FPGA Matrix Multiplier

In Hwan Baek
Henri Samueli School of
Engineering and Applied Science
University of California Los Angeles
Los Angeles, California
Email: chris.inhwan.baek@gmail.com

David Boeck
Henri Samueli School of
Engineering and Applied Science
University of California Los Angeles
Los Angeles, California
Email: dboeck@ucla.edu

***Abstract*—This paper describes an FPGA design that performs 4x4 matrix multiplication. The goal of the design is to optimize throughput, area, and accuracy. The design of our matrix multiplier consists of four main parts: fractional binary numbers (fixed point notation), binary multiplication, matrix addition, and fetch routine. Each part is designed and optimized to find the optimal balance among the throughput, the area, and the accuracy. According to the test results, the design with the optimal result used a 3-stage pipeline from the BRAM block to the output of the summation block, 13-bit representation of binary values, shifting and addition to replace multipliers, and an inexpensive fetch module.**

I. Introduction

This paper describes an FPGA design that performs 4x4 matrix multiplication. The design is implemented with Virtex-5 using Xilinx ISE. A matrix with input integer values as its elements is multiplied with another matrix whose elements have constant values as shown in Figure 1. For fetching input values, a good candidate is Xilinx Block RAM (BRAM). Since the second matrix contains fractional values, the binary values must be able to represent fractional values. Although IEEE floating point is the standard representation, the design uses fixed point notation for enhanced performance, which is explained in more details in section II-A.

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix} \times \begin{bmatrix} 1 & 1/8 & 3 & 1/4 \\ 3/4 & 3/2 & 3/8 & 2 \\ 1/2 & 5 & 7/15 & 3 \\ 1 & 140/123 & 1/4 & 3/4 \end{bmatrix}$$

Figure 1. Maxtrix A x Martix B

The number of bits chosen for the fixed point notation directly affects the accuracy, the area, and the throughput of the design. Because the fixed point notation only estimates the fractional values, the accuracy may not be perfect. The throughput is given as the number of matrices calculated per second. The area is given as the number of used devices such as registers. The goal of the design is to optimize throughput, area, and accuracy. There is a trade off between the three criteria. For instance, increasing the number of bits to represent the matrix element values will improve the accuracy, but it will increase the area because more registers are required to store more bits. Performance, defined as throughput divided by area, has a cost weight of 0.9. Accuracy has a cost weight of 0.1. Using this evaluation method, the performance, the area, and the accuracy need to be balanced in order to achieve the best result.

II. System Design

The design of our matrix multiplier consists of four main parts: fractional binary numbers (fixed point notation), binary multiplication, matrix addition, and fetch routine. These are explained in the subsections below.

A. *Fractional Binary Numbers*

Binary numbers in general represent integer values. Since Matrix B contains fractions and mixed numbers, the output can be fractions and mixed numbers. Binary strings can represent fractions and mixed numerals if explicitly defined beforehand. The notation used for this project is fixed point notation.

For fixed point notation, a fixed point must be chosen to split the binary string. The bits to the left of the fixed point represent integer values, while the bits to the right of the fixed point represent fractional values. Figure 2 shows a 7-bit example with the fixed point before the 3rd LSB. As shown, the weights of each bit are still powers of two, it is just that to the right of the fixed point the weights are inverse powers of two.

It is important to choose an appropriate fixed point location as it will affect accuracy of this system. The range of Matrix A are integer values from 1 to 16. Therefore, maximum value for Matrix C occurs when Matrix A is all 16's. The largest element from this matrix multiplication is $16 \times \left({}^{1}\!/_{8} + {}^{3}\!/_{2} + 5 + {}^{140}\!/_{123}\right) \approx 124.21138$. The integer portion of 124 is represented in binary by 1111100. Thus, there must be 7 bits to the left of the floating point to represent this maximum integer number.

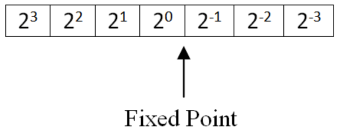


Figure 2. 7 Bit Fixed Point Example

Each of the elements in Matrix B can be represented exactly by a 16-bit fixed point binary string except $^7/_{15}$ or $^{140}/_{123}$. Thus, these numbers must be estimated which affects the accuracy of the system. Table I shows fixed point estimations for $^7/_{15}$ and $^{140}/_{123}$.

