ELEC522 – Fall 2022

Project 5: QR Decomposition using CORDIC arithmetic for a 4 x 4 matrix

By

Hsuan-You (Shaun) Lin

Rice University, Houston, TX

Nov. 29, 2022

Supervised by

Dr. Joseph R. Cavallaro

Contents

List of Figures

Description of QRD design architecture:

QR decomposition system architecture:

In this project, I implemented the QR decomposition system through the architecture as shown in *Figure* 1 below, blue squares represent ArcTan mode, black circles represent Rotation mode, Input matrix will be loaded into the system by columns in a systolic manner and unloaded the results from left to right.

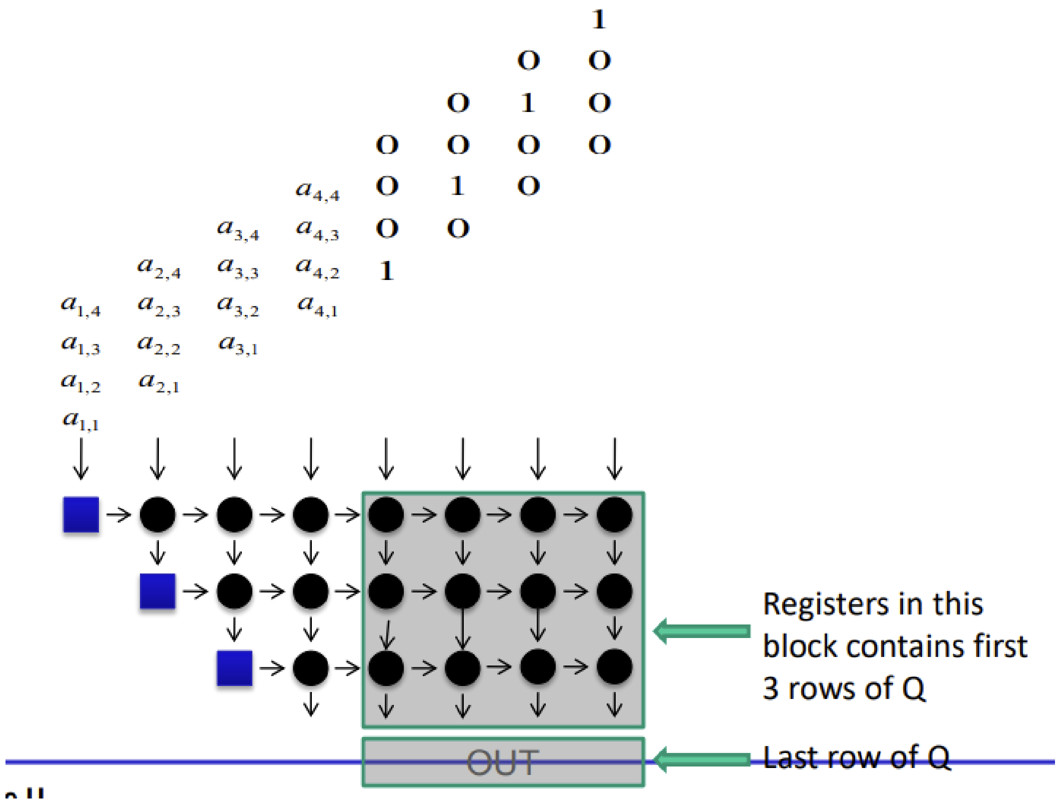
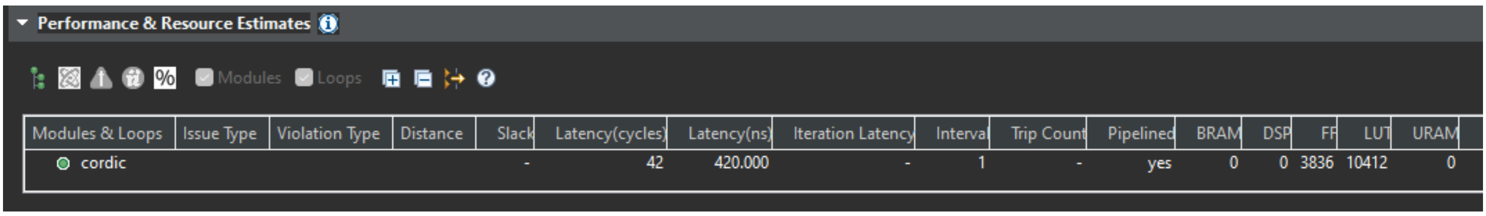


Figure 1. My QR decomposition system architecture

Vitis HLS CORDIC module's performance & Resource Estimates:

We will use the CORDIC module from the previous project to complete the ArcTan mode and Rotation mode operations. *Figure* 2 shows the final synthesis results from the Vitis HLS CORDIC module, we can see that **42 latency** and 10412 LUT are obtained after the Pipeline optimization, where 42 latency will affect the clock cycle in the Model Composer later.



