Project 6: 4x4 Linear System Solver with ARM Core and CORDIC QRD Accelerator Integration

Jennifer Hellar

ELEC 522 - Fall 2018

Introduction

In this project, I explored the integration of a processing core (the embedded ARM cortex) with a custom FPGA accelerator using Xilinx SDK and System Generator. My goal was to solve a 4x4 system of linear equations represented in matrix-vector form by $Ax = b$. The algorithmic implementation of this is given by the following steps:

1. *QR Decomposition of a 4x4 matrix using CORDIC arithmetic: $A \rightarrow Q^T$ and R*
2. *Matrix-vector multiplication: $y = Q^T b$*
3. *Backsubstition:*
    a. $x_4 = y_4 / R_{44}$
    b. $x_3 = (y_3 - R_{34} * x_4) / R_{33}$
    c. $x_2 = (y_2 - R_{23} * x_3 - R_{24} * x_4) / R_{22}$
    d. $x_1 = (y_1 - R_{12} * x_2 - R_{13} * x_3 - R_{14} * x_4) / R_{11}$

I implemented this algorithm entirely in C++ and ran on the ARM core to benchmark the time performance. I also modified the QR Decomposition array accelerator from Project 5 to interface with the ARM core using the AXI-LITE bus and implemented the system solver with the accelerator performing the QRD.

QRD Implementation in C++

To compare the accelerator performance versus the performance of a program executing on the ARM core, I implemented QR decomposition in C++ using Vivado HLS to write a testbench and compile and execute the code (see the *qr_hls* folder).

The code iterates through the lower left elements of *A* (a 4x4 array of type **FIX_16_3**) and does the following:

1. Zeros out the appropriate elements of *A* to obtain *R* using the ATAN mode of the CORDIC function from Project 4 (see ***Figure 1***)

2. Propagates the rotations to the right columns of *R* using the ROTATION mode of the CORDIC function (see ***Figure 2***)
3. Applies the rotations to the appropriate rows of the identity matrix to obtain Q^T using the ROTATION mode the the CORDIC function (see ***Figure 3***)

```
70      for(col = 0; col < 4; col++){
71          for(row = 2; row >= col; row--){
72              x = r_temp[row][col];
73              y = r_temp[row+1][col];
74
75              // Perform atan
76              cordic(0, x, -y, 0, x_out, y_out, z_out);
77
78              r_temp[row][col] = x_out;
79              r_temp[row+1][col] = 0;
80              angle = z_out;
```

Figure 1: QRD in C++, ATAN mode to zero out elements

```
82          // Propagate rotation to the rest of the R matrix
83          for(rot_col = col+1; rot_col < 4; rot_col++){
84              x = r_temp[row][rot_col];
85              y = r_temp[row+1][rot_col];
86              cordic(1, x, y, angle, x_out, y_out, z_out);
87              r_temp[row][rot_col] = x_out;
88              r_temp[row+1][rot_col] = y_out;
89          }
```

Figure 2: QRD in C++, ROTATION mode applying angles to generate *R*

```
91          // Propagate rotation through Q matrix cols
92          for(rot_col = 0; rot_col < 4; rot_col++){
93              x = qt_temp[row][rot_col];
94              y = qt_temp[row+1][rot_col];
95              cordic(1, x, y, angle, x_out, y_out, z_out);
96              qt_temp[row][rot_col] = x_out;
97              qt_temp[row+1][rot_col] = y_out;
98          }
```

Figure 3: QRD in C++: ROTATION mode applying angles to generate Q^T

System Solver Implementation in C++

I implemented the system solver in several steps. First, I assumed inputs of type **double** and converted to **FIX_16_3** in order to use the QRD function described above. Then, I called the QRD function and converted the results back to **double**. This is shown in ***Figure 4*** below.

```
4   void solver(
5           double a[4][4],
6           double b[4],
7           double x[4])
8   {
9       // Convert A to fixed point
10      FIX_16_3 a_fix[4][4];
11      for(int i = 0; i < 4; i++){
12          for(int j = 0; j < 4; j++){
13              a_fix[i][j] = a[i][j];
14          }
15      }
16
17      // QR decomposition of A returns Qt and R in FIX_16_3
18      FIX_16_3 r_fix[4][4], qt_fix[4][4];
19      qr(a_fix, qt_fix, r_fix);
20
21      // Convert Qt and R back to double
22      double qt[4][4];
23      double r[4][4];
24      for(int i = 0; i < 4; i++){
25          for(int j = 0; j < 4; j++){
26              qt[i][j] = qt_fix[i][j];
27              r[i][j] = r_fix[i][j];
28          }
29      }
```

Figure 4: Solver in C++, data type conversions and QRD

Next, I performed the matrix-vector multiplication $y = Q^T b$ and back-substitution to solve for x as shown below in ***Figure 5***.

```
31      // Matrix-vector multiplication y = Qt*b
32      double y[4];
33      double sum;
34      for(int i = 0; i < 4; i++){
35          sum = 0;
36          for(int j = 0; j < 4; j++){
37              sum = sum + qt[i][j]*b[j];
38          }
39          y[i] = sum;
40      }
41
42      // Back-substitution
43      x[3] = y[3]/r[3][3];
44      x[2] = (y[2] - (r[2][3]*x[3]))/r[2][2];
45      x[1] = (y[1] - (r[1][2]*x[2]) - (r[1][3]*x[3]))/r[1][1];
46      x[0] = (y[0] - (r[0][1]*x[1]) - (r[0][2]*x[2]) - (r[0][3]*x[3]))/r[0][0];
```

Figure 5: Solver in C++, matrix-vector multiplication and back-substitution

The rest of the code, including the CORDIC function, can be found in *qr.cpp* and *qr.h* in the *qr_hls* folder.

