一.　　实验目的

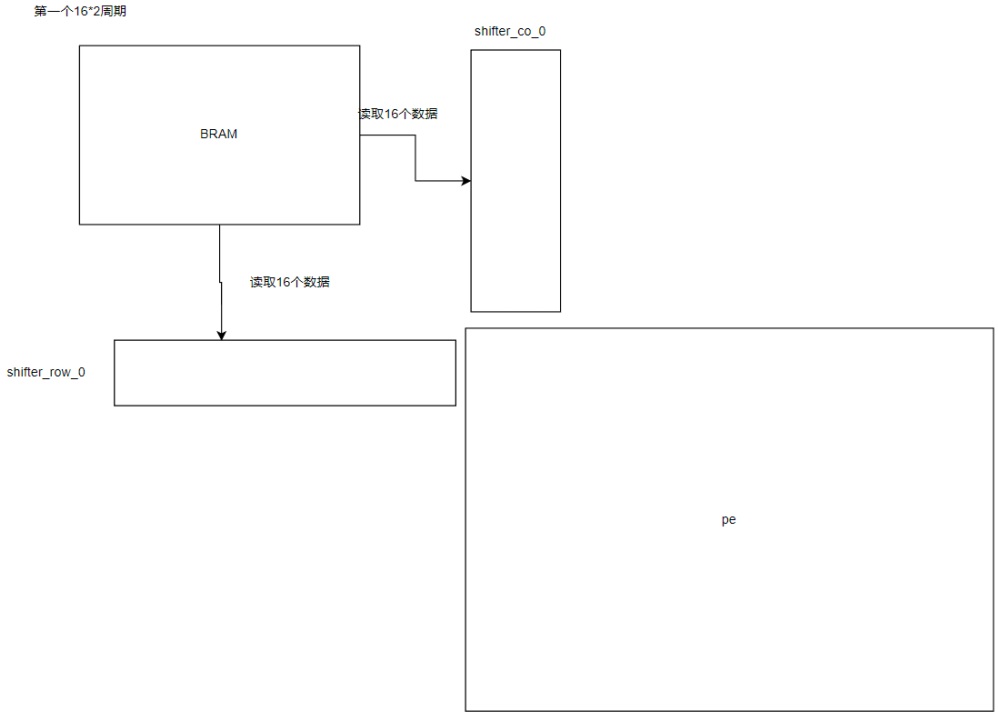
- 学习矩阵乘法单元的设计
- 使用 Verilog 实现所设计的矩阵乘法单元并进行仿真验证