Performance & Resource Estimates

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
cordic				-	42	420.000	-	1	-	yes	0	0	3836	10412	0

Figure 2. Vitis HLS synthesis results

Arctan mode & Rotation mode:

In my CORDIC module, if modes = 1 will operate in the Arctan mode as shown in *Figure* 3, when modes = 0 it will operate in the Rotation mode as shown in *Figure* 4. As you can see, I set the delay block for rst control signal in every subsystem, and also set delay block before the theta_in port, delay blocks use 42, as I mentioned earlier, because the CORDIC module consumes 42 latency. From the lecture's slide, professor told us to store the first valid input in the internal register, for each subsequent input, do the operation and store the result R, then pass the other calculated output to the next block. Finally, I used Mux block implements the multiplexer, to store the output results.

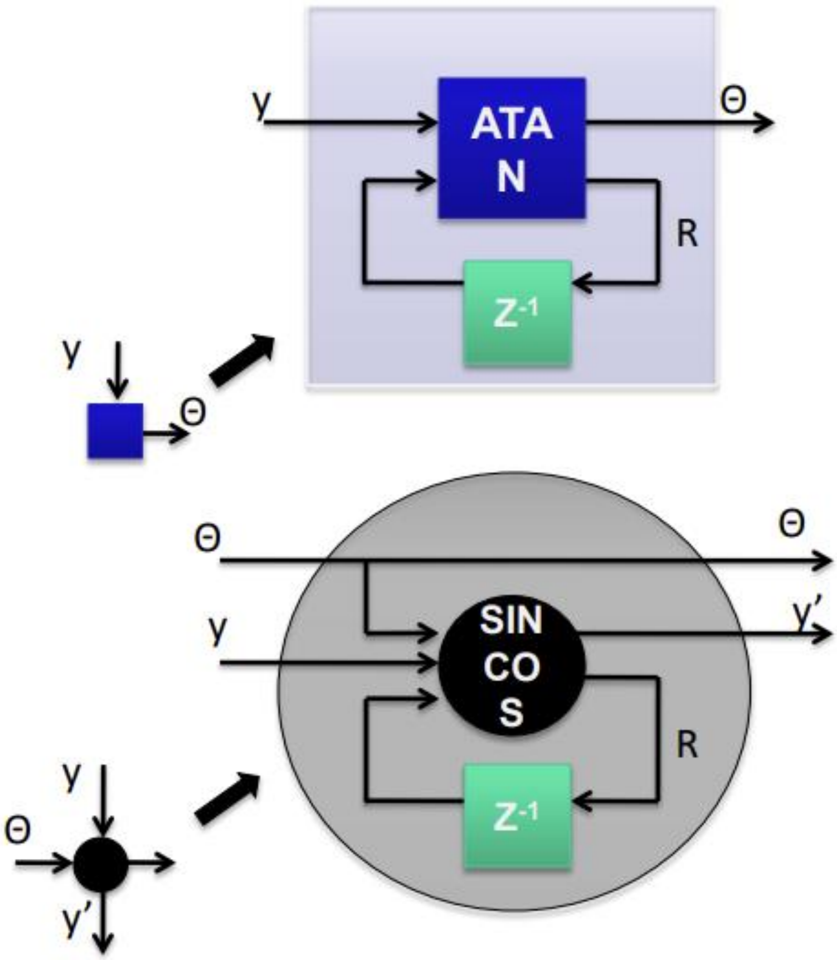


Figure 3. ArcTan block and Rotation block's design

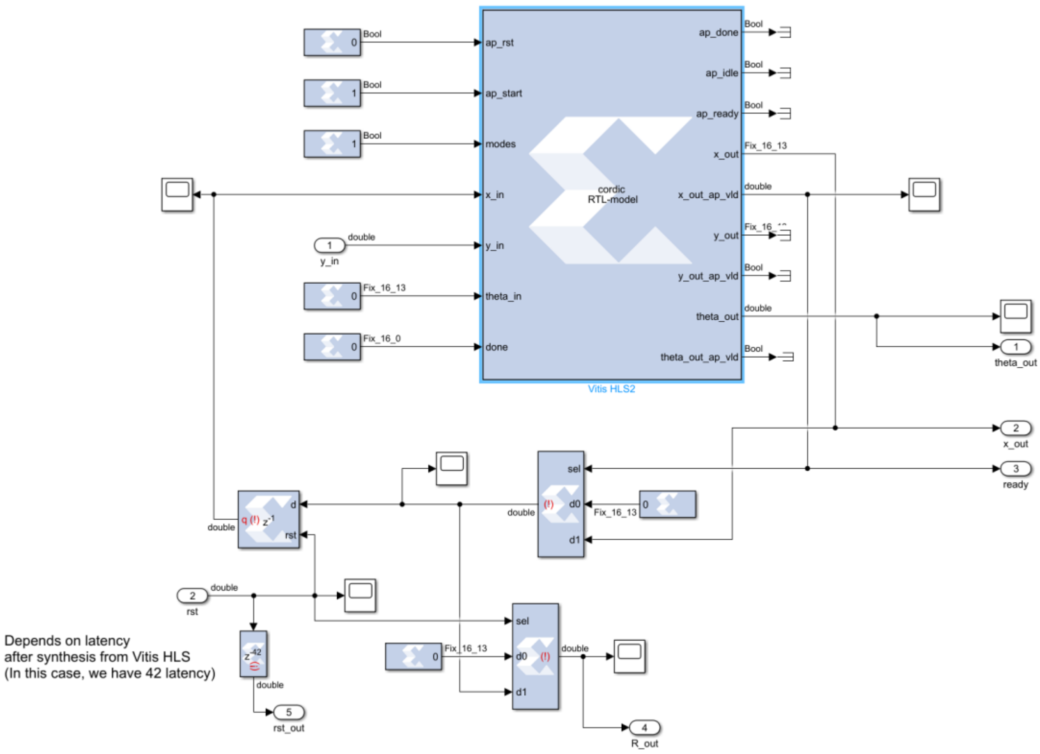


Figure 4. ArcTan mode subsystem in Model Composer

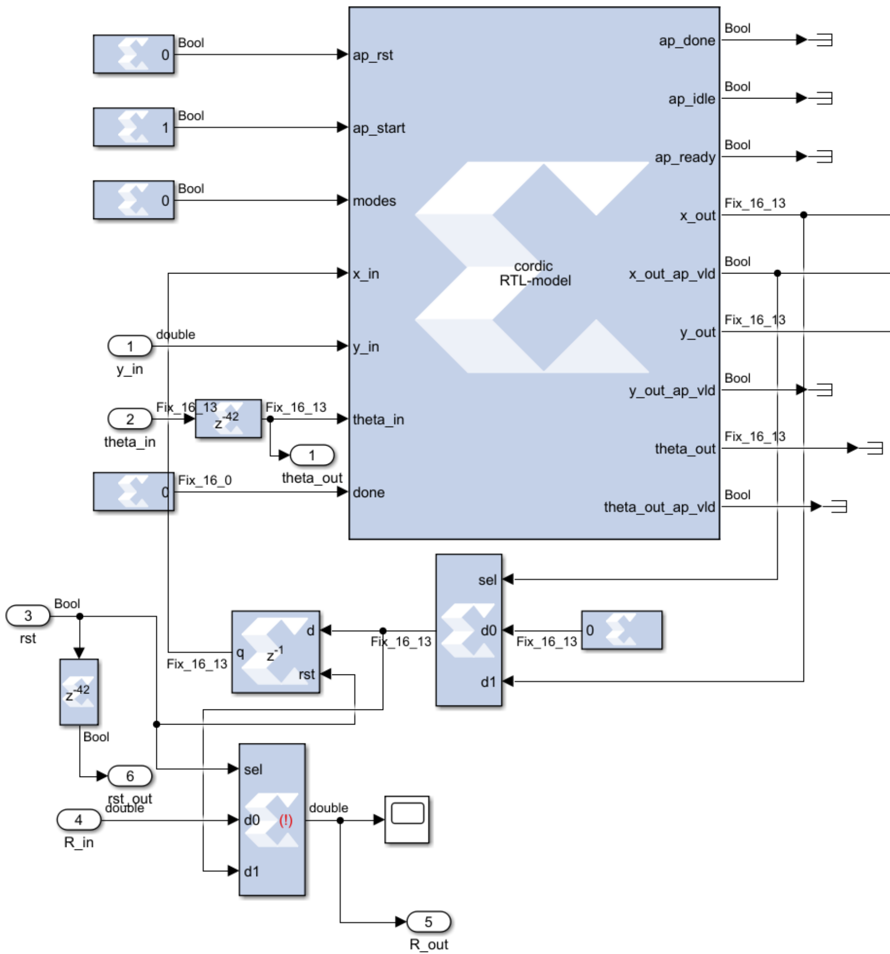


Figure 5. Rotation mode subsystem in Model Composer

Model Composer model using the Vitis HLS block and testing results:

QR Decomposition system architecture:

With the concepts in the above section, I implemented the QR Decomposition system in Model Composer as shown in *Figure* 6, and this system could handle continuous matrix signals, here I make two matrices as an example, it can be found from my m_code “QRD_Input.m” as shown in *Figure* 7, input matrices, Identity matrix and control signal rst are visualized in the *Figure* 8.