Testing of C++ System Solver

I implemented the C++ in Vivado HLS initially for testing purposes and used the testbench code in ***Figure 6***, taken from *qr-test.cpp*.

```
int main(int argc, char **argv)
{
    // Ax = b
    double a[4][4] = {{0.1, 0.1, 0.4, 0.3},
            {0.2, 0.8, 0.6, 0.5},
            {0.9, 0.1, 0.3, 0.2},
            {0.3, 0.1, 0.4, 0.6}};

    double a1[4][4] = {{0.4218,    0.6557,    0.6787,    0.6555},
              {0.9157,    0.0357,    0.7577,    0.1712},
              {0.7922,    0.8491,    0.7431,    0.7060},
              {0.9595,    0.9340,    0.3922,    0.0318}};

    double a2[4][4] = {{0.3517,    0.9172,    0.3804,    0.5308},
              {0.8308,    0.2858,    0.5678,    0.7792},
              {0.5853,    0.7572,    0.0759,    0.9340},
              {0.5497,    0.7537,    0.0540,    0.1299}};

    double b[4] = {0.2, 0.1, 0.8, 0.4};
    double x[4]; double x1[4]; double x2[4];

    // Input to linear system solver
    solver(a, b, x);
    solver(a1, b, x1);
    solver(a2, b, x2);

    printf("x = [%f; %f; %f; %f]\n\n", x[0], x[1], x[2], x[3]); // Sh
    printf("x1 = [%f; %f; %f; %f]\n\n", x1[0], x1[1], x1[2], x1[3]);
    printf("x2 = [%f; %f; %f; %f]\n\n", x2[0], x2[1], x2[2], x2[3]);
```

Figure 6: Solver in C++, testbench code in Vivado HLS

I ran the code above which exercised the system solver function for the three systems shown. I used MATLAB builtin functions to directly compute the correct result for the first, *x = [0.8034, -0.3796, 0.2957, 0.1311]*. I also recomputed *b = A*x* using each of the results and compared to the original *b*, resulting in total errors of 0.0066, 0.029, and 0.011 respectively as shown in below.

```
x = [0.799096; -0.381305; 0.305273; 0.122963]

x1 = [2.017672; -0.542455; -2.853609; 2.507993]

x2 = [0.341865; 0.272358; -1.162742; 0.510324]

error in reconstructed b = 0.006641

error in reconstructed b1 = 0.029405

error in reconstructed b2 = 0.010670
```

Figure 7: Solver in C++, Vivado HLS simulation results

Modifications to Input of QRD Accelerator

After developing a C++ system solver as described above, I switched gears to modify the QRD accelerator from Project 5 to interface with the ARM core over AXI-LITE. On the input side, I quickly realized that I would have to add significant logic to correctly capture values transmitted over the AXI-LITE bus, since they would not arrive with any sort of predictable timing and they would hold the last value transmitted rather than pulse the data as I had done in Project 5.

With this in mind, I first added some basic control signals as follows:

1. *enable:* Positive edge transition enables the input buffers to be able to output values.
2. *start:* Positive edge transition triggers the QR computation once all the data has been received by starting the QR subsystem and controlling the data input into that array.
3. *rst:* Positive edge transition resets the output registers and disables the output logic.

In addition, each data line (*Acol1, Acol2,* etc.) has a corresponding *write* input which on a positive edge transition saves the current value from the data line into a register. The overall input system is shown in ***Figure 8***, where the *qr1* subsystem is the systolic array from Project 5.

