Project 5: QR Decomposition of a 4x4 matrix using CORDIC arithmetic

Jennifer Hellar

ELEC 522 - Fall 2018

Introduction

In Project 4, I implemented a dual-mode CORDIC module in VivadoHLS that can perform either vector rotation (*Equations 1 & 2*) or arctangent and magnitude computations (*Equations 3 & 4*).

Rotation Mode:

$$a = cos(z) * x - sin(z) * y \quad (1)$$
$$b = sin(z) * x + cos(z) * y \quad (2)$$

Arctan Mode:

$$a = \sqrt{x^2 + y^2} \quad (3)$$
$$c = arctan(\tfrac{y}{x}) \quad (4)$$

The module has four inputs:

1. *mode* - boolean mode selection (rotation: 1, arctan: 0)
2. *x, y* - input vector
3. *z* - rotation mode: angle of rotation, arctan mode: 0

The three outputs are similar:

1. *a* - rotation mode: x value of resulting vector, arctan mode: magnitude of input vector
2. *b* - rotation mode: y value of resulting vector, arctan mode: 0
3. *c* - rotation mode: 0, arctan mode: angle of input vector

In this project, I imported that module into System Generator and used it to build an array of arctan & rotation blocks to perform QR decomposition. I chose to implement the alternative triangular systolic array shown in ***Figure 1*** below.

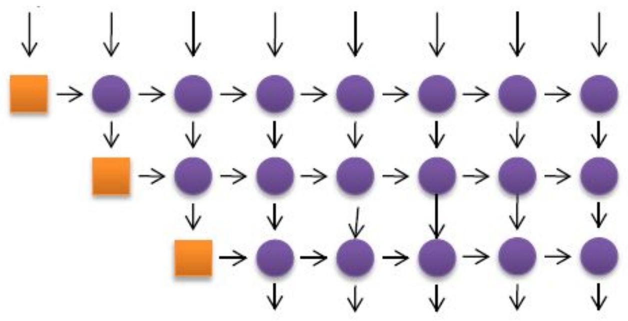


Figure 1: Alternative triangular systolic array

In this case, the orange blocks are ATAN (vectoring) units, and the purple circles are SIN/COS (rotational) units. The left four input data streams are the reversed columns of our matrix. The right four input data streams are the reversed columns of the identity matrix.

Each ATAN block performs the function of zeroing out the necessary elements of each column of the input matrix and propagating the required rotation angles to Rotation blocks. The Rotation blocks take the input angles and apply them to the appropriate rows/columns of the current matrix. At the end, the R matrix comes out as shown in ***Figure 2***, saved in the internal registers of the corresponding blocks.

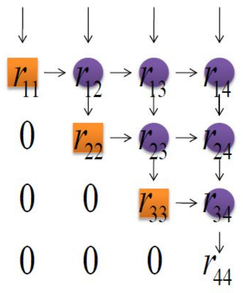


Figure 2: R matrix output in alternative triangular array

Since the rotations are applied to the identity matrix also, the output Q matrix comes out transposed in the blocks shown in ***Figure 3***.

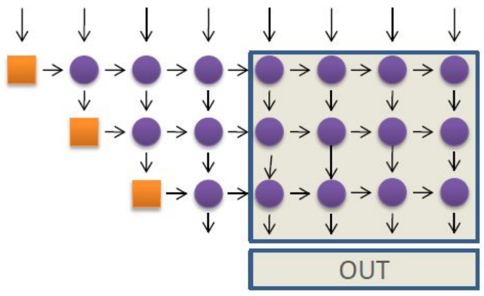


Figure 3: Q matrix output in alternative triangular array

This architecture uses more area than the normal triangular array since it requires the additional 16 rotation blocks to capture the Q matrix, but since each block is in dedicated ATAN or Rotation mode, the control logic is much simpler.

Arctangent Block

The ATAN blocks rotate the input column vectors [a; b] to [R; 0]; they save the resulting R to an internal register and output the angle to the rotation blocks to the right. At the end of the computation, the ATAN blocks hold the main diagonal R values in their internal registers. The SysGen design for this block is shown in ***Figure 2*** below.