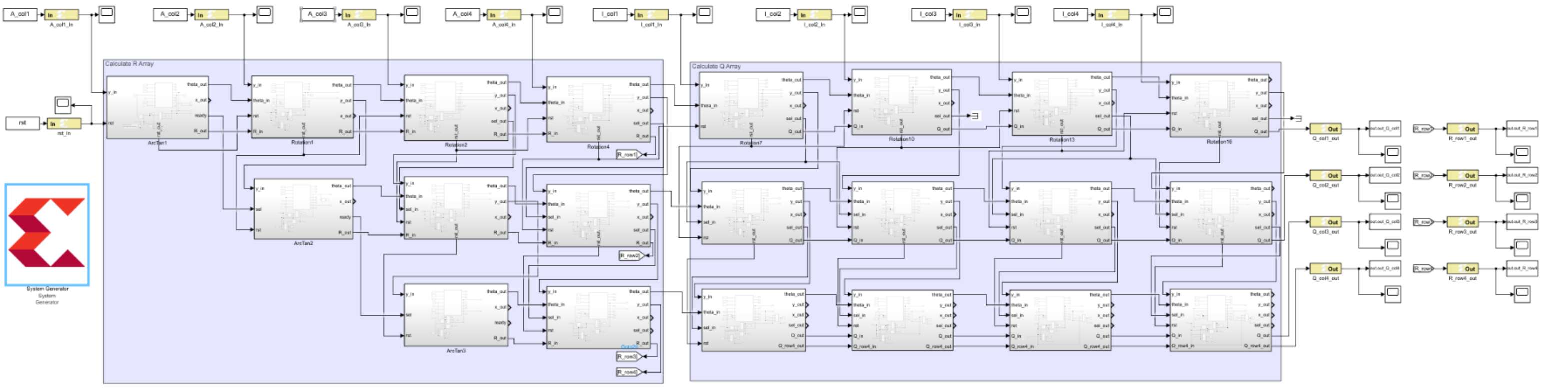


Figure 6. QR Decomposition system architecture implementation

QRD_Input.m QRD_Output.m +

```
% Clears all variables previously assigned.
clear
% Closes all graphs and windows open.
close all

latency = 42;   % check the latency value from Vitis HLS synthesis results.
gap = (latency*5) + 4;

rst_val = [zeros(1,(latency*4)+3), 1, zeros(1,gap), 1, zeros(1,latency)];
rst = timeseries(rst_val);

A1 = [0.6900 0.5054 0.5914 0.5547;     % A1 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];

A2 = [0.3564 0.7160 0.9118 0.0091;     % A2 Matrix
      0.8255 0.3252 0.0785 0.0412;
      0.6638 0.5491 0.9037 0.3417;
      0.7119 0.4366 0.7662 0.6734];

a1_1 = [A1(4,1), zeros(1,latency), A1(3,1), zeros(1,latency), A1(2,1), zeros(1,latency), A1(1,1), zeros(1,(latency*2)+1)];
a1_2 = [A2(4,1), zeros(1,latency), A2(3,1), zeros(1,latency), A2(2,1), zeros(1,latency), A2(1,1), zeros(1,latency)];
a2_1 = [zeros(1,latency), A1(4,2), zeros(1,latency), A1(3,2), zeros(1,latency), A1(2,2), zeros(1,latency), A1(1,2), zeros(1,(latency*2)+1)];
a2_2 = [A2(4,2), zeros(1,latency), A2(3,2), zeros(1,latency), A2(2,2), zeros(1,latency), A2(1,2), zeros(1,latency)];
a3_1 = [zeros(1,latency*2), A1(4,3), zeros(1,latency), A1(3,3), zeros(1,latency), A1(2,3), zeros(1,latency), A1(1,3), zeros(1,(latency*2)+1)];
a3_2 = [A2(4,3), zeros(1,latency), A2(3,3), zeros(1,latency), A2(2,3), zeros(1,latency), A2(1,3), zeros(1,latency)];
a4_1 = [zeros(1,latency*3), A1(4,4), zeros(1,latency), A1(3,4), zeros(1,latency), A1(2,4), zeros(1,latency), A1(1,4), zeros(1,(latency*2)+1)];
a4_2 = [A2(4,4), zeros(1,latency), A2(3,4), zeros(1,latency), A2(2,4), zeros(1,latency), A2(1,4), zeros(1,latency)];
A_col1 = timeseries([a1_1, a1_2]);
A_col2 = timeseries([a2_1, a2_2]);
A_col3 = timeseries([a3_1, a3_2]);
A_col4 = timeseries([a4_1, a4_2]);

I = [1 0 0 0;   % Identity matrix
     0 1 0 0;
     0 0 1 0;
     0 0 0 1];

i1 = [zeros(1,latency*4), I(1,1), zeros(1,gap), I(1,1), zeros(1,latency)];
i2 = [zeros(1,(latency*6)+1), I(2,2), zeros(1,gap), I(2,2), zeros(1,latency)];
i3 = [zeros(1,(latency*8)+2), I(3,3), zeros(1,gap), I(3,3), zeros(1,latency)];
i4 = [zeros(1,(latency*10)+3), I(4,4), zeros(1,gap), I(4,4), zeros(1,latency)];
I_col1 = timeseries(i1);
I_col2 = timeseries(i2);
I_col3 = timeseries(i3);
I_col4 = timeseries(i4);
```

