Course: Advanced Digital System Design
Course Code: ESE507
Term: Fall2015
Topic: Project 3 – Hardware Generation Tool
Students Name and IDs:
Sanket Keni (110451820)
Sagar Ramesh Thakkar (110533717)

Q1. In hardware generators, scalability and flexibility can be difficult. In your report, explain how you made our generator capable of handling the flexibility required by the parameters (k, p, b, g). Were any of these particularly easy or difficult to support? How did you handle the fact that there were no maximum defined values on k and b? Are there practical limits on these values? If so, what are they, and why?

Solution1:

a) First of all we made mvm modules for 4 types:
   1) P=1 & g=0
   2) P=1 & g=1
   3) P=k & g=0
   4) P=k & g=1

   And wrote the code for each mvm module parameterizing 'k' and 'b' while keeping 'p' and 'g' constant.

b) Using 'b' as a parameter helps us to adjust the number of bits required for the data being used.

c) Parameterizing 'k' enables to adjust the number of times that the MAC operation should execute and also during pipelining stage, where we need to create 'k' memories to store a vector from matrix A.

d) Parameterizing the parallel case when p=k was harder than unparalleled case as number of states increased as 'k' increased.

e) As 'k' and 'b' increases, area and power both increase. As a result it will be practically difficult to use designs with large 'k' and 'b' to satisfy the design cost constraint p. This puts a direct limit on these two parameters.

f) Also increasing 'b' increases the time of critical path thereby taking more time to compute. As a result performance is reduced.

----

2) Describe how you designed your control module/FSM. How did you structure the design so that it can be changed as the k parameter changes? Explain how the structure changes as k grows.

Solution2:

    A) The Control module works with below State Diagram.

    1) Initially the system is in **State 0**. This is an idle state. The FSM switches to state 0 when it gets a 'reset=1' signal. The following control signals are set when we get reset=1 :

    addr_a=0;   addr_x=0;   addr_y=0;

    wr_en_a=0;   wr_en_x=0; wr_en_y=0;

    clear_acc=0;   done=0

The FSM will continue to remain in state 0 until it receives either start=1 or loadVector=1 or loadMatrix=1

FSM till state3
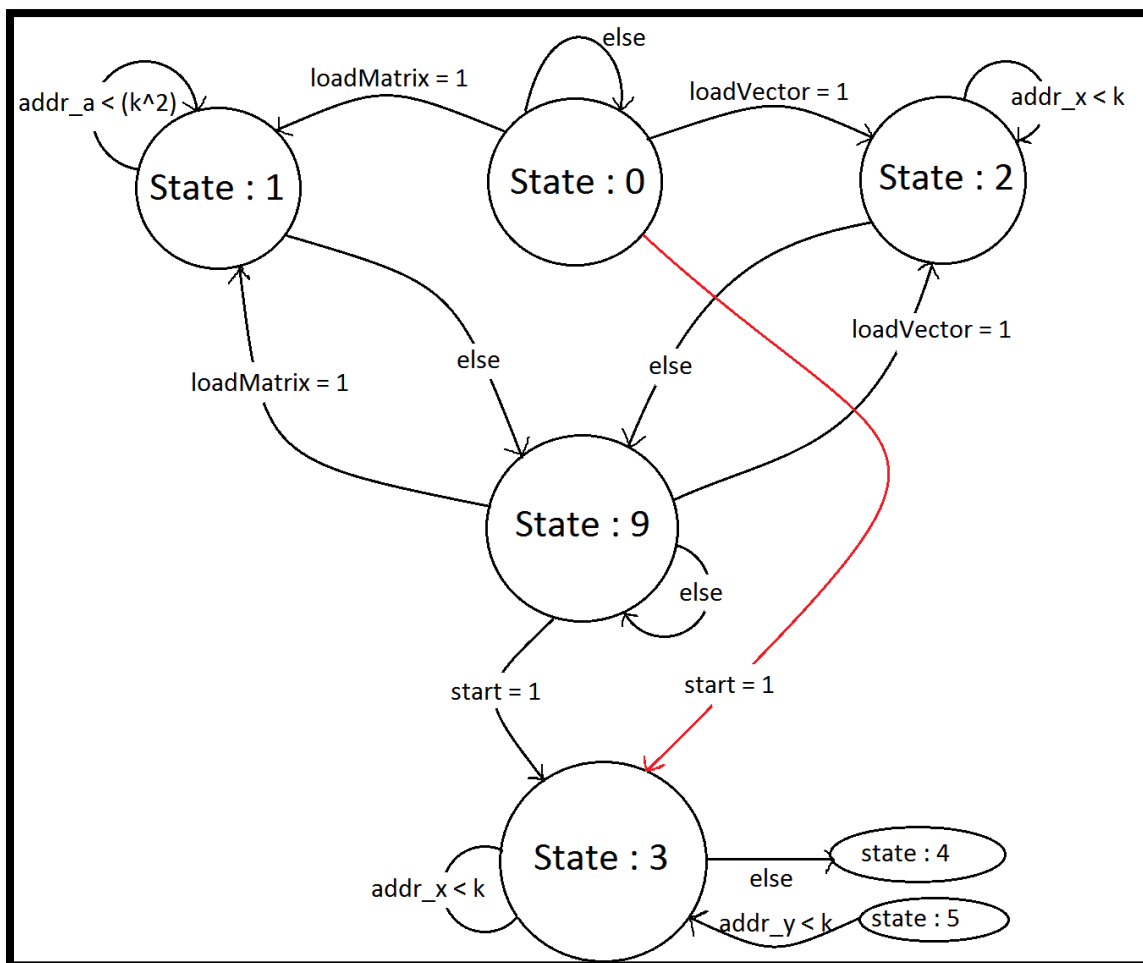
2) When loadMatrix=1, state changes to state 1. In this state, the Matrix A is copied into memory mem_a. For this, the address is incremented until [addr_a <(k^2)] and the write signal is activated wr_en_a=1. Thus this state is executed for the next (k^2) cycles. After this the state goes to state 9.

3) When loadVector=1, state changes to state 1. In this state, the Vector X is copied into memory mem_x. For this, the address is incremented until (addr_x <k) and the write signal is activated wr_en_x=1. Thus this state is executed for the next k cycles. After this the state goes to state 9.

4) After completion of either states 1 or 2, the next state is state 9. This is a wait state and it will stay in this state until it receives either loadMatrix or loadVector as input signal for which it will go to state1 or state2 respectively.

