ESE 507
Project 2: Matrix Vector Multiplication

Lars Folkerts and Xuan Chen

October 26th 2015

1.1 Questions

a) Multiplying a 3x3 matrix with a 1x 3 vector requires:

9 multiplications
+ 6 additions
15 total operations

Multiplying a k x k matrix with a 1xk vector requires

k^2 multiplications
+ $(k^2 - k)$ additions
$(2k^2 - k)$ total operations

b) The control module works based on the following chart on the next page.

c) The testbench module is straight forward. First we reset the system, then assert the start signal. Then we send random 8 bit inputs to data_in, one for each clokc cycle. These inputs include 9 inputs for the 3x3 matrix [lines 55-65] and 3 inputs for the 1x3 vector [lines 67-73]. We also print these values to the file *test_input.txt* as we receive them for additional hand verification.

Now that the inputs are generated, we can and do calculate the expected outputs directly in the testbench. These solutions are printed in *test_solution.txt* for hand verification. The simplefied solution code is:

```
for (j = 0; j < MAT_SCALE; j++) begin
    y[j] = 0;
    for (k = 0; k < MAT_SCALE; k++) begin
        y[j] = y[j] + a[j * MAT_SCALE + k] * x[k];
    end
end
```

Then we finally run the clock until our module is done with its calculation. We read the outputs from our module and compare this with out test solution. The outputs are reported in *test_output.txt* for hand verification. If the results are different, we report an error and abort. We continue this for a number of iterations, currently 1000, and if all the tests pass we report that our testing was successful and close.

d) View Table 1 for a comparison of power and area for different frequencies.

Max Frequency: 1/1.17ns = 854MHz
Critical Path: Memory Write
Startpoint: mat_mult_data_path/mem_a/data_out_reg[1]
Endpoint: mat_mult_data_path/mem_y/mem_reg[1][12]

e) It takes 25 time cycles to perform one operation. View Illustration 2 for details on the number of clock cycles per operation.

f) The area-delay product is displayed in Table 1.

g) Energy per operation = 1050.7uW * 1.17ns = 1229.3 fJ
Energy per 3x3 matrix multiplication = 1229.3 fJ * 27 ops = 33.2 pJ

h) Number of computational cycles = 9 multiply adds
From 1.1a, number of arithmatic operations = 15
Energy per arithmetic operation = 1229.3 fJ * 9/15 = 737.6 fJ

