# ESE507 Report

## Goal

The goal of this project was to create and test a System Verilog hardware generator for matrix vector multiplication. Input parameters to this hardware generation tool include:

b - the number of bits per input datafield

k - the k-by-k matrix and k-vector sizes

g - the degree of pipelining in the design

p - the degree of parallelism in the design

## Introduction[[1]](#footnote-1)

Hardware generation tools are useful to generate code specific to your application. The more options the hardware generation tool designer gives to the user means more engineering hours for the designer and possibly lower performance of the tool.

In our design, we simplified our generation to contain only the following options:

**b - the number of bits per input datafield**

This field must be a positive integer, but has no upper bound. With the synthesis tool in our lab, this design is doable up to 64 bit inputs, 128 bit outputs.

**k - the k-by-k matrix and k-vector sizes**

This field must be a integer greater than 3, but also has no upper bound. We had trouble synthesizing values where b=4, k=64 for both g and p values with very long synthesis times.

**g - the degree of pipelining in the design**

This value can be equal to 0 for no intermediate pipeline, or 1 for a 2-stage pipleline. The 2 stage pipeline has a flip-flop in between the multiply and accumulate stages of matrix multiplication.

**p - the degree of parallelism in the design**

The value of p can be 1 for no parallelism, or equal to k to have all of the vector indexes multiply-accumulate in parallel.

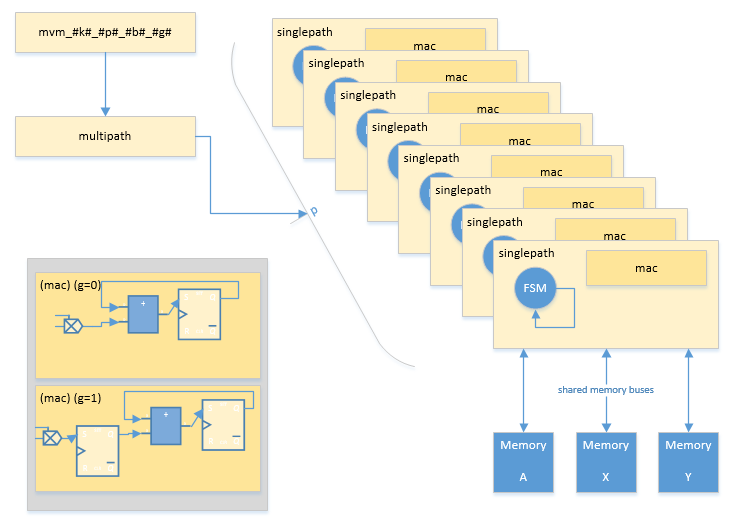
While we do not enforce practical limits on them, but issue warnings to the user when we think they are unfeasible. Certainly with a different synthesis library, but same generated code, these values can become feasible, so it is hard to determine exact boundaries. It is best left to the knowledge of the user to figure out - the best we can do is issue our warnings.

## Software Implementation

Hardware Generator Tool Implementation1

In order to implement our design, we simply placed tags in our main module and testbench files to indicate where the parameter values should get inserted. These tags were of the nature #b#, #k#, #g# and #p#. Our C code replaced these tags in a straightforward manner.

Parameters #b# and #k# were already linked into our code from our previous project. Tags #g# and #p# will be discussed in our the following System Verilog description.

System Verilog Design Implementation [[2]](#footnote-2),[[3]](#footnote-3)

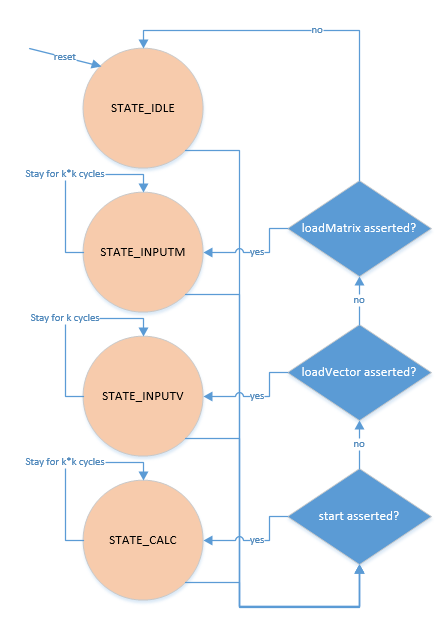
The System Verilog has a top level wrapper function with the appropriate name mvm\_#k#\_#p#\_#b#\_#g#. It passes its main parameters to a module named multipath.

The multipath module was developed in order to deal with the parallelism parameter p. It creates p singlepath hardware modules for us. In addition, it calculates the parameter CALC\_CYCLE, which is the number of operations needed to perform one matrix-vector multiplication. Finally, lets the user know we are done CALC\_CYCLE cycles after start is asserted. The *start* and *reset* signals are also utilized to help with this.

Moving deeper into the code, singlepath contains the datapath and control for the system. Our FSM has four states: STATE\_IDLE, STATE\_INPUTM, STATE\_INPUTV and STATE\_CALC. In our code the variable names a\_ is used for the matrix, variable names x\_ is used for the vector, and y\_ are used for the output.

STATE\_IDLE js the self-explanatory state where the hardware is not performing a task.

STATE\_INPUTM is the state in which our hardware reads the matrix inputs from data\_in. We enter this state when the *loadMatrix* signal is asserted. After k2 cycles when the matrix is fully loaded, the signal decides what state to go to next. If the *start* signal is asserted, the next state will be STATE\_CALC. Else if the *loadVector* signal is asserted, it will go to STATE\_INPUTV. Else if the *loadMatrix* signal is asserted, it will go to STATE\_INPUTM. Otherwise, it will return to STATE\_IDLE. Note that if the *loadVector* and *start* signals are ignored during the k2 cycles.