5) If start signal goes high in either state 0 or state 9, then the next state is state 3. It is in this state that the MAC operation starts. This state will compute the multiplication of one of the vectors in Matrix A with Vector X one at a time. When the MAC operation is completed, the state moves to state 4.

6) When the system is in state 4. It waits for one clock cycle to let the output of accumulator register to reflect in memory Y when Write enable is high for one clock period. After this the system goes into state 5.

7) In state 5 we write into memory location Y and clear the accumulator. Then the FSM goes back to state 3 where it does MAC operation again of next vector of Matrix A. This continues k times till the output of each row and column matrix multiplication is complete and stored in k memory locations. Once this is done the system moves to state 6.

8) In State 6 the system asserts a done signal. Once the done signal is given the system goes to state 7.

9) In State 7, the output from memory Y is displayed onto data_out for k clock cycles. Once the system completed outputting data after k clock cycles, it goes back to state 0.

Q3. Describe how you implemented the parallel (p=k) designs. Explain your structure. What extra logic and storage elements did you need to add? Did you find any clever optimizations to reduce cost?

Solution3:

Parallel design works in the following way:

a) First, we create k memories to store matrix A such that each vector from matrix A is stored in different memory.

b) When start signal is asserted, the multiply and accumulate operation is computed such that the different A-vectors created will multiply and accumulate with the same x-vector.

c) Thus all the MAC operations are done in parallel and output is stored in the form of logic (eg. f1,f2,f3,...) . At the end of MAC operation, all the output values are available at the same clock cycle.
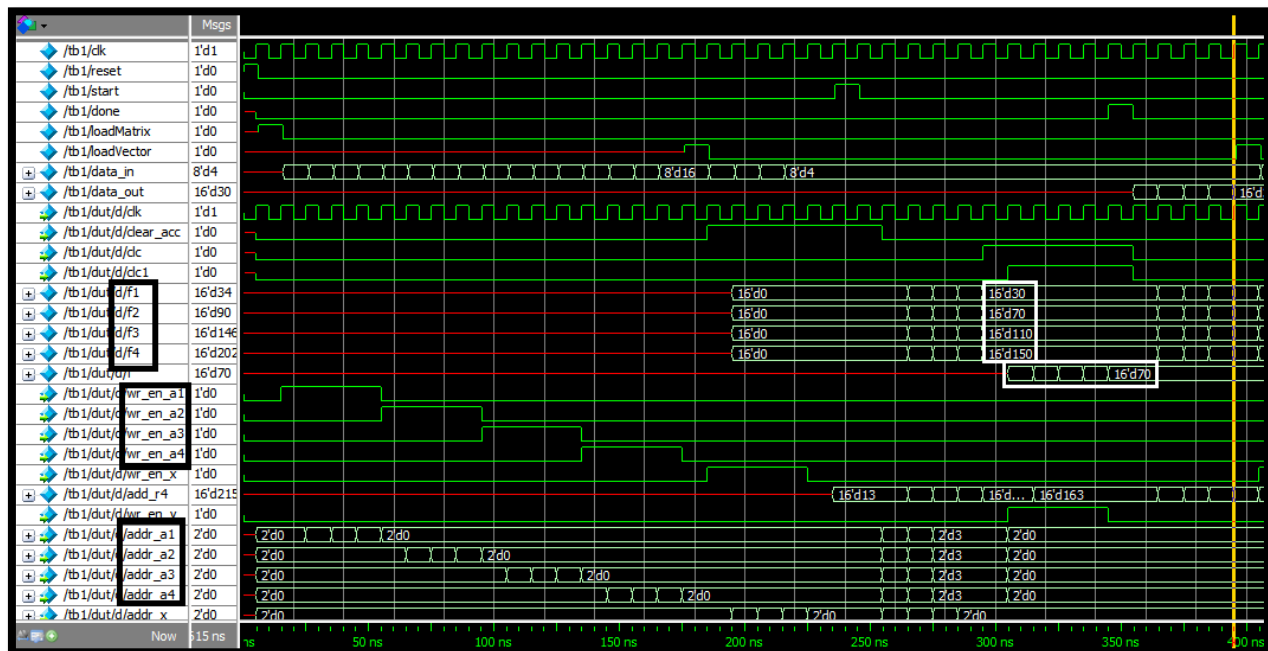
d) Finally, the values from f1,f2,f3,.... Are copied into memory Y and this is displayed when done signal is asserted.

e) Now, one extra cycle is needed to start copying the values from logic f's to memory. Hence, one extra state is used here.

f) Thus compared to unparalleled design, we need (k-1) extra memories of the size same as Vector X.

g) One clever optimization that we can use is to output the values from logic f's to data_out directly. This can be done by using multiplexer with f's as input, data_out as output and addr_y as select input. This will help us save 'k' clock cycles for every Matrix multiplication.

5

Q4.Explain your testbench strategy. How does your testbenches work? Justify why your testbenches test the designs sufficiently. How difficult was it to incorporate the flexibility of the various parameters into the testbench itself?

Solution4:

```
[sthakkar@lab04 k16p1b8g1]$ make
gcc genHardware_testbench_final.c -o genHardware_testbench -lm
[sthakkar@lab04 k16p1b8g1]$ ./genHardware_testbench 16 1 8 1
[sthakkar@lab04 k16p1b8g1]$ vlog tb_16_1_8_1.sv
Model Technology ModelSim SE vlog 10.4c Compiler 2015.07 Jul 19 2015
Start time: 00:20:26 on Dec 05,2015
vlog tb_16_1_8_1.sv
-- Compiling module mvm_16_1_8_1
-- Compiling module memory
-- Compiling module datapath                    tb1 = Noramal + 1 extra vector
-- Compiling module ctrlpath
-- Compiling module tb1 ─────────────────── tb2 = 1(Start without loadVector and
-- Compiling module tb2 ─────────────────── loadMatrix),1Normal ,1 Extra Matrix
-- Compiling module tb3 ──────────────────── tb3 = 1 Set of Random values for matrix and vector
-- Compiling module tb4
-- Compiling module tb6
-- Compiling module tb5
                                                tb4 = 1 Normal + 50 Extra Random Vector
Top level modules:
        tb1
        tb2
        tb3                                     tb6 = 1 Normal + 50 Extra Random Vector
        tb4                                     and 50 Random Matrix
        tb6                             tb5 = 1 Noraml + 50 Extra Random Matrix
        tb5
```

Test benches used:

1. **tb1:** This test bench tests the design in **two iterations**. In the first iteration the *loadMatrix* and *loadVector* both are asserted before *start* signal. In the second iteration, only the *loadVector* is asserted. Thus in the second iteration the A-matrix does not change but X-vector values changes.The output of the same is stored in proj3_outValuestb1 file.