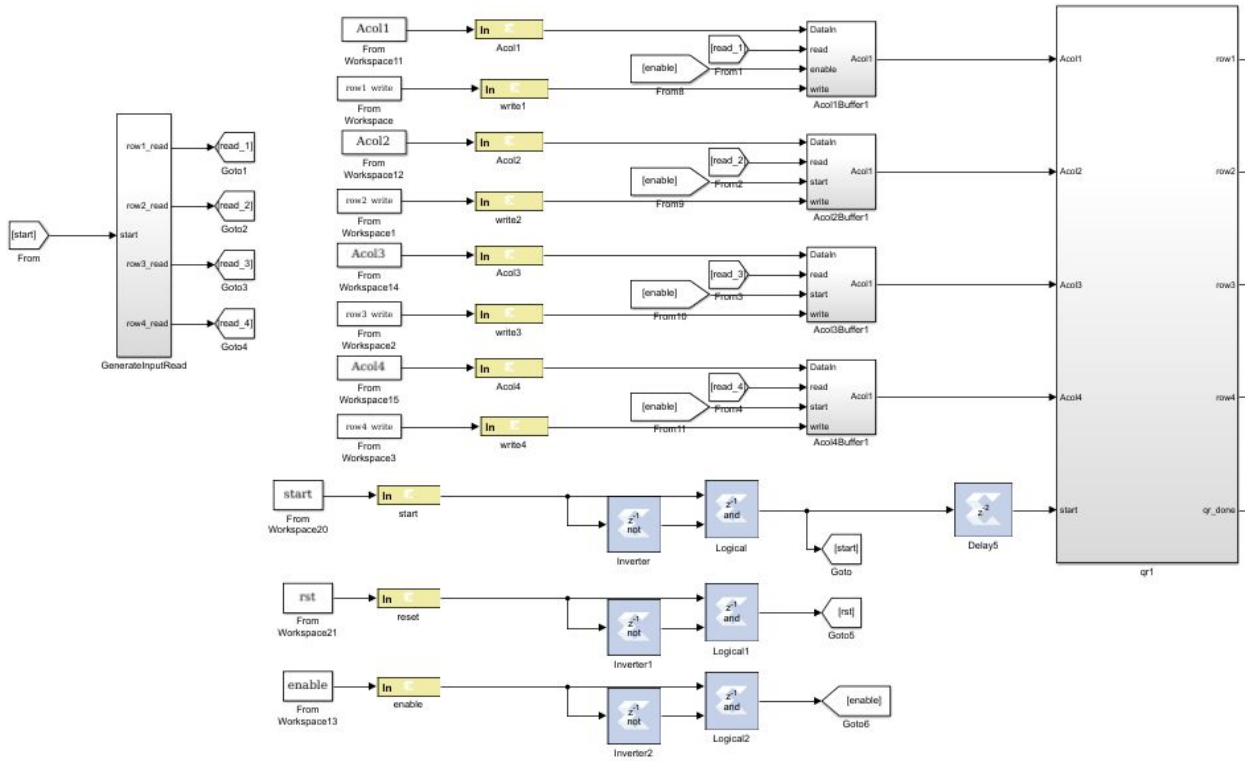

Figure 8: QRD accelerator, input additions

The input buffer for *Acol1*, which saves the values as they arrive and reads them out in order when triggered by *read* is shown below in ***Figure 9***.

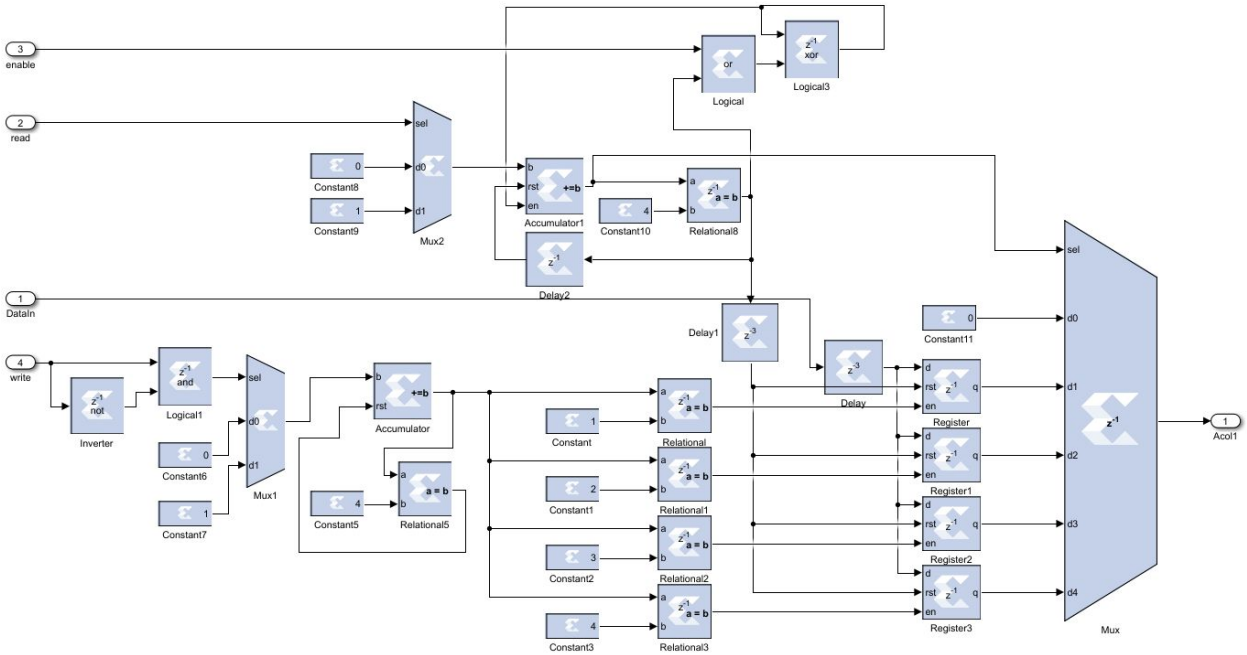

Figure 9: QRD accelerator, input buffer

The data arrives on *DataIn* and is saved into the appropriate register on the right when *write* is triggered. The *read* input adds to an accumulator (enabled by *enable*) which selects the appropriate output for the mux connected to each of the data registers. This allows data to be read in from an asynchronous system (the ARM core).

The subsystem *GenerateInputRead* generates the *read* signals for each of the input buffers and is shown below in ***Figure 10***. The accelerator input *start* triggers the data flow into the systolic array.