Mixed Number	Binary Fixed Point
$^7/_{15}$	0.011101110...
$^{140}/_{123}$	1.001000110...

Table I
MIXED NUMBER ESTIMATION

In fact, the entire 16 bit allowance is not needed to represent the other numbers in Matrix B be exactly. Only 3 bits to right of the fixed point are needed for the elements in Matrix B except for $^7/_{15}$ or $^{140}/_{123}$. This allows the accuracy to depend only on $^7/_{15}$ and $^{140}/_{123}$. Thus, the minimum bound for output number size is 10 bits: 7 bits for the integer and 3 bits for the fraction. To increase accuracy, the fraction portion can increase to 9 bits. However, this will affect performance as the area will increase. We chose 11 bits for the initial design to estimate $^7/_{15}$ by 0.0111 and estimate $^{140}/_{123}$ by 1.0010. There is a trade off between performance and accuracy, and the number of bits used for the fractional portion is revisited during the optimization phase.

B. Binary Multiplication

Binary multiplication is performed with one number from Matrix A and one number from Matrix B. The value in Matrix A is an integer value from 1 to 16, while a value in Matrix B is a fixed point binary number. The multiplication function must also return a fixed point binary number of the same size as the number from Matrix B.

A general binary multiplication circuit that takes two unknown binary numbers and multiplies them together is very complex. This circuit complexity will increase area size and delay. Both these issues degrade performance. However, Matrix B is essentially a constant and can be hardcoded into the design. Therefore, a general binary multiplier circuit is not needed for this design. A circuit that multiplies by a constant can be optimized as there is only one unknown input. This approach was used for the binary multiplication of an element in Matrix A with an element of Matrix B.

To optimize the multiplication circuit, 16 different multiplication modules were created. Each module was optimized to multiply one of the elements in Matrix B with an input number. The easiest element to implement was $B_{11}, B_{41} = 1$. However, this module cannot be a simple wire because the input is a binary integer while the output is a fixed point binary number. Therefore, the input needs to be shifted to the left by the fractional bit width. The fractional bit width is the number of bits to the right of the fixed point. This is needed to convert the integer number input to the fixed point format.

The elements in Matrix B that are purely integer powers of two only involve bit shifting. These numbers are $B_{12} = {}^1/_8$, $B_{14} = {}^1/_4$, $B_{24} = 2$, $B_{31} = {}^1/_2$, and $B_{41} = {}^1/_4$. Dividing by 8 is shifting input three bits to the right, 4 is shifting input two bits to the right, and 2 is shifting input one bit to the right. Multiplying by 2 is shifting the input one bit to the left. However, as with multiplying by unity, the input is first shifted by the fractional bit width. Multiplying an input by integer powers of two only requires one shifting step.

Multiplying by integers other than powers of two adds an additional complexity. These elements are $B_{13}, B_{34} = 3$ and $B_{32} = 5$. These integers can be rewritten as $3 = (2 + 1)$ and $5 = (4 + 1)$. These equations contain a power of 2 and adds 1. To multiply a given input A_{ij} by 3, first A_{ij} is multiplied by 2 by shifting. Then A_{ij} is added to the previous shifted result. Likewise, the same concept is applied to multiplying by 5. These multiplication modules requires two steps to perform the operation. There is a shifting step and an addition step.

The next step in complexity from multiplying by integers are multiplying by fractions with powers of two in the denominator. These numbers are $B_{21}, B_{44} = {}^3/_4$, $B_{22} = {}^3/_2$, and $B_{23} = {}^3/_8$. These numbers can be expressed as a sum of inverse powers of two: ${}^3/_4 = \left({}^1/_4 + {}^1/_4 + {}^1/_4\right)$, ${}^3/_2 = \left(1 + {}^1/_2\right)$, and ${}^3/_8 = \left({}^1/_8 + {}^1/_8 + {}^1/_8\right)$. Adders in practice contain many combinational logic, and should be minimized to increase performance. Multiplying by ${}^3/_4$ and ${}^3/_8$ contains two adders. It is desirable to optimize ${}^3/_4$ and ${}^3/_8$ to reduce the number of adders. Xilinx's synthesis tool contains primitives for adders and subtractors. If subtractor primitives are used, then the following expressions can be used: ${}^3/_4 = \left(1 - {}^1/_4\right)$, and ${}^3/_8 = \left({}^1/_2 - {}^1/_8\right)$. Instead of using two adders, one subtractor is used. To multiply by ${}^3/_8$, a given input A_{ij} is multiplied by ${}^1/_2$ and ${}^1/_8$. The two results are then subtracted. The same concept is applied to ${}^3/_4$ and ${}^3/_2$. Therefore, these multiplication modules contain two steps as well. There is a shifting step and an addition/subtraction step.