2. **tb2:** This test bench tests the design in **three iterations**. In the first iteration the system is given only *Start* signal with no load vector or load matrix signal. Here the outputs are X (don't care). Next in the second iteration test, both *loadMatrix* and *loadVector* are asserted. In the third iteration, only *loadMatrix* is asserted. Thus in the third iteration, only the matrix values change without any change in vector values. The output of the same is stored in proj3_outValuestb2 file.

3. **tb3:** This test bench tests for only **one iterations**, but it checks using **random values**. Firstly, the random inputValues and expectedOutput files are created using the genRandom code (this code is written in the same *genHardware_testbench.c* file). The loadMatrix and loadVector are asserted so that it takes the inputValues as input and generates outValues file. The inputs are taken in hex format and outputted in decimal format. The system verilog output of the same is stored in proj3_outValuestb3 file. This is then compared with values stored in file proj3_expectedOutputtb3. This file has the results computed by the c code. The expectedOutput and outValues files matches for all cases.

4. **tb4:** This test bench tests for **fifty one** iterations. In each case it checks for **random values.** In the first iteration the normal operation of asserting the *loadMatrix, loadVector* and *start* is executed. On insertion of the signals the system takes random inputs created from genRandom file. After this the value of the matrix is kept as it is and the *loadVector* is asserted followed by start signal. This operation computes the values for same matrix values but different vector values. This operation of inserting the loadVector and start signal is carried for 50 times. So the system computes the matrix multiplication for one matrix value and 51 different random vector values. The number of times we want to assert the random vector is controlled by the variable **numvector**. For testing purpose we have set it to 50. We can change the value for checking more outputs.
The matrix multiplication results are stored in proj3_outValuestb4. This is compared with the c generated results stored in proj3_expectedOutputttb4. The values match perfectly.

5. **tb5:** This test bench tests for **fifty one** iterations. In each case it checks for **random values.** In the first iteration the normal operation of asserting the *loadMatrix, loadVector* and *start* is executed. On insertion of the signals the system takes random inputs created from genRandom file. After this the value of the vector is kept as it is and the *loadMatrix* is asserted followed by start signal. This operation computes the values for same vector values but different matrix values. This operation of inserting the *loadMatrix* and *start* signal is carried for 50 times. So the system computes the matrix multiplication for one set of vector value and 51 different random matrix values. The number of times we want to assert the random vector is controlled by the variable **nummatrix1**. For testing purpose we have set it to 50. We can change the value for checking more outputs. The matrix multiplication results are stored in proj3_outValuestb5. This is compared with the c generated results stored in proj3_expectedOutputttb5. The values match perfectly.

6. **tb6:** This test bench tests for **fifty one** iterations. In each case it checks for **random values.** The entire process of asserting *loadMatrix, loadVector* and *start* signals is repeated for 50 sets of random matrix values and random vector values. The number of times we want to assert the random values of matrix and vector is controlled by the variable **numentire**. For testing purpose we have set it to 50. We can change the value for checking more outputs.The matrix multiplication results are stored in proj3_outValuestb6. This is compared with the c generated results stored in proj3_expectedOutputttb6. The values match perfectly.

The 6 test-benches used have accurately shown the values of the output. The first tow test-benches compute the values for known inputs. Third test bench tests for random values and fourth to upto sixth computes for random values for large set of values and different conditions. Moreover the sample test-bench used for milestone has operated perfectly.

The value of K and B were challenging in designing the test-benches for different conditions. The number of bits of designed had effected the designing of C code. Here there were instances where the number of bits produced outputs greater than the bounds of data type int. Apart from this the limitation of output bits of System Verilog design generated values which were out of bounds.so it saved the answer in 2's complement form. This was incorporated in the C code.

7

Q5) 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.

Solution5:

1) **Changing 'k':**
    a) As 'k' increases, our control path/datapath will not change, so problem size will not change.
    b) The power dissipated by the design will increase as 'k' increases since the number of computations will increase, this is valid for low values of 'b'. Also area will increase only due to the increase in the memory sizes of A,X and Y.
    c) Increasing 'k' does not affect precision.
    d) With increasing the value of K the frequency of operation almost remains same , however when we increase the value of k sufficiently high the frequency of operation reduces. In our analysis , for a constant b when we increased the K value from 4 to 32 at value K= 32 the frequency of operation has reduced.

2) **Changing 'p':**
    a) As 'p' increases, the complexity of design will increase. This is because we need to add more memories and registers to the design.
    b) Power dissipated will remain fairly constant or might depend on individual designs because the number of computations for paralleled and unparalleled designs are the same, only difference is that the computations are done at different times.
    c) Increasing 'p' does not affect precision.
    d) Increasing parallelism will increase performance greatly. This is because the computations are performed in parallel at the same time. However, parallelism adds a great amount of complexity in circuit design.

