

به نام خدا

فاز سوم پروژه طراحی سیستم های دیجیتال



نیمسال دوم سال تحصیلی ۱۴۰۰-۱۴۰۱

دانشگاه صنعتی شریف

دانشکده مهندسی کامپیوتر

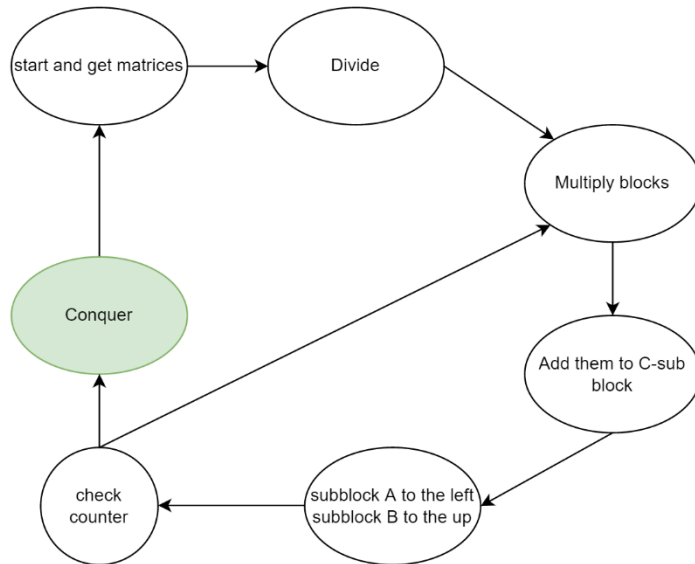
توضیحات پروژه:

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 ایی انجام شده که انجام عملیات ضرب ماتریس به صورت موازی و بهینه انجام شود.

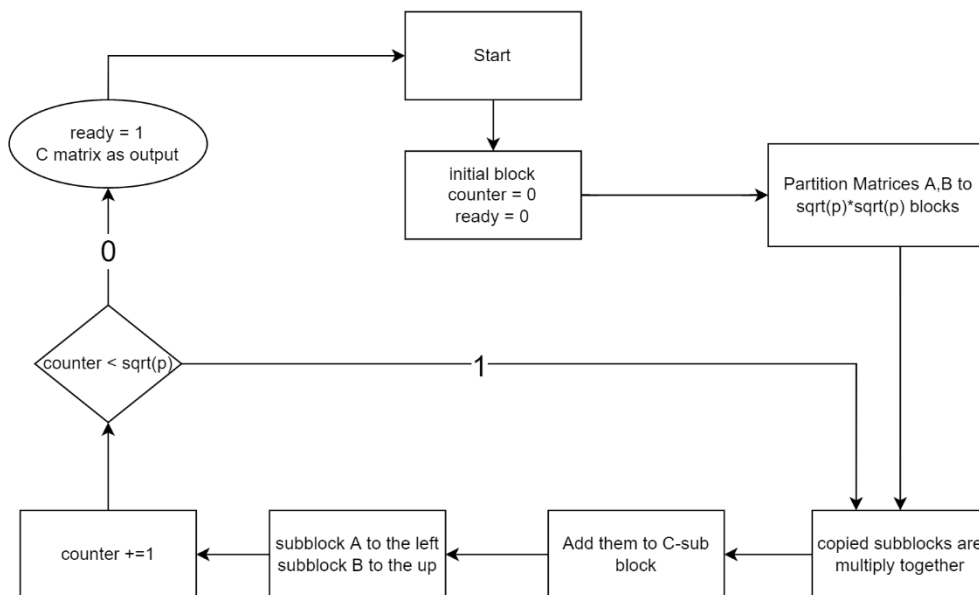
اعضای تیم:

- سهیل نظری مندجین
- علیرضا حقی
- مهدی سعادت بخت

:Fsm and Asm



ASM



با استفاده از طراحی Fsm انجام شده یک ماژول کنترلر تعریف کرده ایم که به وسیله آن استتیت های مختلف ماشین ما تغییر کرده و سه بیت کنترلی برای خواندن، جمع کردن و شیفت دادن در اختیار داریم که میان استتیت ها مقادیرشان را تنظیم میکنیم.