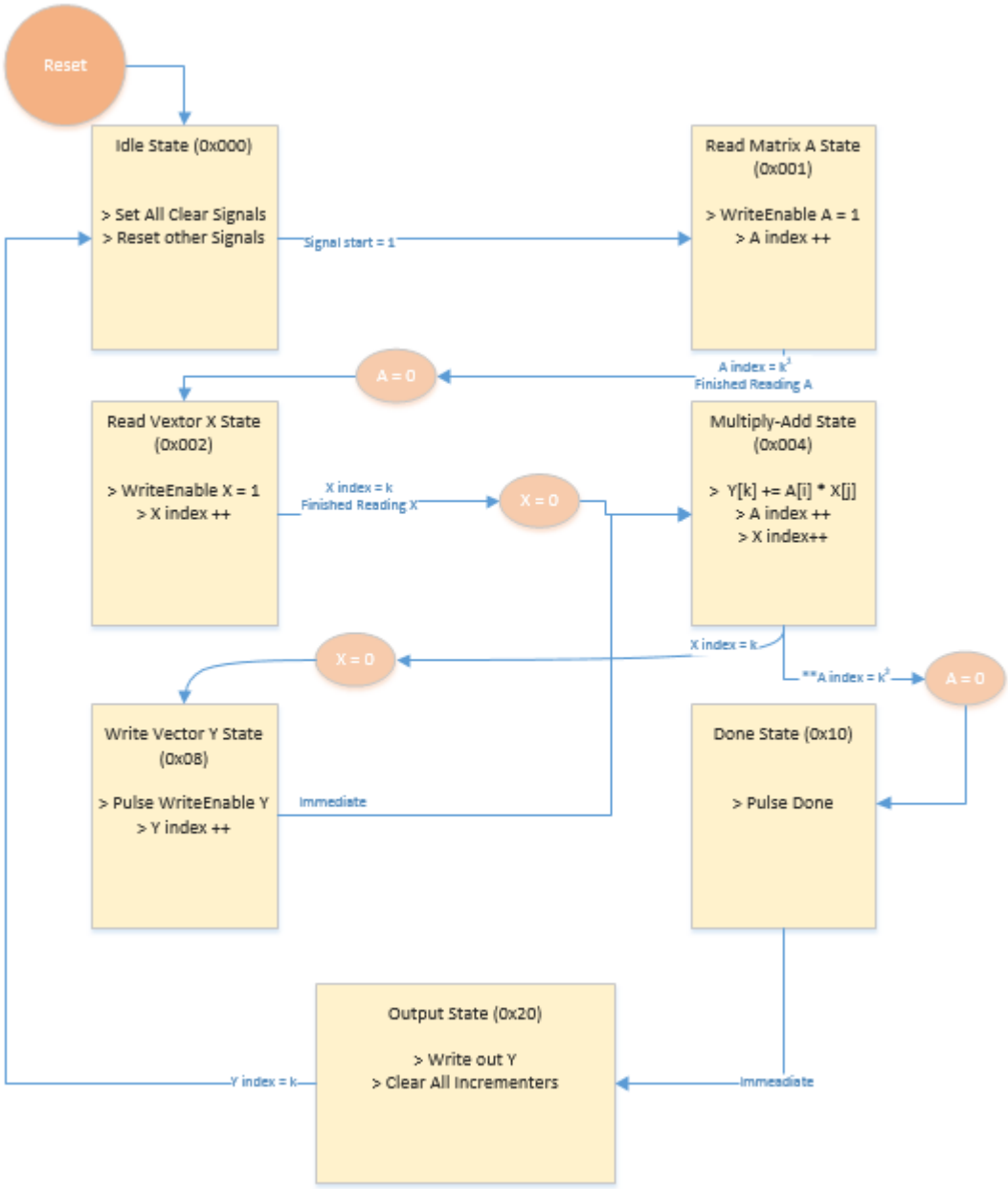

Illustration 1: Control Path Logic

*The above image illustrates our control path. The control path will stay in a state until the appropriate signal is set or condition is met. This can be an input from the overall system (i.e. start) or it can be a condition from the data path (e.g. X index = k). The ** symbol menas that this path take precedence. Also, Immediate means the control path switches from one state to another immediately, only staying in the state for 1 clock cycle.*

The operations in purple are completed by the datapath, but it helps to see what the overall system is doing base on the control path signals.

