Systolic Array for Applying Matrix Multiplication

Table of contents

Table of figures

1. Introduction

This section is for clarifying the design method used in the design.

The design depends on skew array (put the inputs in a diagonally) for feeding the right sequence at the right time to the PE unit.

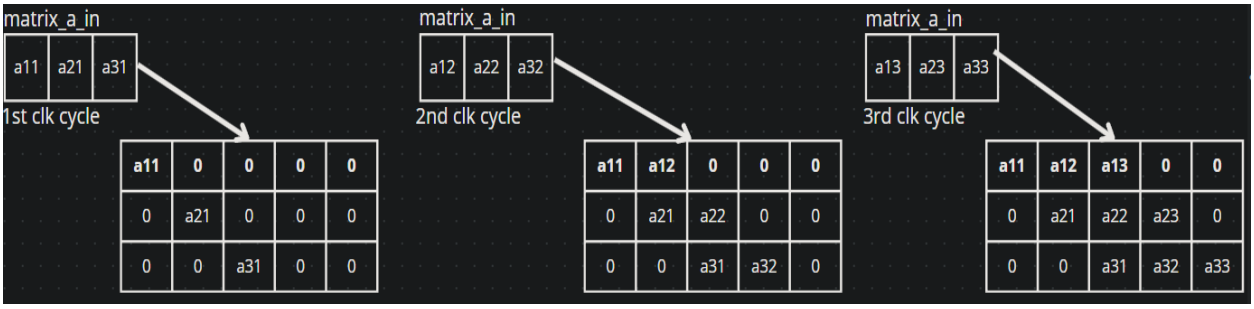

Figure 1 : input matrix_A method

While taking the input and assigning it to a 2-D array as shown in the figure, in parallel we send the first column of this array to the structural_imp module and the second column in the next cycle and so on, to ensure the parallelism of the design.

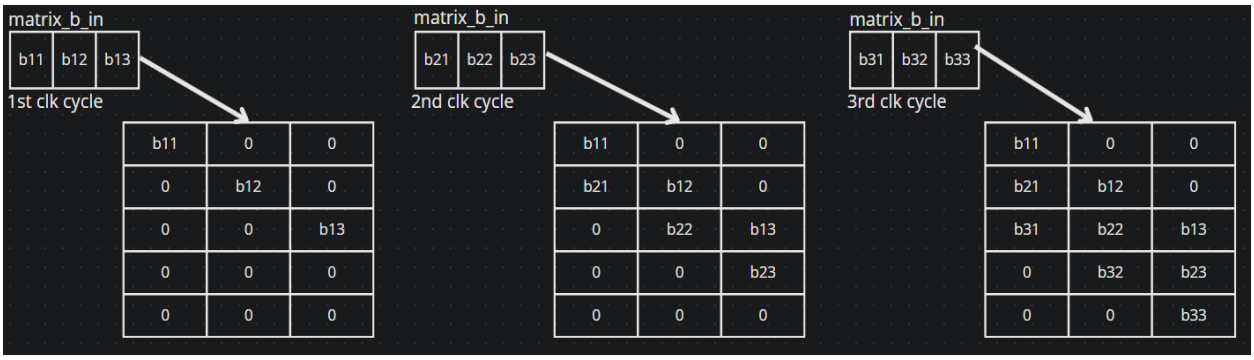

Figure 2: input matrix_B method

Same as matrix A method but it sends raw instead of column.