3) **Changing 'b':**
    a) Problem size does not depend on 'b'.
    b) Area increases with increase in 'b' due to increase in memory to accommodate the extra number of bits.
    c) Precision greatly depends on the value of 'b'. If 'b' is low then output will overflow and we will get wrong answer. If 'b' is much higher, then the MSB bits remain redundant.
    d) The number of bits to be used 'b' should be optimum, neither less or too high. In general 'b' does not affect performance of design.
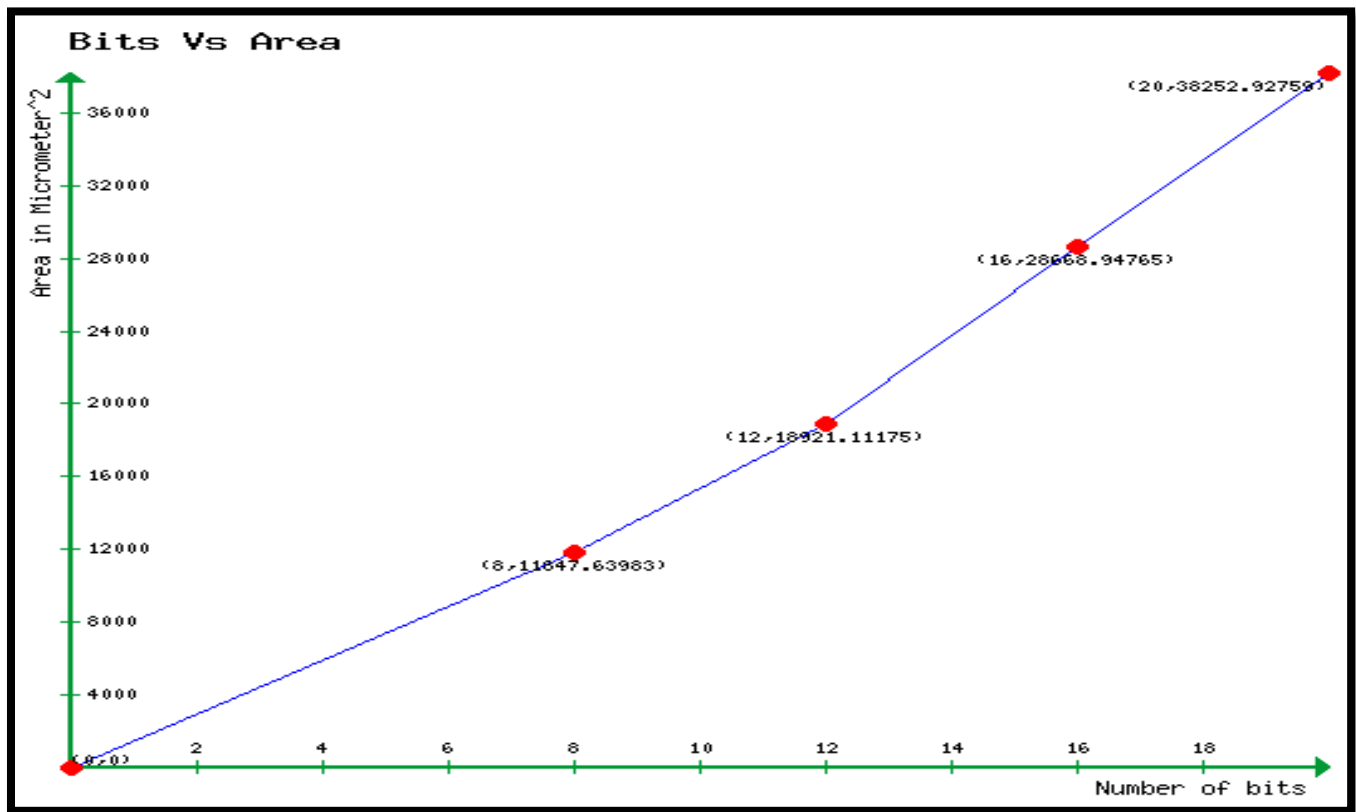
4) **Changing 'g':**
    a) Complexity increases with increase in 'g'. This is because we not only have to add registers in between a path, but that path should be critical one if the performance has to be increased. Thus pipelining is not that difficult but understanding where to pipeline is a difficult job.
    b) Adding number of pipeline stages will accordingly increase the cost of the design. This is due to accommodation of extra registers.
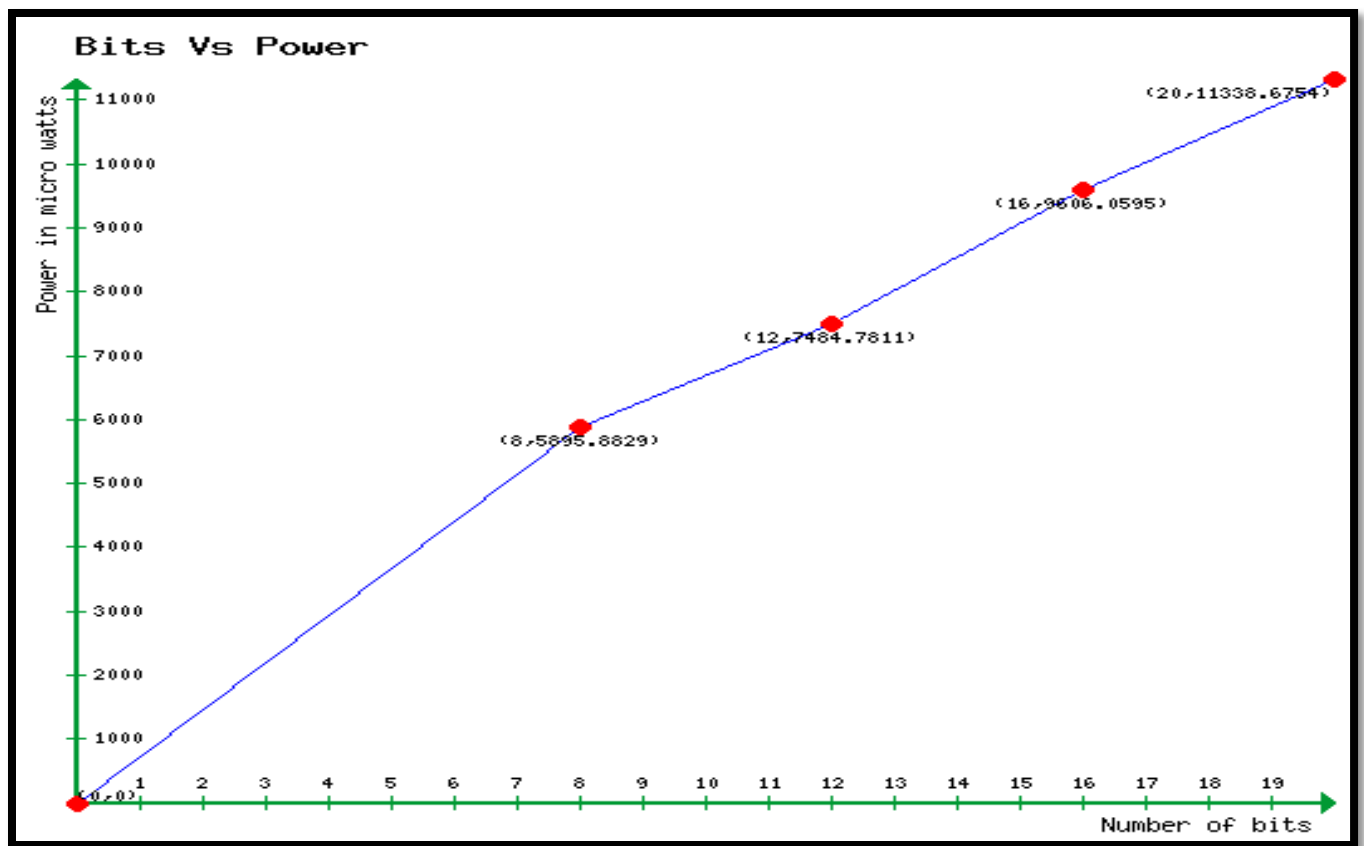   c) Increasing 'g' does not affect precision.

