Project 2: Systolic Matrix Multiplication

Jennifer Hellar, ELEC 522, Fall 2018

I. Design Approach & Tradeoffs

My design consists of 16 MAC blocks with data pipelined between them as shown in ***Figure 1*** below. To perform a matrix multiplication *C = A*B*, the data is loaded in the hardware blocks as follows:

- The four rows of *A* are loaded from the left, and the four columns of *B* are loaded from the top.
- Each row/column is delayed one cycle behind the previous.
- Each element of a row/column is loaded serially one cycle at a time into the appropriate MAC block.
- If *A*B* is a new matrix multiplication following a previous one, it can start being loaded 6 cycles after the previous one started.

Data output *C* is unloaded systolically to the left. Each MAC in a row passes its result to the left when it completes it, and the first MAC in the row outputs the results for the row serially to an output gateway. This unloads the data as quickly as possible without using individual gateways for each MAC, since the MAC on the left finishes first, then the second from the left, and so on.

My reset logic consists of a counter and a comparator block which outputs a pulse every 5 cycles, with a delay of 2 so that the first reset pulse starts propagating at cycle 7, when the first MAC finishes its computation. The pulse propagates right and down through the array, acquiring a delay of one along each diagonal. Those delays could have been generated outside of the array in a control system, which would have used fewer delay blocks than having a delay in each MAC. But a design like that would have made wiring and routing more challenging, so I chose the more systolic approach. In ***Figure 2*** below, you can see that each MAC propagates the RST signal to the nearby MACs.

Each MAC also propagates the input data along the rows and columns of the array as shown in ***Figure 2***, picking up a delay of 1 in each MAC. Like the RST signal, these connections could have been wired outside the MACs, but this approach allowed for carefully controlled data flow through the array.

For matrix-vector multiplication, I added an additional boolean input called *notvector*, which would only be false if a vector was being passed as the first column of *B*. This input was given to the first MAC in the array and then propagated down through the first column. If it was indeed a vector, then those MACs would not propagate the *A* matrix values to the right. This effectively left the remaining 12 MACs unused. The logic for this can be seen the the MAC schematic in ***Figure 3*** below.

