Project 3: Vivado HLS Matrix Multiplication Optimization

Jennifer Hellar

ELEC 522 - Fall 2018

Introduction

In this project, I used the Xilinx Vivado HLS tool to design and optimize a matrix multiplication system. I tested and verified this design, simulated it in software and hardware, and compared the resulting performance and resource utilization to that of the system from Project 2.

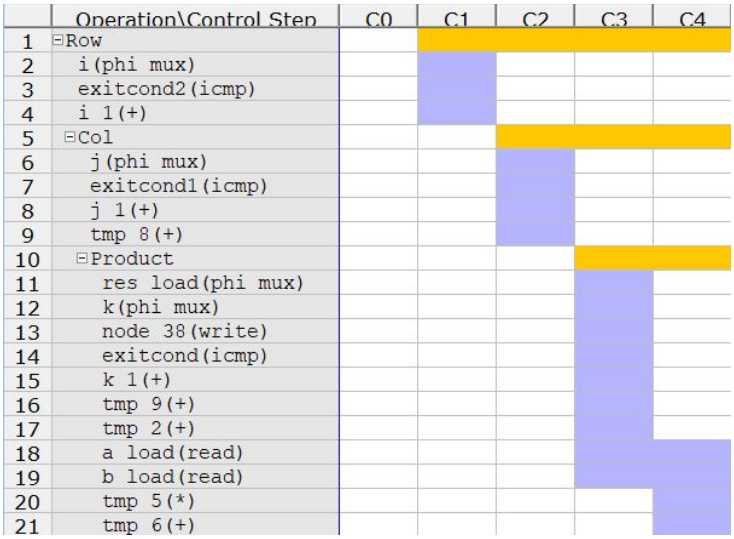
Initial Design

The starting point for my design was the ***matrixmul*** project from the Vivado HLS Tutorial Lab 1, shown in ***Figure 1*** below. It is a simple function that takes 2-D arrays of "short" data types, loops over the indices of the matrices, performs the inner product of the rows and columns, and outputs the resulting matrix.

```
67 #include "matrixmul.h"
68
69 void matrixmul(
70       mat_a_t a[MAT_A_ROWS][MAT_A_COLS],
71       mat_b_t b[MAT_B_ROWS][MAT_B_COLS],
72       result_t res[MAT_A_ROWS][MAT_B_COLS])
73 {
74   // Iterate over the rows of the A matrix
75   Row: for(int i = 0; i < MAT_A_ROWS; i++) {
76     // Iterate over the columns of the B matrix
77     Col: for(int j = 0; j < MAT_B_COLS; j++) {
78       // Do the inner product of a row of A and col of B
79       res[i][j] = 0;
80       Product: for(int k = 0; k < MAT_B_ROWS; k++) {
81         res[i][j] += a[i][k] * b[k][j];
82       }
83     }
84   }
85 }
```

Figure 1: Original matrixmul code

The Analysis view of the timing performance of this design is shown in ***Figure 2*** below, and the timing profile in ***Figure 3***.



	Operation\Control Step	C0	C1	C2	C3	C4
1	Row					
2	i(phi mux)					
3	exitcond2(icmp)					
4	i 1(+)					
5	Col					
6	j(phi mux)					
7	exitcond1(icmp)					
8	j 1(+)					
9	tmp 8(+)					
10	Product					
11	res load(phi mux)					
12	k(phi mux)					
13	node 38(write)					
14	exitcond(icmp)					
15	k 1(+)					
16	tmp 9(+)					
17	tmp 2(+)					
18	a load(read)					
19	b load(read)					
20	tmp 5(*)					
21	tmp 6(+)					

Figure 2: Analysis view of timing performance for original ***matrixmul***

	Pipelined	Latency	Initiation Interval	Iteration Latency	Trip count
matrixmul	-	169	170	-	-
Row	no	168	-	42	4
Col	no	40	-	10	4
Product	no	8	-	2	4

Figure 3: Analysis Timing Profile for original ***matrixmul***

From this, we can see that the overall latency is 169 and the interval is 170, so there is no parallelism. Calculating a single inner product takes 8 cycles, a single row of results takes 40 cycles, and the entire matrix takes 168 cycles. Note the two cycle memory accesses in lines 18-19 of ***Figure 2*** will cause difficulties in pipelining, since the memory accesses across loop iterations will start to collide if there is a pipeline initiation interval of 1.