The last two elements of Matrix B, $B_{34} = {}^7/_{15}$ and $B_{42} = {}^{140}/_{123}$, must be estimated as discussed previously. The initial design used 4 bits to represent the fractional portion. Therefore from Table I, these numbers are estimated by truncation with $B_{34} = 0.0111$ and $B_{42} = 1.0010$. With these estimations, $B_{34} = \left({}^1/_4 + {}^1/_8 + {}^1/_{16}\right)$ and $B_{42} = \left(1 + {}^1/_8\right)$. Multiplying by B_{42} is easily implemented with a shifting step and addition step as was done with previous values. However, multiplying by B_{34} will involve two adders. Using subtraction, we can rewrite as $B_{34} = \left[1 - \left({}^1/_2 + {}^1/_{16}\right)\right]$. Even though this result contains one adder and one subtractor, there are two numbers to shift rather than three numbers to shift. Therefore, B_{34} is implemented with a subtractor for improved optimization.

C. Matrix Addition Module

Every element in result Matrix C is a sum of four products. Equation (1) shows the required products for an element of Matrix C for a given row (i is the row number). This equation demonstrates that to calculate a row of Matrix C, only the same corresponding row is needed from Matrix A. To calculate row 1 of Matrix C, only row 1 of Matrix A is needed; to calculate row 2 of Matrix C, only row 2 of Matrix A is needed, etc. All

16 elements of Matrix A do not need to be present in order to start computation.

$$
\begin{aligned}
C_{i1} &= A_{i1}(B_{11}) + A_{i2}(B_{21}) + A_{i3}(B_{31}) + A_{i4}(B_{41}) \\
C_{i2} &= A_{i1}(B_{12}) + A_{i2}(B_{22}) + A_{i3}(B_{32}) + A_{i4}(B_{42}) \\
C_{i3} &= A_{i1}(B_{13}) + A_{i2}(B_{23}) + A_{i3}(B_{33}) + A_{i4}(B_{43}) \\
C_{i4} &= A_{i1}(B_{14}) + A_{i2}(B_{24}) + A_{i4}(B_{33}) + A_{i4}(B_{44})
\end{aligned} \tag{1}
$$

Taking advantage that only one row of Matrix A is needed to calculate one row of Matrix C, the matrix addition module only needs a four element input and a four element output. If Matrix A row 1 is the input, then Matrix C row 1 is the output; if Matrix A row 2 is the input, then Matrix C row 2 is the output, etc. This module basically implements (1) with A_{i1}, A_{i2}, A_{i3}, and A_{i4} as the inputs; and C_{i1}, C_{i2}, C_{i3}, and C_{i4} as the outputs.

This approach would require four steps to output a single matrix. Even though this affects the throughput, this approach was used for two main reasons. The first reason is that the area would be one fourth of the area if the circuit calculated all 16 elements of Matrix C at the same time. This will help the performance of the circuit. The second, and main reason, is that fetching the elements of Matrix A from BRAM is a bottleneck. The BRAM can only output a limited number of Matrix A elements at a time. Therefore, to access all 16 elements of Matrix A form BRAM will take multiple clock cycles. Therefore, it is vital to start calculating Matrix C as soon as the required elements are available. In this case, the first four elements of Matrix C can be calculated when the first four elements of Matrix A are available, the second four elements of Matrix C can be calculated when the second four elements of Matrix A are available, etc. This strategy just matches the throughput of the BRAM, since the BRAM cannot access all 16 elements of Matrix A at the same time, with the added benefit of reducing area.

D. Fetch Routine

A separate fetch module is designed in order to incorporate pipeline concept to our system. As described above, the matrix multiplication is performed for each row at a time. Each element in each row of Matrix A needs to be fetched, then multiplication and addition are performed after awards. Without the pipeline design, this process is performed repetitively in a sequence. In other words, the system will perform fetch, calculation, fetch, calculation, and so on in a sequence. This is a not optimal because the fetch module is not doing any work while the calculations are being performed, and vice versa. The pipeline in our design is explained more thoroughly in section III.