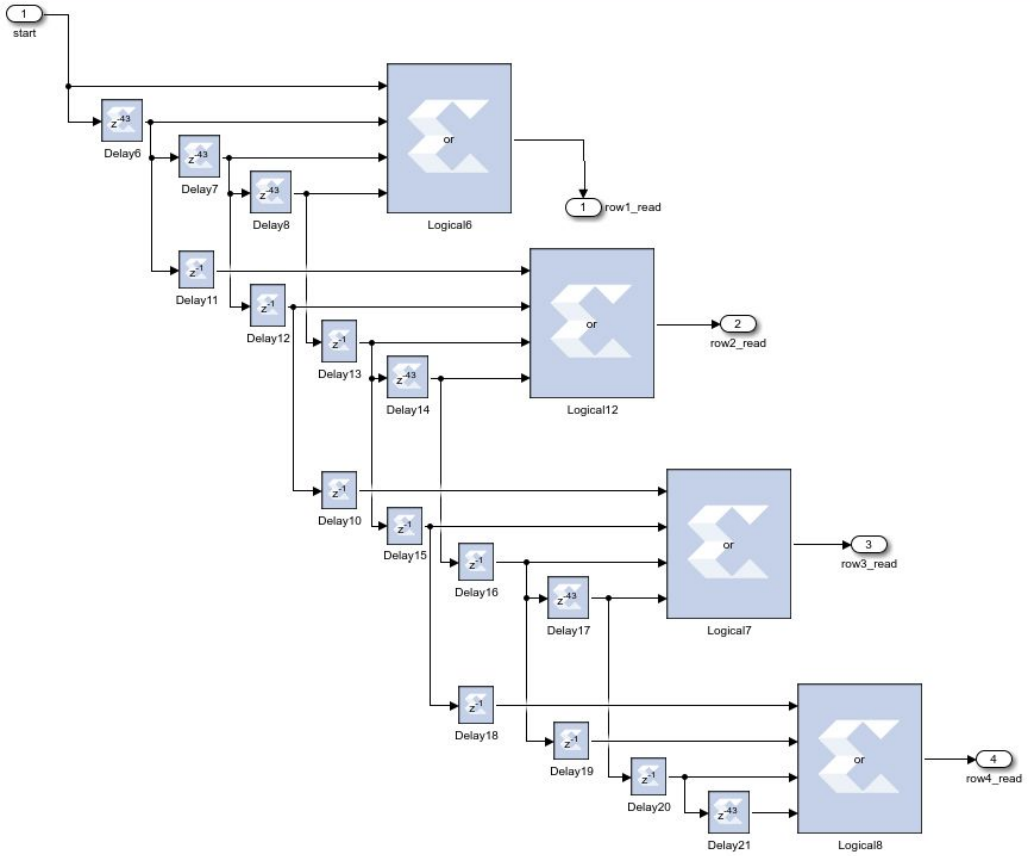

Figure 10: QRD Accelerator, *GenerateInputRead* subsystem

Modifications to Output of QRD Accelerator

After modifying the input logic, I turned my focus to the outputs, which were somewhat easier. The overall output logic is shown in ***Figure 11*** below.

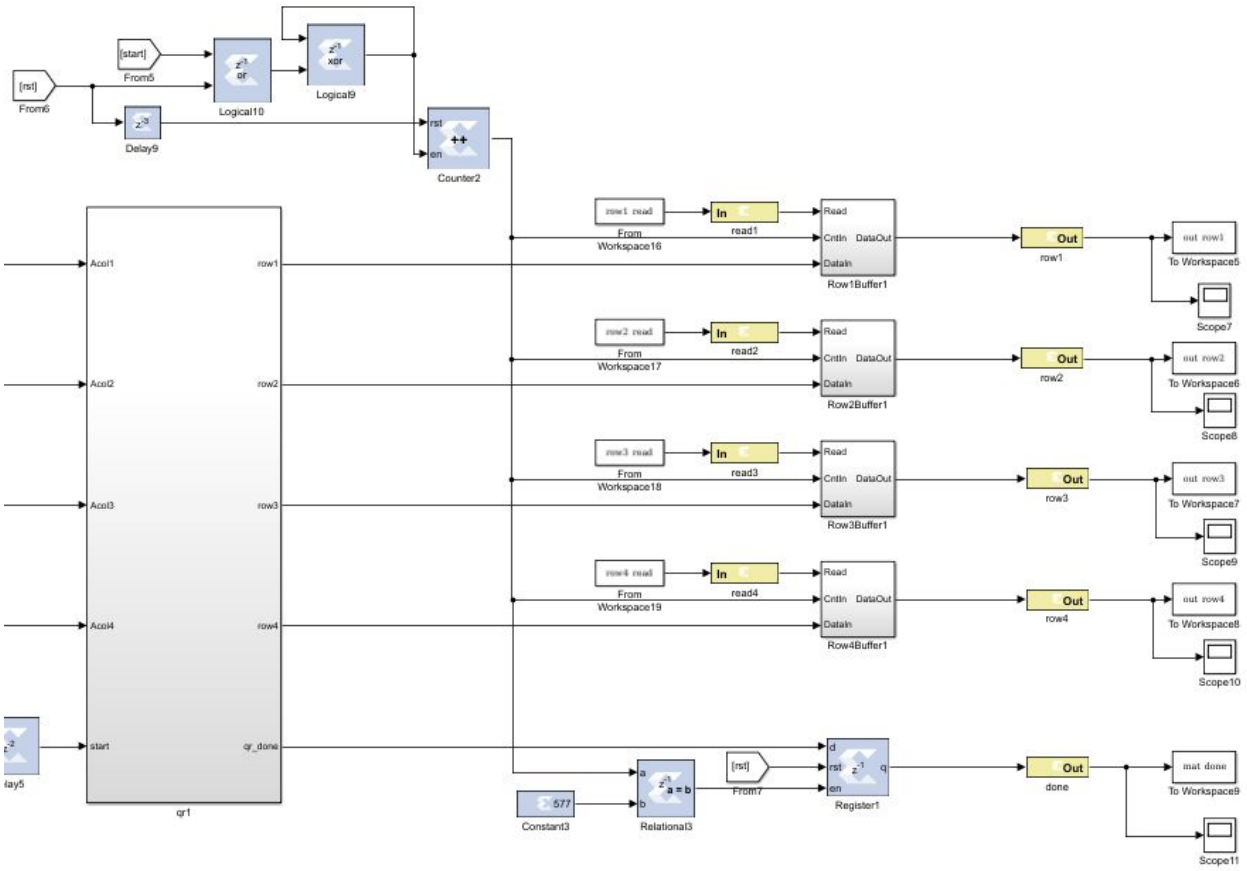

Figure 11: QRD accelerator, output additions

The computation *start* signal also enables a constant up-counter which serves as an input to the output buffer subsystems shown above. The *read* signals associated with each row tell the buffer to place the next output on the line with a positive edge transition. The output buffer subsystem for row 1 is shown in ***Figure 12*** below.