Figure 7. QRD_Input.m m_code for input matrices

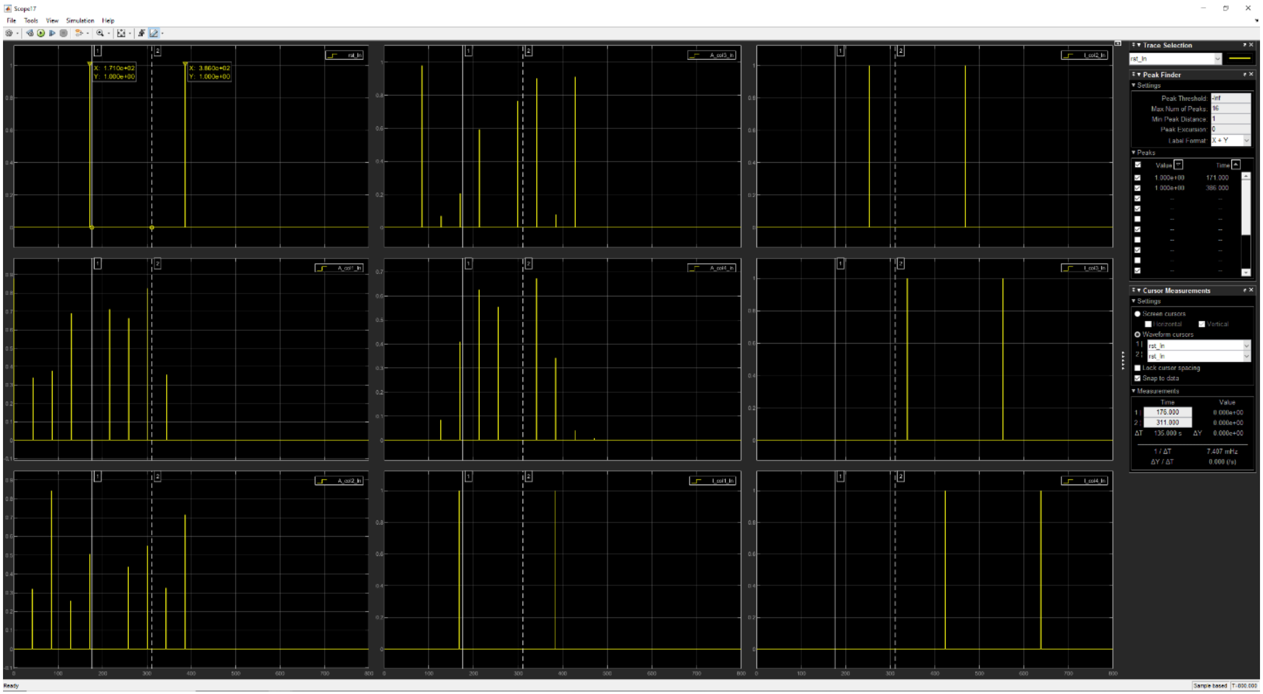


Figure 8. Input signal visualization from workspace

Testing results:

Figure 9 shows the output QR matrices results, it will store the QR array to workspace, and I wrote the “QRD_Output.m” m_code to print the final matrices to command window as shown in *Figure* 10, it could compare the result from the *Figure* 11, which is the screenshot from the QR decomposition online calculator website. Despite there are some positive and negative symbol problems here, the error in the calculation is mostly less than **0.05**.