二.　　实验步骤

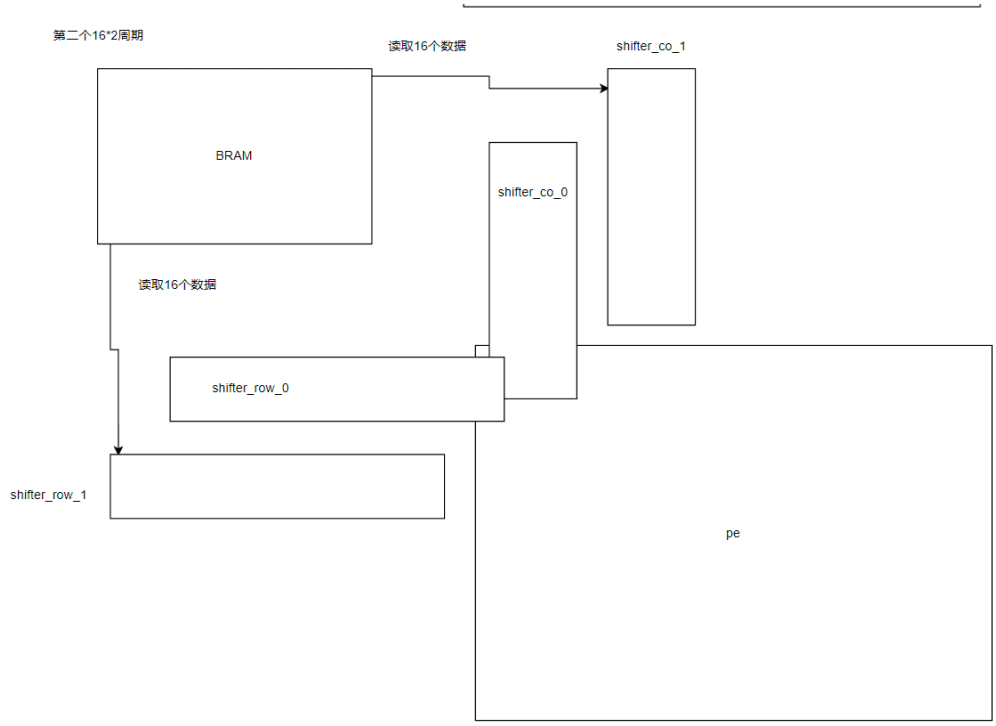
1，设计脉动阵列输入和控制逻辑

对于本实验的场景，每个 clk 的输入只有 32bit，对脉动阵列来说，**输入带宽远远低于计算的能力，IO 带宽是本题最大的瓶颈**，所以需要对输入进行缓冲，累积一定输入数据后进行一次计算

在我的设计中，为脉动阵列的横向和纵向都分别设置移位寄存器作为缓冲，每次取 2 个 16 个 32bit 数据作为横向和纵向的移位寄存器的输入，需要 32 个 clk。完成一个 32clk 的周期后，以移位寄存器作为输入，脉动阵列可以脉动一次



脉动后，再次进行 32 个周期的输入，再次脉动，此时脉动时，横向和纵向的移位寄存器 row_0 和 col_0 已经移位过一次，其输出是第二个元素，而 row_1 和 col_1 移位寄存器的输出是第一个元素，这样就完成了数据错位输入脉动阵列，见下图。在输入的过程中，每 32clk 附加一次脉动，这样**使输入和计算得以并行**



2， 实现脉动阵列输入和控制模块

输入的地址由**地址生成模块**进行计算，横向移位寄存器的输入为第一个矩阵按行进行输入，其地址计算公式为 row*16 + col，纵向移位寄存器输入为第二个矩阵按列进行输入，其地址计算公式为 256+col+row*16，其 verilog 实现如下

其中 row_en 表示是否计算横向输入地址

```
module addr_gen #(ADDRLEN = 9) (
    input clk, rst, row_en,
    output [ADDRLEN-1:0] addr
);
    reg [3:0] row;
    reg [3:0] col;
    reg [3:0] cnt;

    always @(posedge clk) begin
        if (rst) begin
            row <= 0;
            col <= 0;
            cnt <= 0;
        end else begin
            if (cnt == 15) begin
                row <= row + row_en;
                col <= col + !row_en;
                cnt <= 0;
            end
            else
                cnt <= cnt + 1;
        end
    end

    assign addr = row_en ? ((row << 4) | cnt) : (9'h100 | col | (cnt << 4));
endmodule
```

移位寄存器的控制与平常的 shifter 稍微有不同，不能在每个时钟周期时都移位，使用专门的输入使能 load 和输出使能 en 来控制，只有当期中一个为 1 时才进行移位

```
module shifter #(LENGTH = 16, WIDTH = 32) (
    input clk, rst, load, en,
    input [WIDTH-1:0] idata,
    output [WIDTH-1:0] out
);

    reg [WIDTH-1:0] shift_data [LENGTH-1:0];
    integer i;

    always @(posedge clk) begin
        if (rst) begin
            for (i = 0; i < LENGTH; i = i + 1) begin
                shift_data[i] <= 0;
            end
        end else if (load) begin
            shift_data[0] <= idata;
            for (i = 1; i < LENGTH; i = i + 1) begin
                shift_data[i] <= shift_data[i-1];
            end
        end else if (en) begin
            shift_data[0] <= 0;
            for (i = 1; i < LENGTH; i = i + 1) begin
                shift_data[i] <= shift_data[i-1];
            end
        end
    end

    assign out = shift_data[LENGTH-1];
endmodule
```

还需要一个控制模块来控制横向或纵向的哪一个移位寄存器进行输入，该模块的输入是降频 16 倍的 clk

```
module controller #(WIDTH = 16) (
    input clk, rst,
    output [WIDTH-1:0] row_shifter_en,
    output [WIDTH-1:0] col_shifter_en,
    output reg row_en
);
    reg [WIDTH-1:0] shift_en;

    always @(posedge clk) begin
        if (rst) begin
            shift_en <= 1;
            row_en = 0;
        end else begin
            shift_en <= row_en ? shift_en << 1 : shift_en;
            row_en <= ~row_en;
        end
    end

    assign row_shifter_en = {WIDTH{row_en}} & shift_en;
    assign col_shifter_en = ~{WIDTH{row_en}} & shift_en;
endmodule
```