Made with MS Visio

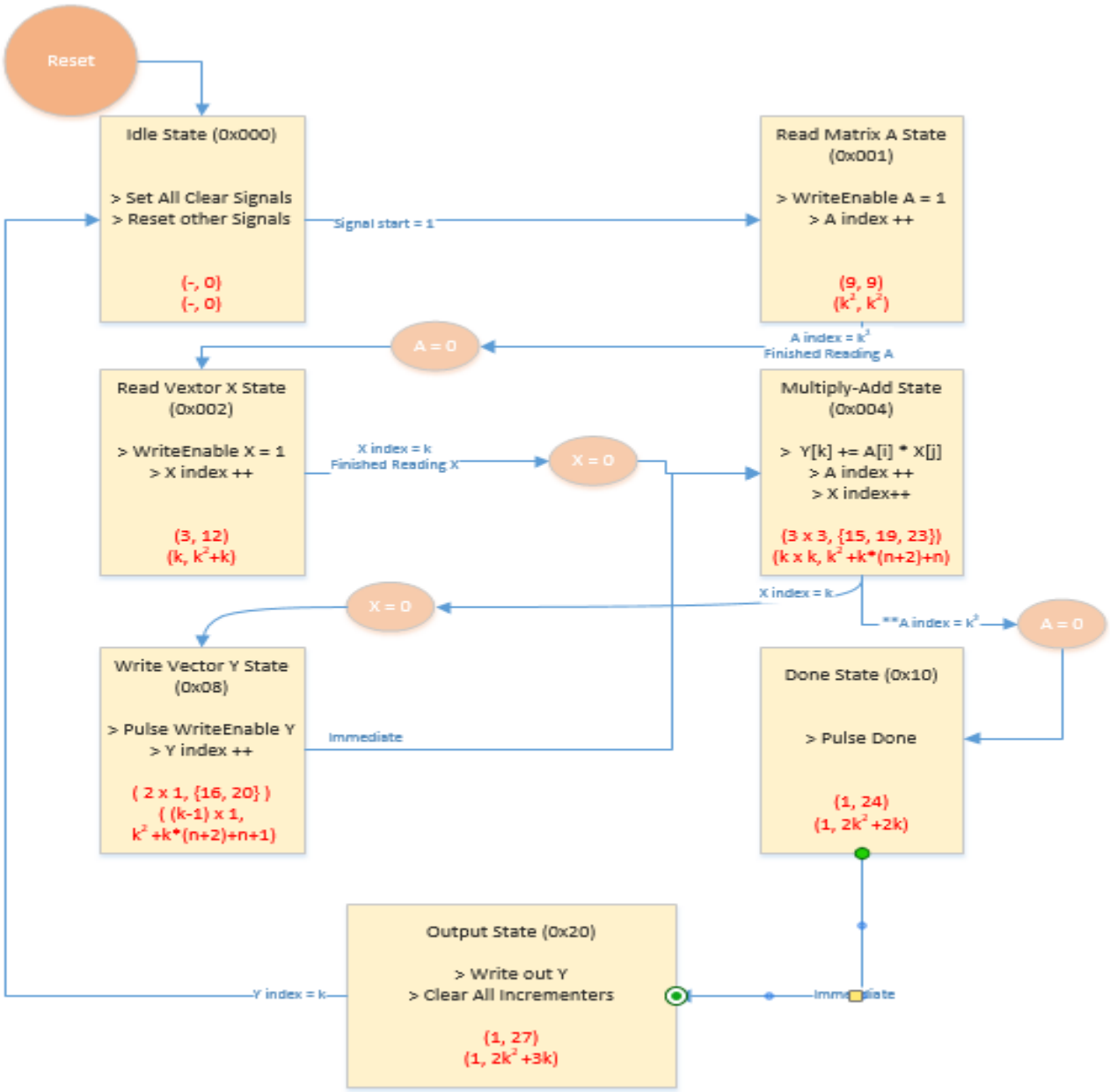

Illustration 2: Timing Diagram

The first number in the set represents the amount of time cycles T the control path stays in that state. For states that are reentrant, we show the N x T, where N is the number of times the state is reentered.
The second number in the set represents the time t at which the control path leaves that state. For states that are reentrant, we show {t_0, t_1, ...} for each time the control path leaves that state.
Below this numerical data, we generalized this data for multiplying a kxk matrix with a 1xk vector. We added the variable n to reentrant states to represent the number of times the user has entered the state. The the values for each t_n in the above {t_0, t_1, ...} sets can be calculated.
Made with MS Visio.