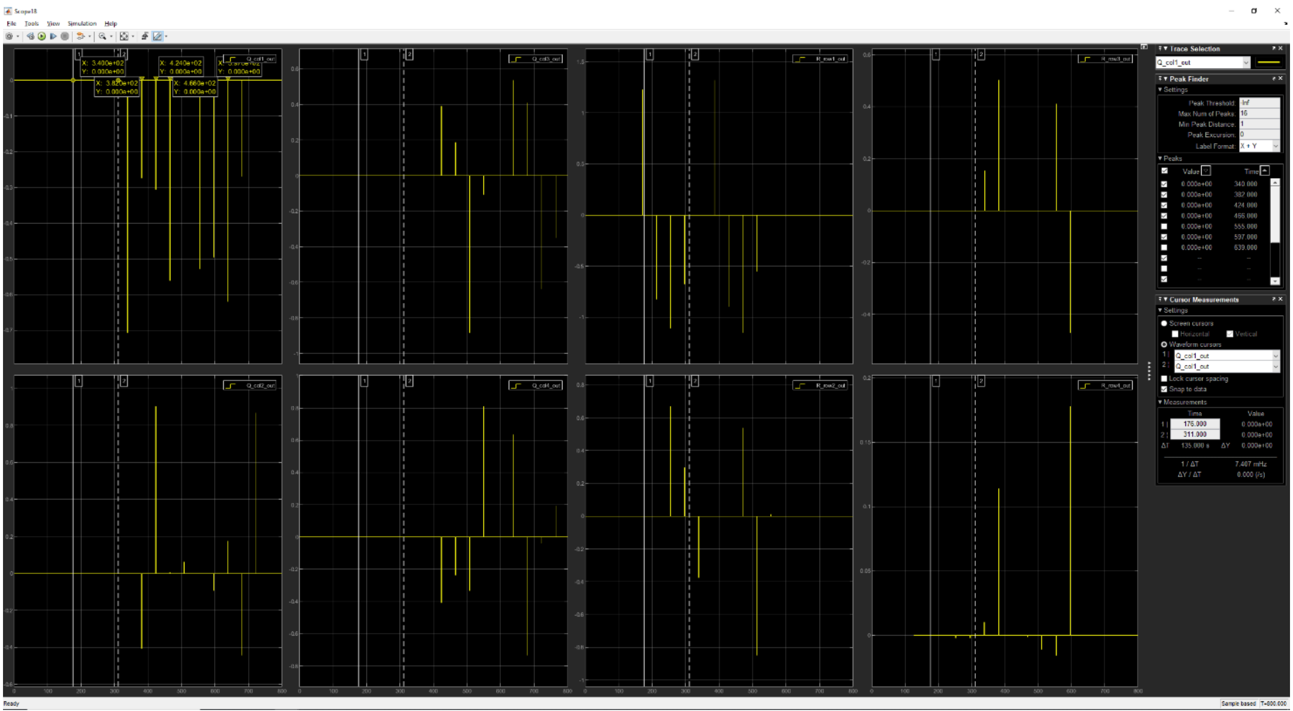


Figure 9. Output QR matrices signal visualization

```
Command Window
>> QRD_Input
>> QRD_Output
Q1:
   -0.5612    0.0613   -0.1067    0.8098
   -0.3063    0.0037   -0.8832   -0.3312
   -0.2737    0.9028    0.1866   -0.2367
   -0.7069   -0.4047    0.3910   -0.4070

R1:
    1.2269   -0.8223   -1.1075   -0.6766
         0    0.6697    0.2980   -0.3732
         0         0    0.1533    0.5021
         0         0         0    0.1139

Q2:
   -0.2695    0.8696   -0.3505    0.1942
   -0.6202   -0.4427   -0.6370   -0.0403
   -0.4967    0.1744    0.4104   -0.7333
   -0.5286   -0.0919    0.5365    0.6351

R2:
    1.3193   -0.8992   -1.1514   -0.5531
         0    0.5378   -0.8481    0.0131
         0         0    0.4120   -0.4711
         0         0         0    0.1779
```