First Improvement

The first optimization that I tried was partitioning the input and output arrays, as well as pipelining the Row loop, as shown in ***Figure 4***.

```
67 #include "matrixmul.h"
68
69 void matrixmul(
70       mat_a_t a[MAT_A_ROWS][MAT_A_COLS],
71       mat_b_t b[MAT_B_ROWS][MAT_B_COLS],
72       result_t res[MAT_A_ROWS][MAT_B_COLS])
73 {
74 #pragma HLS ARRAY_PARTITION variable=res complete dim=1
75 #pragma HLS ARRAY_PARTITION variable=b complete dim=2
76 #pragma HLS ARRAY_PARTITION variable=a complete dim=1
77   // Iterate over the rows of the A matrix
78   Row: for(int i = 0; i < MAT_A_ROWS; i++) {
79 #pragma HLS PIPELINE rewind
80     // Iterate over the columns of the B matrix
81     Col: for(int j = 0; j < MAT_B_COLS; j++) {
82       // Do the inner product of a row of A and col of B
83       res[i][j] = 0;
84       Product: for(int k = 0; k < MAT_B_ROWS; k++) {
85         res[i][j] += a[i][k] * b[k][j];
86       }
87     }
88   }
89 }
90
```

Figure 4: First optimization code (partitioning and pipelining)

Prior to this optimization, the matrices were each treated as a block of memory which had to be indexed into and driven through a single port interface. After this optimization, each matrix was "partitioned" along one dimension and split across multiple ports, allowing memory accesses to happen faster, as described in the Xilinx tutorial Powerpoint on Improving Performance in ***Figure 5***.

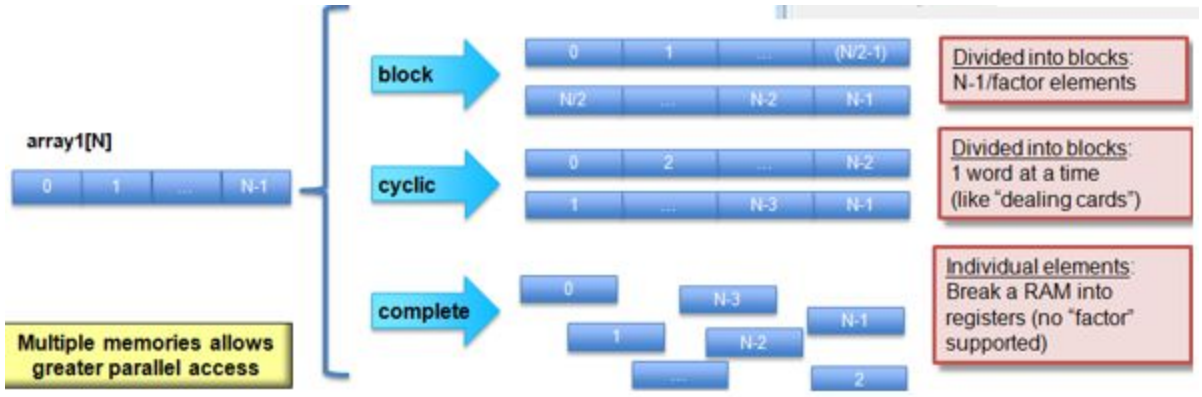
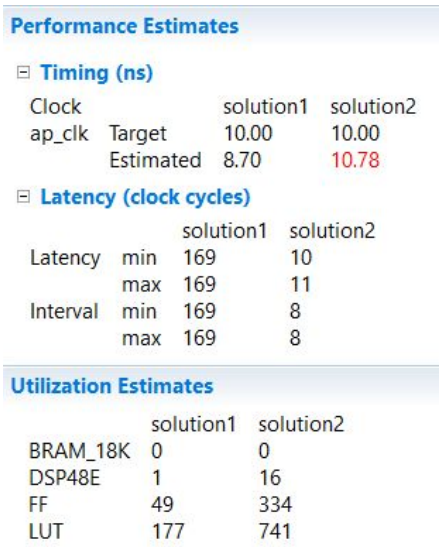


Figure 5: Visual depiction of array partitioning (see Improving Performance ppt)

The second part of the optimization was pipelining the outermost loop, which unrolled all of the loops below it so that they could be performed in parallel. The performance and

utilization estimates for the original code (solution1) and the optimized code (solution2) are shown in ***Figure 6***.



