REPORT

DCS Final Project
Student: 111511076 陳彥宇

Design—FSM:

The following is the logic of my FSM. By putting “next_state” & “counter” together in the same always block, it becomes much easier to modify the timing of each operation to account for the pipelining delay. Also, it’s very simple for anyone reading the code to determine the duration of each state.

```
always_comb begin
    case(state)
        IDLE: begin
            next_state = (i_valid) ? INPUT_I : IDLE;
            counter_next = (i_valid) ? 0 : 10;
        end
        INPUT_I: begin
            next_state = (counter == 130) ? W_READY : INPUT_I;
            counter_next = (counter == 130) ? 0 : counter + 1 ;
        end
        W_READY: begin
            next_state = WAIT_W_VALID;
            counter_next = 0;
        end
        WAIT_W_VALID: begin
            next_state = (w_valid) ? INPUT_W : WAIT_W_VALID;
            counter_next = 0;
        end
        INPUT_W: begin
            next_state = (counter == 10) ? RAT: INPUT_W;
            counter_next = (counter == 10) ? 0 : counter + 1;
        end
        RAT: begin
            next_state = (counter == 7) ? CALC_MW : RAT;
            counter_next = (counter == 7) ? 0 : counter + 1 ;
        end
        CALC_MW: begin
            next_state = (counter == 10) ? OUTPUT : CALC_MW;
            counter_next = (counter == 10) ? 0 : counter + 1 ;
        end
        OUTPUT: begin
            next_state = (counter == 7) ? IDLE : OUTPUT;
            counter_next = (counter == 7) ? 0 : counter + 1 ;
        end
        default: begin
            next_state = state;
            counter_next = counter;
        end
    endcase
end
```

Fig1: State & Counter Transition Logic

Area & Timing Optimization:

➢ Parallel Operation

I realized that $\boldsymbol{I} \times \boldsymbol{I}^{\boldsymbol{T}}$ can be performed while we are still taking inputs for matrix $\boldsymbol{I}$. Since $\boldsymbol{II}^{\boldsymbol{T}}[0][0]$ is $\sum_{i=0}^{15} \boldsymbol{I}[\mathbf{0}][\mathbf{i}] \times \boldsymbol{I}[\mathbf{0}][\mathbf{i}]$, we can compute its value once the first row is completely filled, while still receiving inputs for the 2nd row. Once the 2nd row is filled

II^T[0][1], [1][0], [1][1] can be calculated, and so on…
The following is a visual representation of this process. Once the **1st row** is complete, the **red section** can be acquired. Once the **2nd row** is complete, we can compute the **orange section**, and so on.

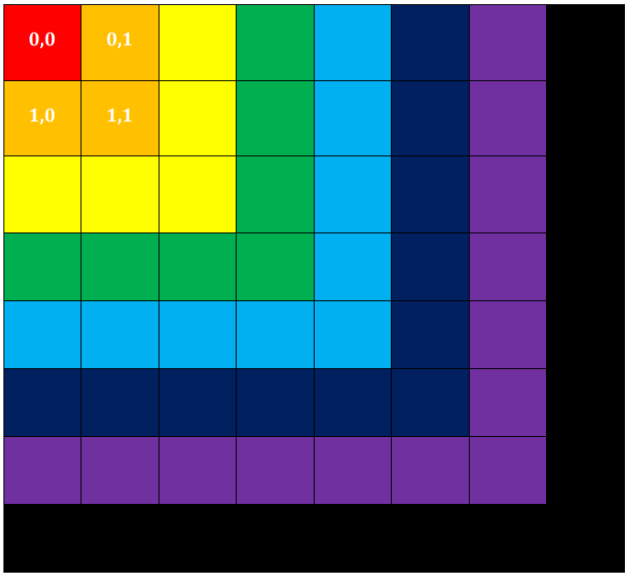

Fig2: II^T Computation Visualization

Furthermore, the result of a matrix multiplied with its transpose is always a symmetric matrix. This simplifies the operation. For example, II^T [2][3] & [3][2] have the same value, so I only need to perform one calculation. The terms [0~7][7] & [7][0~7] are calculated during the input state of the weight vector. Additionally, I am also summing the terms in each row of the matrix at the same time as we will need it later on during the RAT process.

➢ Division:

As for the RAT process, since there are 8 terms in each row, we can simply right shift the bits of the sums by 3.

➢ Pipeline:

In the first version of my code, I did not employ any pipeline and the minimum clock period is 9.1ns. After pipelining the multiplication process of II^T and $II^T \times W$, I'm able to reduce the the minimum clock period to 5.4ns. The pipelining is done by using a modified version of Lab 7's module. For II^T, I instantiated 16 of the same pipeline module since the 16 terms of one of I's row has to be multiplied with the 16 terms of one of I^T's columns.

```
always_ff@(posedge clk) begin
    if (in_valid) begin
        {A_a, A_b} <= in_1; //[7:4][3:0]
        {B_a, B_b} <= in_2; //[7:4][3:0]
    end
    else begin
        {A_a, A_b} <= 0; //[7:4][3:0]
        {B_a, B_b} <= 0; //[7:4][3:0]
    end
end
```

```
always_ff@(posedge clk) begin
    arr[0] <= A_a * B_a; //product(8, A_a, B_a);
    arr[1] <= A_a * B_b; //product(4, A_a, B_b);

    arr[2] <= A_b * B_a; //product(4, A_b, B_a);
    arr[3] <= A_b * B_b; //product(0, A_b, B_b);

end

always_ff@(posedge clk or negedge rst_n) begin
    if (!rst_n) out <= 0;
    else out <= (arr[0] << 8) + ((arr[1] + arr[2]) << 4) + arr[3];
end
endmodule
```

Fig3. Pipeline Module