在顶层模块中实例化如下

```
wire [15:0] row_shifter_en, col_shifter_en;
// wire [15:0] col_shifter_en;
controller ctrl(.clk(clk16x), .rst(rst), .row_en(row_en), .row_shifter_en(row_shifter_en), .col_shifter_en(col_shifter_en)
addr_gen addr_generator(.clk(clk), .rst(rst), .row_en(row_en), .addr(mab_addr));

/* ----------------------------------------------------------------- */
// 输入 buffer
// clk32x 每32周期一次 代表 col 和 row 都加载完一次向量
wire [31:0] shifter_row_0, shifter_col_0, shifter_row_1, shifter_col_1, shifter_row_2, shifter_col_2, shifter_row_3, shift
shifter_row_6, shifter_col_6, shifter_row_7, shifter_col_7, shifter_row_8, shifter_col_8, shifter_row_9, shifter_col_9, sh
shifter_col_12, shifter_row_13, shifter_col_13, shifter_row_14, shifter_col_14, shifter_row_15, shifter_col_15;

shifter row_0(.clk(clk), .rst(rst), .load(row_shifter_en[0]), .en(PE_clk), .idata(mab_dout), .out(shifter_row_0));
shifter col_0(.clk(clk), .rst(rst), .load(col_shifter_en[0]), .en(PE_clk), .idata(mab_dout), .out(shifter_col_0));
shifter row_1(.clk(clk), .rst(rst), .load(row_shifter_en[1]), .en(PE_clk), .idata(mab_dout), .out(shifter_row_1));
shifter col_1(.clk(clk), .rst(rst), .load(col_shifter_en[1]), .en(PE_clk), .idata(mab_dout), .out(shifter_col_1));
shifter row_2(.clk(clk), .rst(rst), .load(row_shifter_en[2]), .en(PE_clk), .idata(mab_dout), .out(shifter_row_2));
shifter col_2(.clk(clk), .rst(rst), .load(col_shifter_en[2]), .en(PE_clk), .idata(mab_dout), .out(shifter_col_2));
shifter row_3(.clk(clk), .rst(rst), .load(row_shifter_en[3]), .en(PE_clk), .idata(mab_dout), .out(shifter_row_3));
```

3， 实现脉动阵列 PE 模块并在顶层模块中实例化

计算模块 PE 实现如下

```
module PE #( WIDTH = 32) (
    input clk, rst,
    input [WIDTH-1:0] in_col,
    input [WIDTH-1:0] in_row,
    output reg [WIDTH-1:0] out_col,
    output reg [WIDTH-1:0] out_row,
    output reg [WIDTH-1:0] out_sum
);

    always @(posedge clk) begin
        if (rst) begin
            out_col <= 0;
            out_row <= 0;
            out_sum <= 0;
        end else begin
            out_col <= in_col;
            out_row <= in_row;
            out_sum <= in_col*in_row + out_sum;
        end
    end
endmodule
```

由于有 256 个 PE，也不可能手动去连线，于是采用了 python 脚本生成的方式来实例化，在第 0 行和第 0 列时输入为移位寄存器，其他的 PE 输入由上一个 PE 结果连线而来

```
# 脉动阵列生成
wire_str = "wire [31:0] "

for r in range(16):
    for c in range(16):
        base = "sa_" + str(r) + "_" + str(c)
        wire_str += base + "_row, "
        wire_str += base + "_col, "
        # wire_str += base + "_out, "

        in_row = "sa_" + str(r) + "_" + str(c-1) + "_row" if c != 0 else "shifter_row_" + str(r)
        in_col = "sa_" + str(r-1) + "_" + str(c) + "_col" if r != 0 else "shifter_col_" + str(c)
        print("\tPE %s(.clk(clk32x), .rst(rst), .in_col(%s), .in_row(%s), .out_col(%s_col), .out_row(%s_row), .out_sum(%s));"
            % (base, in_col, in_row, base, base, "sa_out[%s]" % str(r*16+c)))

        if c %4 == 0:
            wire_str += "\n\t"

print(wire_str)
```

