# 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 greater than 3, but has no upper bound. In order to move past 32 bits, one of the key challenges was to generate random input longer than 32bits, which is accomplished by function LongRandom function. Another problem is the power operator (\*\*) function, which is used to calculate the value range (i.e. 2512). With inputs greater than 512 bits, it does not produce the correct output.

**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. Intermediate values can be chosen in our model, but are blocked by the generate code as described by this assignment.

While we do not enforce practical limits on the **b** and **k** maximum values, 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.

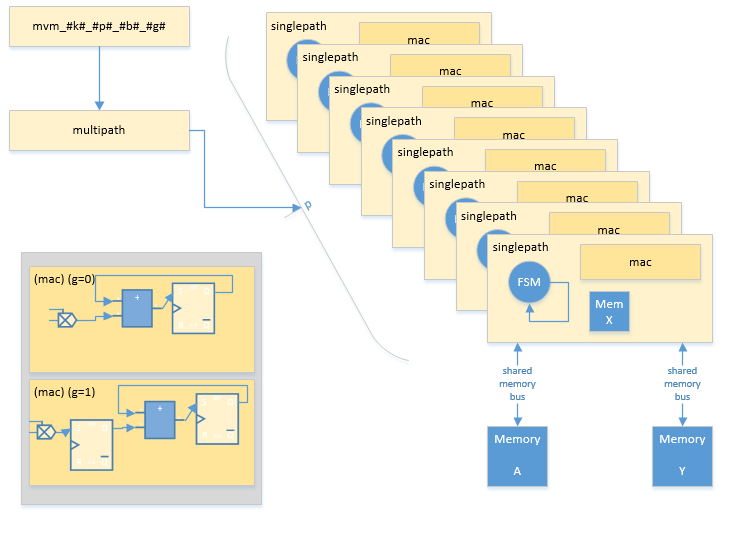
We do enforce the hard limits as discussed above (i.e. 3<**b**, 3<**k**, **g**={0,1}, **p**={1,**k**}). The generate script will not run within these limits.

## 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)

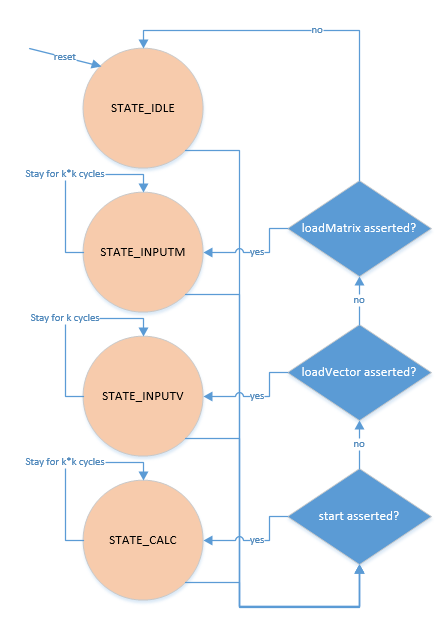
The System Verilog has a top level wrapper function with the appropriate name mvm\_#k#\_#p#\_#b#\_#g#. It passes its main parameters to a module named multipath.

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 *calculation* *cycles*. A calculation cycle is different than a clock cycle as we can have **p** parallel units. Thus a clock cycle contains **p** calculation cycles as data is read on the memory bus.

Thus when **p**=1, we have k2/1 clock cycles in STATE\_CALC.

When **p**=**k**, we spend **k**2/**k**=**k** clock cycles in STATE\_CALC.

Our system is unique in that **p** can be any value, not just 1 or **k**. Changes to the generate file are needed for this as we restrict the user to the values of 1 and k for the purposes of this assignment.

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 input 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.

The X memory in our design is not shared, and each singlepath has its own memory for X. This is a flaw in our design that leads to increased area when p>1.

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.

Testing Script

A script called Run.bash was written in src/ to synthesize our modules. This script does not look at logical correctness and only grabs certain information from modules. The script is also not robust, as it was only written for it-only-needs-to-work-this-one-time purpose. Thus it is probably best only to look at instead of attempting to run it yourself. Information is in the README if you are interested in running the script.