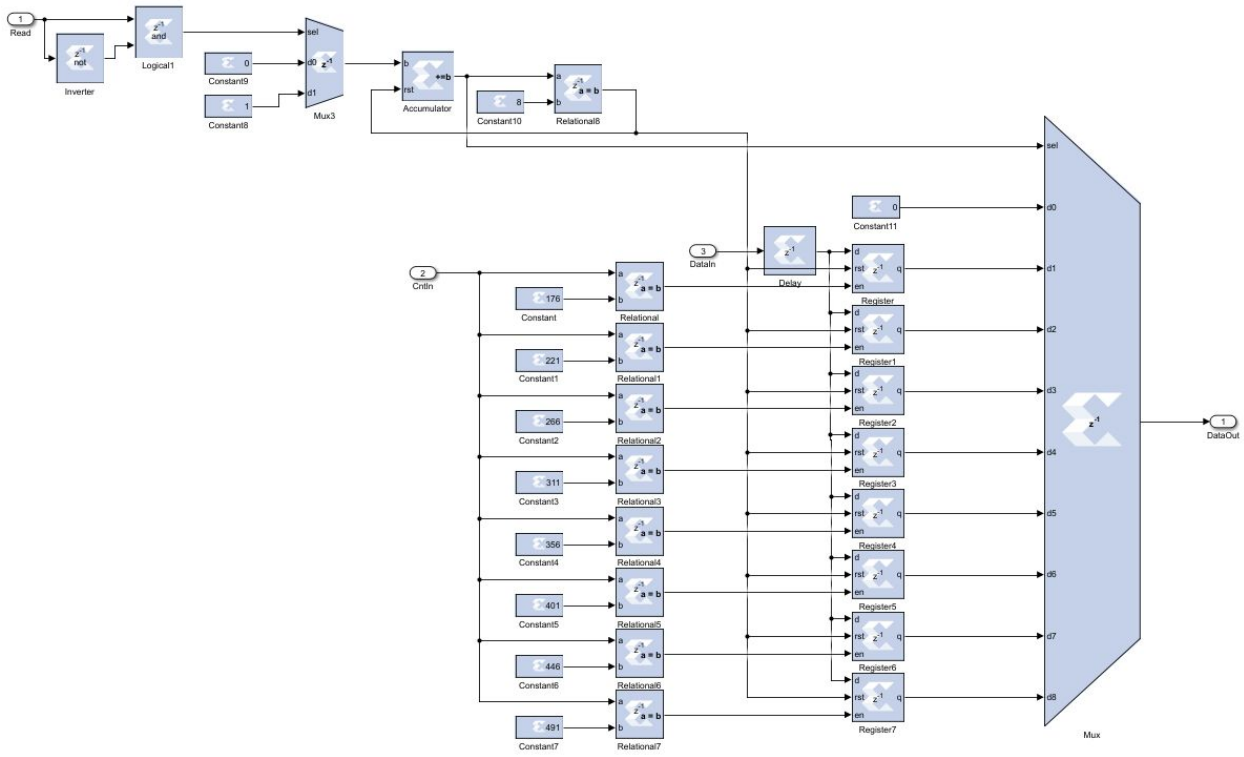

Figure 12: QRD accelerator, output buffer

When the counter reaches different values, the current output of the QR array is saved into the corresponding register. The values were determined by scoping the output to know the exact time each result appeared. The *read* signal incremented an accumulator on positive edge transitions to change the mux select signal.

MATLAB Simulation of QRD Accelerator

I used the code in *load_data.m* to simulate the accelerator performance in SysGen. The code is shown below in ***Figure 13***.

```
A1 = [0.1, 0.1, 0.4, 0.3;
    0.2, 0.8, 0.6, 0.5;
    0.9, 0.1, 0.3, 0.2;
    0.3, 0.1, 0.4, 0.6];

% Input data at various times, for various lengths of time
A1col1 = [kron(fliplr(A1(:,1)'),ones(1,3)),zeros(1,617)];
A1col2 = [kron(fliplr(A1(:,2)'),ones(1,4)),zeros(1,617)];
A1col3 = [kron(fliplr(A1(:,3)'),ones(1,2)),zeros(1,617)];
A1col4 = [kron(fliplr(A1(:,4)'),ones(1,3)),zeros(1,617)];

Acol1 = timeseries([zeros(1,4),A1col1]);
Acol2 = timeseries([zeros(1,4),A1col2]);
Acol3 = timeseries([zeros(1,4),A1col3]);
Acol4 = timeseries([zeros(1,4),A1col4]);

% Write after data is available
row1_write = timeseries([zeros(1,4),ones(1,2),zeros(1,1),ones(1,2),zeros(1,
row2_write = timeseries([zeros(1,4),ones(1,3),zeros(1,1),ones(1,3),zeros(1,
row3_write = timeseries([zeros(1,4),ones(1,1),zeros(1,1),ones(1,1),zeros(1,
row4_write = timeseries([zeros(1,4),ones(1,2),zeros(1,1),ones(1,2),zeros(1,
```

```
% Enable and start
enable = timeseries([zeros(1,3),ones(1,3),zeros(1,750)]);
start_array = [zeros(1,30),ones(1,10),zeros(1,800)];
start = timeseries(start_array);

% Read results out
row1_read = timeseries([zeros(1,700),ones(1,2),zeros(1,1),ones(1,1),zeros(1
row2_read = timeseries([zeros(1,700),ones(1,3),zeros(1,1),ones(1,1),zeros(1
row3_read = timeseries([zeros(1,700),ones(1,4),zeros(1,1),ones(1,1),zeros(1
row4_read = timeseries([zeros(1,700),ones(1,1),zeros(1,1),ones(1,1),zeros(1

% Reset the system
rst = timeseries([zeros(1,614),ones(1,7),zeros(1,750)]);

%% Check result

R = [out_row1(705),out_row1(708),out_row1(710),out_row1(712);
    0, out_row2(705), out_row2(709), out_row2(711);
    0, 0, out_row3(705),out_row3(710);
    0, 0, 0, out_row4(705)];
Qt = [out_row1(714), out_row1(716), out_row1(718), out_row1(720);
    out_row2(713), out_row2(715), out_row2(717), out_row2(719);
    out_row3(712), out_row3(714), out_row3(716), out_row3(718);
    out_row4(707), out_row4(709), out_row4(711), out_row4(713)];
Anew = Qt'*R
```