生成出来结果如下：

```
wire [31:0] sa_out [255:0];
wire [31:0] sa_0_0_row, sa_0_0_col,
sa_0_1_row, sa_0_1_col, sa_0_2_row, sa_0_2_col, sa_0_3_row, sa_0_3_col, sa_0_4_row, sa_0_4_col,
sa_0_5_row, sa_0_5_col, sa_0_6_row, sa_0_6_col, sa_0_7_row, sa_0_7_col, sa_0_8_row, sa_0_8_col,
sa_0_9_row, sa_0_9_col, sa_0_10_row, sa_0_10_col, sa_0_11_row, sa_0_11_col, sa_0_12_row, sa_0_12_col,
sa_0_13_row, sa_0_13_col, sa_0_14_row, sa_0_14_col, sa_0_15_row, sa_0_15_col, sa_1_0_row, sa_1_0_col,
```

```
PE sa_0_0(.clk(PE_clk), .rst(rst), .in_col(shifter_col_0), .in_row(shifter_row_0), .out_col(sa_0_0_col), .out_row(sa_0_0_row), .out_sum(sa_out[0]));
PE sa_0_1(.clk(PE_clk), .rst(rst), .in_col(shifter_col_1), .in_row(sa_0_0_row), .out_col(sa_0_1_col), .out_row(sa_0_1_row), .out_sum(sa_out[1]));
PE sa_0_2(.clk(PE_clk), .rst(rst), .in_col(shifter_col_2), .in_row(sa_0_1_row), .out_col(sa_0_2_col), .out_row(sa_0_2_row), .out_sum(sa_out[2]));
PE sa_0_3(.clk(PE_clk), .rst(rst), .in_col(shifter_col_3), .in_row(sa_0_2_row), .out_col(sa_0_3_col), .out_row(sa_0_3_row), .out_sum(sa_out[3]));
PE sa_0_4(.clk(PE_clk), .rst(rst), .in_col(shifter_col_4), .in_row(sa_0_3_row), .out_col(sa_0_4_col), .out_row(sa_0_4_row), .out_sum(sa_out[4]));
PE sa_0_5(.clk(PE_clk), .rst(rst), .in_col(shifter_col_5), .in_row(sa_0_4_row), .out_col(sa_0_5_col), .out_row(sa_0_5_row), .out_sum(sa_out[5]));
```

4， 实现输出逻辑

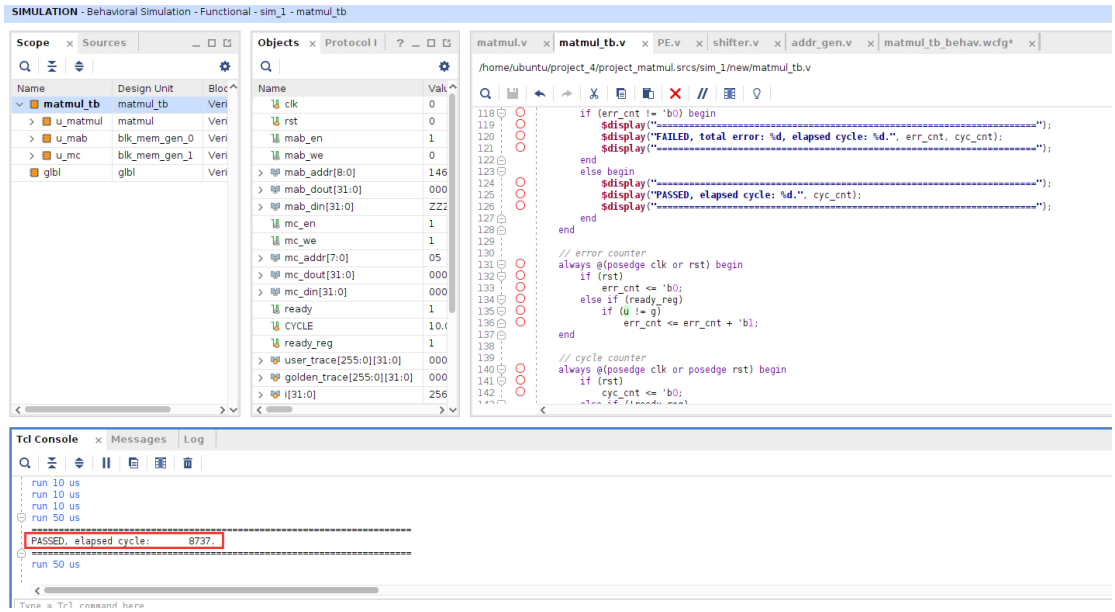
当横向和纵向移位寄存器 load 使能信号全 0 时表示移位寄存器输入已完成，此时脉动阵列已经计算出来了（0，0）的值，可以进行输出