Figure 10. QR matrices in command window

$$Q = \begin{pmatrix} -0.5617 & 0.0618 & -0.1175 & 0.8166 \\ -0.3080 & 0.0050 & -0.8885 & -0.3401 \\ -0.2768 & 0.9129 & 0.1900 & -0.2322 \\ -0.7162 & -0.4035 & 0.4008 & -0.4044 \end{pmatrix} \qquad Q = \begin{pmatrix} -0.2690 & 0.8748 & 0.3531 & -0.1940 \\ -0.6230 & -0.4414 & 0.6442 & 0.0460 \\ -0.5009 & 0.1774 & -0.4156 & 0.7382 \\ -0.5372 & -0.0916 & -0.5363 & -0.6445 \end{pmatrix}$$

$$R = \begin{pmatrix} -1.2285 & -0.8256 & -1.1173 & -0.6788 \\ 0.0000 & 0.6740 & -0.2953 & 0.3773 \\ 0.0000 & 0.0000 & 0.1524 & -0.5096 \\ 0.0000 & 0.0000 & 0.0000 & 0.1105 \end{pmatrix} \qquad R = \begin{pmatrix} -1.3251 & -0.9048 & -1.1585 & -0.5611 \\ 0.0000 & 0.5403 & 0.8532 & -0.0113 \\ 0.0000 & 0.0000 & -0.4139 & -0.4734 \\ 0.0000 & 0.0000 & 0.0000 & -0.1816 \end{pmatrix}$$

$$\mathrm{A} = \mathrm{Q}\,\mathrm{R} = \begin{pmatrix} 0.6900 & 0.5054 & 0.5914 & 0.5547 \\ 0.3784 & 0.2577 & 0.2073 & 0.6262 \\ 0.3401 & 0.8438 & 0.0687 & 0.4099 \\ 0.8799 & 0.3194 & 0.9805 & 0.0850 \end{pmatrix} \qquad \mathrm{A} = \mathrm{Q}\,\mathrm{R} = \begin{pmatrix} 0.3564 & 0.7160 & 0.9118 & 0.0091 \\ 0.8255 & 0.3252 & 0.0785 & 0.0412 \\ 0.6638 & 0.5491 & 0.9037 & 0.3417 \\ 0.7119 & 0.4366 & 0.7662 & 0.6734 \end{pmatrix}$$

Figure 11. Results of QR decomposition from online calculator

System Generator results:

Figure 12 shows the resource result for QRD system, there are **36525 LUTs** and **21977 Registers** in my design without any DSPs.

Resource Analyzer: QRD_System

Post Synthesis Resources: Clicking on an instance name highlights corresponding block/subsystem in the model

Name	BRAMs (140)	DSPs (220)	LUTs (53200)	Registers (106400)
QRD_System	0	0	36525	21977
Rotation9	0	0	1818	1073
Rotation8	0	0	1820	1075
Rotation7	0	0	1820	1074
Rotation6	0	0	1774	1030
Rotation5	0	0	1820	1075
Rotation4	0	0	1820	1074
Rotation3	0	0	1776	1032
Rotation2	0	0	1820	1074
Rotation18	0	0	1818	1073
Rotation17	0	0	1820	1075
Rotation16	0	0	1820	1074
Rotation15	0	0	1818	1073
Rotation14	0	0	1820	1075
Rotation13	0	0	1820	1074
Rotation12	0	0	1802	1073
Rotation11	0	0	1820	1075
Rotation10	0	0	1820	1074
Rotation1	0	0	1776	1031
ArcTan3	0	0	1307	924
ArcTan2	0	0	1307	924
ArcTan1	0	0	1309	925

Figure 12. QR decomposition system resource cost

Vivado:

QR decomposition Vivado block design:

In this section, basically we just import the IP catalog which generated from Model Composer, and when we generate the QRD system, we need to set each gateway to AXILITE interfaces, and integrated the IP package with the ARM core in Vivado as shown in *Figure* 13, then verified and wrapped the design, after that synthesis the design and run implementation, finally generated bitstream, exported the hardware to Vitis IDE. Also, the *Figure 14* is my QRD system utilization and timing results table.