Figure 13: QRD accelerator, MATLAB testbench

After running this code, I correctly received *R* and Q^T that recombined to form a close approximation of *A* as shown in ***Figure 14*** below.

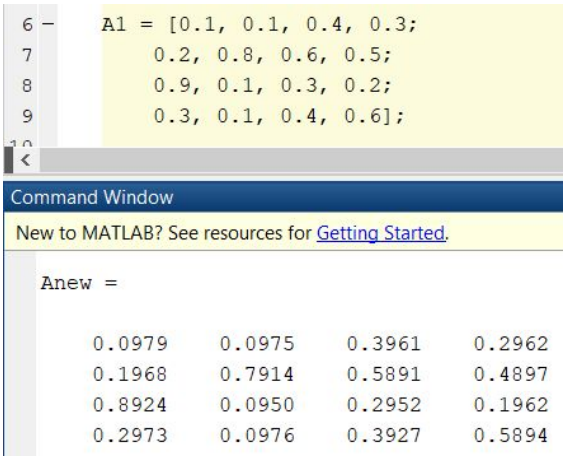

Figure 14: QRD accelerator, MATLAB simulation results

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	11105 (20.87%)
FFs	4053 (7.62%)
Slices	3578 (26.9%)
IOPADs	130 (100%)
DSPs	0
WNS (ns)	2.136
Max clock (MHz)	127

My block diagram is shown in ***Figure 15***.

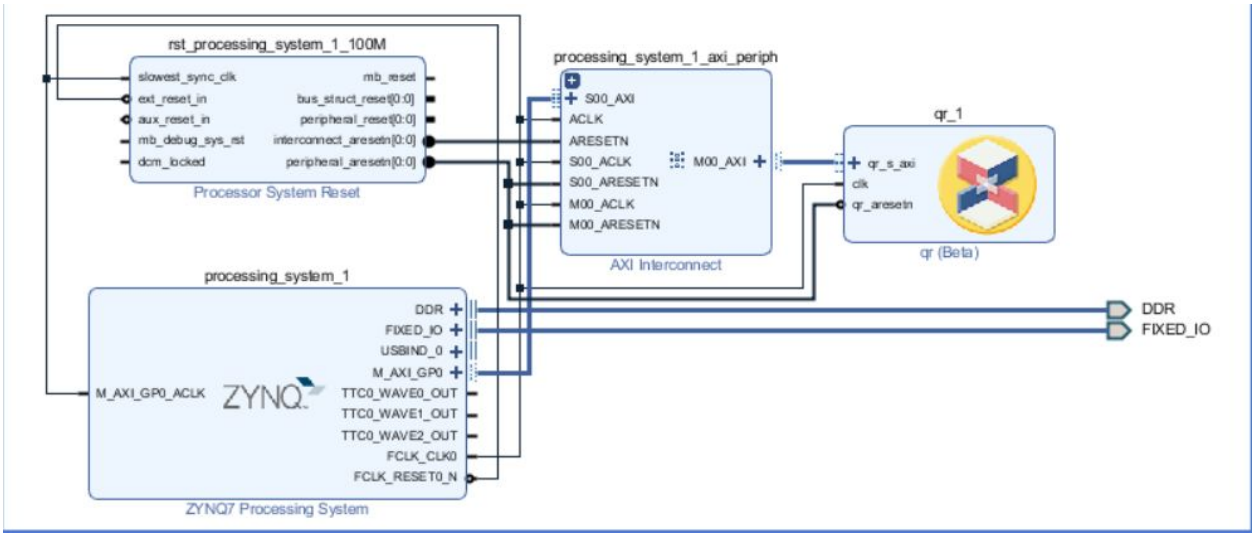

Figure 15: Vivado block diagram

Export to SDK

From there, I exported the design to SDK. I added a new C++ application project and added all of the .h files that Dr. Cavallaro provided in his example. I also added the repository for the drivers, and made an empty main.cc file with the appropriate headers shown below in ***Figure 16***.

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
// .h file for software solver
#include "solver.h"
// .h file based on your .hls design
#include "ap_fixed.h"
// The .h files from C:\ELEC522\cavallar\fixed_SDK\SG\netlist_I
#include "qr.h"
#include "qr_hw.h"
// Memory map file from C:\ELEC522\cavallar\fixed_SDK\SG\netlis
#include "xparameters.h"
// The Xilinx time function file for measuring ARM core cycles
#include "xtime_l.h"
```

Figure 16: SDK, include appropriate headers

In this case, *solver.h* and *solver.cpp* are the C++ files for performing the system solver in code only, copied over from the HLS file where I developed them. I also copied over Dr. Cavallaro's functions *get_int* and *get_double* for converting data types back and forth to send and receive over the AXI-LITE bus.

I set up the systems and timing variables as shown in ***Figure 17***.

