HW5 QR Cordic Systolic Array design

VLSI DSP HW5

Shun-Liang Yeh, NCHU Lab612

5/18/2023

INDEX

I. Design Specification and Goal

Problem

1. Introduction

In this assignment, you are asked to develop a hardware QR factorization module. Given a N×N matrix, the QR factorization convert it to the product of a unitary matrix Q and an upper triangular matrix R.

- Goal is to design a hardware QR factorization module, capable of generating the Q matrix and R matrix for a given NxN A matrix, i.e. A = QR

2. QR factorization scheme

Assume matrix $\mathbf{A}_{4\times4}$, you may apply a sequence of Givens rotation to convert it into an upper triangular matrix $\mathbf{R}_{4\times4}$. However, to obtain the $\mathbf{Q}_{4\times4}$, you need extra computations to obtain the product of these Givens rotations. An easy way to accomplish it is augmenting matrix $\mathbf{A}_{4\times4}$ with an identity matrix $\mathbf{I}_{4\times4}$ and updating the identity matrix along with every Givens rotation applied to matrix $\mathbf{A}_{4\times4}$.

$$\mathbf{Q}_n \times \mathbf{Q}_{n-1} \times \cdots \times \mathbf{Q}_2 \times \mathbf{Q}_1 \times [\mathbf{A} \mid \mathbf{I}] = [\mathbf{R} \mid \mathbf{Q}]$$
$$\mathbf{Q} = \mathbf{Q}_n \times \mathbf{Q}_{n-1} \times \cdots \times \mathbf{Q}_2 \times \mathbf{Q}_1$$

- Using the above QR factorization scheme we can get our Q & R matrix through a series of givens rotations, just like what we did in HW1

For the triangularization part (**R** matrix calculation), a triangular systolic array structure as shown in the lecture note is needed. For **Q** matrix calculation, an additional rectangular array is needed to serve the purpose. This makes the array structure of the entire design a trapezoidal one (Fig. 1). Not shown in the figure includes a controller module (or FSM) to control the computations.

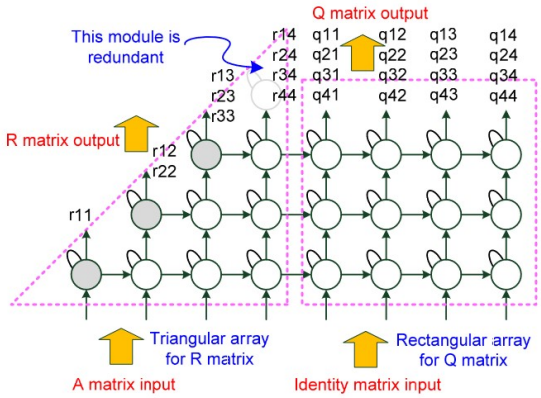


Figure 1. Trapezoidal array for QR factorization

- The systolic array structure is listed above, left triangular PE arrays calculate the R matrix while right hand side calculate the Q matrix.

3. Hardware design guidelines

- **CORDIC module**

CORDIC module is employed to implement the Givens rotations. The word length is set as 12 bits and you can determine how many bits are for the integral part and how many bits

are for the fractional part. The iteration number is set as 12. To reduce the computing latency iterations, 4 iterations are performed in one clock cycle and it takes 3+1 (extra clock cycle for normalization) to complete one Givens rotation. An unfolded CORDIC architecture with two iterations per clock cycle is shown below.

