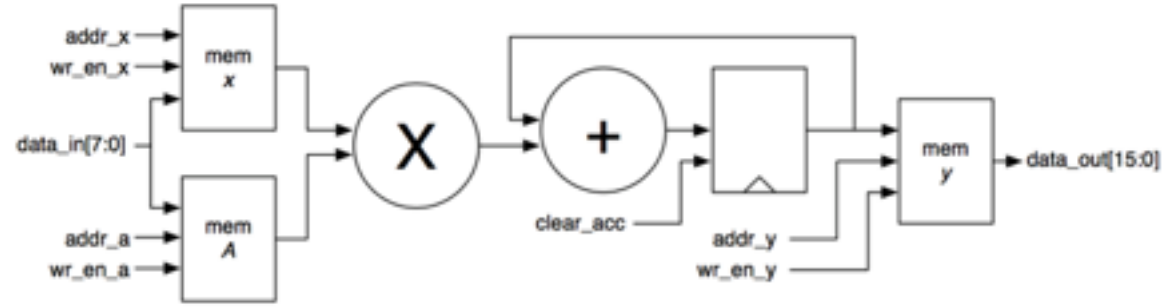
Project 2 - Report

Matrix Vector Multiplier



$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} a_0 & a_1 & a_2 \\ a_3 & a_4 & a_5 \\ a_6 & a_7 & a_8 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} a_0x_0 + a_1x_1 + a_2x_2 \\ a_3x_0 + a_4x_1 + a_5x_2 \\ a_6x_0 + a_7x_1 + a_8x_2 \end{bmatrix}$$

Project by

Rishikanth Chandrasekaran & Gunjan Shrivastava

110284413 110615473

Part 1

a. By pen and paper, multiplying a 3 x 3 Matrix and a vector of size 3 will take 9 multiplication operations and 6 additions. Thus it will require a total of 15 arithmetic operations.

Generalizing this, a k x k Matrix will require k^2 multiplications and 2k additions, giving a total of k(k+2) arithmetic operations.

b. The control module is implemented using a Finite State Machine (FSM). The FSM consists of 5 discrete states, with each state performing specific functions. The transition from one state to other is triggered by flag signals which indicate the completion of the previous states.
The 5 states are listed down below:

State 0 : Idle / Initiation state – all counters and flags are set to default values – waits in this state till start signal – Comes to this state when reset is 1
State 1 : Input state (X) – The values for vector X are written into memory
State 2 : Input state (A) – The values for Matrix A are written into memory
State 3 : MAC state – The matrix and vector are fed into the MAC and the results are written into the output memory Y
State 4 : Display state – The output values are sent out on data_out from the memory Y. The done signal is also generated before the values are sent.

The state flow diagram of the Finite State Machine is shown by the figure below.