```
    // Variables for timing and counts
    unsigned long long tt;
    int tt_print;
    double tt_seconds;
    XTime start_time_co;
    XTime stop_time_co;

    // Ax = b
    double a[4][4] = {{0.1, 0.1, 0.4, 0.3},
            {0.2, 0.8, 0.6, 0.5},
            {0.9, 0.1, 0.3, 0.2},
            {0.3, 0.1, 0.4, 0.6}};

//    double a[4][4] = {{0.4218,    0.6557,    0.6787,    0.6555},
//            {0.9157,    0.0357,    0.7577,    0.1712},
//            {0.7922,    0.8491,    0.7431,    0.7060},
//            {0.9595,    0.9340,    0.3922,    0.0318}};

//    double a[4][4] = {{0.3517,    0.9172,    0.3804,    0.5308},
//            {0.8308,    0.2858,    0.5678,    0.7792},
//            {0.5853,    0.7572,    0.0759,    0.9340},
//            {0.5497,    0.7537,    0.0540,    0.1299}};

    double b[4] = {0.2, 0.1, 0.8, 0.4};
    double x[4];

    printf("-- Start of the Program --\n");
```

Figure 17: SDK, system setup

I inputted these on separate runs to the purely C++ solver first and calculated the timing and error as shown in ***Figure 18***.

```
// Get the starting time in cycle counts
XTime_GetTime(&start_time_co);

// Input to linear system solver
solver(a, b, x);

// Capture the stop time
XTime_GetTime(&stop_time_co);

printf("x = [%f; %f; %f; %f]\n", x[0], x[1], x[2], x[3]);   // Should be [0.

// Error calculation by recalculating b = A*x
double error = 0.0;
double b_new[4];
double sum;
for(int i = 0; i < 4; i++){     // Matrix-vector multiplication
    sum = 0;
    for(int j = 0; j < 4; j++){
        sum = sum + a[i][j]*x[j];
    }
    b_new[i] = sum;
    error = error + abs_double(b[i] - (double)b_new[i]);
}
printf("b_new = [%f; %f; %f; %f]\n", b_new[0], b_new[1], b_new[2], b_new[3])

printf("error in reconstructed b = %f\n", error);       // gives 0.006641
```

Figure 18: SDK, C++ solver timing & error calculation

I then inputted the matrix into the QRD accelerator to compare the timing, first converting to **int** and then writing to the appropriate registers as shown in ***Figure 19***. For the sake of space, only the first few writes are shown.

```
// Get the starting time in cycle counts
XTime_GetTime(&start_time_co);

// Convert A to int for transmitting to PL
int a_int[4][4];
for(int i = 0; i < 4; i++){
    for(int j = 0; j < 4; j++){
        a_int[i][j] = get_int(a[i][j]);
    }
}

// Generate the result on hardware using the integer converted valu
// Send data over AXI - find these parameters from the ip folder fr
// The base address is in the xparameters.h file which should be lo

// Enable QR module
qr_WriteReg(XPAR_QR_1_QR_S_AXI_BASEADDR, QR_ENABLE, 1);
qr_WriteReg(XPAR_QR_1_QR_S_AXI_BASEADDR, QR_ENABLE, 0);

// Transmit 1st row
qr_WriteReg(XPAR_QR_1_QR_S_AXI_BASEADDR, QR_ACOL1, a_int[3][0]);
qr_WriteReg(XPAR_QR_1_QR_S_AXI_BASEADDR, QR_WRITE1, 1);
qr_WriteReg(XPAR_QR_1_QR_S_AXI_BASEADDR, QR_ACOL2, a_int[3][1]);
qr_WriteReg(XPAR_QR_1_QR_S_AXI_BASEADDR, QR_WRITE2, 1);
qr_WriteReg(XPAR_QR_1_QR_S_AXI_BASEADDR, QR_ACOL3, a_int[3][2]);
qr_WriteReg(XPAR_QR_1_QR_S_AXI_BASEADDR, QR_WRITE3, 1);
```

Figure 19: SDK, converting data type and writing to the accelerator

After transmitting all the data, I sent the *start* signal and polled the *done* signal until it went high, at which point I began reading out the data as shown in ***Figure 20***.

```
// Start computation
qr_WriteReg(XPAR_QR_1_QR_S_AXI_BASEADDR, QR_START, 1);
qr_WriteReg(XPAR_QR_1_QR_S_AXI_BASEADDR, QR_START, 0);

int done_out;
do
    done_out = qr_ReadReg(XPAR_QR_1_QR_S_AXI_BASEADDR, QR_DONE);
while(done_out == 0);

int r_int[4][4];
int qt_int[4][4];

// Read first row of R
qr_WriteReg(XPAR_QR_1_QR_S_AXI_BASEADDR, QR_READ1, 1);
r_int[0][0] = qr_ReadReg(XPAR_QR_1_QR_S_AXI_BASEADDR, QR_ROW1);
qr_WriteReg(XPAR_QR_1_QR_S_AXI_BASEADDR, QR_READ1, 0);

qr_WriteReg(XPAR_QR_1_QR_S_AXI_BASEADDR, QR_READ1, 1);
r_int[0][1] = qr_ReadReg(XPAR_QR_1_QR_S_AXI_BASEADDR, QR_ROW1);
qr_WriteReg(XPAR_QR_1_QR_S_AXI_BASEADDR, QR_READ1, 0);

qr_WriteReg(XPAR_QR_1_QR_S_AXI_BASEADDR, QR_READ1, 1);
r_int[0][2] = qr_ReadReg(XPAR_QR_1_QR_S_AXI_BASEADDR, QR_ROW1);
qr_WriteReg(XPAR_QR_1_QR_S_AXI_BASEADDR, QR_READ1, 0);
```

Figure 20: SDK, polling for QR completion and reading data out of the accelerator

Once all that was done, I reset the accelerator, converted the received R and Q^T back to **double** type, and performed the same matrix-vector multiply $y = Q^T*b$ as shown in ***Figure 21***.