Increasing parallelism greatly increases performance. Adding pipeline is most efficient if the timing for critical path is twice or more times greater than the 2nd critical path. Otherwise adding more pipelines will not increase performance to that extent. -

----

Q6. 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.

Solution6:

Sr.No.	Type of Design	Clock Period (ns)	Frequency (GHz)	Area(um^2)				Power(uW)			Critical Path	
				Combinational area	Buf/Inv area:	Noncombinational area:	Total	Total Dynamic Power	Cell Leakage Power	Total	Start Point	End Point
1	k = 8,p=8,b =8,g=1	1.17	0.854	6747.887977	622.706	4477.045848	11847.63983	5651.37	244.5129	5895.8829	d/mem_a4/data_out_reg[2]	d/f4_reg[14]
2	k = 8,p=8,b =12,g=1	1.41	0.709	11318.03397	961.324	6641.753778	18921.11175	7094.4	390.3811	7484.7811	d/mem_x/data_out_reg[8]	d/f3_reg[23]
3	k = 8,p=8,b =16,g=1	1.55	0.645	18151.57395	1769.432	8747.9417	28668.94765	9012.8	593.2595	9606.0595	d/mem_x/data_out_reg[1]	d/f7_reg[29]
4	k = 8,p=8,b =20,g=1	1.69	0.591	25025.01395	2297.708	10930.20563	38252.92759	10553.5	785.1754	11338.675	d/mem_x/data_out_reg[10]	d/f2_reg[38]

Bits Vs Area
Area in Micrometer^2
Number of bits

(20,38252.92759)
(16,28668.94765)
(12,18921.11175)
(8,11847.63983)
(0,0)

Bits Vs Power

Power in micro watts — Number of bits

(0,0)
(8,5895.8829)
(12,7484.7811)
(16,9606.0595)
(20,11338.6754)

Q7. 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.

Solution7

| Sr.No.	Type of Design	Value of K	Clock Period (ns)	Frequency(GHz)	Area(um^2)				Power(uW)			Critical Path	
					Combinational Area:	Buf/Inv Area:	Non Combinational Area:	Total Area	Total Dynamic Power	Cell Leakage Power	Total	Start Point	End Point
1	k = 4,p=1,b =8,g=1	4	0.88	1.136	1407.405993	91.77	1370.431951	2869.607944	2050.4	58.3069	2108.7069	d/mem_a/data_out_reg[6]	d/mul_out_r_reg[15]
2	k = 8,p=1,b =8,g=1	8	0.88	1.136	3323.936012	420.014003	3562.271872	7306.221887	5759.9	126.8659	5886.7659	d/mem_a/data_out_reg[2]	d/mul_out_r_reg[14]
3	k = 16,p=1,b =8,g=1	16	0.87	1.149	10270.79206	1512.47601	11398.63159	23181.89966	20694.2	433.8115	21128.0115	d/f_reg[0]	d/f_reg[15]
4	k = 32,p=1,b =8,g=1	32	1.05	0.952	36177.06427	5821.676033	40937.39852	82936.13882	63407.3	1517.7	64925	c/addr_a_reg[0]	d/mem_a/data_out_reg[1]

As seen from the table the critical path changes with the value of K.
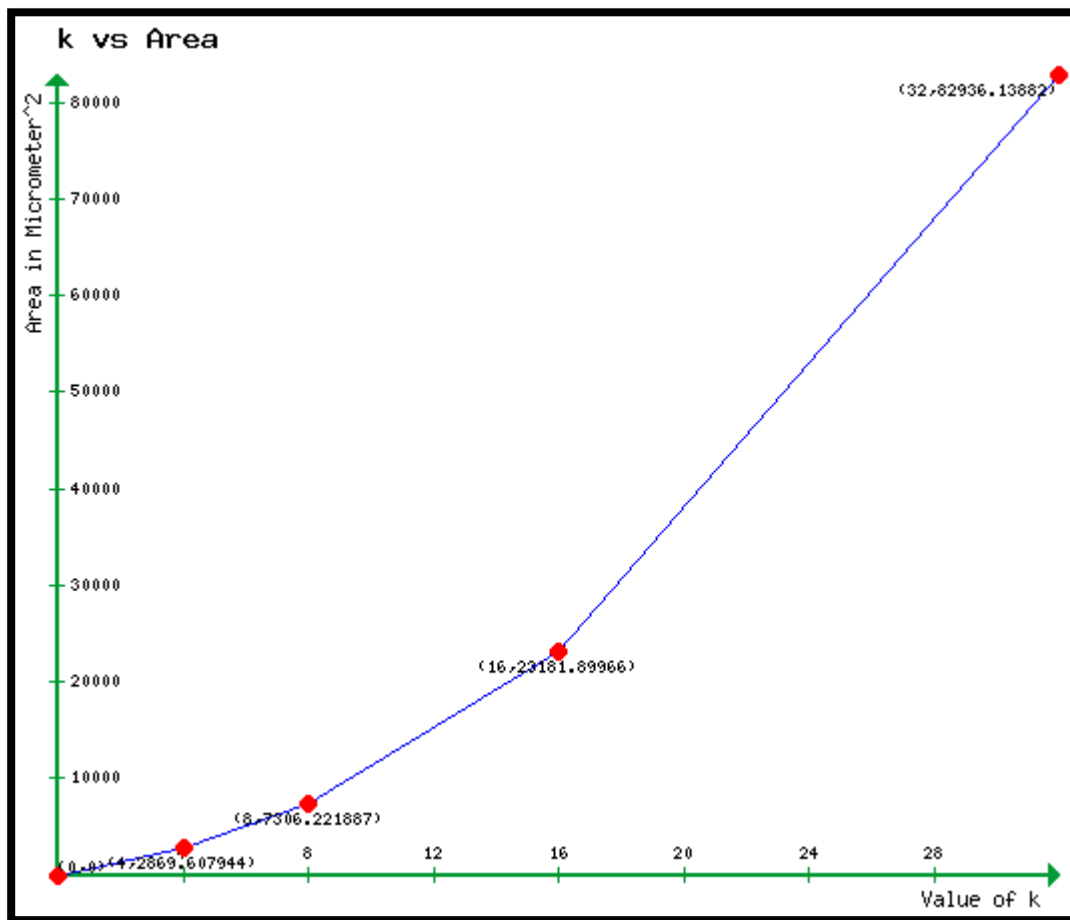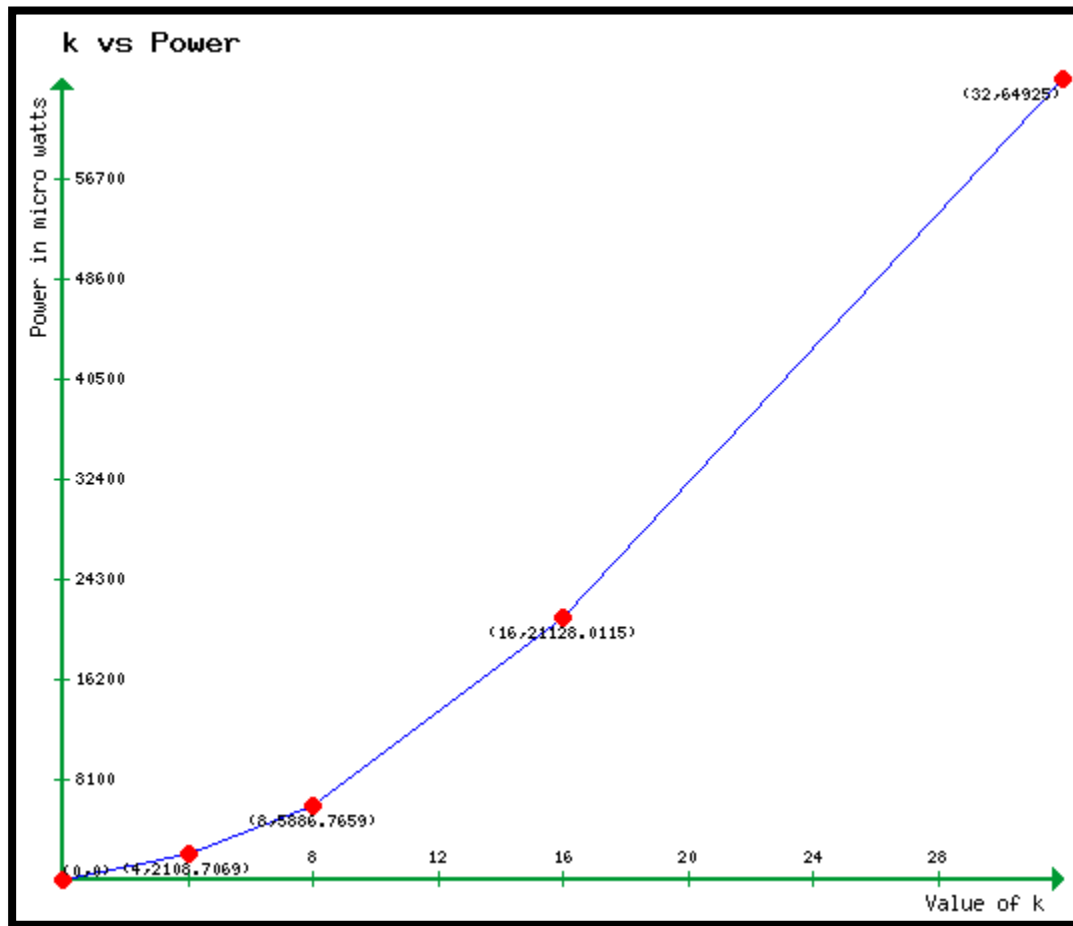
To calculate c, count the number of clock cycles when loadVector is set to high till when the done signal goes high.