Performance Estimates

Timing (ns)

Clock		solution1	solution2
ap_clk	Target	10.00	10.00
	Estimated	8.70	10.78

Latency (clock cycles)

		solution1	solution2
Latency	min	169	10
	max	169	11
Interval	min	169	8
	max	169	8

Utilization Estimates

	solution1	solution2
BRAM_18K	0	0
DSP48E	1	16
FF	49	334
LUT	177	741

Figure 6: Performance & utilization estimates (original: solution1, improved: solution2)

As expected, HLS reported a scheduling error for accessing memory through port a and was unable to pipeline with an initiation interval of 1. Nevertheless, this solution achieved a very low latency of 10 cycles using 16 DSPs. The estimated clock overshot the target of 10 ns, though, making this solution not ideal.

Second Improvement

Next, I tried to fix the memory access problem by optimizing the I/O interface. To do this, I rewrote the code so that each array element would only be accessed once and in order; such a dataflow would allow us to use FIFO interfaces. The resulting code is shown in ***Figure 7***.

```
1  #include "matrixmul.h"
2
3  void matrixmul(
4        char a[MAT_A_ROWS][MAT_A_COLS],
5        char b[MAT_B_ROWS][MAT_B_COLS],
6        short res[MAT_A_ROWS][MAT_B_COLS])
7  {
8      char a_row[MAT_A_ROWS];
9      char b_copy[MAT_B_ROWS][MAT_B_COLS];
10     short tmp = 0;
11
12     // Iterate over the rows of the A matrix
13     Row: for(int i = 0; i < MAT_A_ROWS; i++) {
14       // Iterate over the columns of the B matrix
15       Col: for(int j = 0; j < MAT_B_COLS; j++) {
16         // Do the inner product of a row of A and col of B
17         tmp = 0;
18         if(j==0)
19             RD_ROW: for(int k = 0; k < MAT_A_ROWS; k++)
20                 a_row[k] = a[i][k];
21         if(i==0)
22             RD_COL: for(int k = 0; k < MAT_B_ROWS; k++)
23                 b_copy[k][j] = b[k][j];
24         Product: for(int k = 0; k < MAT_B_ROWS; k++) {
25           tmp += a_row[k] * b_copy[k][j];
26         }
27         res[i][j] = tmp;
28       }
29     }
30 }
```

Figure 7: Second optimization code

From this, we see that each row of A is accessed and copied once (line 19), as is each column of B (line 22). In addition, a temporary variable holds and accumulates each result before it is finally copied to the output res.

With this new code, I tried several optimizations as follows:

1) Solution1: No optimization.
2) Solution2: Partitioned the arrays and pipelined Row as described above.
3) Solution3: Reshaped the arrays, made them FIFO port Interfaces, and pipelined Col as shown in ***Figure 8*** below.
4) Solution4: Applied the Dataflow pragma to the whole function, in addition to the optimizations in Solution3.

```
#include "matrixmul.h"

void matrixmul(
      char a[MAT_A_ROWS][MAT_A_COLS],
      char b[MAT_B_ROWS][MAT_B_COLS],
      short res[MAT_A_ROWS][MAT_B_COLS])
{
#pragma HLS INTERFACE ap_fifo port=res
#pragma HLS ARRAY_RESHAPE variable=b complete dim=1
#pragma HLS INTERFACE ap_fifo port=b
#pragma HLS ARRAY_RESHAPE variable=a complete dim=2
#pragma HLS INTERFACE ap_fifo port=a
    char a_row[MAT_A_ROWS];
    char b_copy[MAT_B_ROWS][MAT_B_COLS];
    short tmp = 0;

    // Iterate over the rows of the A matrix
    Row: for(int i = 0; i < MAT_A_ROWS; i++) {
      // Iterate over the columns of the B matrix
      Col: for(int j = 0; j < MAT_B_COLS; j++) {
#pragma HLS PIPELINE
        // Do the inner product of a row of A and col of B
        tmp = 0;
        if(j==0)
            RD_ROW: for(int k = 0; k < MAT_A_ROWS; k++)
                a_row[k] = a[i][k];
        if(i==0)
            RD_COL: for(int k = 0; k < MAT_B_ROWS; k++)
                b_copy[k][j] = b[k][j];
        Product: for(int k = 0; k < MAT_B_ROWS; k++) {
          tmp += a_row[k] * b_copy[k][j];
        }
        res[i][j] = tmp;
      }
    }
}
```