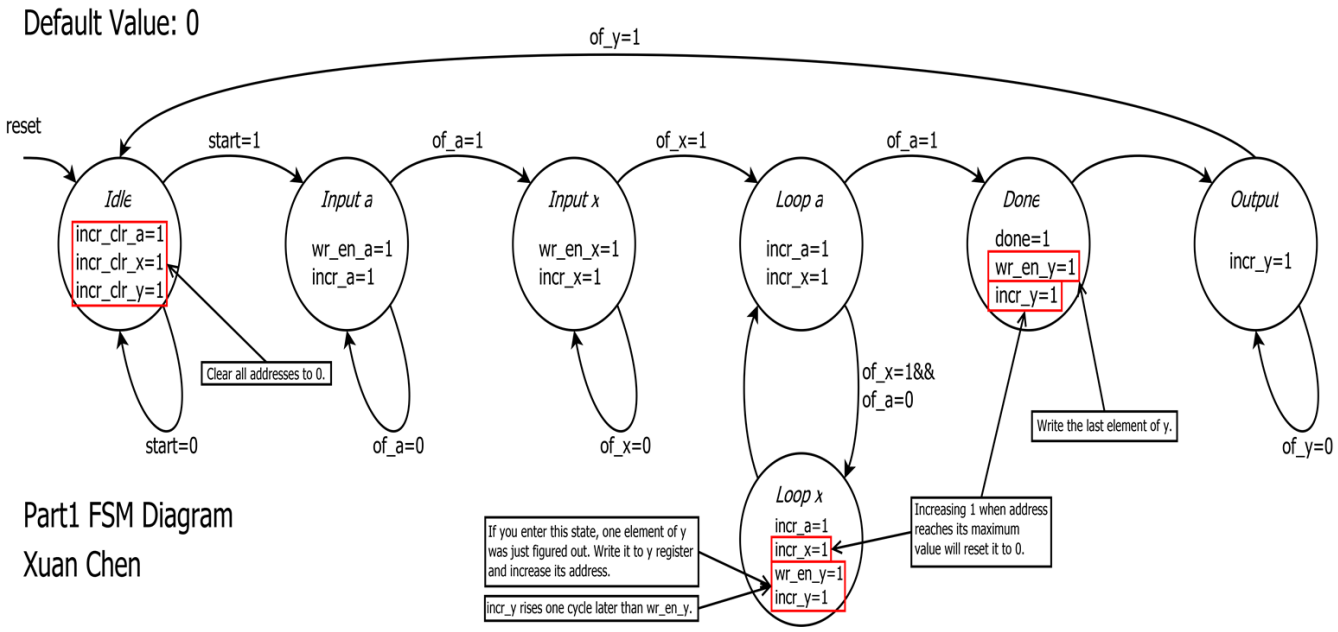

Illustration 3: Generated FSM with State Variables

Part 1	k=3								
Period (ns)	Comb. Area (um²)	Non-Comb. Area (um²)	Total Area (um²)	Area Delay Product	Total Dyn. Power (uW)	Cell Leak Power (uW)	Total Power (uW)	Is met?	Slack
1.1	996.4	981.0	1977.4	60903.9	1105.6	38.5	1144.1	-0.08	(VIOLATED)
1.11	1000.7	979.4	1980.1	61541.5	1096.4	38.8	1135.2	-0.07	(VIOLATED)
1.12	1003.1	981.0	1984.1	62221.4	1088.9	38.8	1127.7	-0.05	(VIOLATED)
1.13	994.6	985.3	1979.8	62640.9	1080.2	38.9	1119.1	-0.05	(VIOLATED)
1.14	993.5	978.3	1971.9	62943.0	1069.5	38.4	1107.9	-0.03	(VIOLATED)
1.15	996.4	978.6	1975.0	63595.0	1065.5	38.3	1103.8	-0.02	(VIOLATED)
1.16	985.8	978.3	1964.1	63794.0	1057.5	38.0	1095.5	-0.02	(VIOLATED)
1.17	1000.2	977.8	1978.0	64799.3	1050.7	38.7	1089.4	0	(MET)
1.2	973.8	976.8	1950.6	65540.2	1019.1	37.7	1056.8	0	(MET)
Part 2	k=4								
Period (ns)	Comb. Area (um²)	Non-Comb. Area (um²)	Total Area (um²)	Area Delay Product	Total Dyn. Power (uW)	Cell Leak Power (uW)	Total Power (uW)	Is met?	Slack
1	1436.4	1350.7	2787.1	122634.5	1648.9	58.2	1707.1	-0.16	(VIOLATED)
1.1	1405.3	1357.1	2762.4	133700.6	1495.0	58.0	1553.0	-0.05	(VIOLATED)
1.11	1417.2	1352.9	2770.1	135292.9	1484.6	58.1	1542.7	-0.06	(VIOLATED)
1.12	1414.6	1356.6	2771.2	136564.1	1470.4	58.0	1528.4	-0.05	(VIOLATED)
1.13	1412.7	1351.8	2764.5	137452.8	1453.4	57.9	1511.3	-0.03	(VIOLATED)
1.14	1416.7	1350.7	2767.5	138816.0	1443.6	58.2	1501.8	0	(MET)
1.2	1419.1	1339.0	2758.2	145630.5	1357.6	58.0	1415.6	0	(MET)

*Table 1: Area and power over a range of tested frequencies. There are differences between the two circuits. This is slightly unexpected. It is likely that the circuitry for k=4 adds an extra bit. For example, for k = 3, the incrementers would overflow at k=00**11** or k^2=**1001**; for k=4, this becomes k=0**100** or k^2=**1 0000.***

1.2 Questions

Since our module is generic, all we needed to do was change parameter MAT_SCALE from 3 to 4 for multiplying bigger matrices.

d) View Table 1 for a comparison of power and area for different frequencies.

Max Frequency: 1/1.14ns = 877.2MHz

Critical Path: Memory write

Startpoint: mat_mult_data_path/mem_a/data_out_reg[1]

Endpoint: mat_mult_data_path/mem_y/mem_reg[1][12]

e) Based on Illustration 2, we can plug in for k=4

$2k^2+3k = 2(4)^2+3(4) = 44$ operations

f) The area-delay product is displayed in Table 1.

g) Energy per operation = 1443.6uW * 1.14ns = 1645.7 fJ

Energy per 4x4 matrix multiplication = 1645.7 fJ * 44 ops = 72.4 pJ

h) Number of computational cycles = 16 multiply adds

From 1.1a, k=4, number of arithmetic operations = 16 multiplies + 12 adds = 28 ops

Energy per arithmetic operation = 1645.7 fJ * 16/28 = 940.4 fJ

1.3 Questions

a) We did several things to boost the speed. First, we divided our FSM into 2 FSMs - one for input and one for computation and output. With only one input port and k^2+k inputs, data input provides a bottleneck to our system. By making the input operations a stage in our pipeline, we can improve throughput. Thus while we are computing the output and writing the output to memory, we can be queuing up the next round of inputs.