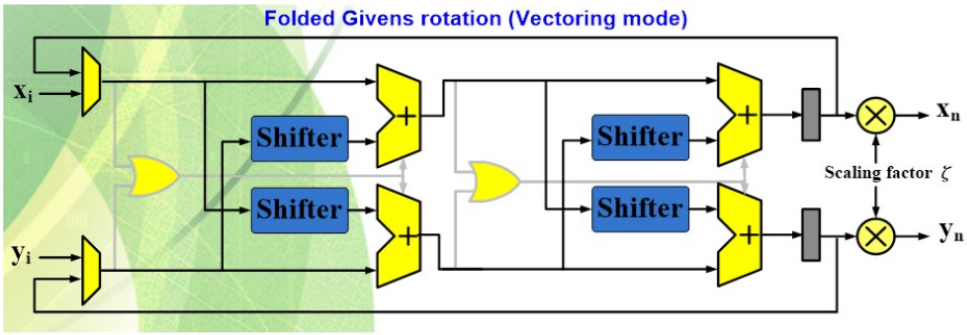


Figure 2. unfolded CORDIC processor design

Note that there is no need of computing the rotation angle θ explicitly. In each row of the CORDIC array, the leading one (performing in the vectoring mode) passes the rotation directions sequence, which consists of +1 or -1, to the trailing modules (performing in the rotation mode) in the same row. In other words, these trailing modules simply follow the sequence of micro-rotations performed in the leading one. Note that there are 4 rotation directions to be passed in each clock cycle because 4 iterations are performed.

- As suggested, each PE should calculate their result using only 4 cycles, where 3 cycles are used for rotation and the last cycle used for the multiplication of K constants.

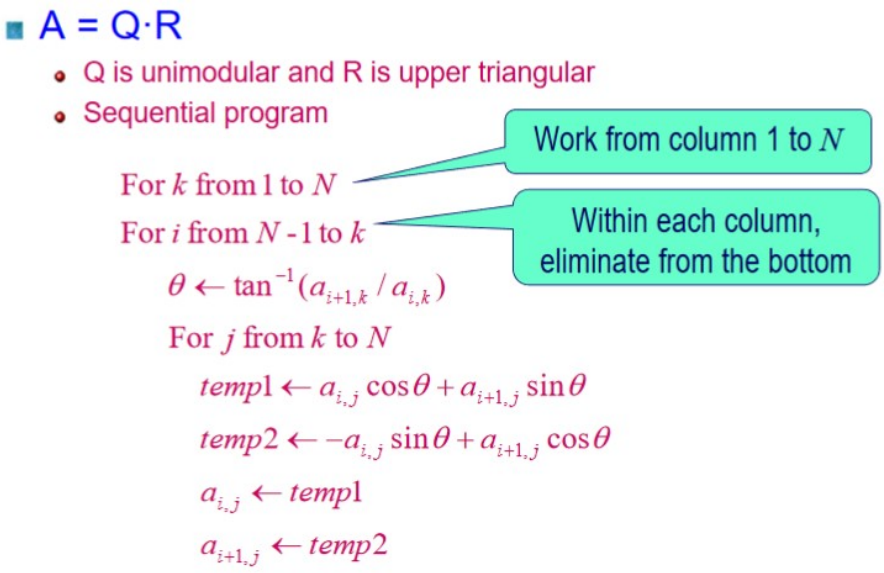
- **I/O requirements**

  As shown in Figure 1, matrix A is inputted from the bottom of the triangular array and an identity matrix is inputted from the bottom of the rectangular array. The resultant upper triangular matrix R can be obtained along the diagonal border of the triangular array. The Q matrix will reside on the rectangular array. Four additional clock cycles are needed to propagate the results upward so that the Q matrix can be obtained from the top row of the rectangular array. Also assume the entries of input matrix A are all integral, signed and 8-bit wide. The entries of R and Q matrices are all 12-bit wide (you need to indicate how many bits are for integral part, and how many bits are for fractional part).

- Input is an 8-bit integer, value between 128~ -127, while we have to output the Q & R matrices in 12-bit outputs.

II. QR Decomposition & QR Cordic Algorithm

QR Decomposition



- According to the algorithm provided within the handout, we can generate the matlab QR algorithm model with slight modification.

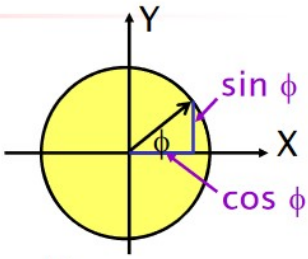
```matlab
function [Q, R] = qr_decomposition(M_in)
    % Algorithm from VLSI DSP lecture notes, 5-54, modify it using CORDIC algorithm
    M = M_in;
    N = length(M);
    % Augmenting the matrix I to the left, updating I alongside with A when performing rotation
    Q = eye(N);

    for k = 1:N
        for i = N:-1:k + 1
            % This needed to be replaced with cordic, vectoring mode.
            theta = atan(M(i, k) / M(i - 1, k));

            for j = k:N
                % For R
                % This needed to be replaced with cordic, rotation mode.
                tmp1 = M(i - 1, j) * cos(theta) + M(i, j) * sin(theta);
                tmp2 = -M(i - 1, j) * sin(theta) + M(i, j) * cos(theta);

                M(i - 1, j) = tmp1;
                M(i, j) = tmp2;
            end

            for j = 1:N
                % For Q, after calculation, take its transpose to get the correct Q, same for this portion.
                tmp1 = Q(i - 1, j) * cos(theta) + Q(i, j) * sin(theta);
                tmp2 = -Q(i - 1, j) * sin(theta) + Q(i, j) * cos(theta);

                Q(i - 1, j) = tmp1;
                Q(i, j) = tmp2;
            end
        end

    end

    Q = Q';
    R = M;
end
```