If you do wish to run the script, you must:

Create the directories **src/tmp/**, **src/tmp/out** and **report/**.

Run the setup script: **source ese507setup-csh**

Run **nohup bash Run.bash**

**nohup** is no-hangup, which will keep the script running even when ssh fails.

The start of each simulation the parameters b,k,g,p,T,# are printed the the screen, where T is the clock period (misnamed frequency f in the script) and # is the sate the script is in. You can also redirect the output if you want.

The script runs through all the following possible parameterizations:

**k =** {8,12,16,20}

**p =** {1, k}

**b =** {8,12,16,20}

**g** = {0,1}

It will start from a predetermined base period and increase the period by 0.1 ns until it synthesizes correctly.

Then it will decrease the clock period by 0.01 to get a more precise maximum frequency.

Finally it will gather several more data points at slower frequencies.

Note that the script takes about 26 hours to run.

## Results7

The results are stored in **report/Results.xlsx** .

The first tab, **RawData**, contains the data directly from the script output. Frequency, Latency, Throughput and Total Power were also caluclated and appended onto this worksheet, where:

*where T is the period in ns*

*Of a loadVector and calculation, ignore loadMatrix times*

Note that we define throughput as **k** input words as we are assuming a constant matrix with a loadVector and a Cal

The second tab is named **MaxFreq** and contains the fastest synthesized module for each of our 64 results. This is interesting for a quick lookup, but trends are hard to extrapolate from this data.

The final tab is named **Graphs** and it is the most interesting and is worthwile to open and play around with. In rows 2-5, you can change the parameters for (k,p,b,g). There are five graphs which will update based on the parameters entered in rows 2-5. Row 6 may also need to be updated based for x axis-values on PARAM graphs. The 5 graphs are:

Area vs. Throughput

Area vs PARAM (at max frequencies)

Throughput vs PARAM (at max frequencies)

Power vs Throughput

Power vs PARAM

where PARAM can be either **k**,**p**,**b**,**g**. The sheet is currently set up to compare 4 different sets of (**k**,**p**,**b**,**g**) parameters

The interesting results are discussed below.

## Discussion[[5]](#footnote-5),[[6]](#footnote-6),[[7]](#footnote-7),[[8]](#footnote-8),[[9]](#footnote-9)

Here is how parameterization should affect the circuit:

**b –** The number of bits should be directly proportional to area and power, as we would need more gates to perform operations. There may also be a slight exponential trend on **b** as the architecture will get more complex as you increase the logic size.

**k, p**  - Our hardware runs in O(**k2/p**) time with O(**p**) area and power. Thus by increasing **k**, we are lengthening the time of the calculation. By increasing p, we are indirectly minimizing the time, while directly increasing the amount of area and power our circuit uses.

**g –** By increasing g, we should see a small decrease in latency and an increase in throughput as the minimum viable clock period is reduced. The added flip flops to the pipeline will increase area. Power will increase slightly if we intend to always run our design at the maximum frequency. However, by running our processor at a constant frequency not dependent on **g, g** should have little effect on power consumption as the energy per operation should be roughly the same.

Starting on the next page, we can view selected actual results.

Parameter b

According to Graph A.1, we can see dramatic area increases for area as we increase **b** when **k**=8, **p**=8. **g**=0.. The added flip-flops and computation logic increase the area roughly linearly with b (Area≈**b\***2500). As expected, the area is not directly proportional to b and seems to increase slightly more than 2500 when b gets large.

Also included is graph A.1.1 where we plot the area for several different frequencies. This shows that area is generally not a function of throughput for the same design. Only by increasing the throughput via a hardware change will the area change.

The throughput graph A.2 is linear as by increasing b, we can increase the number of bits we process without dramatically decreasing the clock frequency. Note that for b=16 and for b=20, there is no improvement in throughput. This likely has to do with our synthesis missing a local minimum for b=16. As a comparison, we added graph A.2.1 when k=12, which we see is much more linear.