Another this we did is parallelized the computation circuit, as shown in Illustrations 4 and 6. By doing this, we cut down our computation time from k^2 to k. Furthermore, we computed the multiplication is parallel, leading to a $\log_2(k)+1$ time complexity for the computation. For the output we kept the module the same as before, as all the outputs are ready at the same time for our parallel computation.

Finally, we pipelined our computation circuit to an absolute maximum with a 6 stage multiplier. This allowed us to improve the frequency. This increased our computational complexity from $\log_2(k)+1$ to $\log_2(k)+6$ for all of the different stages of our pipeline.

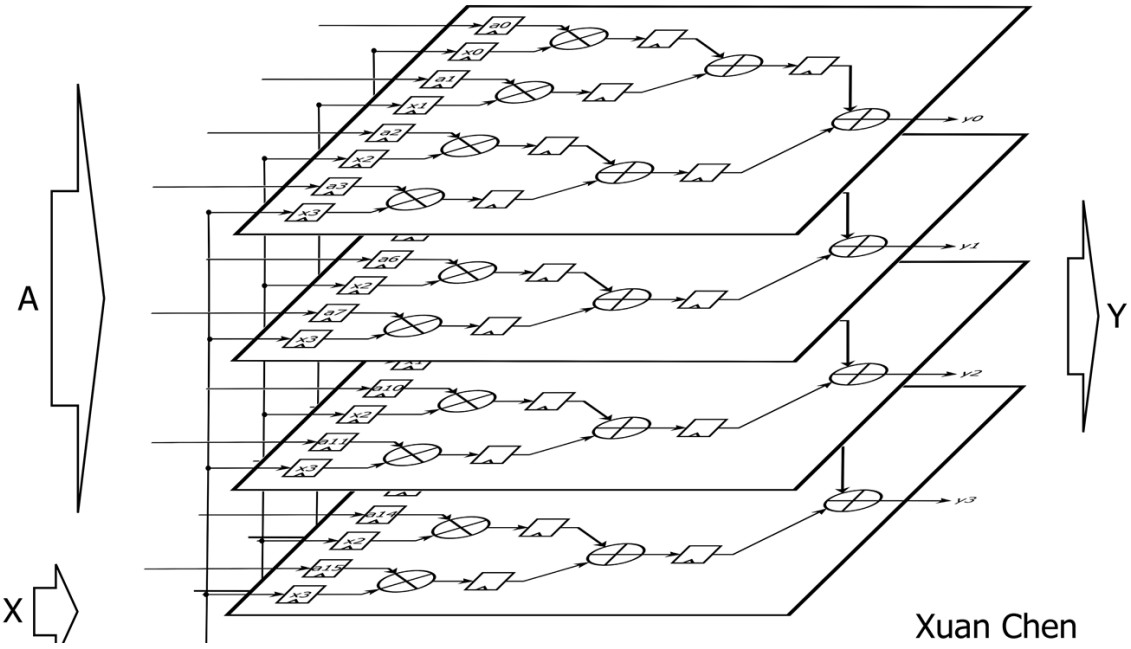

Illustration 4: Part 3 Datapath

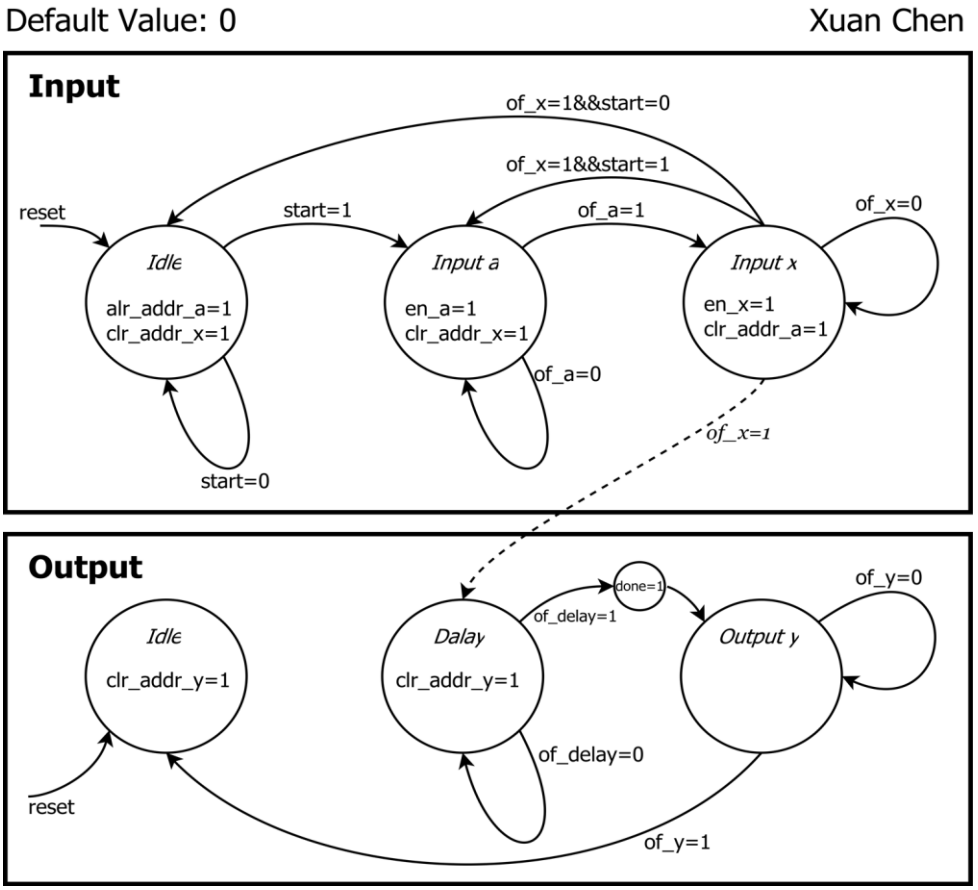

Illustration 5: Part 3 FSM

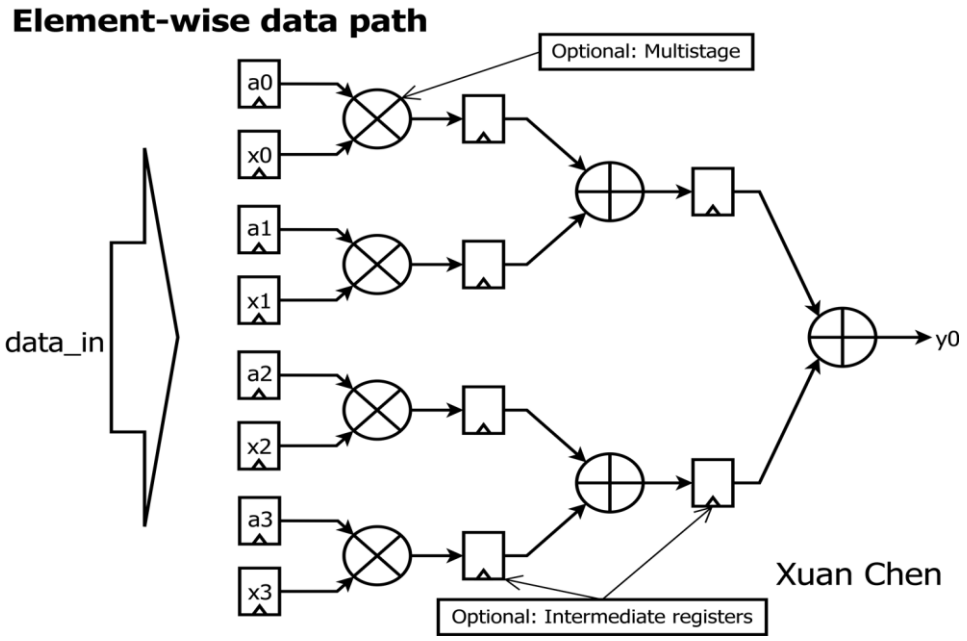

Illustration 6

One additional improvement we can add is to start the computation pipeline as the inputs are ready. That is, we can load in all indexes of i in A_{ij} and $X_{j.}$ can be loaded into our module first. We then can start our multiply-accumulate with the parallel $A_{ij}*X_j$ while we load in the next j index. This would require the addition of more control hardware and states and would not improve throughput for large values of k, due to the k^2+k input bottle neck. However, we can reduce latency from 8k to 8 with this implementation, since the computation would be completed at the same time as we are receiving inputs. The additional overhead might slow down the max frequency as well, which would result in a lower throughput, but still an improved latency for large k values.

b.d) This information is located in Table 2

Max Frequency = 1/0.47ns = 2.13 GHz

Critical Path: Adder

Startpont: datapath/genblk1[2].element/add/genblk2.next_layer/add_in_reg[0][11]

Endpoint: datapath/genblk1[2].element/add/add_in_reg[1][1]

b.e) The number of operations was implied in 1.3a. Our system has the following timing

Input: k^2+k

Calculation: 8k

Output: k

This gives us a latency of $(k^2+k)+(8k)+(k) = k^2+10k$. Our throughput is determined by the input bottle neck and is equal to k^2+k, or the computation/output 9k for $k<8$. For k = 4 the numbers are 46 cycles for latency and 36 cycles for throughput. View Table 3 for a latency and throughput comparison of parts 2 and 3.