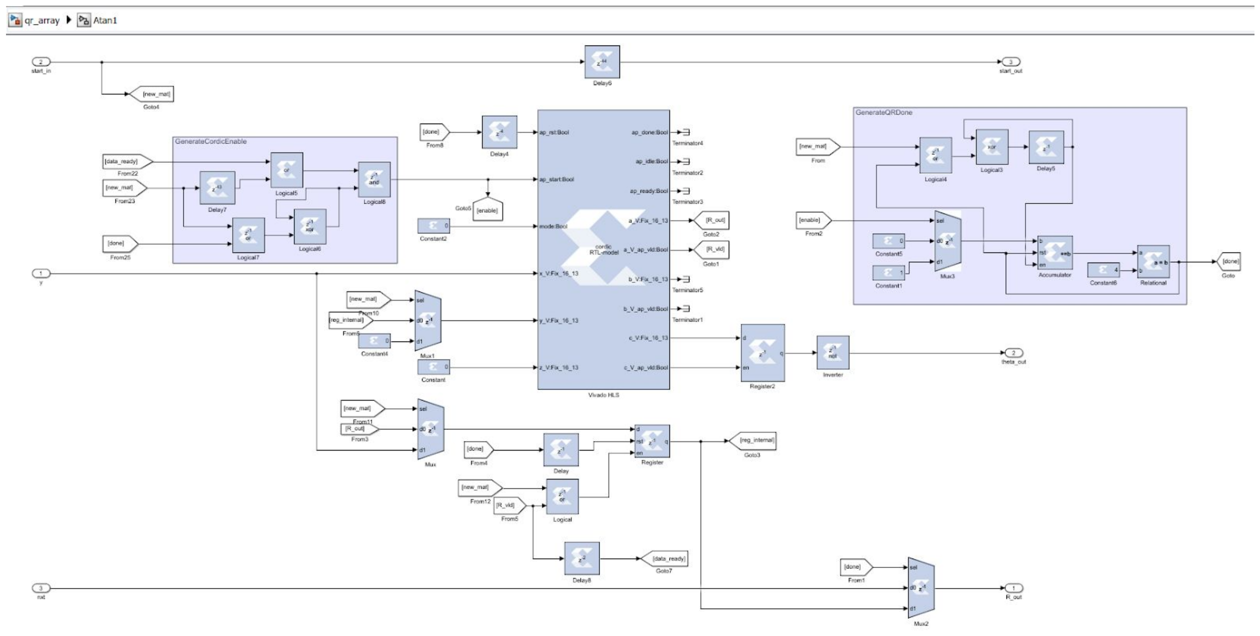


Figure 2: ATAN block implementation in SysGen

The first y input is stored into the internal register whose value is *reg_internal* in my block diagram above. That register takes a new value only when triggered by a new matrix computation (to save the first valid input) or when the output of the CORDIC module is ready. When a value has been successfully saved into the internal register the value *data_ready* goes high for one cycle. If the ATAN block is not finished with its computations, this triggers the CORDIC block again to compute the next output. This zeros out elements of A from the bottom of the column upwards.

When the block has finished its computations (3 for the top row, 2 for the second, etc.), the signal *done* goes high for one cycle, which causes the block to output the current R value in the register, clears the internal register one cycle later, stops triggering the CORDIC module, and resets the CORDIC module. This effectively resets and clears the ATAN block until another matrix is inputted. When not actively outputting an R value, the block outputs values that have been passed to it by the next block over.

A. CORDIC “Enable” Signal Generation Subsystem

The ATAN and Rotation blocks each contain a subsystem to generate the enable signal for the CORDIC block, shown above in the ATAN block and enlarged in ***Figure 3*** below.

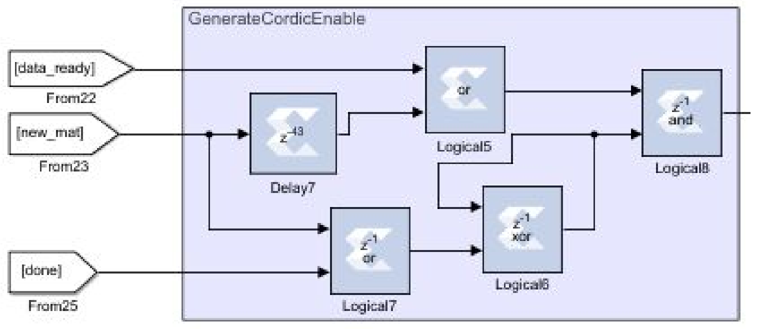


Figure 3: CORDIC “enable” signal generator

The core part of the subsystem is the finite state machine defined by the XOR loop. This FSM toggles its output in response to a pulse. It will output high when the block is actively computing (*new_mat* is a pulse which propagates through the array indicating the start of a new computation), and it will output low after the block is finished (*done* pulses). The *enable* output of the subsystem will be high when the block is actively computing and the next set of data is ready.

B. “Done” Signal Generation Subsystem

Both blocks also contain a subsystem to generate the computation “done” signal, as shown in ***Figure 4*** below.

