده است (p^2 را تعداد پردازنده‌ها است).

عملیات‌های ضرب دو ماتریس به شکل تقسیم و حل به ۵ مرحله تقسیم می‌شود که به شکل زیر است.



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 به‌واسطهٔ جنریتورها توانستیم برنامهٔ کاملا 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 ساب‌بلاک‌های 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
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];
    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;
endgenerate
```

مرحله سوم (ضرب ساب‌بلاک‌ها 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]);

    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
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
endgenerate

genvar ii, jj, kk, ll;
```

مرحله چهارم (شیفت دادن ماتریس a به راست و ماتریس b به پایین)

```
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[0][jj][(kk * n_divide_ps + ll) * 32 + 31 : (kk * n_divide_ps + ll)*32] <= tmp_A[ii][jj][(kk * n_divide_ps + ll) * 32 + 31 : (kk * n_divide_ps + ll) * 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 + ll) * 32 + 31 : (kk * n_divide_ps + ll)*32] <= tmp_A[ii][jj][(kk * n_divide_ps + ll) * 32 + 31 : (kk * n_divide_ps + ll)*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 + ll) * 32 + 31 : (kk * n_divide_ps + ll) * 32] = tmp_B[ii][jj][(kk * n_divide_ps + ll) * 32 + 31 : (kk * n_divide_ps + ll)*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 + ll) * 32 + 31 : (kk * n_divide_ps + ll)*32] = tmp_B[ii][jj][(kk * n_divide_ps + ll) * 32 + 31 : (kk * n_divide_ps + ll)*32];
endgenerate
```