b.f) The area delay product is found in Table 2. Here we used latency (=36) times area, since latency and throughput differ.

b.g) Energy per operation = 35.65mW * 0.48ns = 17.1 pJ

Energy per 4x4 matrix multiplication = 17.1 pJ * 46 cycles = 786.6 pJ

b.h) Number of computational cycles = 4 multiply adds (time 4 in parallel)

From 1.1a, k=4, number of arithmetic operations = 16 multiplies + 12 adds = 28 ops

Energy per arithmetic operation = 17.1 pJ * 4/28 = 2.44 pJ

Part 3	k=4	Piplined							
Period (ns)	Comb. Area (um^2)	Non-Comb. Area (um^2)	Total Area (um^2)	Area Delay Product	Total Dyn. Power (uW)	Cell Leak Power (uW)	Total Power (uW)	Is met?	Slack
0.2	7843.0	14804.2	22647.2	163060.1	74586.0	437.8	75023.8	-0.24	(VIOLATED)
0.3	7850.2	14763.0	22613.2	244222.5	49782.3	437.2	50219.5	-0.14	(VIOLATED)
0.4	7811.9	14830.6	22642.5	326051.3	37622.7	440.6	38063.3	-0.04	(VIOLATED)
0.41	7828.4	14817.8	22646.2	334257.5	36731.9	439.6	37171.5	-0.03	(VIOLATED)
0.42	7739.3	14580.5	22319.8	337475.3	34179.5	431.2	34610.7	-0.02	(VIOLATED)
0.43	7747.0	13966.1	21713.0	336118.0	31894.2	421.1	32315.3	-0.01	(VIOLATED)
0.44	7518.0	13488.1	21006.0	332735.3	30123.0	407.0	30530.0	-0.02	(VIOLATED)
0.45	7520.9	13406.9	20927.8	339030.6	29183.3	405.5	29588.8	-0.01	(VIOLATED)
0.46	7479.4	13177.4	20656.8	342076.0	27864.9	400.8	28265.7	-0.01	(VIOLATED)
0.47	7497.2	13140.1	20637.3	349183.9	27319.6	401.1	27720.7	increas	(VIOLATED:
0.48	7459.4	12718.5	20178.0	348675.2	25256.5	392.5	25649.0	0	(MET)
0.5	7547.0	12277.5	19824.4	356840.1	24117.9	388.7	24506.6	0	(MET)

Table 2: This is the table for part 3.

c) The comparison of the different multiply - accumulate architectures is shown on the right. The design for part 3 takes over 10x as much area and consumes 23 times as much power as the multiplier in part 2. The speed benefits gained are minimal, with the new module improving throughput by about 3 fold. The faster design would probably be worse in the system, due to these relatively poor metrics.

Design	Part 2	Part 3
Min Period (ns)	1.14	0.48
Max Frequency (MHz)	877.193	2083.33
Latency (ns)	50.16	17.28
Throughput (ns)	50.16	22.08
Area (uW^2)	2767	20178
Area Delay Product (uW^2*ns)	138816	348675
Total Power (mW)	1.50	25.65
Energy per Multiply (pJ)	0.94	2.44

Table 3: Comparison of Multiply-Accumulator Architectures

d) The best thing we could do to minimize energy per operation is to have fewer components. Our designs for parts 1 and 2 accomplished this pretty well, with only one multiplier, one adder and 32 bits worth of flip flops (2 8-bit inputs, 1 16-bit output). We can also optimize energy per operation by lowering the amount of memory we are using at a given time. That is do something like fetch A_{00} and X_0 and send them trough our multiply-accumulate path. Then next operation (or even pipelined) we can fetch A_{01} and X_1 and send them through our multiply-accumulate path. Our code for parts 1 and 2 of this project simply stored the entire matrix A and vector X throughout the operation, which caused power to be dissipated as our circuit held onto the signals (likely in registers).

Lowering the maximum frequency would also lower the amount of energy per operation. By

giving the synthesis tool some slack in frequency, it can pick lower energy components. This can be seen in Table 1, as an example; as we increased the clock period from 1.17ns to 1.2ns, our total power dropped by 32uW. There is also likely some threshold where the synthesis tool picks a slower but more energy efficient adder and multiplier architectures (e.g. carry lookahead adder to ripple-carry adder), and we should see the amount of energy per matrix-vector multiplication drop significantly at this inflection point.

e. Given we have k^2+k inputs and one input, we are constrained to this as an lower bound for our performance. This is compared to other parameters such as with k outputs and k^2 operations. Thus our maximum throughput would be $1/((k^2 + k)*f)$. If we add in one cycle to compute the value given the last input, and one cycle for output, we have a latency of k^2+k+2.