- The algorithm is modified to allow it for computing the Q and R matrices.

The Cordic Algorithm

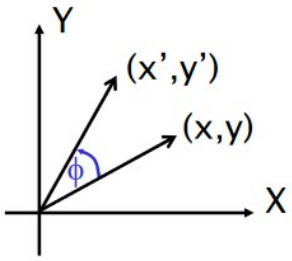
Basic CORDIC Transformations

- Basic idea
  - Rotate (1,0) by ϕ degrees to get (x,y): x=cos(ϕ), y=sin(ϕ)



- Rotation of any (x,y) vector:

$$x' = x.\cos(\phi) - y.\sin(\phi)$$
$$y' = y.\cos(\phi) + x.\sin(\phi)$$



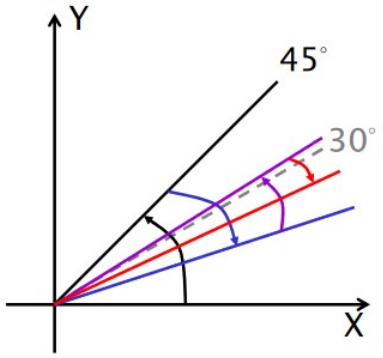
- Rearrange as:

$$x' = \cos(\phi).[x - y.\tan(\phi)]$$
$$y' = \cos(\phi).[y + x.\tan(\phi)]$$

Note : $\frac{\sin(\phi)}{\cos(\phi)} = \tan(\phi)$

Example: Rewriting Angles in Terms of α_i

- Example: ϕ=30.0°
  - Start with α_0 = 45.0 (> 30.0)
  - 45.0 – 26.6 = 18.4 (< 30.0)
  - 18.4 + 14.0 = 32.4 (> 30.0)
  - 32.4 – 7.1 = 25.3 (< 30.0)
  - 25.3 + 3.6 = 28.9 (< 30.0)
  - 28.9 + 1.8 = 30.7 (> 30.0)
  - . . .
  - ϕ = 30.0
    ≈ 45.0 – 26.6 + 14.0 – 7.1 + 3.6
    + 1.8 – 0.9 + 0.4 – 0.2 + 0.1
    = 30.1



Spring 2006 EE 5324 - VLSI Design II - © Kia Bazargan 374

- The basic idea is trying to rotates an (x,y) vector to (x',y') using the cosine and sine function, by approaching the desire angle iteratively. Approaching the desired angle in a number of iterations.

Rotation Reduction

$$x' = \cos(\phi).[x - y.\tan(\phi)]$$
$$y' = \cos(\phi).[y + x.\tan(\phi)]$$

- Rewrite in terms of α_i: $(0 \le i \le n)$

$$x_{i+1} = \cos(\alpha_i).[x_i - y_i.d_i.\tan(\alpha_i)] \rightarrow x_{i+1} = K_i.[x_i - y_i.d_i.2^{-i}]$$
$$y_{i+1} = \cos(\alpha_i).[y_i + x_i.d_i.\tan(\alpha_i)] \qquad y_{i+1} = K_i.[y_i + x_i.d_i.2^{-i}]$$

- After Understanding the Cordic Algorithm, one can try to code it out in matlab for verification.