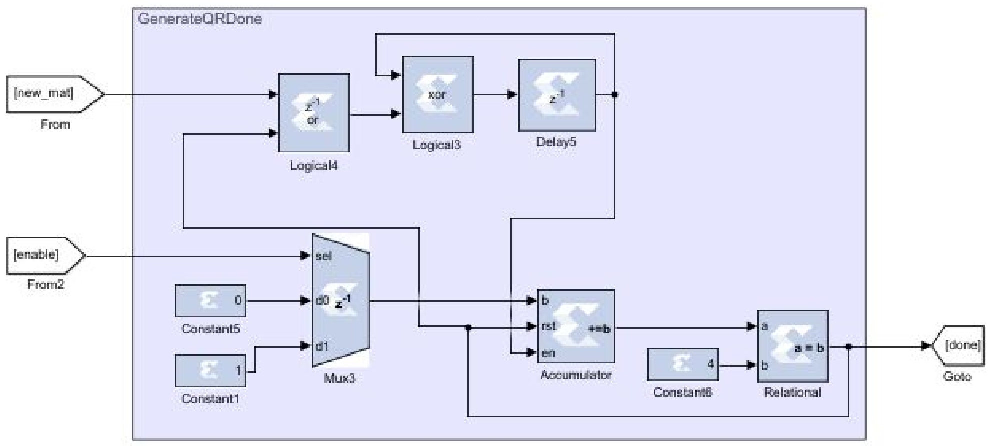


Figure 4: Block “done” signal generator

This block counts the number of times that “enable” has gone high i.e. how many computations have been started. If it has finished the correct number of computations (3 for the top row) and is starting another one, the subsystem outputs the *done* pulse which stops and resets the system, including disabling and resetting the accumulator.

Rotation Block

The Rotation blocks take in a data stream and interpret the first pair as a vector and each subsequent input as the new “x” value for the vector with the “y” value being the internal register. They are responsible for applying the angles generated by the ATAN blocks to the rest of the matrix and to the identity matrix. The SysGen design is shown in ***Figure 5*** below.

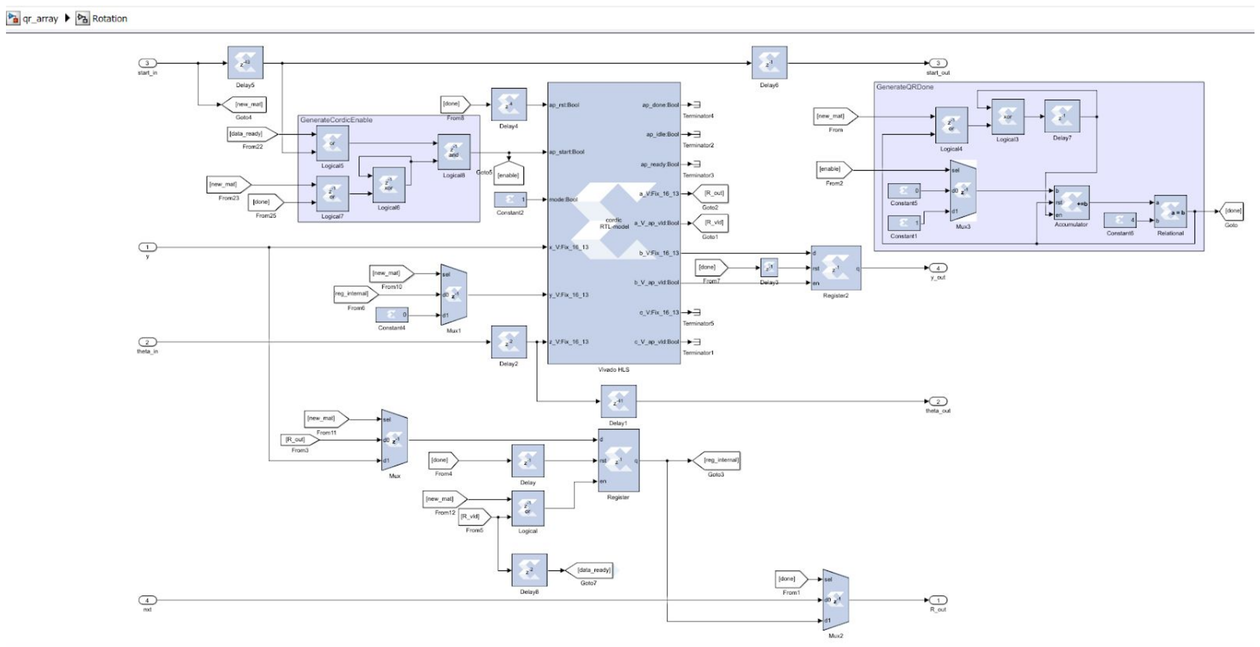


Figure 5: Rotation block implementation in SysGen

This block contains the same Enable and Done subsystems and is constructed much like the ATAN block. The only major difference is the additional input *theta* and output *y_out*. The input *theta* and *start* are propagated through with the appropriate delays.

Triangular Array

The overall SysGen design for the triangular array is shown in ***Figure 6*** below.