Figure 8: Solution3 optimization code with pragmas

Matrix A was reshaped along dimension 2 (column-wise) so that different column elements could then be accessed at the same time and the RD_ROW loop could easily be unrolled. By the same logic, Matrix B was reshaped along dimension 1 so that each row could be accessed independently for the RD_COL loop. The RESHAPE pragma essentially converted them to single very wide ports. To maintain correct data accesses of the FIFO ports, the pipeline had to be placed on the Col loop rather than the Row loop. The resulting performance and utilization estimates for these solutions are shown in ***Figure 9*** below.

Performance Estimates

Timing (ns)

Clock		solution1	solution2	solution3	solution4
ap_clk	Target	10.00	10.00	10.00	10.00
	Estimated	8.70	10.78	7.19	7.19

Latency (clock cycles)

		solution1	solution2	solution3	solution4
Latency	min	201	10	21	22
	max	457	11	21	22
Interval	min	201	8	21	23
	max	457	8	21	23

Utilization Estimates

	solution1	solution2	solution3	solution4
BRAM_18K	0	0	0	0
DSP48E	1	8	2	2
FF	128	510	315	254
LUT	411	1205	713	393

Figure 9: Performance and Utilization Estimates for New Code

The new code greatly increased the original latency without any pragmas (solution1). Moreover, the same optimizations from the first iteration applied to this code (partitioning and pipelining) resulted in a clock estimate of 10.78, which overshot our target of 10; it also used the most LUTs of any solution (solution2).

The solution which implemented the FIFO interface (solution3) achieved the best latency without overshooting the target clock, and used only an estimated 2 DSPs. Therefore, I chose this for my final design.

Final HLS Code Design

My final code design is shown in ***Figure 10*** below.

```
#include "matrixmul.h"

void matrixmul(
      char a[MAT_A_ROWS][MAT_A_COLS],
      char b[MAT_B_ROWS][MAT_B_COLS],
      short res[MAT_A_ROWS][MAT_B_COLS])
{
#pragma HLS INTERFACE ap_fifo port=res
#pragma HLS ARRAY_RESHAPE variable=b complete dim=1
#pragma HLS INTERFACE ap_fifo port=b
#pragma HLS ARRAY_RESHAPE variable=a complete dim=2
#pragma HLS INTERFACE ap_fifo port=a
    char a_row[MAT_A_ROWS];
    char b_copy[MAT_B_ROWS][MAT_B_COLS];
    short tmp = 0;

    // Iterate over the rows of the A matrix
    Row: for(int i = 0; i < MAT_A_ROWS; i++) {
      // Iterate over the columns of the B matrix
      Col: for(int j = 0; j < MAT_B_COLS; j++) {
#pragma HLS PIPELINE
        // Do the inner product of a row of A and col of B
        tmp = 0;
        if(j==0)
            RD_ROW: for(int k = 0; k < MAT_A_ROWS; k++)
                a_row[k] = a[i][k];
        if(i==0)
            RD_COL: for(int k = 0; k < MAT_B_ROWS; k++)
                b_copy[k][j] = b[k][j];
        Product: for(int k = 0; k < MAT_B_ROWS; k++) {
          tmp += a_row[k] * b_copy[k][j];
        }
        res[i][j] = tmp;
      }
    }
}
```

Figure 10: Final HLS Code

The Analysis view of the Performance of this code is shown in ***Figure 11*** below.