We used a dual BRAM to fetch values for Matrix A. Since the matrix multiplication is performed for one row at a time, four values need to be fetched from the dual BRAM while a dual BRAM outputs two values at a time. In order to have four values to be fetched, the fetch module has four sets of intermediate registers and four sets of output registers. Let's assume that the BRAM outputs the following sequence of integer value sets: (16, 15), (14, 13), (12, 11), (10, 9), (8, 7), (6, 5), (4, 3), (2, 1). Two sets of intermediate registers, namely X1 and X2, alternate with the other two sets, namely X3 and X4, to store the outputs of the BRAM. Thus, the first output values 16 and 15 are each stored in X1 and X2. After one clock cycle, the next output values 14 and 13 are each stored in X3 and X4. Then these values in X1, X2, X3, and X4 are stored in the output register sets, namely R1, R2, R3, and R4, respectively. This alternating storing of the values is controlled by a 1-bit register named count, which flips its value every cycle to indicate which register sets must store the BRAM output. Such process is well shown in figure 3.

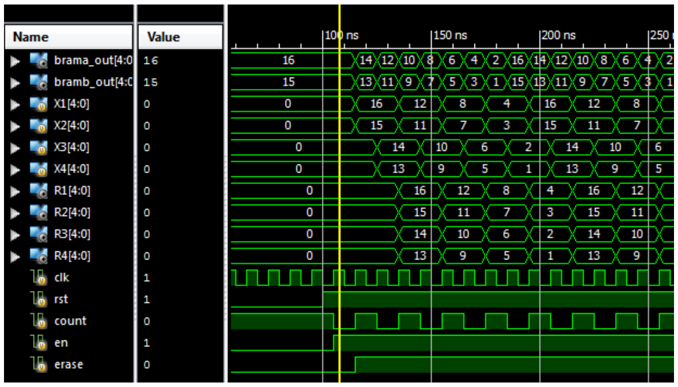


Figure 3. Fetch Module Testbench Waveform

As shown, whenever count is 1, the BRAM outputs are stored into X1 and X2. Whenever count is 0, the outputs are stored into X3 and X4. However, this is not true until the first rising edge of count. Because the BRAM output values are 16 and 15 until the second rising edge of clk after a reset (rst), incorrect values will be stored in X3 and X4. Such problem is resolved with a register named erase to make sure the value storing process waits until the second rising edge of clk after the reset. Another problem is the delay of BRAM, which results in storing of incorrect values. Another register named en resolved this problem.

III. Implementation

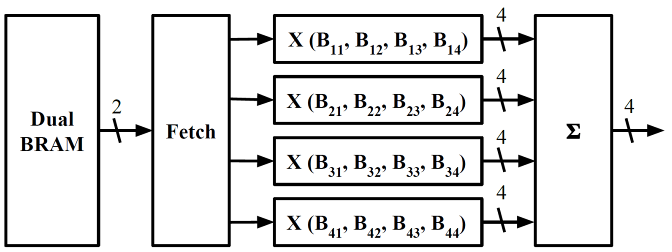


Figure 4. System Block Diagram

The separate modules are placed together as shown in Figure 4. The fetch routine cycles through the dual BRAM and outputs four elements of Matrix A at a time. After two clock cycles the fetch block outputs the first four elements of Matrix A, after two more clock cycles the fetch block outputs the next four elements of Matrix A, etc. The output of the fetch block feeds into the binary multiplier block. Each element is multiplied by four different constants (Matrix B). There are 16 outputs after the multiplier block that are fed to a summation block. The output of the summation block is a row of output Matrix C.

The system forms a pipeline from the BRAM block to the output of the summation block. When the output of the summation block is Matrix C row 1, Matrix C row 2 is in the summation phase, Matrix C row 3 is in the multiplier phase, and Matrix C row 4 is in the fetching phase. Pipelining increases latency, but reduces the critical path delay between registers. A lower critical delay path allows for faster clock frequencies. However, adding more in between registers, to reduce critical path delay, increases the circuit area. There is a balance needed with pipelining between clock speed and area. This is explored during optimization.