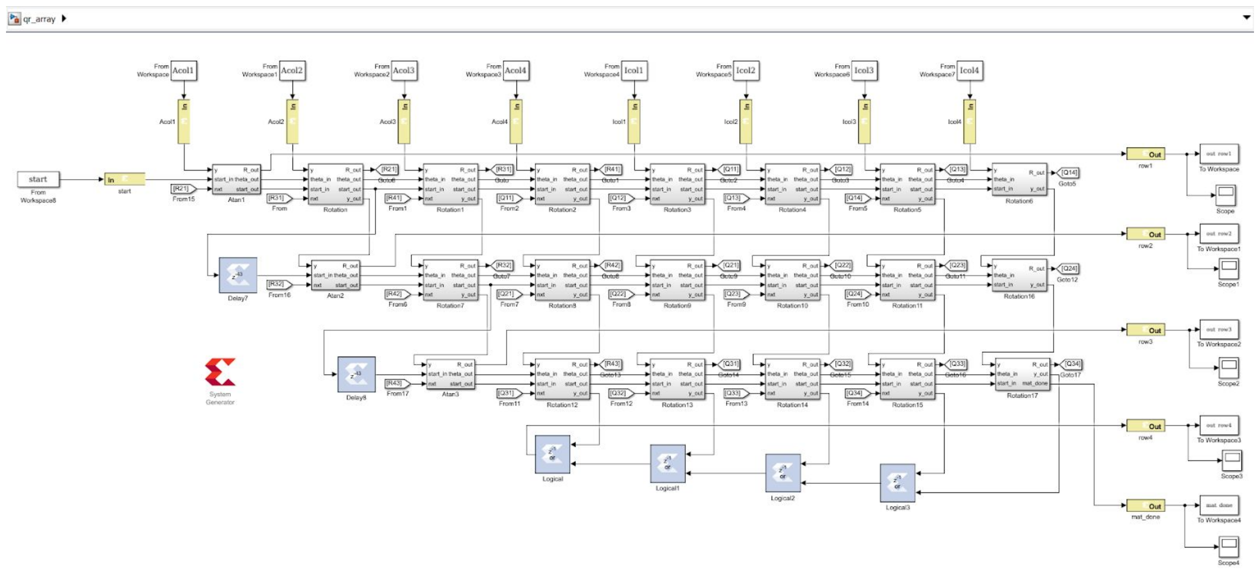


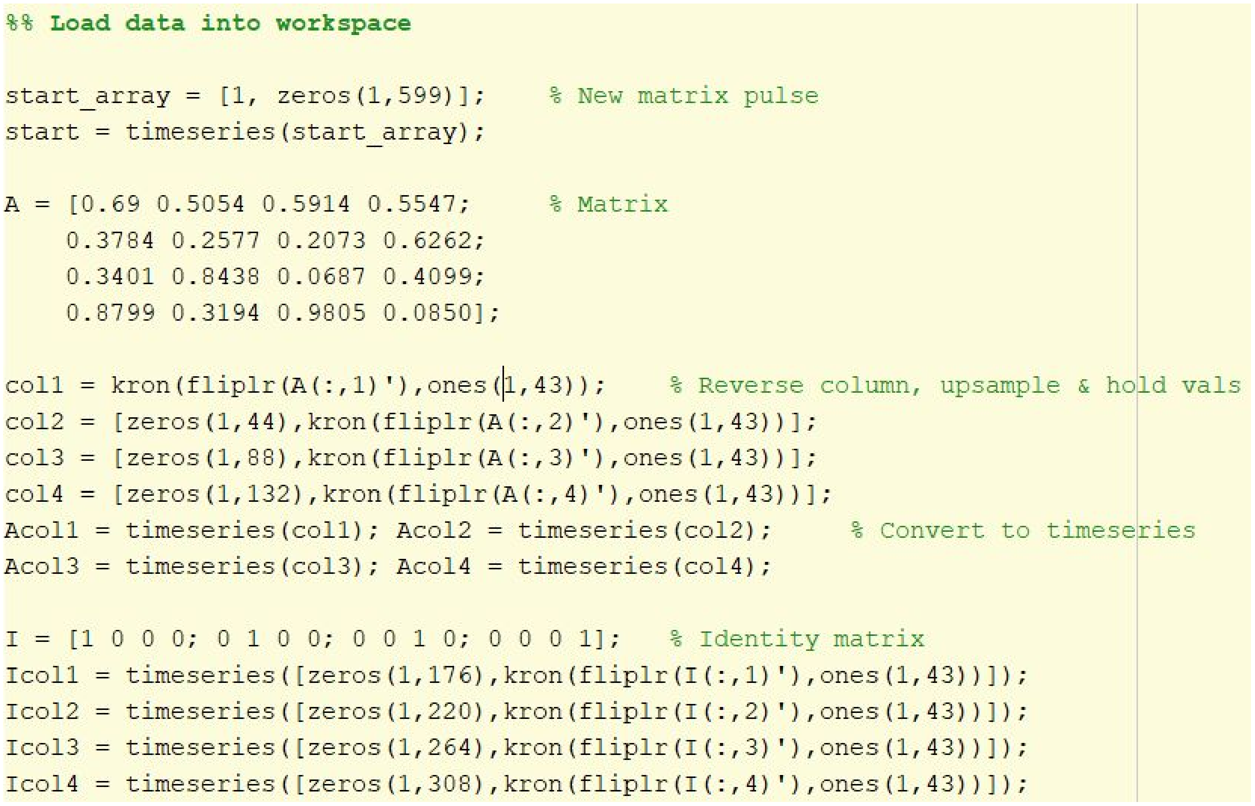
Figure 6: Triangular array implementation SysGen

The matrices are inputted from above, the start signal from the left. Data is unloaded systolically to the left, since the leftmost blocks are directly connected to the output gateways. Each row is delayed 43 cycles behind the one above it, since it must wait for the data to be computed and passed through. The final row of ATAN and Rotation blocks computes the third and fourth row of the output, which is why the fourth row of the design is implemented only as an OR-gate tree.