Rotation Mode

Taking Care of the Magnitude

$$x_{i+1} = K_i.[x_i - y_i.d_i.2^{-i}]$$
$$y_{i+1} = K_i.[y_i + x_i.d_i.2^{-i}]$$

- Observations:
  - We choose to always use ALL α_i terms, with +/- signs
  - $K_i = \cos(\alpha_i) = \cos(-\alpha_i)$
  - At each step, we multiply by $\cos(\alpha_i)$ [constant?]
- Let the multiplications aggregate to:

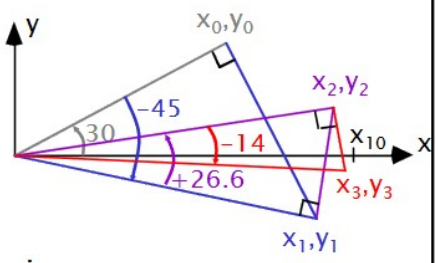
$$K = \prod_{i=0}^{n} K_i \qquad n \to \infty, K = 0.607\,252\,935\ldots$$

  - Multiply this constant ONLY ONCE at the end

- The correct K must be found for the correct iteration numbers, for our case, we have to find K_12. Thus we have to first calculation what K_12 is.

Hardware Realization: CORDIC Rotation Mode

- To simplify the hardware:
  - First rotate by ϕ, then rotate by $-d_i.\alpha_i$ to get 0 (no subtraction to compare ϕ and current angle)
- Algorithm: (z is the current angle)
  - Mode: rotation: "at each step, try to make z zero"
  - Initialize x=0.607253,y=0,z=ϕ
  - For i=0 $\rightarrow$n
  - d_i = 1 when z>0, else -1
  - $x_{i+1} = x_i - d_i \cdot 2^{-i} \cdot y_i$
  - $y_{i+1} = y_i + d_i \cdot 2^{-i} \cdot x_i$
  - $z_{i+1} = z_i - d_i \cdot \alpha_i$
  - Result: $x_n=\cos(\phi)$, $y_n=\sin(\phi)$
  - Precision: n bits $(\tan^{-1}(2^{-i}) \approx 2^{-i})$



[© Oxford U Press]

- Above is the pseudoCode for cordic rotation mode, we can implement it in matlab code below.

```matlab
function [x_result, y_result, angle] = cordic_rotation_mode(x, y, angle, iters_num, T)
    % Rotation mode
    % Description: Rotation mode using Linear rotation and multiplied by a constant K = 0.607252935 for stretching
    % Input:  vector (x,y) rotates it with the angle z
    % Output: vector (xcos(z),ysin(z))
    % Enables shift add multiply instead of calculation trigonemetric function
    % tan(a^i) = 2^(-i), thus tan(a^i) is simply shift.
    % Try to approach the desirable angle, we multiply the cos(a^i) only after the complete rotation.
    % Simply multiply K = 0.607252935 after the rotations. See if the angle difference approach to zero.

    alpha = [45, 26.565, 14.0362, 7.12502, 3.57633, 1.78991, 0.895174, 0.447614, 0.223811, 0.111906, 0.055953, 0.027977];
    K = 0.60725334371201; % This is Products of K_12

     % Turning into fixed point
     alpha = cast(alpha,'like',T.lut_coef);
     x_result = cast(x,'like',T.x_output);
     y_result = cast(y,'like',T.y_output);
     angle    = cast(angle,'like',T.theta_output);
     x_new = cast(x,'like',T.x_output);


    for i = 1:iters_num
        % Z is the current angle, and also the angle I want to shift toward to.
        if angle > 0
            % If angle is above 0, I would like to rotate in a clockwise manner
            x_new(:) = x + bitsra(y, i - 1);
            y(:) = y - bitsra(x, i - 1);
            x(:) = x_new;
            angle(:) = angle - alpha(i);
        else
            % Otherwise, rotate counter clockwise.
            x_new(:) = x - bitsra(y, i - 1);
            y(:) = y + bitsra(x, i - 1);
            x(:) = x_new;
            angle(:) = angle + alpha(i);
        end

    end

    x_result(:) = x * K;
    y_result(:) = y * K;
end

```

- Rotation mode rotates the vector (x,y) by a certain angle to another new vector (x',y').
- The normal rotation mode uses a look up table to search for the angle needed for each iteration, above implements the cordic psuedoCode in matlab.