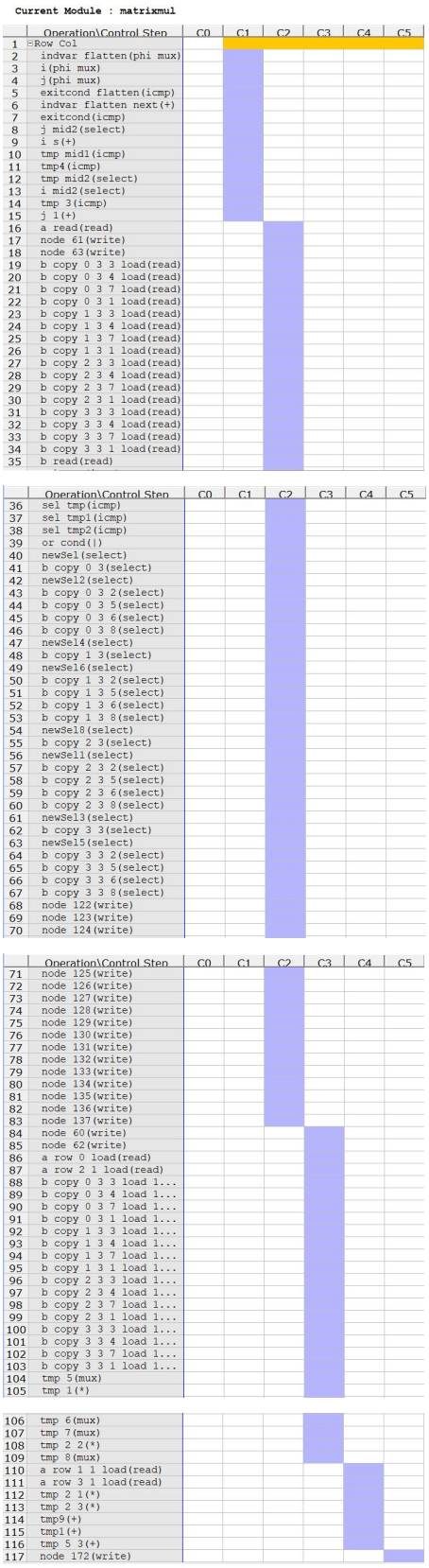


Figure 11: Analysis view of final solution timing performance

From this, we can see that no operation takes more than one cycle to complete, so pipelining with initiation interval 1 is entirely possible. The timing profile is shown in ***Figure 12***.

	Pipelined	Latency	Initiation Interval	Iteration Latency	Trip count
matrixmul	-	21	22	-	-
Row_Col	yes	19	1	5	16

Figure 12: Analysis timing performance profile for final design

As expected, the pipelined initiation interval is 1, with an iteration latency of 5. The overall initiation interval is still 22, however, and the overall latency is 21.

C/RTL Cosimulation Results

The new version of Vivado HLS did not allow me to run VHDL and Verilog RTL cosimulation simultaneously, but both passed and gave the latency of 21 shown in ***Figure 13***. I was not able to get an interval estimate from this.

Cosimulation Report for 'matrixmul'

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	Pass	21	21	21	NA	NA	NA
Verilog	NA	21	21	21	NA	NA	NA

Figure 13: C/RTL cosim results

SysGen Simulation Model

I exported the RTL for the final design described above as a System Generator token and simulated it there. My model is shown in ***Figure 14***.