| k	c	f(Ghz)	Throughput (words/second)
4	35	1.136	129828571.4
8	99	1.136	91797979.8
16	323	1.149	56916408.67
32	1155	0.952	26375757.58
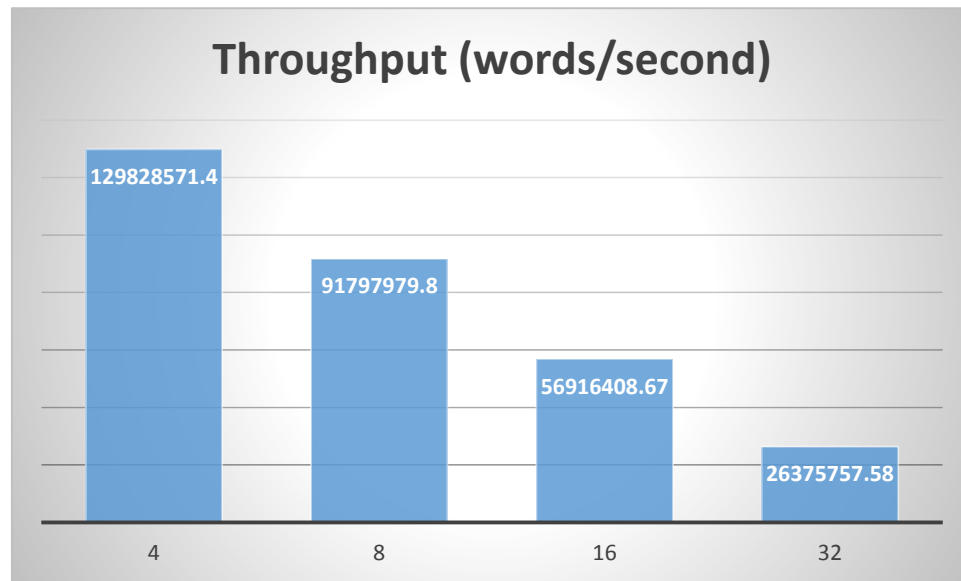
Q8. In the previous step, you used pipelined designs (g=1). How do the costs and performance change if you use un-pipelined designs? Generate and synthesize designs without pipelining using parameters (k, p, b, g) = (4, 1, 8, 0) and (32, 1, 8, 0) and compare these two designs to their pipelined counterparts from question 7. How does the critical path change?