Vector Mode

```
function [x_result, y_result, angle] = cordic_vector_mode(x, y, angle, iters_num,T)
    % Vectoring mode
    % Description: Vectoring mode uses linear rotation s.t. y approach to 0 iteratively
    % Input:  vector (x,y) rotates it with the angle z
    % Output: Value after rotation (x,y) also the angle of tan(y/x)
    % Goal is trying to nullify y accumulating the angle when rotating.
    alpha = [45, 26.565, 14.0362, 7.12502, 3.57633, 1.78991, 0.895174, 0.447614, 0.223811, 0.111906, 0.055953, 0.027977];
    % Turning into fixed point
    alpha = cast(alpha,'like',T.lut_coef);
    x_result = cast(x,'like',T.x_output);
    y_result = cast(y,'like',T.y_output);
    angle    = cast(angle,'like',T.theta_output);
    x_new    = cast(x,'like',T.x_output);
    for i = 1:iters_num
        % Z is the current angle, and also the angle I want to shift toward to.
        if sign(x * y) > 0
            % Since it is at the first dimension, we shoudl shift in a clockwise manner.
            % Since we are trying to obtain the angle, we must add when we are rotating.
            x_new(:) = x + bitsra(y, i - 1);
            y(:)     = y - bitsra(x, i - 1);
            x(:) = x_new;
            angle(:) = angle + alpha(i);
        else
            % Otherwise shift in a counter clockwise manner
            x_new(:) = x - bitsra(y, i - 1);
            y(:) = y + bitsra(x, i - 1);
            x(:) = x_new;
            angle(:) = angle - alpha(i);
        end
    end

    x_result = x;
    y_result = y;
end
```

- The normal Vector mode also uses a look up table for approaching desired angle, however, the condition for rotation differs from the one presented in rotation mode. Simply changing the condition allows for conversion from rotation mode into vector mode.
- However, the above two implementation is not optimized for hardware design, since a huge LUT must be stored to provide the angle. Also, the rotation mode has to wait for the angle produced by vector mode which is a waste of time. Consequently, some optimization must be performed on the algorithm above.

Optimizing Cordic Algorithm

For the triangularization part (**R** matrix calculation), a triangular systolic array structure as shown in the lecture note is needed. For **Q** matrix calculation, an additional rectangular array is needed to serve the purpose. This makes the array structure of the entire design a trapezoidal one (Fig. 1). Not shown in the figure includes a controller module (or FSM) to control the computations.