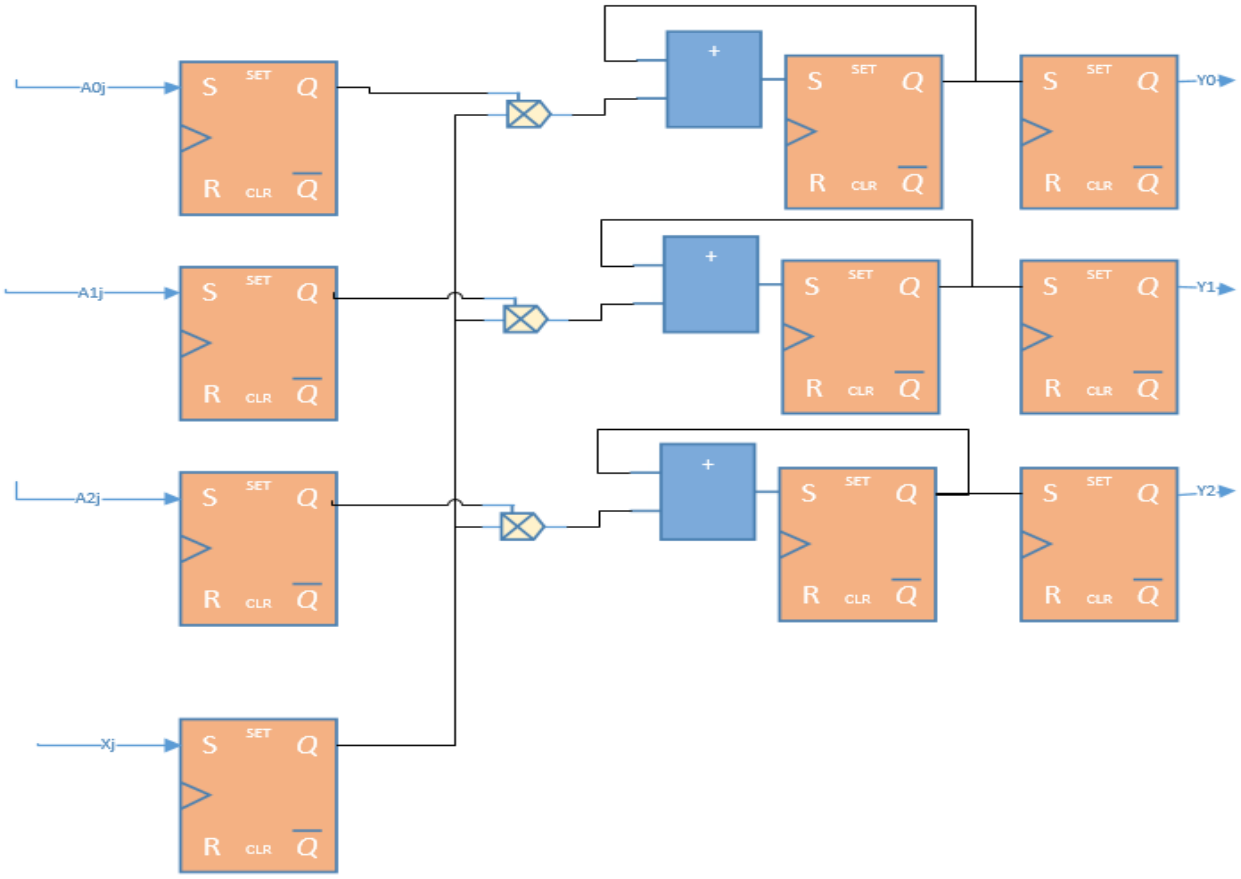

Illustration 7: Improved multiply-add with multiple input ports. Made with MS Visio.

Lets say we were able to get all the inputs in one time cycle. Then we could run the k multiply-accumulate operations in parallel, resulting in k cycles of multiply-accumulate. Then we can output all of the cycles in one time period. This gives us k+2 operations.

For this architecture k+1 input ports and k output ports are needed. For the input port, all indexes of i in A_{ij} can be loaded in parallel with $X_{j.}$ Then during the next cycle when these

values are sent through the multiply accumulators, we can increment j and load in all indexes of i in A_{ij} and X_j. Since all of the outputs will be ready at the same time, when j = k-1, then we can output all of these k outputs during that one time period.