Solution8:

Sr.No.	Type of Design	Value of K	Clock Period (ns)	Frequency(GHz)	Area(um^2)				Power(uW)			Critical Path	
					Combinatio nal Area:	Buf/Inv Area:	Non Combinatio nal Area:	Total Area	Total Dynamic Power	Cell Leakage Power	Total	Start Point	End Point
1	k = 4,p=1,b =8,g=0	4	1.1	0.909	1362.185993	91.504	1306.059955	2759.749948	1577	56.2933	1633.2933	d/mem_a/data_out_reg[1]	d/f_reg[14]
2	k = 32,p=1,b =8,g=0	32	1.13	0.884	36094.60427	5773.264033	40866.11052	82733.97882	58721.8	1514.1	60235.9	c/addr_a_reg[0]	d/mem_a/data_out_reg[2]

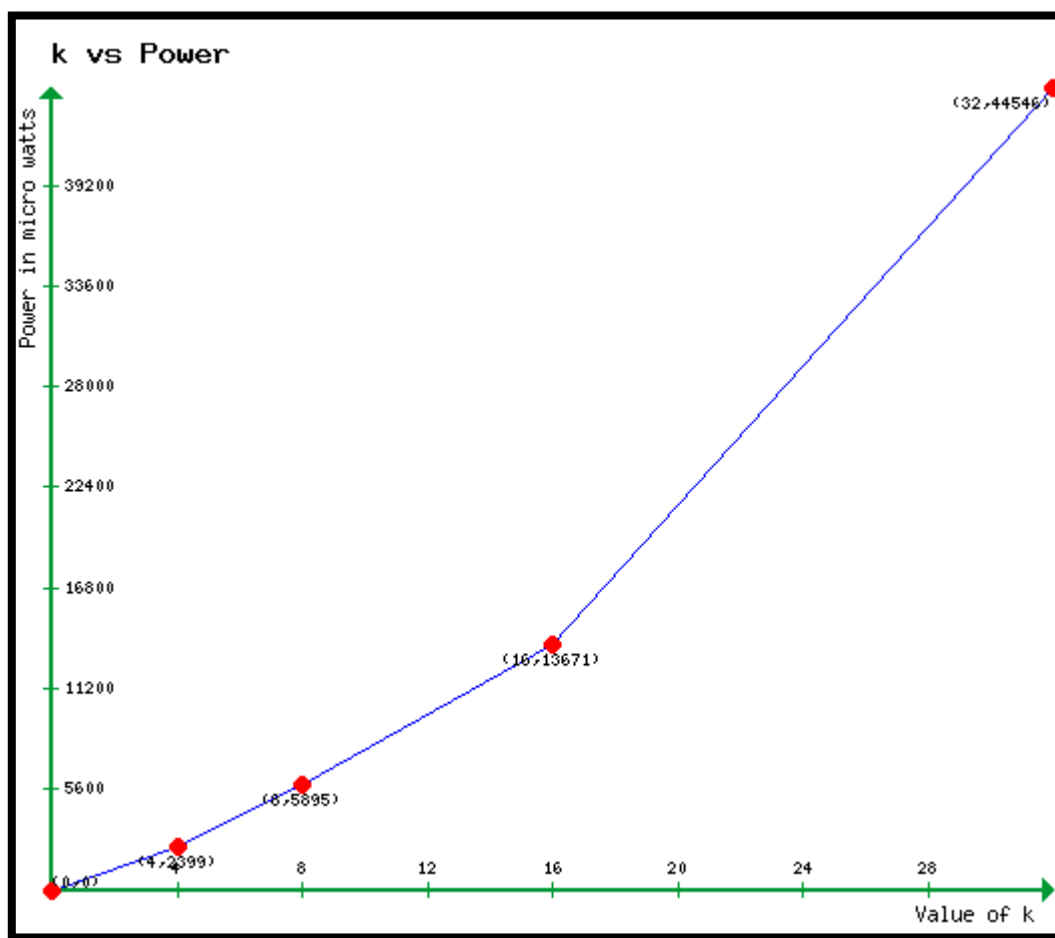Comparison between Pipelined and Un-pipelined Design

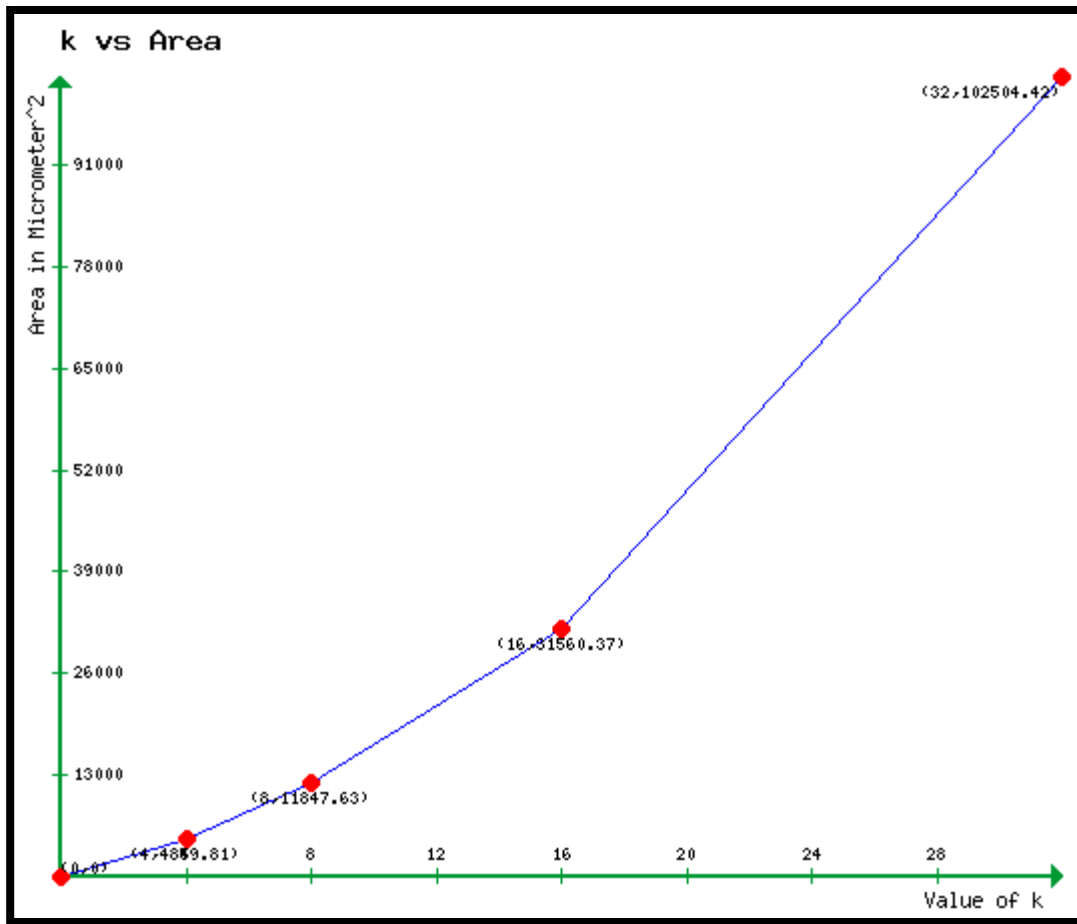| Pipelined	Parameter	Pipelined	
		Yes	No
K=4	Area (um^2)	2,869.608	2,759.750
	Power(uw)	2,108.707	1,633.293
	Throughput(words/cycle)	129,828,571.000	117290322.6
	Critical Path -Start Point	d/mem_a/data_out_reg[6]	c/addr_a_reg[0]
	Critical Path -End Point	d/mul_out_r_reg[15]	d/mem_a/data_out_reg[1]
K=32	Area (um^2)	82,936.139	82,733.979
	Power(uw)	64,925.000	60,235.900
	Throughput(words/cycle)	26375757.58	25189670.53
	Critical Path -Start Point	d/mem_a/data_out_reg[1]	c/addr_a_reg[0]
	Critical Path -End Point	d/f_reg[14]	d/mem_a/data_out_reg[2]