فاز سوم پروژه طراحی سیستم های دیجیتال

تکه کد ریست و استارت در استیت 0:

```
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 استیت ها و ست کردن بیت های 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
    end
    else
        nx_state = state;
    end
end
```

عملیات های ضرب دو ماتریس به شکل تقسیم و حل به 5 مرحله تقسیم می شود 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 جنریتور ها پیاده سازی کرده ایم که کد مراحل مختلف مطابق روبرو است:

مرحله اول و دوم (پارنتیشن بندی ماتریس به ساب بلاک های کوچک تر)

```
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
  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
      end
    end
    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
    end
  end
endgenerate
```

فاز سوم پروژه طراحی سیستم های دیجیتال

مرحله سوم (ضرب ساب بلاک ها در هم و سپس جمع ان ها 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

//multiplying 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 رادیکال پی قرار دارند.

فاز سوم پروژه طراحی سیستم های دیجیتال

در مرحله سوم برای ضرب دو زیر ماتریس در هم نیازمند یک ماژول دیگر هستیم که به اینگونه پیاده سازی کرده ایم.

```
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

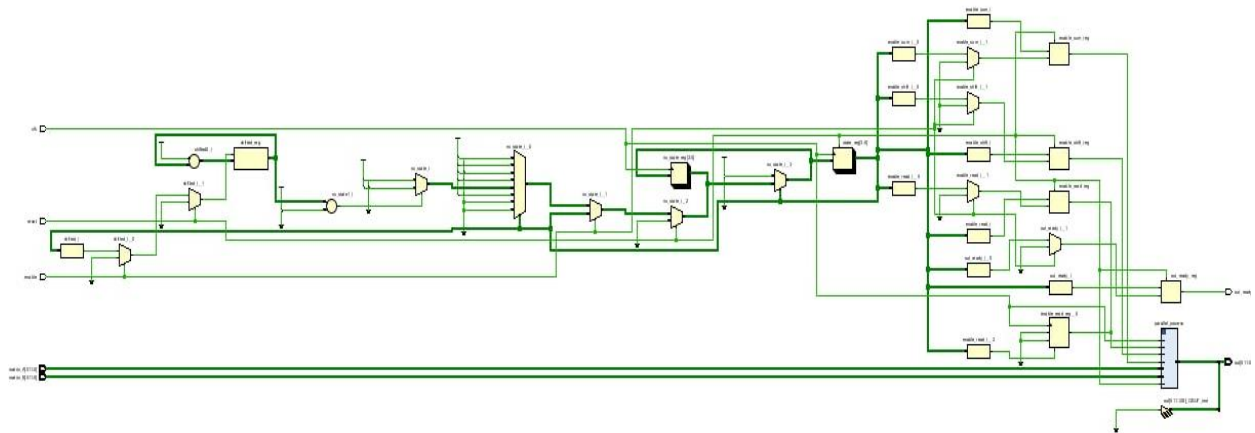
    for (i = 0; i < n; i = i + 1)
        for (j = n - 1; j < n; j = j + 1)