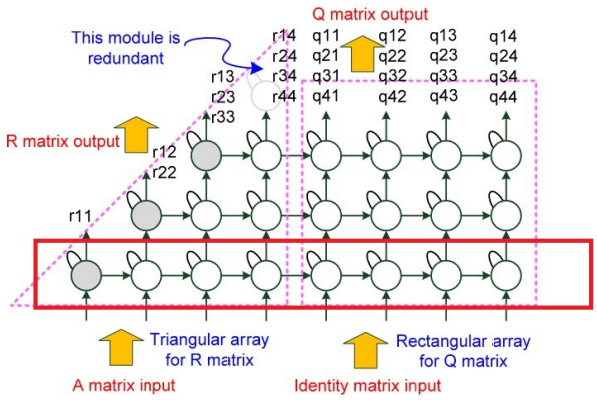


Figure 1. Trapezoidal array for QR factorization

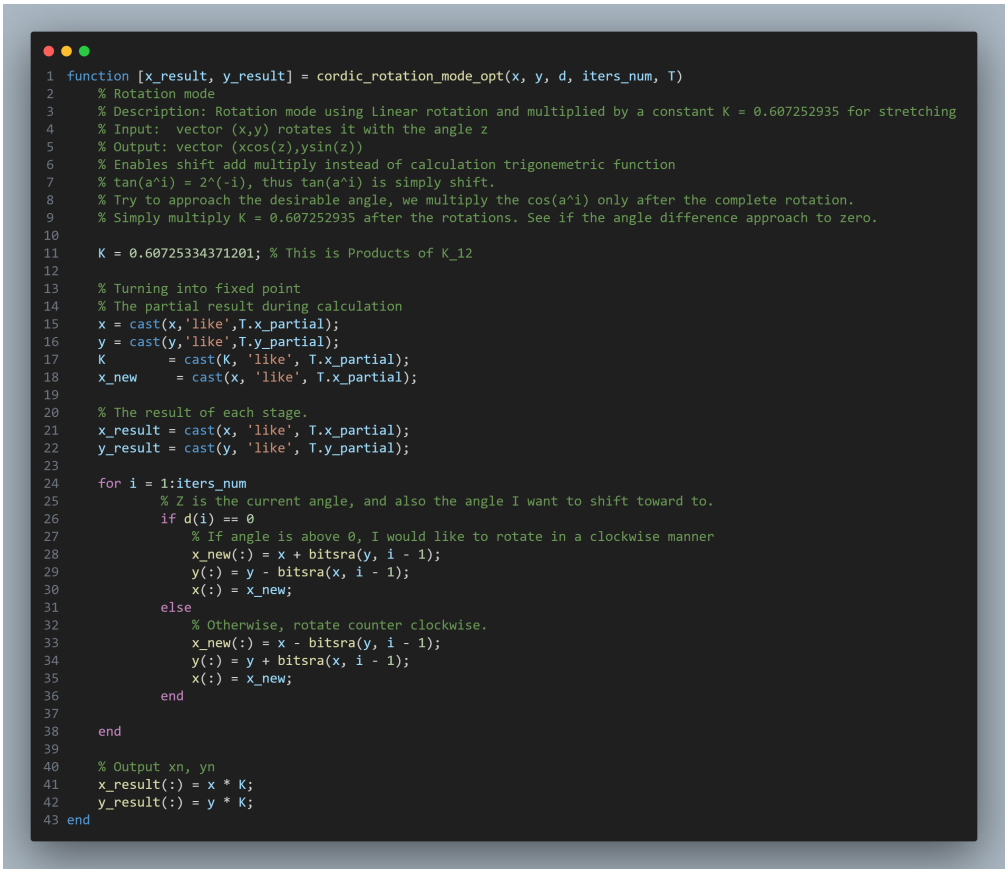
- Due to the hardware structure provided above , we notice that the vector mode(gray node) in red box has to rotates the same amount of degree as other rotation mode(white node), thus we can further optimize by simply propogate the direction vector generated by Vector Mode to other Rotation Mode.

Optimized Vector Mode

```
function [r,d] = cordic_vector_mode_opt(x, y, iters_num,T)
    % Vectoring mode
    % Description: Vectoring mode uses linear rotation s.t. y approach to 0 iteratively
    % Input:  vector (x,y) rotates it with the angle z
    % Output: Value after rotation (x,y) also the angle of tan(y/x)
    % Goal is trying to nullify y accumulating the angle when rotating.

    % Beware the boundary condition!!! The first qr cordic starts from the N+1 rows. Thus the first operation
    % Is (x,0) -> (r,0)!!
    K = 0.60725334371201;
    % Turning into fixed point
    % The partial result during calculation
    r = 0;
    x = cast(x,'like',T.x_partial);
    y = cast(y,'like',T.y_partial);
    r = cast(r,'like',T.x_partial);
    x_new     = cast(x, 'like', T.x_partial);
    K         = cast(K, 'like' , T.x_partial);

    % The result of each stage.
    d         = zeros(1,iters_num);
    for i = 1:iters_num
        % Z is the current angle, and also the angle I want to shift toward to.
        if sign(x * y) > 0
            % Since it is at the first dimension, we shoudl shift in a clockwise manner.
            % Since we are trying to obtain the angle, we must first add when we are rotating.
            x_new(:) = x + bitsra(y, i - 1);
            y(:)     = y - bitsra(x, i - 1);
            x(:) = x_new;
            d(i) = 0;
        else
            % Otherwise shift in a CounterClockwise(CC) manner
            x_new(:) = x - bitsra(y, i - 1);
            y(:) = y + bitsra(x, i - 1);
            x(:) = x_new;
            d(i) = 1;
        end
    end

   r(:) = x*K;
end
```

- Notice that now we does not need an LUT to store the angle alpha. However, we still need to send out the direction vector out such that the rotation mode rotates as vector mode did. Thus Vector Mode simply generates the direction of rotation. Since we do this in a Givens Rotation manner, we call this mode Givens Generation(GG).