COMPARISON OF PIPELINED AND UN-PIPELINED DESIGN
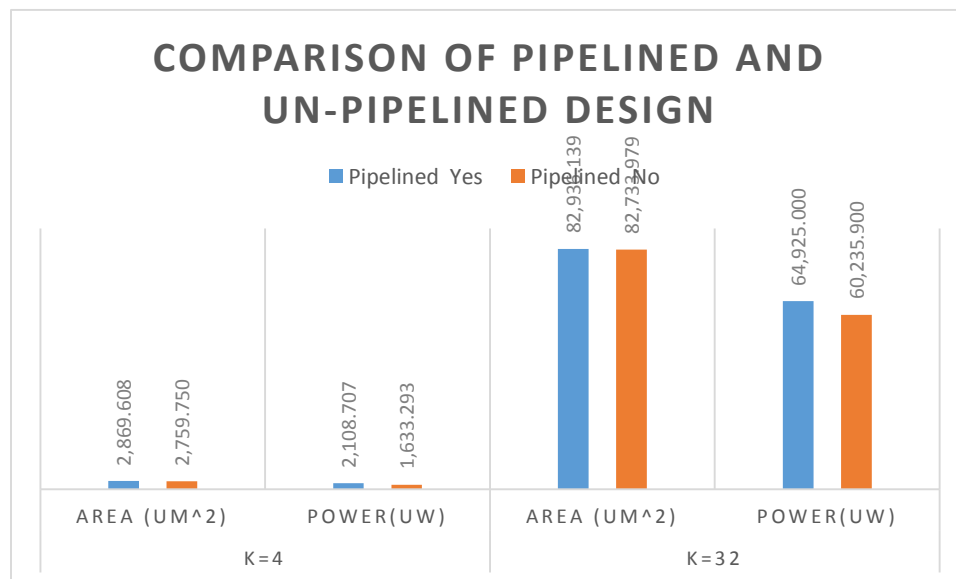
Q9 Lastly, you need to evaluate how the designs change when you increase parallelism (by setting p=k). Now, generate and synthesize four parallel designs: k=4, 8, 16, and 32, with p=k, b=8, and g=1. Graph: (1) power versus k, (2) area versus k, and (3) throughput versus k.

Solution9:
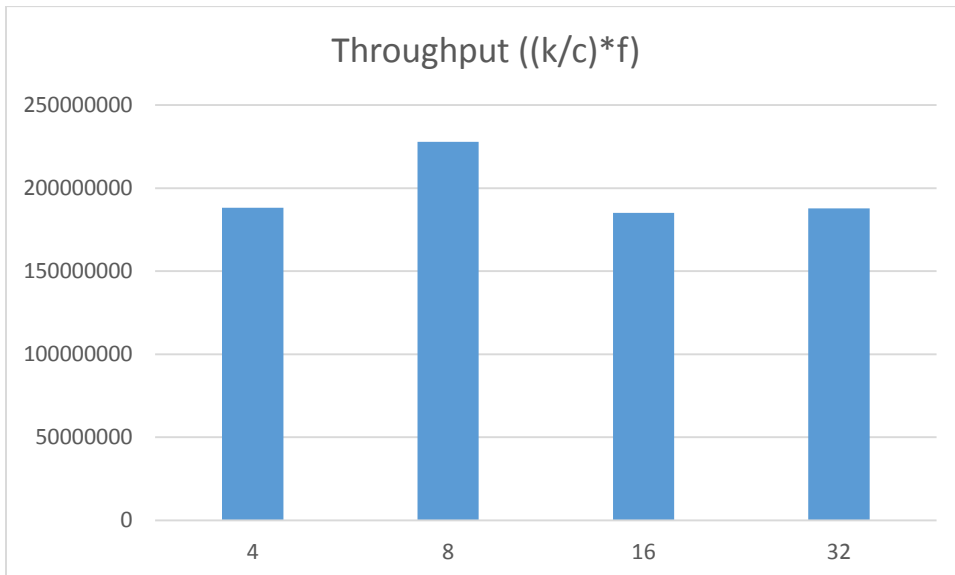
k	Power (uwatts)	Area (um^2)	Throughput (words / second)
4	2399	4859.81	188222222
8	5895	11847.63	227920000
16	13671	31560.37	185185185
32	44546	102504.42	187921568

k vs Area

Area in Micrometer^2 (y-axis)
Value of k (x-axis)

Data points:
(0,0)
(4,4859.81)
(8,11847.63)
(16,31560.37)
(32,102504.42)

y-axis labels: 13000, 26000, 39000, 52000, 65000, 78000, 91000
x-axis labels: 8, 12, 16, 20, 24, 28

Throughput versus *k*

Throughput ((k/c)*f)

y-axis: 0, 50000000, 100000000, 150000000, 200000000, 250000000
x-axis: 4, 8, 16, 32

Paralleled and Unparalleled COST vs THROUGHPUT differences:

k	Cost Power (uwatts)		Cost Area (um^2)		Throughput	
	Unparalleled	Paralleled	Unparalleled	Paralleled	Unparalleled	Paralleled
4	2108	2399	2869.6	4859.81	129828571.4	188222222
8	5886	5895	7306.2	11847.63	91797979.8	227920000
16	21128	13671	23181.9	31560.37	56916408.67	185185185
32	64925	44546	82936.1	102504.42	26375757.58	187921568

1) We can see that for low-k values (eg. k=4), unparalleled designs dissipate approximately 14% less power, have 41% less area and have 30% less throughput than paralleled designs.
2) Similarly for high-k values (eg k=32), paralled designs dissipate approximately 30% less power, have 20 % less area but gives more than 600% throughput than unparalled.
3) Looking at these stats, we can see that for high-k values, parallel designs are very favourable, while for low-k designs, the use of either of the two design depends on the usage required.