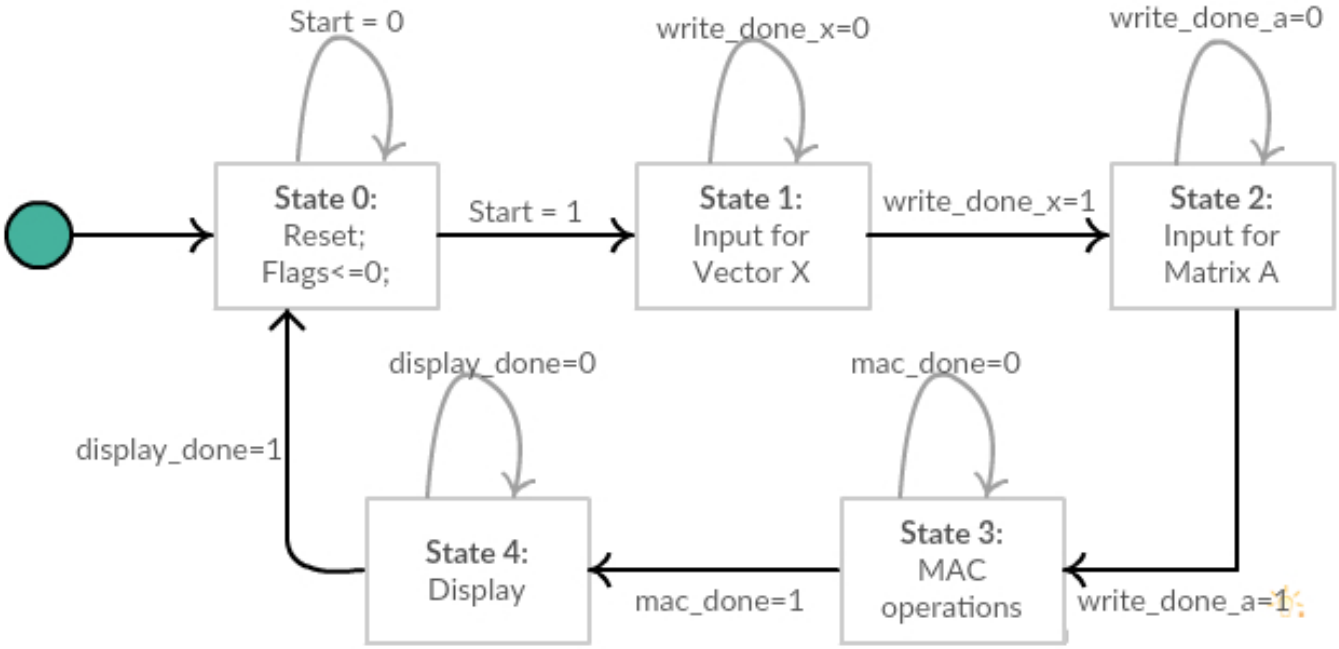


Figure 1: State Transition Diagram of FSM

The control path module makes use of counters and flag signals to generate the addresses for the read and write operations of the memories of A, X and Y. It also synchronizes the write enable signals for the memories. It effectively controls and directs the data path module.

In **State 0**, it waits for the start signal.
In **State 1**, it generates addresses and write enable signals for memory X to store the Vector X in constant sync with the clock edge. Once the input for the vector has been stored, it **raises** the **write_done_x** flag indicating the end of this state.
In **State 2**, it generates addresses and write enable signals for memory A to store the Matrix A in constant sync with the clock edge. Once the input for the vector has been stored, it **raises** the **write_done_a** flag indicating the end of this state.
In **State 3**, it feeds the row wise the matrix and vector elements into the MAC, and after the end of calculation for one row, the answer is written to memory Y, for which addresses and write enable signals are generated at the correct instant. After all calculations are over, it **raises** the **mac_done** flag which signals the end of this state.
In **State 4**, the **done** signal is **raised** indicating that the results are ready to be

displayed. The clock cycle immediately after done is raised the outputs stored in memory Y are pushed out on the data_out line. After all values are displayed, the **display_done** flag is **raised** which brings the FSM back to state 0.

At any instant of time the control module knows precisely in which state the system is operating in by monitoring the flag signals and the previous states. Since each flag signal uniquely represents the operation of each state, the control module is able to keep track of the states.

c. Verification Strategy

The verification strategy consists of 2 parts. A C-program which generates the test data, and a System Verilog Testbench module which instantiates the main module implemented.

The C-Program randomly generates a 3 x 3 matrix and 3 x 1 Vector where each element can be positive or negative. The program has a #define COUNT parameter at the top, which the user can specify the number of such matrix vector test pairs to be generated for testing. The default value is 10.

The program generates the specified number of test cases and writes it to a file called “inputData”. It also calculates the output for those values and writes them to a file called “expectedOutput”.

The Verilog testbench code reads from the file and runs the top level module for those values. The testbench runs the design against the input values and generates output values. The testbench then prints the output values and also prints the expectedOutput values.

The testbench also triggers the reset signal to test the functioning of reset signals. It also at certain iterations doesn’t trigger start signal to test the system.

The testbench thus effectively tests the system for a big set of random inputs, also tests functionality of reset and start signals.

d. Part 1 – Synthesis Analysis

Total Cell Area : 2036.761968 um^2
Power : Total Power - 1.3234 mW
Dynamic Power - 1.2824 mW
Leakage Power - 41.0074 uW

Clock Period : 1.1097 ns
Frequency : 0.9012 Ghz
Critical Path : Critical path is from Memory X which stores the Vector to the output flip flop of the MAC module (f)
Startpoint: d/x/data_out_reg[1]
Endpoint : d/m/f_reg[14]

e. The system takes **28 clock** cycles for 1 matrix-vector multiplication.
Delay of the system : 28 clocks * 1.1097ns = **31.0716ns**

f. Area: 2036.761968 um^2
Delay: 31.0716ns
Area-Delay Product: 2036.761968 x 31.0716 = **63.286 m^2ps**

g. Energy per matrix-vector multiplication : Power x Time x No of clocks
= 1.3234 mW x 31.0716 nS x 28
= **1.151 pW**

h. Operation count of the system: 18
Energy per Operation: Energy per matrix-vector multiplication / Operation Count
= 1.151 pW **/** 18
= 0.0639 pW