II. Completed Model

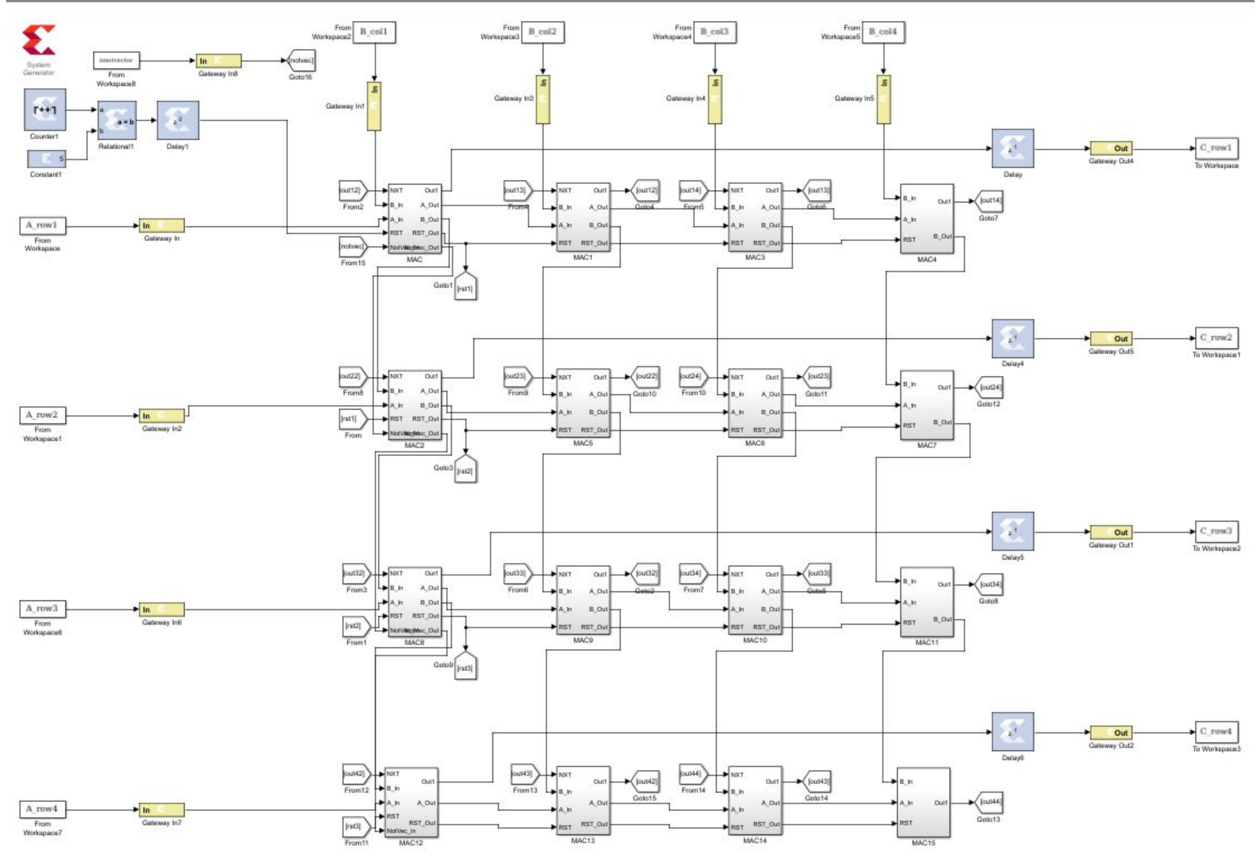

Figure 1: Completed Simulink Design

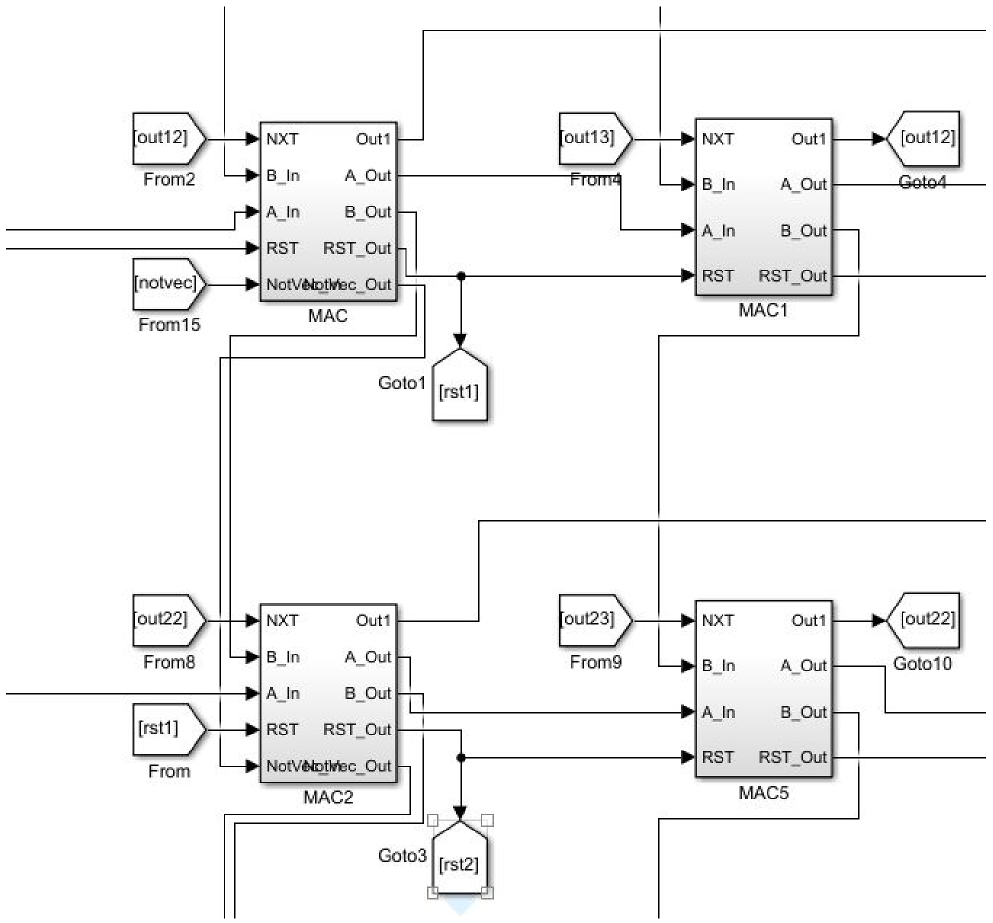

Figure 2: Upper left four MACs of the array

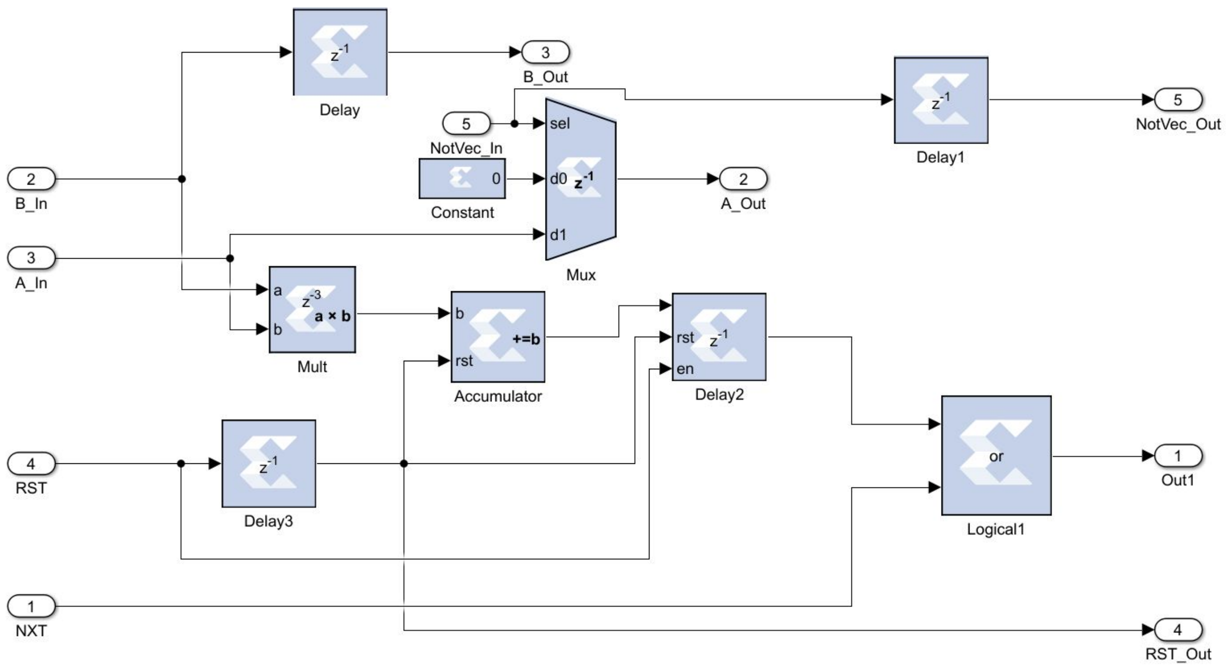

Figure 3: MAC Unit (from left column containing additional vector logic)

III. Resource Utilization & Timing

Resource utilization

Slices: 224
LUTs: 331
Slice Registers (all FFs): 988
IOBs: 194
DSP48s: 16

Timing

Maximum clock rate: 338 MHz
Clk: 100 MHz
Slack time (WNS): 7.047 ns

IV. Simulation & Testing Methodology

I used MATLAB code to preprocess and format the matrices for multiplication. Both ***load_matrix.m*** and ***format_inputs.m*** must be in the same folder as the .slx file. The latter function, ***format_inputs***, is a helper function. If you open ***load_matrix***, then you will find it is divided into three sections. The first two sections are different test scenarios, one with three matrix pairs to be multiplied, and the second with a matrix-vector multiplication in the middle between two regular matrix multiplies.