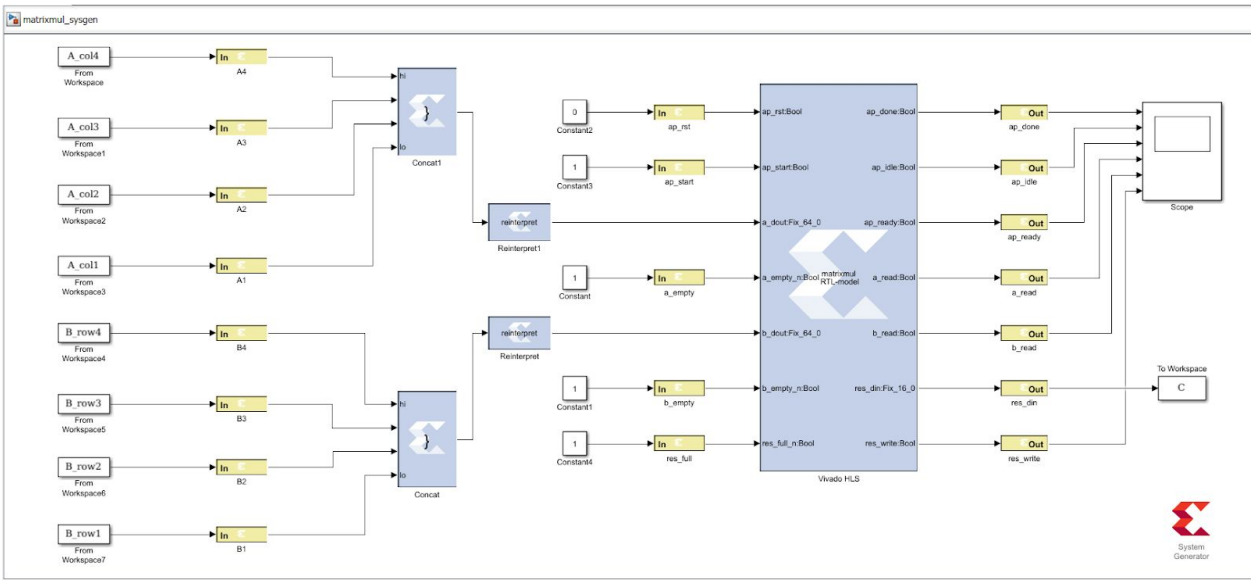


Figure 14: System Generator Model for Final Design

On the left, the columns of A and the rows of B stream in from the Workspace in parallel. Each element is reinterpreted as UFix_16_0 by the Gateway In, concatenated, and then reinterpreted as Fix_64_0 before being fed to the Vivado HLS block. In the case of A, this allows us to pass in a whole row at the same time; in the case of B, we are passing in a whole column at a time.

MATLAB Code & Simulation Procedure

The MATLAB code which formats the matrices in the workspace to be read into the SysGen model is shown in ***Figure 15*** below.

```
%% Load matrices
A = [1 2 8 4; 5 2 7 0; 9 10 4 2; 7 0 3 1];    % First set of matrices
B = [5 1 6 2; 0 1 2 8; 5 9 2 7; 0 5 6 3];
A1 = 4*ones(4,4);                                % Second set of matrices
B1 = [1 0 0 0; 0 2 0 4; 8 2 5 0; 3 4 6 2];

A_col1_val = [zeros(2,1); upsample(A(:,1),4); zeros(6,1); upsample(A1(:,1),4); zeros(6,1)];
A_col2_val = [zeros(2,1); upsample(A(:,2),4); zeros(6,1); upsample(A1(:,2),4); zeros(6,1)];
A_col3_val = [zeros(2,1); upsample(A(:,3),4); zeros(6,1); upsample(A1(:,3),4); zeros(6,1)];
A_col4_val = [zeros(2,1); upsample(A(:,4),4); zeros(6,1); upsample(A1(:,4),4); zeros(6,1)];

B_row1_val = [zeros(2,1); B(1,1:4)'; zeros(18,1); B1(1,1:4)'; zeros(18,1)];
B_row2_val = [zeros(2,1); B(2,1:4)'; zeros(18,1); B1(2,1:4)'; zeros(18,1)];
B_row3_val = [zeros(2,1); B(3,1:4)'; zeros(18,1); B1(3,1:4)'; zeros(18,1)];
B_row4_val = [zeros(2,1); B(4,1:4)'; zeros(18,1); B1(4,1:4)'; zeros(18,1)];

A_col1 = timeseries(A_col1_val);
A_col2 = timeseries(A_col2_val);
A_col3 = timeseries(A_col3_val);
A_col4 = timeseries(A_col4_val);

B_row1 = timeseries(B_row1_val);
B_row2 = timeseries(B_row2_val);
B_row3 = timeseries(B_row3_val);
B_row4 = timeseries(B_row4_val);
```

Figure 15: MATLAB code for formatting the input matrices (***loadmatrix.m***)

This code must be run prior to running the Simulink simulation. It splits the columns/rows of the matrices, upsamples them so that it provides the data as slowly as the HLS block required, concatenates the matrices with the appropriate delay in between (6 cycles), and converts them to timeseries for loading into the From Workspace block.

After running the simulation, the second section of MATLAB code shown in ***Figure 16*** can be run to extract the results from the raw output C and compare them the expected output. If the results are correct, the variable ***test_passed*** will be a logical 1.

```
%% Extract and format outputs

C1 = A*B;
C1_out = [C(6:9)';C(10:13)';C(14:17)';C(18:21)'];
C2 = A1*B1;
C2_out = [C(28:31)'; C(32:35)'; C(36:39)'; C(40:43)'];
test_passed = (sum(sum(C1 == C1_out))+sum(sum(C2 == C2_out))) == 32
```

Figure 16: MATLAB code for extracting and checking the simulation outputs (***loadmatrix.m***)

SysGen Simulation Results & Throughput

After running the MATLAB code and System Generator model shown in the sections above, I correctly obtained the expected output matrices, and ***test_passed*** was a logical 1. The raw output can be viewed in the attached file.