STATE\_INPUTV follow the same description as STATE\_INPUTM except that it reads the vector inputs for k cycles.

STATE\_CALC follows he same logic as STATE\_INPUTM and it runs for k2 cycles.

One final thing to note about the FSM is that *start* takes precedence over *loadVector* which takes precedence over *loadMatrix*.

The datapath implementation if straightforward. Firstly, we have the memories for matrix A, input vector X and output vector Y. These memory locations are shared between all the singlepath modules via time multiplexing. For inputs X and A, it doesn't matter what they output when read signal is not asserted. However, if the system is paralleled, multiple outputs from different data path must be connected together at the end. They are set to high impedance by time division so that they can share the output bus. While this eases the implementation code, it uses tri-state buffers which can be expensive in our system.

Another solution the multiple singlepath problem is placing a MUX to select one path at a time to output. One of the advantages of this solution is that a MUX can avoid any potential conflict rather than a time-sharing bus if the timing control logic is not properly designed. Fortunately, it seems unlikely to happen in the current code implementation.

The mac module takes in parameter g for it pipelining. Here there are 2 different sets of code, one for pipelining and one without. The pipelining code simply adds in a flip flop in between the multiplier and the adder. This is a logically easy place for the flip-flop, but it does not equally divide the timing since multiplication takes significantly longer than addition, especially as parameter b increases.

The singlepath module must also keep track of g and delay the output to memory location y when it is asserted, as it will take an extra cycle to compute the output.

Testbench Implementation [[4]](#footnote-4)

The testbench is logically straightforward. A random signal is asserted for a clock cycle – that is either *loadMatrix, loadVector*, or *start*.

For *loadMatrix* and *loadVector* the signal is asserted as random inputs are generated and fed into the data\_in line. This is k inputs for *loadVector* and k2inputs for *loadMatrix*.

If *start* is asserted, the testbench will wait until *done* is asserted for the module to simulate its output, then it will read its output back to the user. These sample outputs are verified by the testbench itself and are also outputted to a file for hand verification.

## Results & Discussion

We wrote a bash script, Run.bash that generated a lot of data for us. We discuss what we found interesting from the data.

5. In Section 3, we discussed how the parameters (k, p, b, g) allow various tradeoffs between problem size, costs, precision, and performance. Explain how these tradeoffs work at a conceptual level. In other words, explain how you would expect changing each parameter to affect these metrics. Be specific. Now, you will use your generator to produce several designs, synthesize them, and evaluate their cost and performance as the parameters change. For each of the designs you will be synthesizing below, turn in your .sv file and the Synopsys output log file (use the same naming convention but with extension .txt). Each time you synthesize, aim for the highest reachable clock frequency. As we have previously discussed, it is very important to carefully understand your synthesis reports. If there are any errors listed at all, then it means your entire design did not synthesize correctly. This can be caused by things like missing files or typos as well as serious design problems. If any error is shown, you must correct it and re-synthesize. Warnings can also be problematic. Some (“signed to unsigned conversion”) may not matter, but others (“inferred latches” or “unresolved references”) are very big problems. If your synthesis report shows either of these, make sure to correct the problem and re-synthesize.

6. Now, we will use synthesis to evaluate how the area and power of an implementation scale as the input precision b changes. Use your generator to produce four designs with b=8, 12, 16, and 20, while you keep k=8, p=8, and g=1. Then produce two graphs that illustrate: (1) power versus b and (2) area versus b for these designs. Describe where the critical path is located in each design.

7. Next, we will evaluate how throughput, area, and power scale as k changes. Use your generator to produce four designs with k=4, 8, 16, and 32, while you keep p=1, b=8, and g=1. Synthesize each design, and graph: (1) power versus k, (2) area versus k, and (3) throughput versus k. Does the location of the critical path change as k changes? Throughput is defined as the number of data inputs processed per second. To calculate this, you will first determine the average number of data words per cycle, and multiply this by the clock frequency f. We will calculate the words per cycle assuming that our system keeps a fixed matrix A. (That is, assume that a matrix is stored in memory and that we will use loadVector followed by start for every input.) Under these assumptions, for every MVM computed, your system processes k input words. How many cycles does this computation take? Let c represent the number of cycles needed to process these k input words. Then, your throughput will be (k/c) words per cycle times f cycles per second. In your report, include the values of c that you found for each design, and explain how you found them.

8. In the previous step, you used pipelined designs (g=1). How do the costs and performance change if you use unpipelined designs? Generate and synthesize designs without pipelining using parameters (k, p, b, g) = (4, 1, 8, 0) and (32, 1, 8, 0) and compare these two designs to their pipelined counterparts from question 7. How does the critical path change?

9. Lastly, you need to evaluate how the designs change when you increase parallelism (by setting p=k). Now, generate and synthesize four parallel designs: k=4, 8, 16, and 32, with p=k, b=8, and g=1. Graph: (1) power versus k, (2) area versus k, and (3) throughput versus k. These parallel designs will be faster but more expensive than their counterparts from question 7. Which are more efficient: the more-parallel p=k designs, or the lessparallel p=1 designs? Justify your answer quantitatively

1. Part of Question 1 is answered here [↑](#footnote-ref-1)
2. Question 2 is answered here [↑](#footnote-ref-2)
3. Question 3 is answered here [↑](#footnote-ref-3)
4. Question 4 is answered here [↑](#footnote-ref-4)