Since each block clears and resets itself after finishing, the next matrix can begin to be computed while the array is still finishing the first matrix. A given block takes 175 cycles to complete, and 1 cycle to output the result and reset. Therefore, it can accept a new matrix input every 177 cycles. The overall array has an initiation interval of 578 (to complete the first matrix decomposition), and a throughput of 177.

Matlab Code & Simulation Procedure

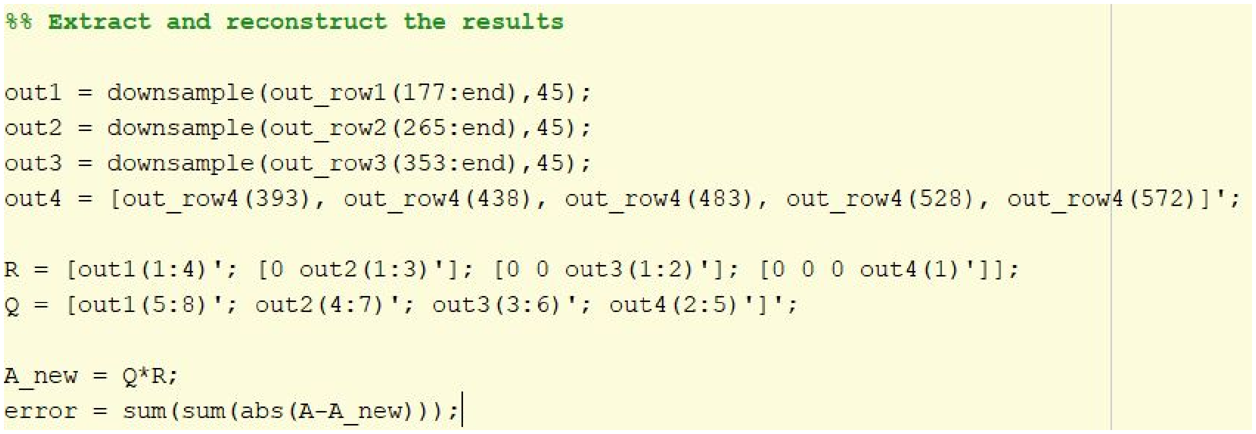
The simulation Matlab code is found in *load_matrix.m*. To simulate the model, run either the first section to perform one matrix decomposition or the second section to perform two matrix decompositions. The matrix (provided by Dr. Cavallaro) is the same in both cases. Run the Simulink model with a simulation time of 567, and then run the final section of the Matlab code to extract the results, reconstruct the matrix, and compute the error.



```
%% Load data into workspace

start_array = [1, zeros(1,599)];    % New matrix pulse
start = timeseries(start_array);

A = [0.69 0.5054 0.5914 0.5547;      % Matrix
    0.3784 0.2577 0.2073 0.6262;
    0.3401 0.8438 0.0687 0.4099;
    0.8799 0.3194 0.9805 0.0850];

col1 = kron(fliplr(A(:,1)'),ones(1,43));     % Reverse column, upsample & hold vals
col2 = [zeros(1,44),kron(fliplr(A(:,2)'),ones(1,43))];
col3 = [zeros(1,88),kron(fliplr(A(:,3)'),ones(1,43))];
col4 = [zeros(1,132),kron(fliplr(A(:,4)'),ones(1,43))];
Acol1 = timeseries(col1); Acol2 = timeseries(col2);     % Convert to timeseries
Acol3 = timeseries(col3); Acol4 = timeseries(col4);

I = [1 0 0 0; 0 1 0 0; 0 0 1 0; 0 0 0 1];    % Identity matrix
Icol1 = timeseries([zeros(1,176),kron(fliplr(I(:,1)'),ones(1,43))]);
Icol2 = timeseries([zeros(1,220),kron(fliplr(I(:,2)'),ones(1,43))]);
Icol3 = timeseries([zeros(1,264),kron(fliplr(I(:,3)'),ones(1,43))]);
Icol4 = timeseries([zeros(1,308),kron(fliplr(I(:,4)'),ones(1,43))]);
```

Figure 7: Matlab code to load a matrix

As shown above in ***Figure 7***, each input column is delayed 44 cycles longer than the previous one, and each input value is held constant for 43 cycles.