2. Architecture

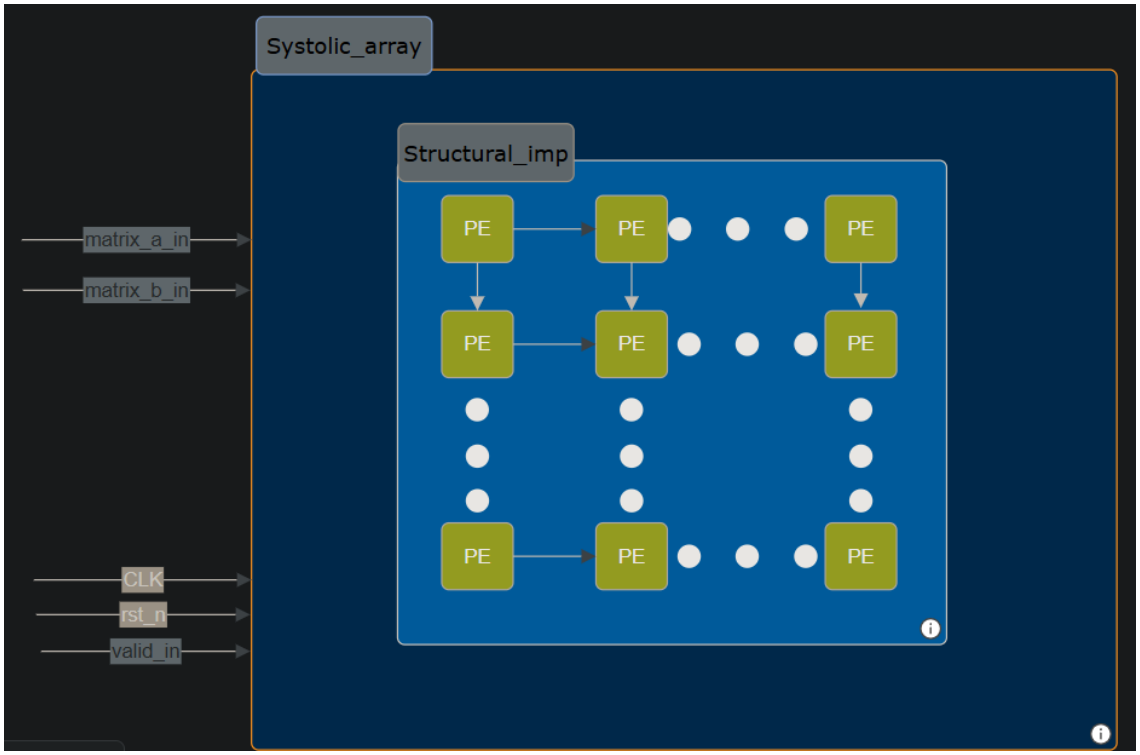

Figure 3: Design architecture

The design consists of three main modules:

- PE (processing element) module
- Systolic_array module
- Systolic_array_top module

2.1 PE (processing element) module

This module is the building block of the design where we multiply and accumulate on the previous output

```
always_ff @(posedge clk, negedge rst_n) begin
    if (!rst_n) begin
        a_out <= 0;
        b_out <= 0;
        c <= 0;
    end
    else begin
        c <= c + (a * b); // multiply and accumulate
        a_out <= a;       // pass data to next PE
        b_out <= b;       // pass data to next PE
    end
end
```

Figure 4: PE code snapshot

2.2 Systolic_array module

In this module we connect the PE's assuming the input is perfectly processed and ready to feed into the PE element.

```
// used to pass the elements row wise and column wise
logic [DATAWIDTH-1:0] row_wire [0:N_SIZE][0:N_SIZE];
logic [DATAWIDTH-1:0] col_wire [0:N_SIZE][0:N_SIZE];

// feeding the matrices
genvar i;
generate
    for (i = 0; i < N_SIZE; i = i + 1) begin
        always_comb begin
            row_wire[i][0] = matrix_A[i];
            col_wire[0][i] = matrix_B[i];
        end
    end
endgenerate
```

Figure 5:Structural_imp snapshot

First, we put the column of A into the first column of row_wire , raw of B in the first row of col_wire.

```
genvar ii, jj;
generate
    for (ii = 0; ii < N_SIZE; ii = ii + 1) begin // row loop
        for (jj = 0; jj < N_SIZE; jj = jj + 1) begin // column loop
            PE #(.DATAWIDTH(DATAWIDTH)) pe_inst(
                .clk(clk),
                .rst_n(rst_n),
                .a(row_wire[ii][jj]),
                .b(col_wire[ii][jj]),
                .c(matrix_C[ii][jj]),
                .a_out(row_wire[ii][jj+1]),
                .b_out(col_wire[ii+1][jj])
            );
        end
    end
endgenerate
```

Figure 6::Structural_imp snapshot

Instantiation of the PE's using the row_wire and the col_wire as an internal connection.

2.3 systolic_array_top module

It is the top module where we take the input and arrange it to ensure proper sequence and timing to the PE to operate correctly.

As mentioned in the introduction section, the feed process is based on a skew array therefore this module consists of several blocks.

- Input data block (where we generate the skew array).
- Feed data block (where we send the typical data to the Structural_imp module).
- Control unit block (where we control the internal signals used and the counters).
- Output block (where we output matrix c and valid out).

2.3.1 input data block

This is where the input is arranged into the 2-D matrix diagonally.

```
integer i;
always_ff @(posedge clk, negedge rst_n) begin : INPUT_BLOCK
    if (!rst_n) begin
        full_matrix_a <= '{default: '{default: '0}};
        full_matrix_b <= '{default: '{default: '0}};
        input_count <= 0;
    end
    else if (valid_in && input_count < N_SIZE) begin
        for (i = 0; i < N_SIZE; i++) begin
            // Matrix A: Store column-wise input with proper skewing for rows
            full_matrix_a[i][input_count + i] <= matrix_a_in[i];
            // Matrix B: Store row-wise input with proper skewing for columns
            full_matrix_b[input_count + i][i] <= matrix_b_in[i];
        end
        input_count <= input_count + 1;
    end
end: INPUT_BLOCK
```

Figure 7: input block in top module

The use of input count is basically for assigning the data N-cycles and it determines also the number of shifts needed.