```
// Reset the accelerator
qr_WriteReg(XPAR_QR_1_QR_S_AXI_BASEADDR, QR_RESET, 1);
qr_WriteReg(XPAR_QR_1_QR_S_AXI_BASEADDR, QR_RESET, 0);

// Convert R and Qt from int
double r[4][4];
double qt[4][4];
for(int i = 0; i < 4; i++){
    for(int j = 0; j < 4; j++){
        r[i][j] = get_double(r_int[i][j]);
        qt[i][j] = get_double(qt_int[i][j]);
    }
}

// Matrix-vector multiplication y = Qt*b
double y[4];
double sum1;
for(int i = 0; i < 4; i++){
    sum1 = 0;
    for(int j = 0; j < 4; j++){
        sum1 = sum1 + qt[i][j]*b[j];
    }
    y[i] = sum1;
}
```

Figure 21: SDK, received data conversion and matrix-vector multiplication

And finally, I performed the same back-substitution and error calculation as previously.

SDK Results

Unfortunately, even though I did get results using the accelerator, they were not correct, and I did not have time to debug the system and discover the problem. My results for each system (all having high condition number verified in MATLAB) are shown in ***Figures 22, 23,*** and ***24*** respectively.

```
-- Start of the Program --
x = [0.799096; -0.381305; 0.305273; 0.122963]
b_new = [0.200777; 0.099420; 0.797231; 0.397485]
error in reconstructed b = 0.006641
Done, Total time steps for PS internal (including ap_fixed emulation) = 2774150
Cycle counts per second for ARM A9 core for PS internal add = 3.333333e+08
Time in seconds for PS ARM software internal add = times steps / COUNTS_PER_SECOND = 8.322450e-03
x = [7.022509; 7.488003; -14.525512; 0.028886]
b_new = [-4.350488; -1.305960; 2.717182; -2.937320]
error in reconstructed b = 11.210950
Done, Total time steps for PL internal (including ap_fixed emulation) = 49888
Cycle counts per second for PL internal = 3.333333e+08
Time in seconds for PL internal = times steps / COUNTS_PER_SECOND = 1.496640e-04
-- End of the Program --
```

Figure 22: SDK, results for system 1

```
-- Start of the Program --
x = [2.017672; -0.542455; -2.853609; 2.507993]
b_new = [0.202612; 0.095406; 0.787928; 0.389873]
error in reconstructed b = 0.029405
Done, Total time steps for PS internal (including ap_fixed emulation) = 2773083
Cycle counts per second for ARM A9 core for PS internal add = 3.333333e+08
Time in seconds for PS ARM software internal add = times steps / COUNTS_PER_SECOND = 8.319249e-03
x = [-121.589797; 307.859397; 243.827366; -486.790909]
b_new = [-3.028978; 1.060195; 2.593710; 251.024408]
error in reconstructed b = 256.607291
Done, Total time steps for PL internal (including ap_fixed emulation) = 49981
Cycle counts per second for PL internal = 3.333333e+08
Time in seconds for PL internal = times steps / COUNTS_PER_SECOND = 1.499430e-04
-- End of the Program --
```

Figure 23: SDK, results for system 2

```
-- Start of the Program --
x = [0.341865; 0.272358; -1.162742; 0.510324]
b_new = [0.198613; 0.099301; 0.794713; 0.396702]
error in reconstructed b = 0.010670
Done, Total time steps for PS internal (including ap_fixed emulation) = 2769556
Cycle counts per second for ARM A9 core for PS internal add = 3.333333e+08
Time in seconds for PS ARM software internal add = times steps / COUNTS_PER_SECOND = 8.308668e-03
x = [4371.955322; -5315.557342; -1151.936842; -95.514545]
b_new = [-3826.708403; 1384.539521; -1642.677161; -1677.683657]
error in reconstructed b = 8532.908741
Done, Total time steps for PL internal (including ap_fixed emulation) = 50578
Cycle counts per second for PL internal = 3.333333e+08
Time in seconds for PL internal = times steps / COUNTS_PER_SECOND = 1.517340e-04
-- End of the Program --
```

Figure 24: SDK, results for system 3

The system solver in C++ worked extremely well, taking 0.0083 seconds with low recomputed errors of 0.0066, 0.03, and 0.01 for *b*. The system solver with the accelerator was faster (taking 0.00015 seconds) but completely wrong. Since the results are non-zero with the accelerator, I

suspect that some small part of my input/output logic is broken in the accelerator, but I am confident that these timing results are valid.

Conclusion

Even though I was not able to get the accelerator portion completely correct, it was obvious that the accelerator did speed up the computation by a factor of 55. Even if I optimized my C++ code, I anticipate a significant speed-up would still occur. But since the accelerator lies in the FPGA fabric outside of the ARM core, there is a big area-speed tradeoff.