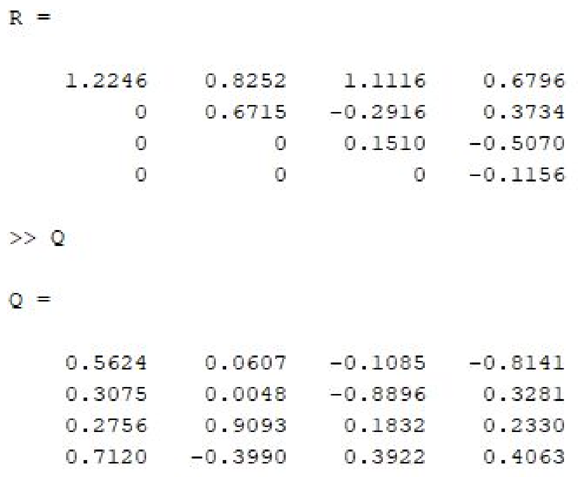
```
%% Extract and reconstruct the results

out1 = downsample(out_row1(177:end),45);
out2 = downsample(out_row2(265:end),45);
out3 = downsample(out_row3(353:end),45);
out4 = [out_row4(393), out_row4(438), out_row4(483), out_row4(528), out_row4(572)]';

R = [out1(1:4)'; [0 out2(1:3)']; [0 0 out3(1:2)']; [0 0 0 out4(1)']];
Q = [out1(5:8)'; out2(4:7)'; out3(3:6)'; out4(2:5)']';

A_new = Q*R;
error = sum(sum(abs(A-A_new)));
```

Figure 8: Matlab code to extract the results

The first output of row 1 occurs at time 177, the first output of row 2 eighty-eight cycles later, and so on. The outputs of row 4 are somewhat different because they are computed simultaneously with those of row 3.

SysGen Simulation Results

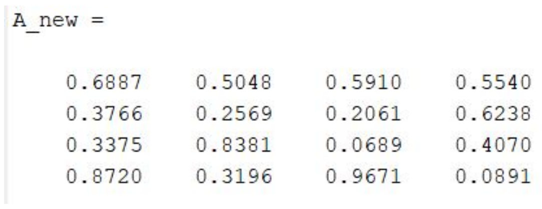
After completing the above simulation procedure, I obtained the QR matrices as shown in ***Figure 9***.



```
R =

    1.2246    0.8252    1.1116    0.6796
         0    0.6715   -0.2916    0.3734
         0         0    0.1510   -0.5070
         0         0         0   -0.1156

>> Q

Q =

    0.5624    0.0607   -0.1085   -0.8141
    0.3075    0.0048   -0.8896    0.3281
    0.2756    0.9093    0.1832    0.2330
    0.7120   -0.3990    0.3922    0.4063
```

Figure 9: Output QR matrices

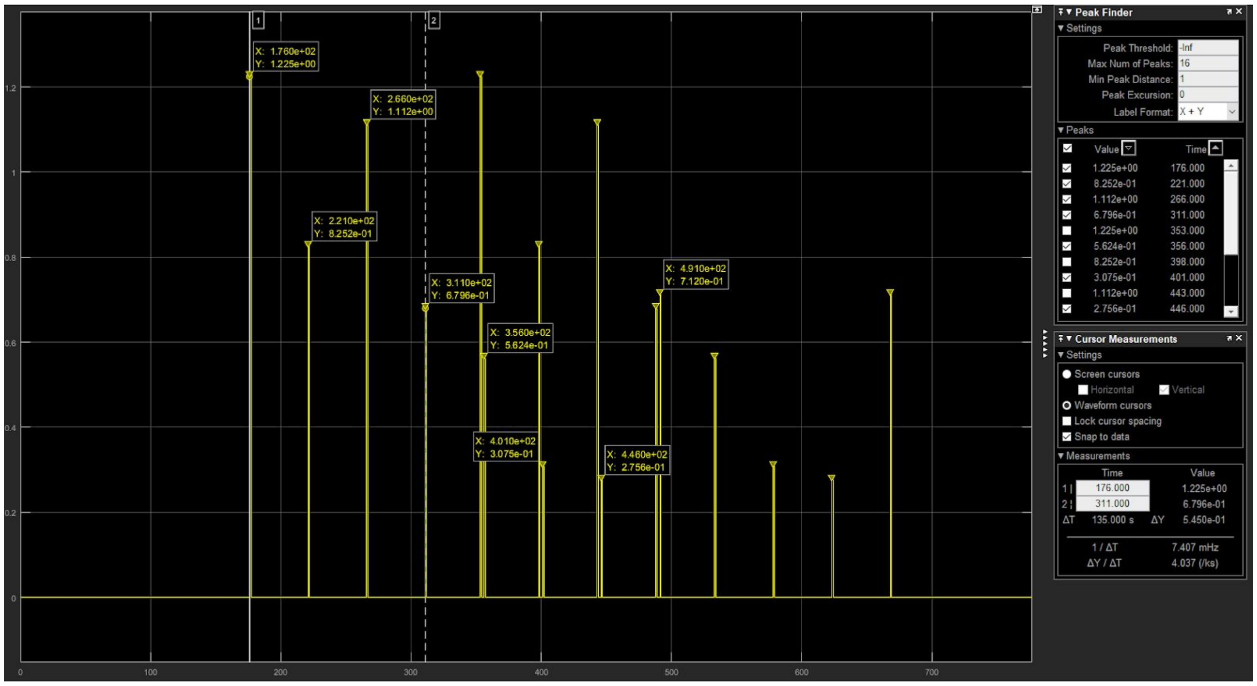
Multiplied together, these formed the reconstructed A matrix as shown in ***Figure 10***, with an error of 0.0462.



```
A_new =

    0.6887    0.5048    0.5910    0.5540
    0.3766    0.2569    0.2061    0.6238
    0.3375    0.8381    0.0689    0.4070
    0.8720    0.3196    0.9671    0.0891
```