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

ضرب ماتریس

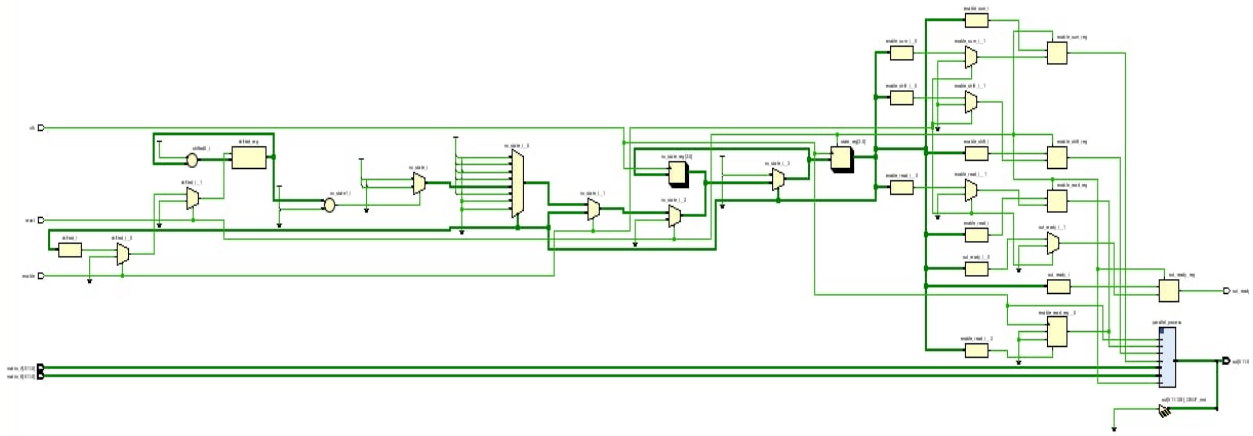
یکی دیگر از بخش‌های مورد نیاز برای پیاده‌سازی ضرب موازی ماتریس‌ها، ضرب دو ماتریس با ابعاد دلخواه است که در آن برای ضرب دو ماتریس $n * n$ از $O(n^3)$ واحد ضرب و واحد جمع استفاده شده است که delay که در این عملیات وجود دارد برابر است با n برابر delay یک واحد جمع‌کننده به‌علاوهٔ یک واحد ضرب‌کننده. هر چند 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 استیت 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
```

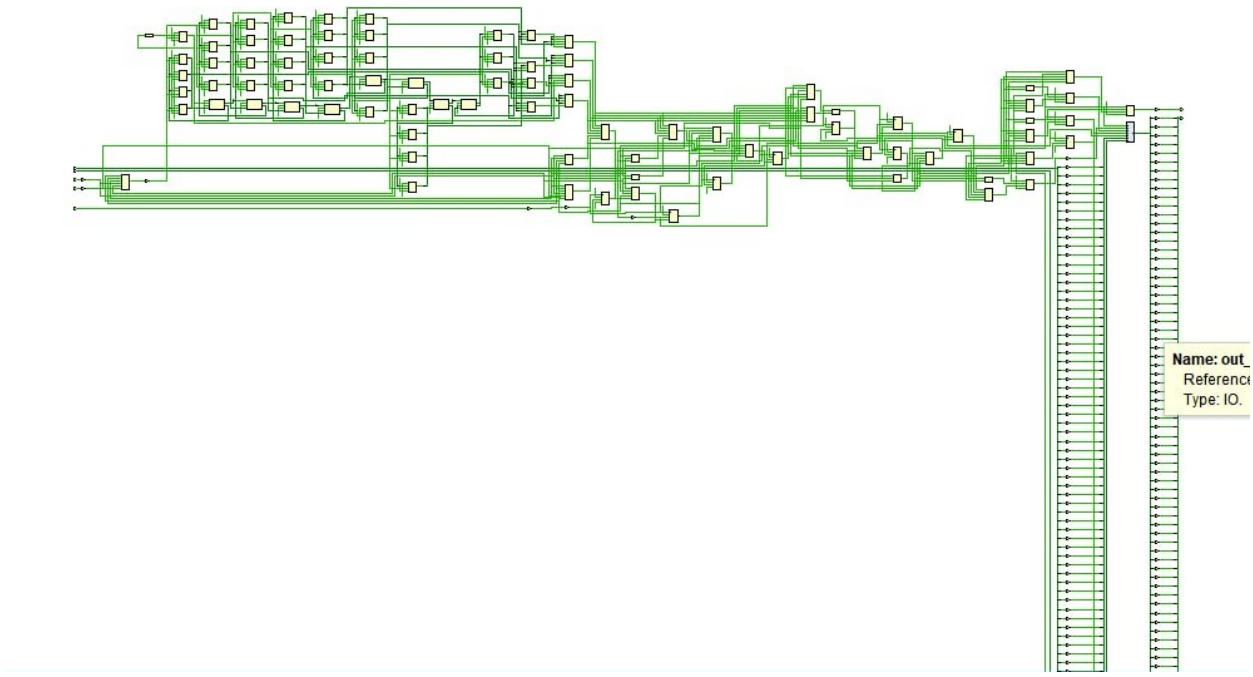
سنتز

ما با استفاده از ابزار سنتز Vivado طراحی خود در وریلاگ را سنتز کرده و کدهای ما کاملا سنتزپذیر 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.

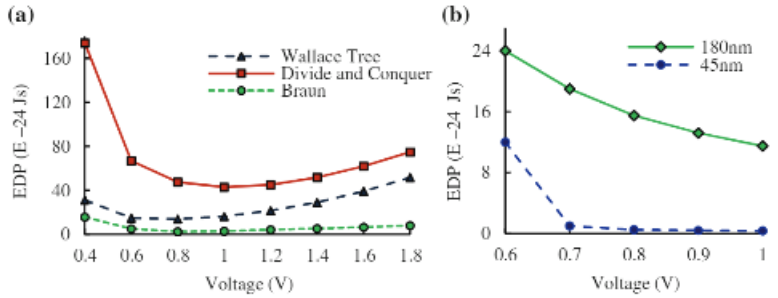


Fig. 4 **a** EDP of different multipliers at 180 nm technology node. **b** EDP comparison of Barun multiplier at 180 and 45 nm technology nodes

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.