IV. Optimization

A. Performance

$$Performance = \frac{throughput}{area} \tag{2}$$

The first step to improved performance is to improve the bottleneck that is accessing BRAM. Equation (2) defines performance and throughput is the number of output matrices per second. The single output BRAM requires 16 clock cycles to access all 16 elements of Matrix A. However, using a dual output BRAM allows two elements of Matrix A to be read per clock cycle. This doubles the throughput as now only 8 clock cycles are needed to access all 16 elements of Matrix A. Implementing a dual output BRAM rather than using a single output BRAM adds minimal area as the dual BRAM still counts as 1 BRAM. Thus, using the dual output BRAM is greatly preferred over the single output BRAM to improve throughput.

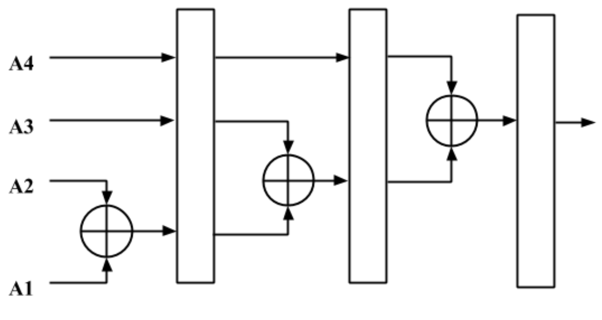


Figure 5. Three Stage Summation Pipeline

Throughput can also be improved by decreasing the minimum clock period. Additional pipeline registers can be used to reduce the critical path delay to reduce the minimum clock period. Figure 5 shows a proposal to add addition pipeline to a four input adder using three stages. In the first stage, input A1 and A2 are added. In the second stage, input A3 is added to the result of stage one. In the third stage, A4 is added to the result of stage two. Each stage is separated by sequential logic registers. This differs from the initial design of one stage pipeline summation block with only registers at the output.

The initial 11 bit output design with only one stage summation block had a minimum clock period of 3.46ns. This allows a throughput of 3.612×10^7 matrices per second. The area is calculated by adding all the registers, LUTs, and BRAM. Two types of LUTS were calculated: Slice LUTs and LUT flip-flop pairs. This area was calculated to 551. The performance with one stage summation block is 6.5567×10^4.

Replacing one stage summation block with a three stage summation block lowered the minimum period to 3.078ns. This increases the throughput to 4.061×10^7 matrices per second. However, the area increased to 894. The performance reduces to 4.542×10^4 for the three stage summation block. Even though increasing the pipeline registers speed up the system, the increase in area makes the three stage summation block unattractive. Performance also goes down using a two stage summation block. Therefore, the initial one stage summation block was kept. After this exercise, it was determined that increasing the pipelines within the system blocks would not improve performance as the area increases more than the throughput increases. Therefore, additional performance improvements would have to reduce area instead of increasing throughput.

In order to reduce the area, the number of registers, LUTs, and LUT flip-flop pairs need to be reduced. The most noticeable type of device among these in the verilog code is registers because the registers are usually declared in the code in spite of Xilinx's optimization of the number of registers during synthesis process. The initial design of fetch routine had larger area because it had extra intermediate registers to deal with the BRAM delay and the resulting incorrectness explained in section II-D above.

The algorithm used in the fetch module also changed. The initial design used data stream concept. Three sets of registers are placed to hold output values from each output port of the dual BRAM. Figure 6 shows this process. The output of port a is first stored in XA3. During the next clock cycle, the value in XA3 is stored in XA2 while the new output of port a overwrites the value in XA3. Similarly, the value in XA2 is stored in XA1 during the next clock cycle and the value in XA3 overwrites the value in XA2 while the new output of port a overwrites the value in XA3. XA1 and XA2 are connected to the output registers. The port b output values are fetched in the identical way.

The new fetch algorithm described in section II-D will not only reduce the area but also increase the speed. While the initial fetch design has value assignment to each of the intermediate register every clock cycle, the new design alternates the value assignment to two sets of register. Thus, a new value is assigned to each of the intermediate registers

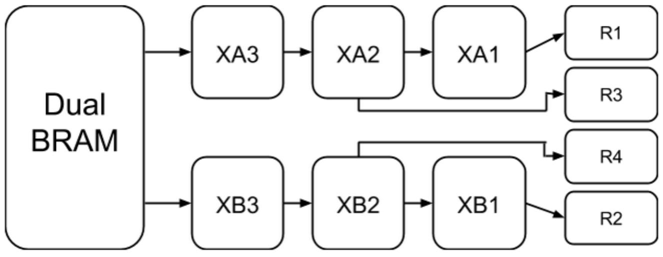


Figure 6. The Initial Design of Fetch Routine

once every two clock cycles. Nonetheless, the number of clock cycles taken to generate the output of the fetch routine stay the same. Less number of value assignment increases the speed of the fetch routine.