For power consumption in Graph A.3 the graph should take throughput into account as power is proportional to the frequency squared. For this graph, we can see that the trend is linear regardless of the number of bits. This is slightly skewed for b=8 probably due control logic overhead.

Since this trend is linear based on throughput, by plotting Power vs b, we would get a graph proportional to the throughput graph A.2. This is not useful to us as it really measures max throughput and not power. Power vs b as shown in Graph A.4 .

Describe where the critical path is located in each design.

Does the location of the critical path change as **k** changes?

Parameter k

Our next set of graphs (Graphs B.1 to B.4) removed parallelism, leading to a dramatic decrease in area. However, we still see a linear area increase. Since the actual multiply add unit is small, this has to do with the size of the memory and size of the counters (increasers) in the system.

Here we also see throughput decrease based on the values, although the shape of the trend is hard to view based on the number of data points provided. Given p=1, k>>g

Thus we should see an exponential decay of throughput. It is hard to tell from so few datapoints, but we can see a general decreasing trend.

In graphs B.3 and B.4 we see power increase with increases in k. This is due to the increased area of the circuit.

How does the critical path change?

Parameter g

Next we look at the effects of pipelining in our system.

The critical path in our system is through our multiplier when **g**=1 and through our entire multiply accumulate module when pipelining is **g**=0. However, note in graph C.2 we see only a slight increase in throughput with pipelining. This means that our pipeline is not directly in the middle of the critical path when **g**=0. In other words, the time for multiplying two **b** bit numbers is greater than the time to add two **b** bit numbers.

We can note that the area does not increase significantly when adding the pipeline. However, more dramatic effects are seen when **p**=**k**.

Graph C.3 we show two values of **b**. However, the impact of g has little do with the power consumption curve. It is still linear based on the throughput of the system.

Parameter p

Finally we wish to test the effect of parallelism in our system. With **p=k** we ran the simulations for different values of k.

Firstly, we see area increase linearly in D.1. Compared to graph B.1, the graph shows a similar change, but the area of our circuit is much greater.

Next we see throughput remains generally unchanged in our system. This is expected as we receive k output data points in k cycles of time, giving us k/k. Also note that the values on this graph are about 10 times higher than the throughput values in Graph B.2. The should, in theory, be k times greater, ignoring the loadVector time.

With increased k, we also see increased power as shown in D.3. Also, we can view the full data. It seems that for k=16 was unable to find a high-throughput local minimum for its data, which explains why this point is significantly lower on Graph D.2

# Conclusion[[10]](#footnote-10)

The hardware generator tool we created, albeit simple, was able to create some good basic multiply accumulate designs. When designing a system, one would likely have **b** and **k** predetermined, leaving the designer to choose between the **g** and **p** parameters. We determined that **g** leadsto a slight increase in area, and gives a slight throughput improvement. However, in terms of power consumption, power was a linear function of throughput, independent of the value of **g**. Thus if a designer does not mind the slight increase in area, adding the pipeline is beneficial so that she can run the module at higher frequencies if she wishes.

For the parameter **p**,the two designs of **p**=**1** and **p**=**k** are dramatically different in terms of area, power and throughput. The choice should be pretty obvious to the hardware designer based on the requirements of his system. Anything that needs to be mobile should probably use the **p**=1 architecture since it is lower power and area. Otherwise, one can jump to **k** parallelism to speed things up **k**-fold.

One thing that remained untested is the possibility of 1<**p**<**k** . This is possible in our architecture and might be useful to a hardware designer. More analysis should be done in this area.

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

   7Part of Question 7 (on throughput) is answered here [↑](#footnote-ref-4)
5. Question 5 is answered here [↑](#footnote-ref-5)
6. Question 6 is answered here [↑](#footnote-ref-6)
7. Part Question 7 is answered here [↑](#footnote-ref-7)
8. Question 8 is answered here [↑](#footnote-ref-8)
9. Part of Question 9 is answered here [↑](#footnote-ref-9)
10. Second half of Question 9 answered in Conclusion [↑](#footnote-ref-10)