由于**脉动阵列每次产生多个输出，而每个周期只能向外部 bram 写一个 32bit 结果**，所以可以直接顺序产生地址写回 bram

```
// gen output addr
reg out;

reg finish;
reg [7:0] oaddr;
always @(posedge clk32x) begin
    if (row_shifter_en == 16'h0 && col_shifter_en == 16'h0 ) begin
        out <= 1;
    end
end

always @(posedge clk) begin
    if (oaddr == 8'd255) begin
        finish <= 1;
    end

    if (out) begin
        oaddr <= oaddr + 1;
    end
end
assign mc_en = 1;
assign mc_we = out;
assign mc_addr = oaddr;
assign mc_din = sa_out[oaddr];
assign ready = finish;
```

5， 输出加速

在第一版实现中，脉动阵列由于输入带宽的限制，所以其计算一次的周期为 32clk，以此作为整体时钟，那么使用的周期数为输入 512*1clk+输出 256*32clk，整体周期达到了 8k+



但是实际上只是输入带宽限制了脉动阵列的计算，当输入完成后，脉动阵列可以一个 clk 计算一次，所以对脉动阵列、移位寄存器的时钟进行修改：

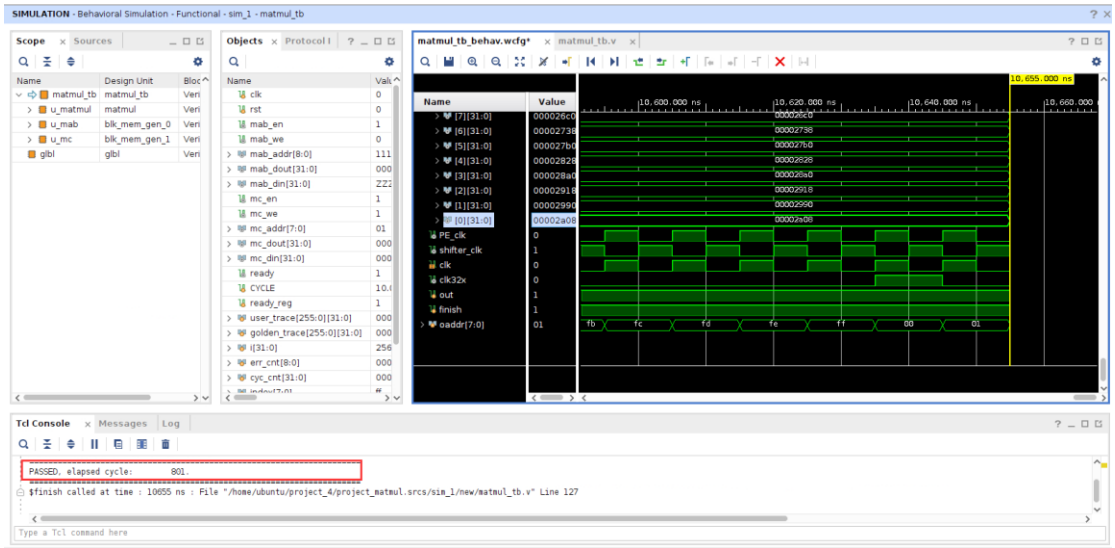
```
wire PE_clk = out ? clk : clk32x;   // for speed up
wire shifter_clk = out ? !clk : clk;    // for speed up
```

```
shifter row_0(.clk(shifter_clk), .rst(rst), .load(row_shifter_en[0]), .en(out ? 1 : clk32x), .idata(mab_dout), .out(shifter_row_0));
PE sa_0_0(.clk(PE_clk), .rst(rst), .in_col(shifter_col_0), .in_row(shifter_row_0), .out_col(sa_0_0_col),
```