To test either scenario,

1. Run the section of MATLAB code to load the data into the workspace.
2. Run the simulation with a stop time of 29.
3. Go back to ***load_matrix*** and run the third section of code to format the output nicely.
4. View the variable ***raw***, which shows the results as they come out skewed in time as shown in **Table 1** below.
5. (Optional) View the other output variables (*output1, output2, output3*), which extract the individual matrix results and unskew them.

10	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**
4	4	4	4	0	0	10	10	10	10	0	0	4	8	16	32	0	0	0
0	4	4	4	4	0	0	26	26	26	26	0	0	4	8	16	32	0	0
0	0	4	4	4	4	0	0	10	10	10	10	0	0	4	8	16	32	0
0	0	0	4	4	4	4	0	0	26	26	26	26	0	0	4	8	16	32

Table 1: Raw simulation Output when three sets of matrices inputted

10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
4	4	4	4	0	0	16	0	0	0	0	0	8	8	8	8	0	0	0
0	4	4	4	4	0	0	16	0	0	0	0	0	8	8	8	8	0	0
0	0	4	4	4	4	0	0	16	0	0	0	0	0	8	8	8	8	0
0	0	0	4	4	4	4	0	0	16	0	0	0	0	0	8	8	8	8

Table 2: Raw simulation Output with matrix-vector multiply in the middle

V. Hardware Cosimulation

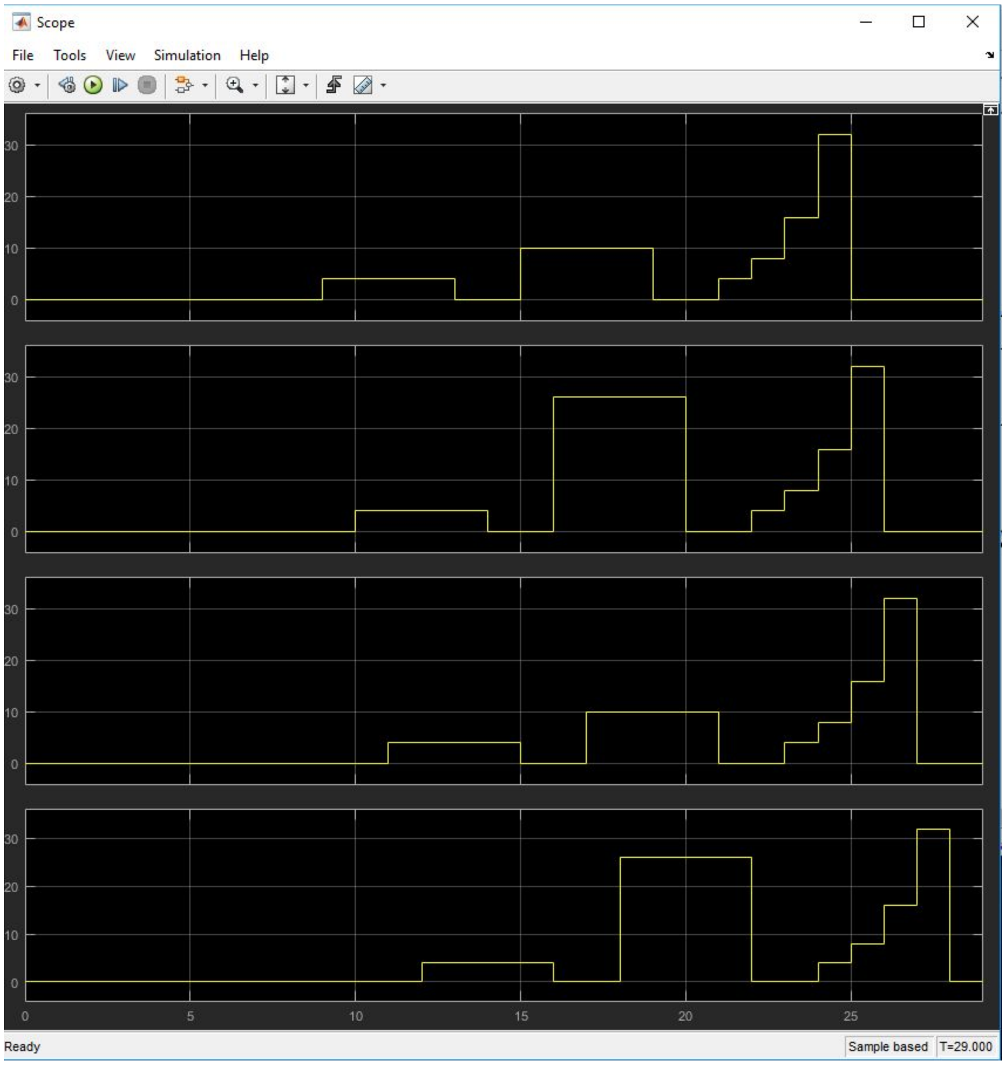