B. Accuracy

$$[C] = \begin{bmatrix} 47.25 & 109.29675 & 63.408333 & 85.75 \\ 34.25 & 78.243902 & 47.041667 & 61.75 \\ 21.25 & 47.191057 & 30.675 & 37.75 \\ 8.25 & 16.138211 & 14.308333 & 13.75 \end{bmatrix}$$

Figure 7. Matrix C Result in Double Format

$$\% error = 100\% \times \frac{1}{16}\sum_{i=1}^{16}\frac{|Val_{fix} - Val_{double}|}{Val_{double}} \qquad (3)$$

The last optimization effort was regarding the binary multiplication. As describe in section II-A, more bits are chosen to have better estimate of ${}^{7}/_{15}$ and ${}^{140}/_{123}$. With 13 bit output, 6 bits are used to represent fractional bits. Therefore, ${}^{7}/_{15} \approx 0.011101$ and ${}^{140}/_{123} \approx 1.001000$. The percent error for 13 bit solution is 0.0982% calculated using (3) and would receive a accuracy grade of 0.3. The ideal Matrix C is shown in Figure 7 and was calculated using Matrix A of 16 down to 1. 13 bits was chosen over the initial 11 bits because the 11 bits error would received a grade of zero for accuracy, with error of 0.173%. Increasing the output bit size to 14 bit would increase the accuracy to grade 0.7 (error of 0.0411%), but would decrease performance by 12% because of increase in area. Since performance is weighted more, the 13 bit solution was chosen.

To represent the values of the resulting matrix elements with fairly good accuracy, 13 bits are more than needed. Therefore, using 13 bits just to have better estimate of these two numbers does not seem to be reasonable. Instead of declaring all values as 13 bit binary, intermediate registers can be declared to estimate these number and multiply with better accuracy. For instance, the module that multiplies the input with ${}^{7}/_{15}$ can have 19-bit intermediate register for the multiplication. The least significant six bits are truncated from the 19-bit multiplication result, which is stored in the output registers. Therefore, there is an improvement of accuracy while maintaining the same number of bits used to represent the binary values. However, this slightly more complicated multiplication algorithm requires more registers, LUTs, and LUT flip-flop pairs.

In order to reduce the area, the number of bits can be reduced. With the new binary multiplication algorithm, it is possible to have fairly good accuracy with less number of bits. 10-bit and 11-bit are chosen for testing. 10-bit results in .0705% error and 11-bit result in .06% error. Since error between .05% and .1% gives the same accuracy score, 10-bit seems to be the better design. After synthesis process, however, Xilinx throws in more registers and LUTs than expected. The resulting area wasn't reduced and hence the performance did not improve. Therefore, the design is reverted to the one before this optimization without intermediate 19 bit registers. All multiplication registers will have 13 bits.

V. Results

$$[B'] = \begin{bmatrix} 1 & 0.125 & 3 & 0.25 \\ 0.75 & 1.5 & 0.375 & 2 \\ 0.5 & 5 & 0.453125 & 3 \\ 1 & 1.125 & 0.25 & 0.75 \end{bmatrix}$$

Figure 8. Matrix B Estimation with 13 Bit Output

$$[C'] = \begin{bmatrix} 47.25 & 109.125 & 63.21875 & 85.75 \\ 34.25 & 78.125 & 46.90625 & 61.75 \\ 21.25 & 47.125 & 30.59375 & 37.75 \\ 8.25 & 16.125 & 14.28125 & 13.75 \end{bmatrix}$$

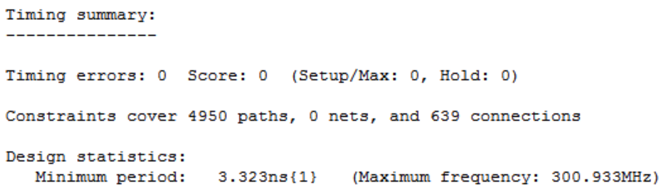
Figure 9. Matrix C Estimation with 13 Bit Output

$$64 \times [C'] = \begin{bmatrix} 3024 & 6984 & 4046 & 5488 \\ 2192 & 5000 & 3002 & 3952 \\ 1360 & 3016 & 1958 & 2416 \\ 528 & 1032 & 914 & 880 \end{bmatrix}$$