Part 2

a. **Changing Design from 3 x 3 to 4 x4**
   The changing of the control module to compute 4 x 4 was straightforward and simple. It mainly involved increasing the size of memory, counter limits and the address lines. The state diagrams doesnot change. Hence there is no complexity involved in changing the design to accept 4x4 inputs.

b. **Changing Design for k x k**
   Specifically for my implementation, the following changes are sufficient to scale the system to k x k :

   1. Increasing size of memory of matrix a and vector x
   2. Increasing the length or number of bits for address lines of memories a and according to k
   3. In my design I use 2 counters called *multiplier* and *counter* for generating the address values for read and write access. The limits of the counter will be (k-1) & k respectively (the condition checked for).
   4. Increase the width or bits for multiplier and counter if required depending on k.

   My implementation is reasonably scalable and does not require any complicated changes, as the state diagram remains the same and maps to the same discrete steps involved. Thus only the number of iterations increase and hence the corresponding limits.

c. **Part 2 – Synthesis Analysis**

   Total Cell Area : 2826.781944 um^2
   Power : Total Power : 1.7655 mW
   Dynamic Power : 1.7063 mW

Leakage Power : 59.2544 uW

Clock Period : 1.1097 ns
Frequency : 0.9012 Ghz
Critical Path : Critical path is from memory A to output register of MAC (f)
Startpoint: d/a/data_out_reg[1]
Endpoint: d/m/f_reg[14]

d. The system takes **44 clock** cycles for 1 matrix-vector multiplication.
Delay of the system : 44 clocks * 1.1097ns = **48.8268ns**

e. **Area:** 2826.781944 um^2
Delay: 48.8268ns
Area-Delay Product: 2826.781944 x 48.8268 = **138.023 m^2ps**

f. **Energy per matrix-vector multiplication** : Power x Time x No of clocks
= 1.7655 mW x 48.8268ns x 44
= **3.793 pW**

g. **Operation count of the system:** 18
Energy per Operation: Energy per matrix-vector multiplication / Operation Count
= 3.793 pW **/** 32
= 0.1185 pW

Part 3

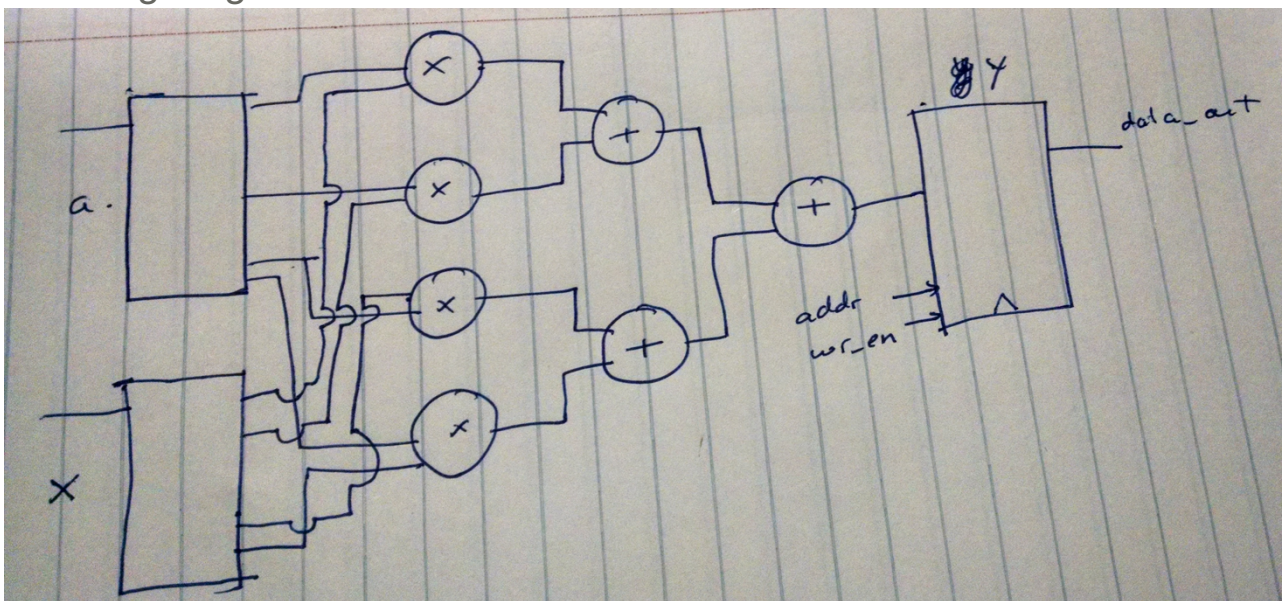
a. **Delay Optimization**

In order to boost speed the following 5 implementations were implemented:

1. Parallelism
2. Parallelism with Pipelining
3. Pipelining – Design Ware components
4. Overlapping input and output data
5. Parallelism with Pipelining and Overlapping input and output data

Parallelism:

Parallelism was achieved by reconstructing the system according to the following diagram:



The implementation of this system provided less than **2%** improvement in performance and hence was not effective.

Parallelism with Pipelining:

In this design in addition to implementing the parallelism architecture

pipelining was introduced. Since the critical path is large, pipelining the multiplier stages would provide gains. Hence the multipliers were pipelined using Design Ware components. This provided some improvement providing with **15%** improvement in speed.

Parallelism with Pipelining & Overlapping of Input & Output:
In this implementation, the system is parallelized and implemented with the above architecture along with pipelining between the adder and multiplier stages. In addition to this, the system is designed to accept new inputs when it is printing the output. This significantly increases the performance of the system to **52%**.

Pipelining :
This design involved pipelining by introducing a flip-flop between the multiplier and adder. This did not provide any significant performance gains. It produced an insignificant **0.6%** improvement.
However when the multiplier stage was pipelined using Design Ware components the performance improved significantly.
The performance improved to **20.3%** when a 2 stage pipelined multiplier was used, using design ware component.

Overlapping Input and Output Data:
In this procedure, the Input Data was fed as soon as the done signal was raised as opposed to waiting for all the outputs to be sent out. This effectively saves clock cycles. It does not provide any large performance improvements. The performance improvement was **2**

b. Synthesis Results for Delay-Optimized Versions

Design	Area (um^2)	Power (mW)	Frequency (Ghz)	Critical Path
Pipelining-Design Ware	2713.4659	2.9508	1.235	Piped multiplier to pipeline fp before adder
Overlapping I/O	2828.6439	1.6636	0.9	A Reg to mac op reg(f)
Parallelism-Pipelining	5430.3899	1.6836	0.68	Piped multiplier to output memory(y)
Parallelism-Pipelining – I/O Overlap	5709.4239	2.9875	1.19	A reg to piped multiplier

c. Delay of System

Design	Clocks	Time Period (ns)	Delay (ns)
Pipelining-Design Ware	48	0.81	38.88
Overlapping I/O	43	1.11	47.73
Parallelism-Pipelining	28	1.48	41.44
Parallelism-Pipelining-I/O Overlap	**28**	**0.84**	**23.52**

d. Area Delay Product

Design	Area-Delay Product (m^2ps)
Pipelining-Design Ware	114.73
Overlapping I/O	135.011
Parallelism-Pipelining	225.035
Parallelism-Pipelining-I/O Overlap	134.285

e. Energy per matrix-vector multiplication:

Design	Energy per matrix-vector multiplication (pW)
Pipelining-Design Ware	105.499
Overlapping I/O	79.4
Parallelism-Pipelining	69.77
Parallelism-Pipelining-I/O Overlap	70.27

f. Operation count & Energy-per-operation of the system

Design	Operation Count	Energy per Operation (pW)
Pipelining-Design Ware	32	3.3
Overlapping I/O	32	2.48
Parallelism-Pipelining	28	2.49
Parallelism-Pipelining-I/O Overlap	28	2.51

g. Optimizing Energy-Per-Operation instead of Delay

By mathematical relation Energy-per-operation can be optimized by any of the following ways:

1. Increasing no. of operations
2. Decreasing Power consumed
3. Decreasing no. of clock cycles required
4. Decreasing clock period

However decreasing clock period, will increase power consumption. Similary increasing number of operations will either lead to more power consumption or more device which in turn will lead to more power consumption.

Thus decreasing the no of clock cycles required, while finding a balance with clock period and no of devices would be an ideal way to optimize energy per operation.
The parallelism architecture implemented above could also be a solution.

h. Optimizing with input and output ports

If the port and timing specifications could be changed, then one intuitive thing to do would be to read and display multiple values at the same time. This will allow us to reduce the delay of the system and more efficient use of clocks.