Figure 10: Reconstructed A matrix

I tested the reset functionality and array throughput by inputting the same matrix a second time and received the following output for the first row of outputs (first row of R and first column of Q).



Peak Finder

Settings

Peak Threshold: -Inf
Max Num of Peaks: 16
Min Peak Distance: 1
Peak Excursion: 0
Label Format: X + Y

Peaks

✓	Value	Time
✓	1.225e+00	176.000
✓	8.252e-01	221.000
✓	1.112e+00	266.000
✓	6.796e-01	311.000
	1.225e+00	353.000
✓	5.624e-01	356.000
	8.252e-01	398.000
✓	3.075e-01	401.000
	1.112e+00	443.000
✓	2.756e-01	446.000

Cursor Measurements

Settings

● Screen cursors
Horizontal ✓ Vertical
Waveform cursors
Lock cursor spacing
✓ Snap to data

Measurements

	Time		Value
1	176.000		1.225e+00
2	311.000		6.796e-01
ΔT	135.000 s	ΔY	5.450e-01
1 / ΔT			7.407 mHz
ΔY / ΔT			4.037 (/ks)

Figure 11: Simulation output for two matrices (first row of results)

The tagged peaks correspond to the output for the first matrix input, the untagged peaks to the output for the second matrix input. From this, we see the benefits of the systolic array structure, which allows us to pipeline the computations.

Hardware Cosim Results

My hardware cosimulation model is shown in ***Figure 12*** below.

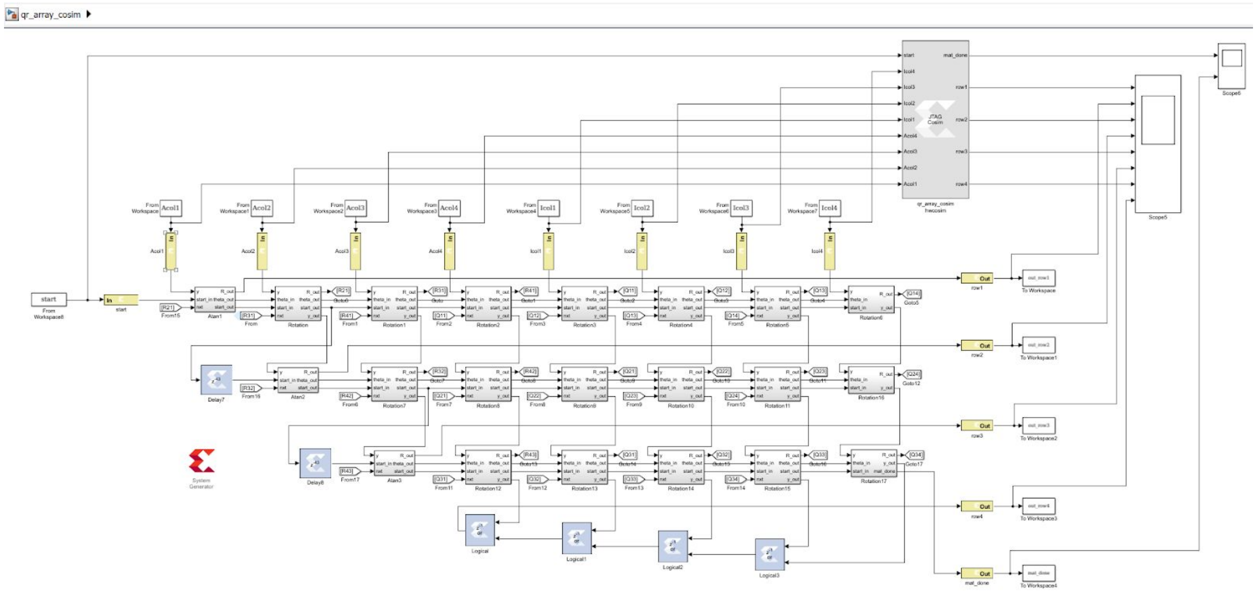


Figure 12: Hardware cosim model

The scope outputs after running the above model are given in ***Figure 13*** and ***Figure 14***.