Figure 10. Shifted Matrix C Estimation with 13 Bit Output

The 13 bit output design has 7 bits to represent integer values and 6 bits to represent fractional values. The estimation matrix B' is shown in Figure 8. Only values B'_{33} and B'_{42} needed to be estimated. However, these two estimation values in Matrix B' propagate throughout the resulting Matrix C' shown in Figure 9. The simulation tool only displays the binary number seen at the output as it is oblivious to the fixed point notation. Since the fractional bit width is 6 bits, shifting Matrix C' 6 bits to the left (or multiplying by 64) will return the unsigned decimal value. Figure 10 shows the resulting shifted matrix for simulation verification.

The 13 bit output design has a minimum clock frequency of 3.323ns as generated by the post place and route timing report. See Figure 11 for post-place and route static timing screenshot. One matrix is outputted every 8 clock cycles, therefore the throughput is 3.7617×10^{7} matrices per second. Figure 12 shows the device utilization summary for this design. This design uses 158 Slice Registers, 145 Slice LUTs, 199 LUT



```
Timing summary:
---------------

Timing errors: 0  Score: 0  (Setup/Max: 0, Hold: 0)

Constraints cover 4950 paths, 0 nets, and 639 connections

Design statistics:
   Minimum period:   3.323ns{1}   (Maximum frequency: 300.933MHz)
```

Figure 11. PAR Timing

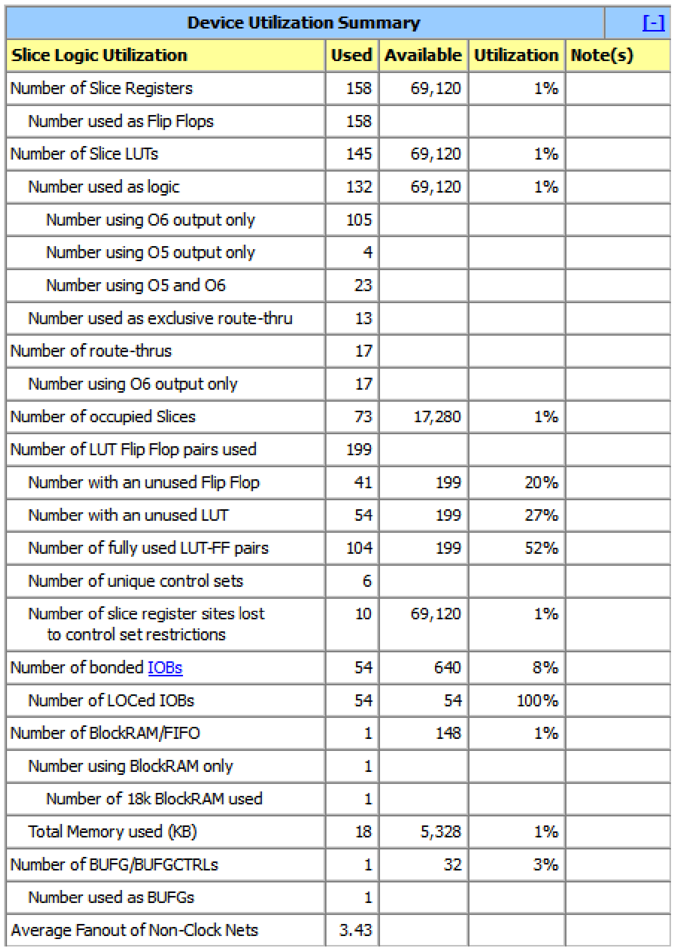
Flip-Flop Pairs, and 1 BRAM. When counting both Slice LUTs and LUT Flip-Flop Pairs as different LUTs the total area is 552. This calculates to a performance of 6.8146×10^4. The design summary also states that there are 54 bonded IOs. This corresponds to the 52 outputs (4×13) and 2 inputs (rst and clk).