此处 PE 和 shifter 的上升沿相同会导致从 shifter 读数据慢一拍，所以做了取反的处理

三. 实验结果

经过改进后，testbench 计数为 **801** 个 clk

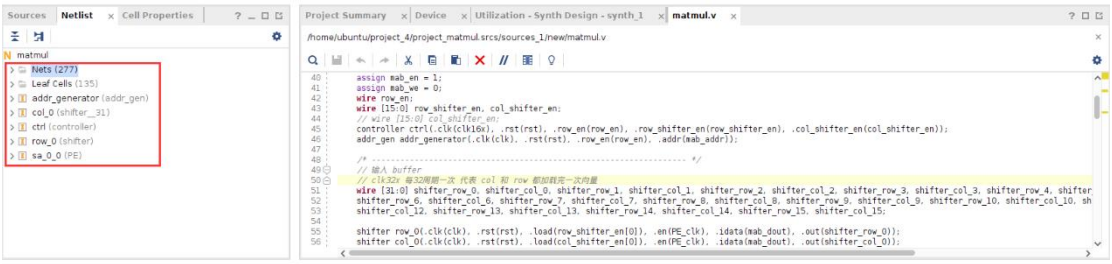


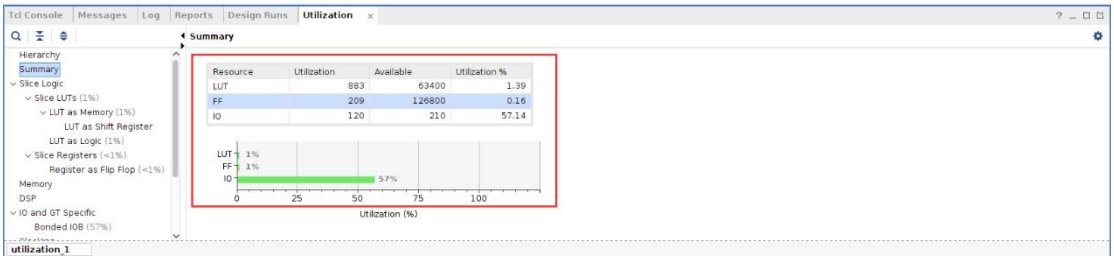
分析：读 512 个 circle，读完之后刚好第一个结果计算出来了，再花 256 个 circle 写回去，中间因为控制器的原因多了一个 32clk 的周期