2.3.2 feed data block

This is where the data is fed into the structural_imp.

```
always_ff @(posedge clk, negedge rst_n) begin :FEED_to_PE_BLOCK
    if (!rst_n) begin
        a_feed_col <= '{default: '0};
        b_feed_row <= '{default: '0};
    end
    else if (computation_started && count_cycle < (2*N_SIZE)-1) begin
        for (int j = 0; j < N_SIZE; j++) begin
            a_feed_col[j] <= full_matrix_a[j][count_cycle];
            b_feed_row[j] <= full_matrix_b[count_cycle][j];
        end
    end
    else begin
        a_feed_col <= '{default: '0};
        b_feed_row <= '{default: '0};
    end
```

Figure 8: feed data block

In parallel with the input being in the data is fed immediately to the PE.

2.3.3 Control unit block

This is where the counter of the clock and internal signals are handled.

```
always_ff @(posedge clk, negedge rst_n) begin :CONTROL_UNIT_BLOCK
    if (!rst_n) begin
        computation_started <= 0;
        count_cycle <= 0;
        computation_done <= 0;
    end
    else begin
        // Start computation after all input data is received
        if (valid_in && !computation_started) begin
            computation_started <= 1;
            count_cycle <= 0;  // Start counting from 0
        end
        // Continue counting once started
        else if (computation_started) begin
            if (count_cycle < (2*N_SIZE)-1) begin
                count_cycle <= count_cycle + 1;
            end
            else if (!computation_done) begin
                computation_done <= 1;
            end
        end
    end
end :CONTROL_UNIT_BLOCK
```

Figure 9:Control unit block

Signal computation_start to trigger and initialize the counter to start count clocks from the moment the PE starts working.

Note that PE is working in parallel with the input feeding which improves efficiency.

Signal computation_done is used to trigger the end of the processing and to start outputting the result matrix.

2.3.4 output block

```
always_ff @(posedge clk, negedge rst_n) begin : OUTPUT_BLOCK
    if (!rst_n) begin
        valid_out <= 0;
        matrix_c_out <= '{default: '0};
        count_out <= 0;
    end
    else begin
        if (computation_done && count_out < N_SIZE) begin
            valid_out <= 1;
            for (k = 0; k < N_SIZE; k++) begin
                matrix_c_out[k] <= output_full_matrix[count_out][k];
            end
            count_out <= count_out + 1;
        end
        else if (count_out >= N_SIZE) begin
            valid_out <= 0;
        end
    end
end : OUTPUT_BLOCK
```

Figure 10: output block

After the end of the processing the module takes N cycle to output the rows of the result matrix.

3.Simulation results

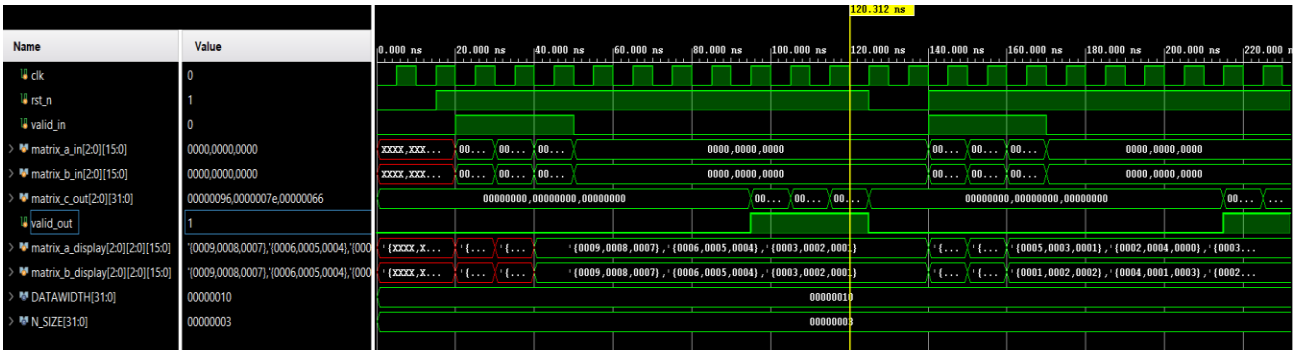

Figure 11: wave form

```
=== First Matrix Multiplication ===
Matrix A (fed column-wise):
1 2 3
4 5 6
7 8 9

Matrix B (fed row-wise):
1 2 3
4 5 6
7 8 9

Result Matrix C = A × B:
30 36 42
66 81 96
102 126 150
```

Figure 12: log file output

```
=== Second Matrix Multiplication ===
Matrix A (fed column-wise):
2 1 3
0 4 2
1 3 5

Matrix B (fed row-wise):
1 0 2
3 1 4
2 2 1

Result Matrix C = A × B:
11 7 11
16 8 18
20 13 19
$finish called at time : 265 ns : File "C:/STM_assesment/systolic_array_MUL/simu/systolic_array_tb.sv" Line 222
```

Figure 13: log file output