To verify the design, the post place and route simulation model was generated. Figure 13 shows the simulation results with a clock frequency of 3.4ns. There is a latency from when reset goes high to when the circuit outputs the first values. However, this is a one-time latency due to initial pipelining and is ignored in the throughput calculation. The first stable output at around 126ns is the first row of Matrix C with $R1 = C_{11}$, $R2 = C_{12}$, $R3 = C_{13}$, and $R4 = C_{14}$. After two clock cycles the next stable output is the second row of Matrix C with $R1 = C_{21}$, $R2 = C_{22}$, $R3 = C_{23}$, and $R4 = C_{24}$. As follows, the next two rows of Matrix C are the next two stable outputs. This simulation results match the expected estimation matrix shown in Figure 10. The fetch routine then loops back to the first address of BRAM and the output repeats outputting Matrix C indefinitely.

The simulation uses a clock frequency of 3.4ns, which is 2% within the minimum clock frequency generated by the post place and route timing report. The design successfully works at the desired minimum frequency. The two markers in Figure 13 show the beginning of a matrix and end of a matrix and are separated by 27.2ns. This is 8 clock cycles and verifies that a matrix is outputted every 8 clock cycles.

VI. Conclusion

The goal of the FPGA matrix multiplier design was to achieve the optimal accuracy and performance, which are measured with the area and the throughput. The trade off among these criteria introduced the main challenge of finding a balance for optimal result. In the overall system level, pipeline concept is used to achieve high throughput. Optimization process involved inspection of accuracy and performance change with respect to the number of bits used for the fixed point notation and the number registers declared in each module's algorithm. Different number of pipeline levels are tested and the optimal level is chosen. 10-bit, 11-bit, 13-bit, 14-bit designs are implemented to find the optimal number of bits to represent the binary values in fixed point notation. Different algorithms for fetch routines and binary multiplication are designed and tested for the best result. According to the test results, the design with the optimal result used one pipeline



Device Utilization Summary				[-]
Slice Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Registers	158	69,120	1%	
Number used as Flip Flops	158			
Number of Slice LUTs	145	69,120	1%	
Number used as logic	132	69,120	1%	
Number using O6 output only	105			
Number using O5 output only	4			
Number using O5 and O6	23			
Number used as exclusive route-thru	13			
Number of route-thrus	17			
Number using O6 output only	17			
Number of occupied Slices	73	17,280	1%	
Number of LUT Flip Flop pairs used	199			
Number with an unused Flip Flop	41	199	20%	
Number with an unused LUT	54	199	27%	
Number of fully used LUT-FF pairs	104	199	52%	
Number of unique control sets	6			
Number of slice register sites lost to control set restrictions	10	69,120	1%	
Number of bonded IOBs	54	640	8%	
Number of LOCed IOBs	54	54	100%	
Number of BlockRAM/FIFO	1	148	1%	
Number using BlockRAM only	1			
Number of 18k BlockRAM used	1			
Total Memory used (KB)	18	5,328	1%	
Number of BUFG/BUFGCTRLs	1	32	3%	
Number used as BUFGs	1			
Average Fanout of Non-Clock Nets	3.43			

Figure 12. Device Utilization

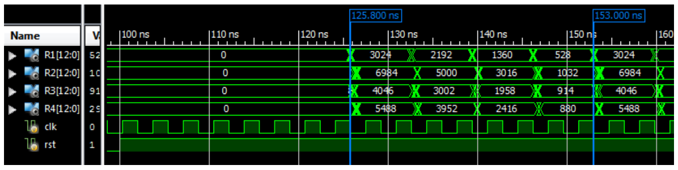

125.800 ns
153.000 ns
Name
V
100 ns
110 ns
120 ns
130 ns
140 ns
150 ns
160
R1[12:0] 52 0 3024 2192 1360 528 3024
R2[12:0] 10 0 6984 5000 3016 1032 6984
R3[12:0] 91 0 4046 3002 1958 914 4046
R4[12:0] 29 0 5488 3952 2416 880 5488
clk 0
rst 1

Figure 13. Simulation Results

from the BRAM block to the output of the summation block, 13-bit representation of binary values, shifting and addition to replace multipliers, and an inexpensive fetch module.

Acknowledgment

The authors would like to thank Prof. Lei He, Tianheng Tu, and Zhuo Jia.

References

[1] *LogiCORE IP Block Memory Generator v7.1*, 2012.
[2] H. So. (2006, Feb. 28). *Introduction to Fixed Point Number Representation* [Online]. Availible: https://inst.eecs.berkeley.edu/ cs61c/sp06/handout/fixedpt.html
[3] D. K. Tala (2014 Feb. 9). *Verilog Tutorial - World of Asic* [Online]. Availible: http://www.asic-world.com/verilog/veritut.html