The first matrix result completed from time 5 to time 20, with one element output per cycle. The total latency was therefore 21 cycles. The second matrix result started to output at time 27 and continued until time 42. Thus, the maximum system throughput is 22 cycles.

For matrix-vector mode, no additional changes are necessary.

Hardware Cosim Results

The hardware cosim System Generator model is shown in ***Figure 17*** below.

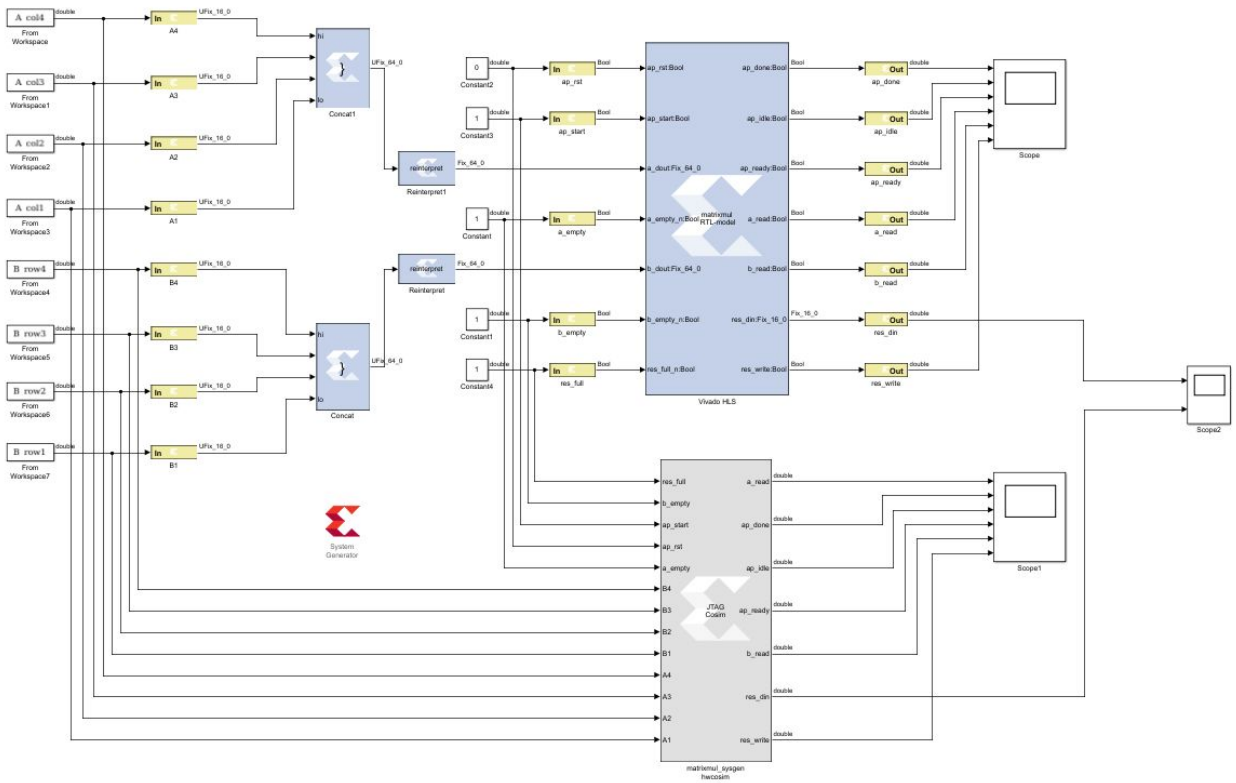


Figure 17: System Generator HW Cosim Model

The scope output for the hardware cosim is shown in ***Figure 18***. As expected, the output matches the simulation results.