512 + 256 + 32 = 800

说明瓶颈主要在 IO 侧，计算时间被 IO 时间重叠掉了

资源使用：

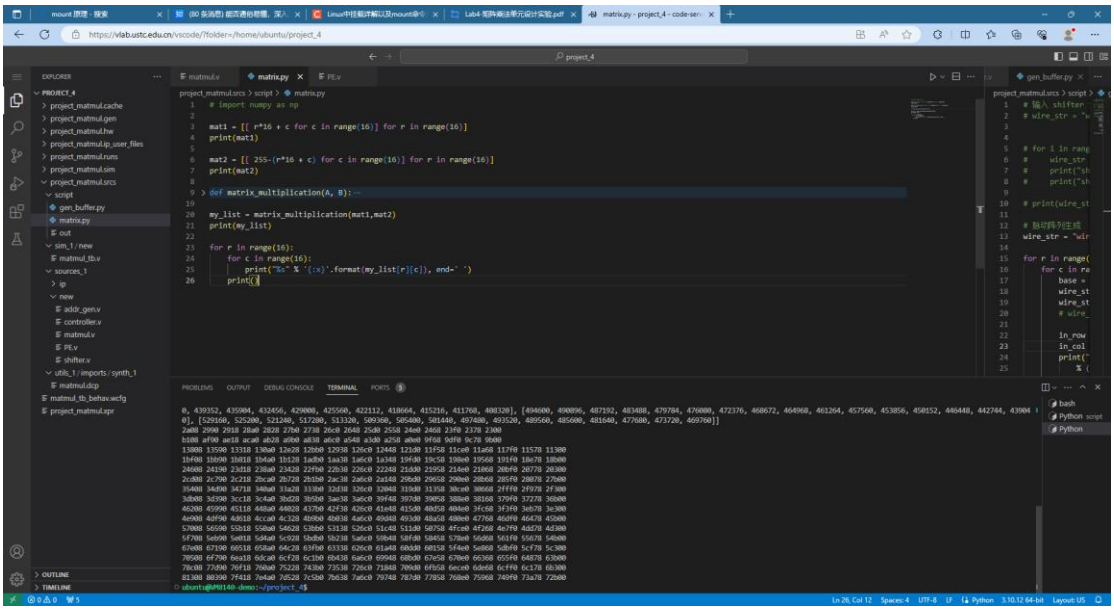


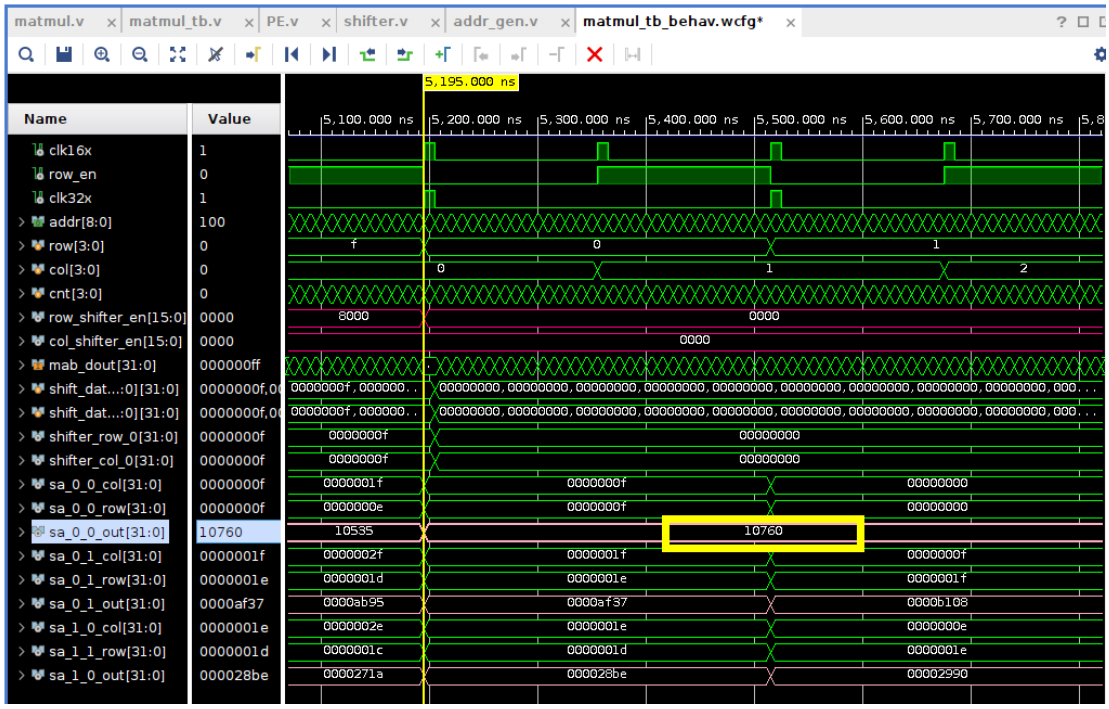


其生成的网表与我实例化的不一样，可能是因为综合时被优化掉了，暂时没找到办法

四． Debug 过程与脚本

由 bram 的 coe 文件可以看出两个矩阵值为 0-255 和 255-0，通过 python 可以计算出其矩阵相乘的结果的值，用于比对 sa_out 的值





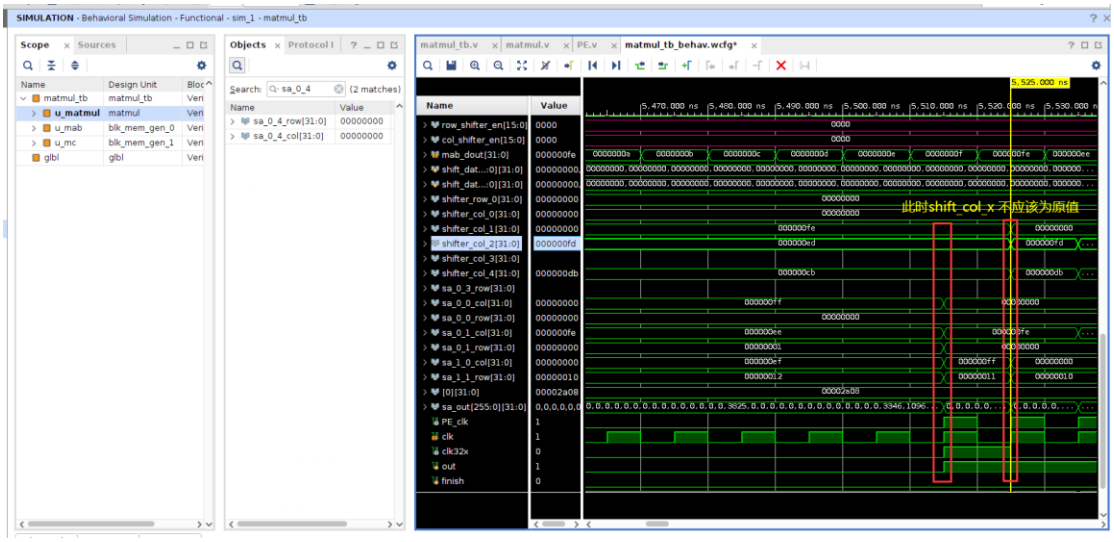
在最开始验证输入控制逻辑时，sa_0_0 输出正确代表前面的部分没有问题

麻烦的 bug 出现在试图在输入完成后将 PE 脉动周期改为 1 个 clk，原来的代码是这么写的，移位寄存器和 PE 的上升沿均由 clk 来产生

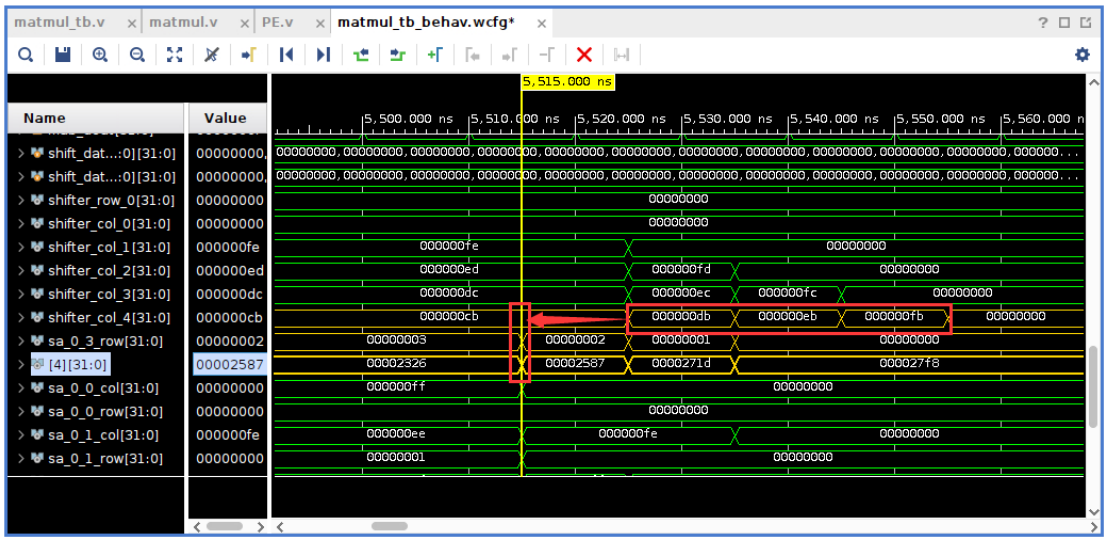
```
wire PE_clk = out ? clk : clk32x;   // for speed up
```

```
shifter row_0(.clk(clk), .rst(rst), .load(row_shifter_en[0]), .en(PE_clk), .idata(mab_dout), .out(shifter_row_0));
```