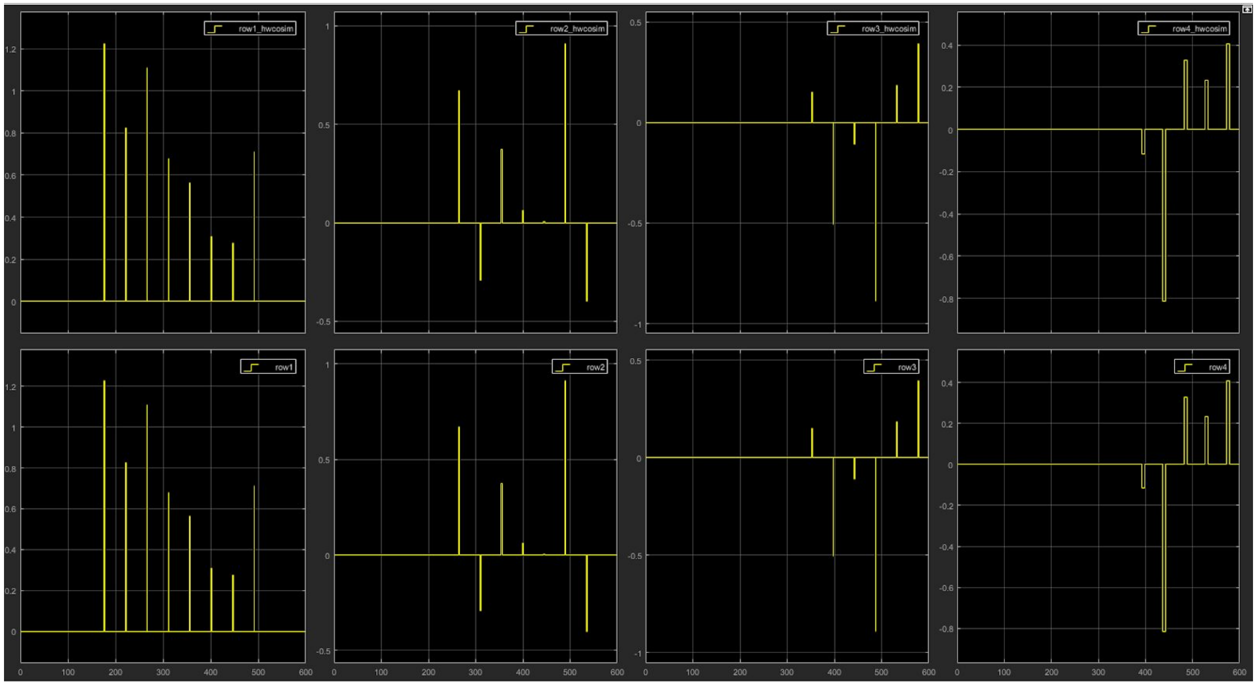


Figure 12: Hardware cosim row results (top) and software row results (bottom)

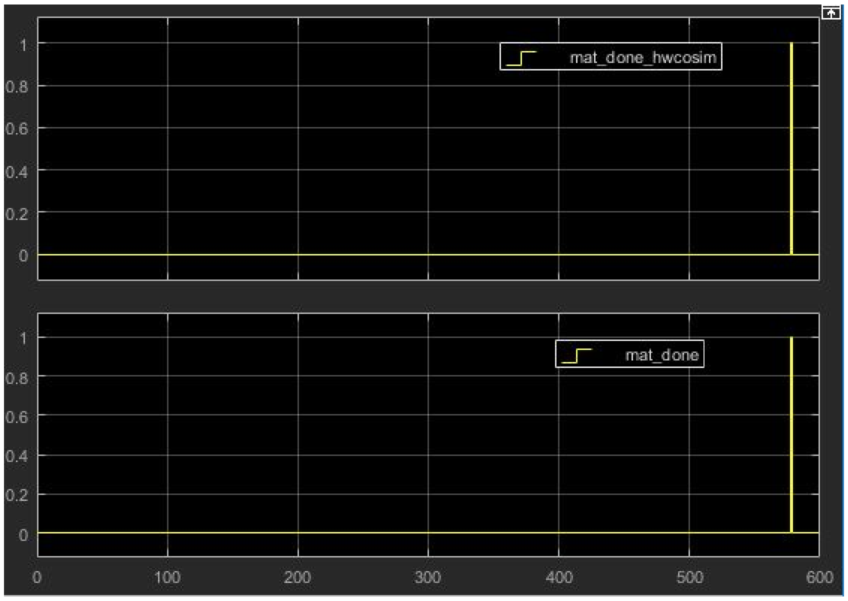


Figure 13: Hardware cosim "done" result (top) and software result (bottom)

As expected, the hardware and software results are identical.

Vivado Synthesis & Implementation Results

I exported my SysGen design into Vivado and ran synthesis and implementation with the following results.

Table 1: Vivado synthesis and implementation utilization and timing results

LUTs	9869 (18.55%)
FFs	5652 (5.31%)
Slices	2991 (22.5%)
I/O	195
DSPs	0
WNS (ns)	2.270
Max clock (MHz)	129