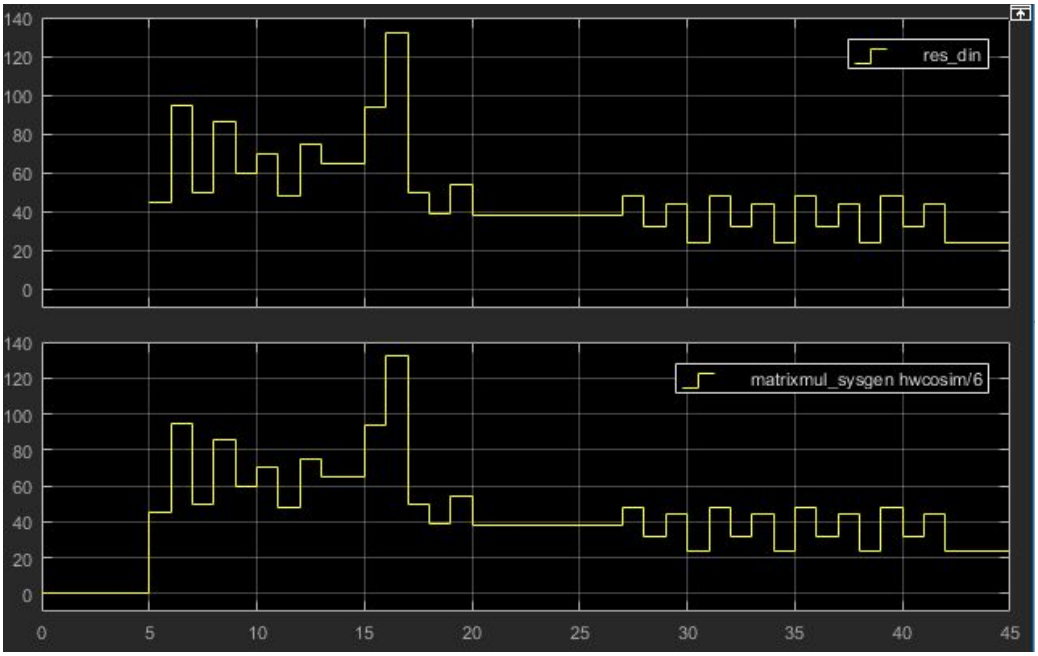


Figure 18: Hardware cosim results (bottom), SysGen simulation results (top)

Vivado Synthesis & Implementation Results

The post-implementation utilization report after RTL export indicated the following resource usage in ***Figure 19***.

Resource Usage

	VHDL
SLICE	123
LUT	121
FF	334
DSP	4
BRAM	0
SRL	0

Final Timing

	VHDL
CP required	10.000
CP achieved post-synthesis	7.350
CP achieved post-implementation	8.812

Timing met

Figure 19: Utilization report after RTL export

After synthesizing and implementing the SysGen model in Vivado, I obtained results very close to this, summarized in ***Table 1*** below. The timing report indicated that the WNS was 2.072 ns, which would permit a maximum clock rate of 126 MHz.

Table 1: Summary of resource utilization & timing for final design

Slices	104
Slice LUTs	123
Slice Registers	336
IOBs	156
DSP48s	4
Max clock rate	126 MHz

Conclusion

The purpose of this project was to explore another means of designing and optimizing a matrix multiplier. The question now arises of which method was more effective: a complete design in System Generator as in Project 2, or a Vivado HLS design exported to System Generator as in this project? A comparison of the two designs is shown in **Table 2**.

Table 2: Comparison of resource utilization & timing for Projects 2 and 3

	Project 2 Design	**Project 3 Design**
Slices	224	104
Slice LUTs	331	123
Slice Registers	988	336
IOBs	194	156
DSP48s	16	4
Max clock rate	338 MHz	126 MHz
Latency	6	21

From this, we see that the Project 2 design achieved a much lower latency and higher clock, but used significantly more resources. The Project 3 design used only 4 DSP48s, 336

Slice Registers, and 156 IOBs. Project 2, on the other hand, used 16 DSP48s, 988 Slice Registers, and 194 IOBs. This shows the inherent tradeoff of area versus latency. Probably the difference arises from the fact that Project 2 was optimized at a very low level, and we directly designed the control signals and data flow, so the timing was completely optimized. In Project 3, we did not design the control signals for the most part, and we optimized I/O and resource utilization much more, resulting in a worse latency.