Figure 4: Hardware cosim results for three matrix multiplies (each scope is an output row)

The hardware cosim results were correct, and each matrix came out skewed as anticipated. The first matrix result completed in time 9-16. The second matrix completed in time 15-22. And the third matrix completed in time 21-28. The values, when compared with the simulation results in **Table 1** above, are correct.

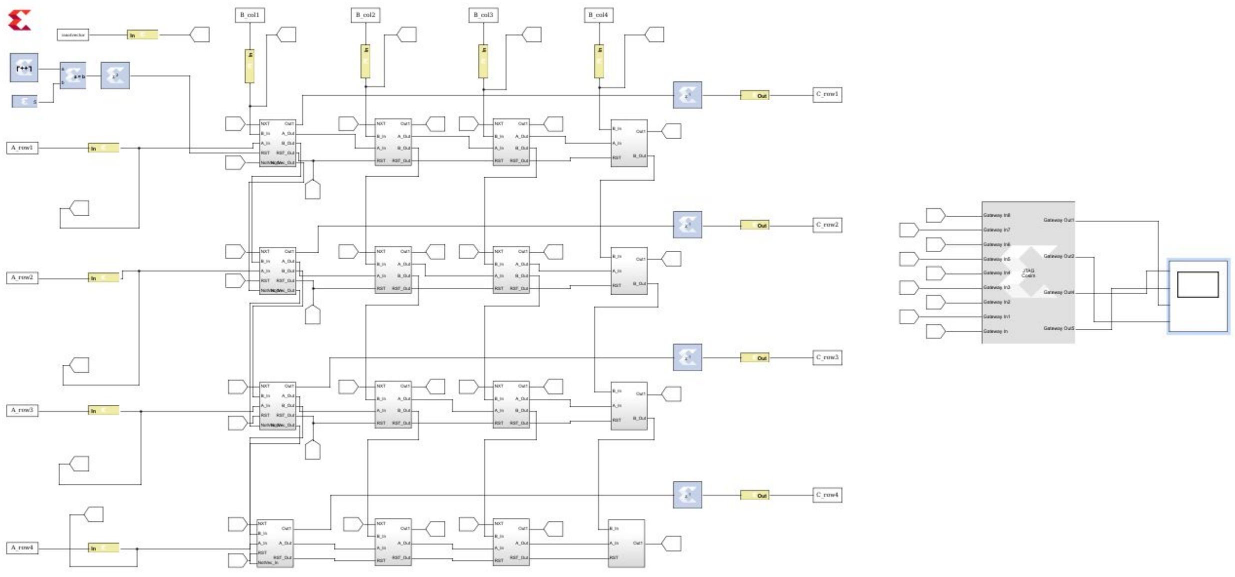

Figure 5: Hardware in-the-loop cosim diagram

<u>VI. Summary</u>

- Latencies:
  - Latency to complete first matrix multiply: 16 cycles
  - Latency to complete each matrix multiply after the first: 6 cycles
  - Delay between the start of each new matrix output on a given row: 2 cycles
- Scalability
  - The overall design will physically scale nicely, since there are no large blocks whose inputs would scale with the size of the matrix.
  - Scaling to an 8x8 or 16x16 matrix multiply would simply require more MACs.
  - Since this design used 194 IO pins out of the 200 available, however, we could not scale larger on this board.
  - Increasing the size of the matrices would also increase computation time and all of the latencies listed above.