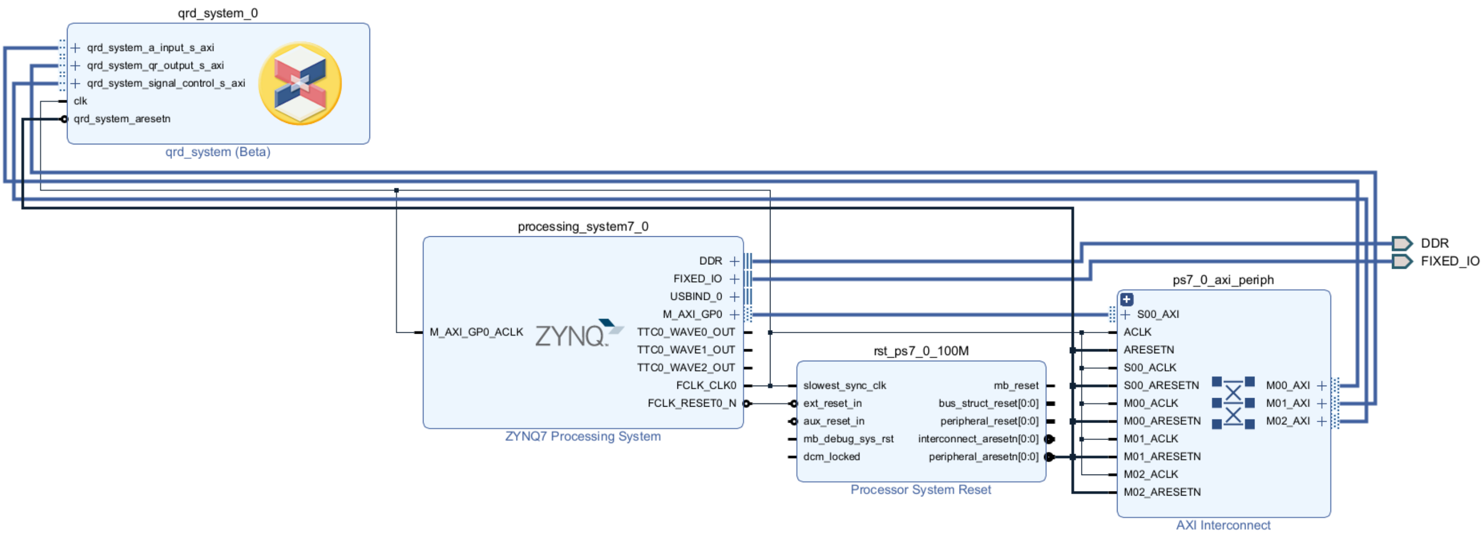


Figure 13. QR decomposition Vivado block design

Tcl Console | Messages | Log | Reports | **Design Runs**

Name	Constraints	Status	WNS	TNS	WHS	THS	WBSS	TPWS	Total Power	Failed Routes	Methodology	RQA Score	QoR Suggestions	LUT	FF	BRAM	URAM	DSP
synth_1 (active)	**constrs_1**	**synth_design Complete!**												**0**	**0**	**0**	**0**	**0**
impl_1	constrs_1	write_bitstream Complete!	0.056	0.000	0.022	0.000		0.000	2.647	0				32576	23031	0	0	0
Out-of-Context Module Runs																		

Figure 14. QRD's Vivado synthesis and implementation utilization and timing results

Vitis IDE: ARM program control

QR decomposition ARM program control in Vitis IDE:

In this section, I implemented the QR decomposition main.cc file and include the qrd_system.h header file which generated from Vivado, and successfully build the program to ARM and connected to Zedboard serial port, then debug the program, and final result as shown in *Figure 15*.

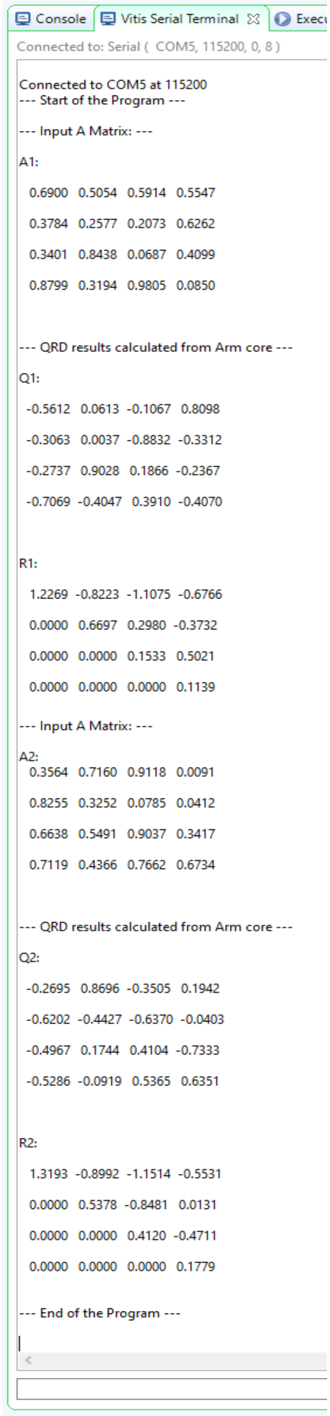


Figure 15. Screen capture of successfully connect to Zedboard and debug the program