导致的问题就是：在一个 PE_clk 上升沿时 PE 从 shifter 取值，问题是 shifter 这个时候没移位，下一个 clk 才移位，PE_clk 是 32 的时候没问题，是 1 的时候，下次移位的时候，PE 已经在计算了，又取到一次原来的值，就错了



这里应该有的效果是



举个例子：PE04 的输入为 shifter_col_4 和 sa_0_3_row（移位寄存器和上一个 PE 的输出）此处红色框时是上升沿，输入 PE 的值为 shifter 的 0xcb 和 0x3，下一个周期本应该是 0xdb*0x2；但是可以看到波形图里下一个上升沿时 shifter 的值仍为 0xcb，所以 PE 读到的值是 0xcb 和 0x2，错误

这个问题是按周期取值的共性问题，我一般是 clk 取反解决，提前半个周期把输入值准备好，结果我给 bram 已经取反了，最后搞得 shifter 取反很麻烦，只能在输入完成之后取反，效果如下

```
wire PE_clk = out ? clk : clk32x;   // for speed up
wire shifter_clk = out ? !clk : clk;    // for speed up
```

```
shifter_row_0[0].clk(shifter_clk), .rst(rst), .load(row_shifter_en[0]), .en(out ? 1 : clk32x), .idata(mab_dout), .out(shifter_row_0));
```

蓝色框的 shifter 在 PE_clk 的前半个周期（红色框处）就准备好了数据，紫色框是上升沿，PE 计算时取得了正确的 shifter 数据，即 0xdb，此时 PE04 计算的就是 0xdb*0x2