            for (k = 0; k < n; k = k + 1)
                assign acc_res[i][k][j][31 : 0] = mul_res[i][k][j][31: 0];

    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]);

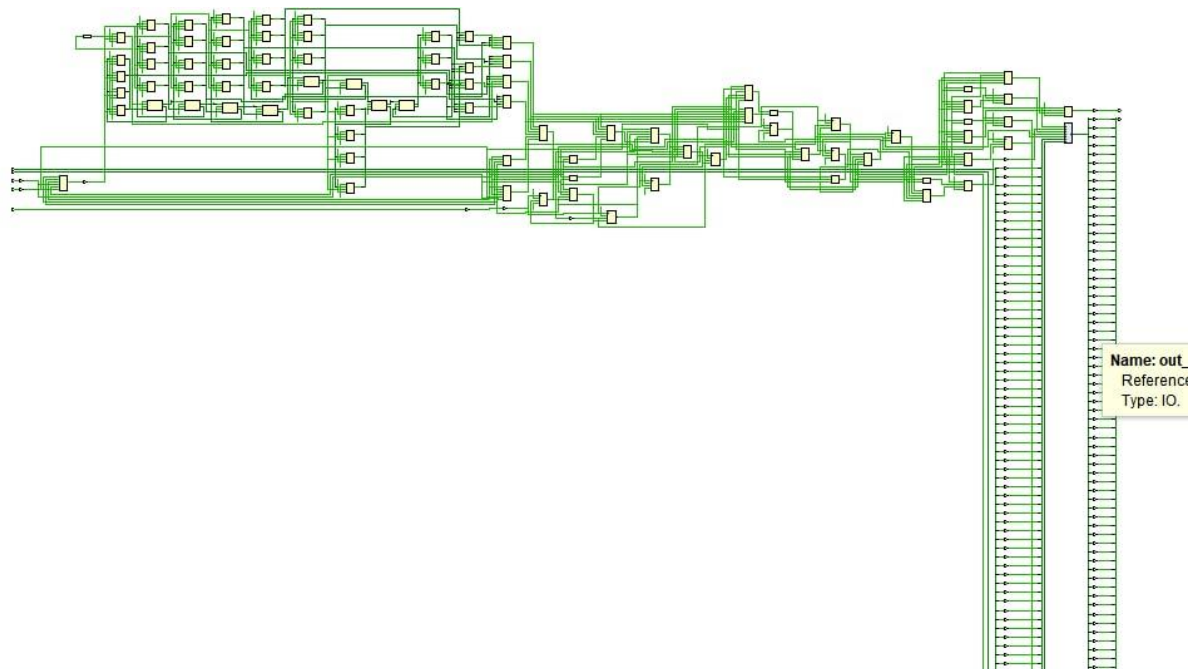
    for (i = 0; i < n; i = i + 1)
        for (k = 0; k < n; k = k + 1)
            if (n > 1)
                assign mat_out[(i + n * k)*32 + 31 : (i + n * k)*32] = acc_res[i][k][0][31 : 0];
            else
                assign mat_out[(i + n * k)*32 + 31 : (i + n * k)*32] = mul_res[i][k][0][31 : 0];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

ما با استفاده از ابزار سنتز Vivado طراحی خود در ورپلاگ را سنتز کرده و کد های ما کاملاً سنتز پذیر بوده و خروجی های آن به شکل زیر می باشد.

Rtl schematic



Synthesis schematic



ما با استفاده از روش تقسیم و حل و همچنین موازی سازی عملیات های پیدا کردن هر یک از درایه های ماتریس خروجی در توان مصرفی نهایی قطعه خود صرفه جویی بسیاری انجام داده ایم طبق آزمایش ها و محاسباتی که بر روی این قضیه انجام شده است جدول های زیر برای سه الگوریتم مختلف انجام اینکار نشان داده شده است.

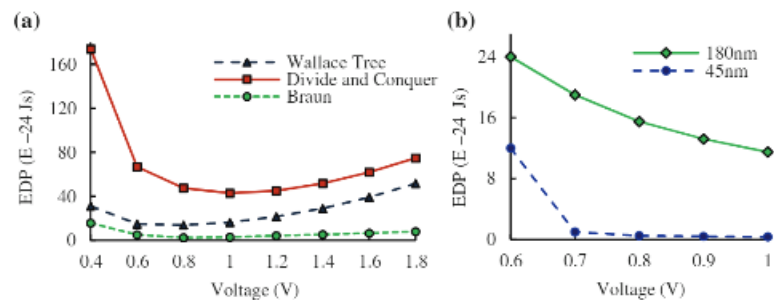


Fig. 4 a EDP of different multipliers at 180 nm technology node. b EDP comparison of Barun multiplier at 180 and 45 nm technology nodes

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.