Optimized Rotation Mode



```
function [x_result, y_result] = cordic_rotation_mode_opt(x, y, d, iters_num, T)
    % Rotation mode
    % Description: Rotation mode using Linear rotation and multiplied by a constant K = 0.607252935 for stretching
    % Input:  vector (x,y) rotates it with the angle z
    % Output: vector (xcos(z),ysin(z))
    % Enables shift add multiply instead of calculation trigonemetric function
    % tan(a^i) = 2^(-i), thus tan(a^i) is simply shift.
    % Try to approach the desirable angle, we multiply the cos(a^i) only after the complete rotation.
    % Simply multiply K = 0.607252935 after the rotations. See if the angle difference approach to zero.

    K = 0.60725334371201; % This is Products of K_12

    % Turning into fixed point
    % The partial result during calculation
    x = cast(x,'like',T.x_partial);
    y = cast(y,'like',T.y_partial);
    K        = cast(K, 'like', T.x_partial);
    x_new    = cast(x, 'like', T.x_partial);

    % The result of each stage.
    x_result = cast(x, 'like', T.x_partial);
    y_result = cast(y, 'like', T.y_partial);

    for i = 1:iters_num
            % Z is the current angle, and also the angle I want to shift toward to.
            if d(i) == 0
                % If angle is above 0, I would like to rotate in a clockwise manner
                x_new(:) = x + bitsra(y, i - 1);
                y(:) = y - bitsra(x, i - 1);
                x(:) = x_new;
            else
                % Otherwise, rotate counter clockwise.
                x_new(:) = x - bitsra(y, i - 1);
                y(:) = y + bitsra(x, i - 1);
                x(:) = x_new;
            end

    end

    % Output xn, yn
    x_result(:) = x * K;
    y_result(:) = y * K;
end
```

- Rotation mode receive the direction vector generated by vector mode to rotates the (x,y) vector to the desired (x',y'). Since we do this in a Givens rotation manner, we call this mode Givens Rotation(GR).

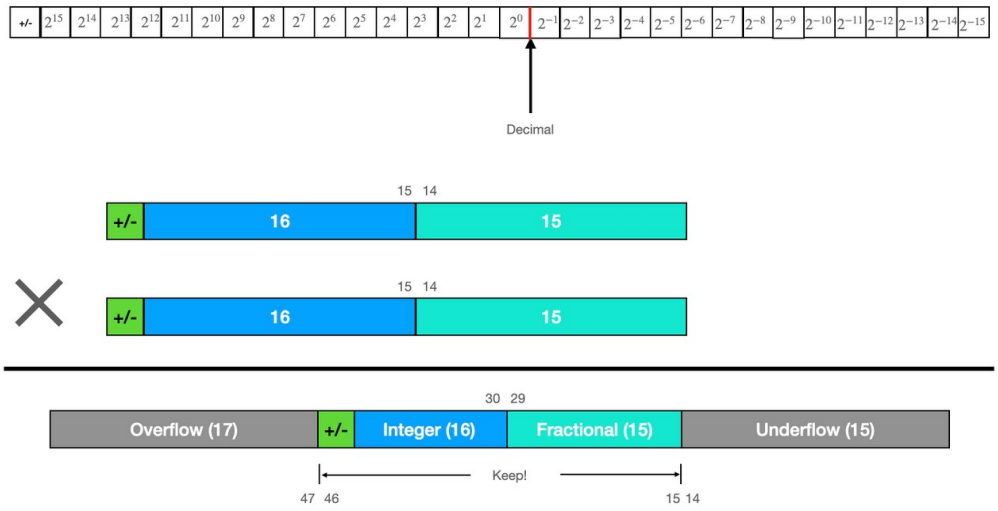
Optimized QR Cordic Algorithm

```m
function [Q, R] = qr_cordic_opt(M,T,S)
    % Algorithm from VLSI DSP lecture notes, 5-54, modify it using CORDIC algorithm
    N = length(M);
    % Augmenting the matrix I to the left, updating I alongside with A when performing rotation
    Q = eye(N);


    tmp1 = 0;
    tmp2 = 0;

    % Turning into fixed point
    M_        = cast(M, 'like',T.x_partial);

    R         = cast(M_,'like',T.x_output);
    Q         = cast(Q,'like' ,S.x_output);

    tmp1 = cast(tmp1,'like',T.x_partial);
    tmp2 = cast(tmp2,'like',T.y_partial);
    d=0;

    iters_num = 12;
    for k = 1:N

        for i = N:-1:k
            % This needed to be replaced with cordic, vectoring mode.
            % Use vector mode to calculate of givens rotation

            % The boundary condition of cordics. While N = 4, the input value is 0.

            if i == 4
                if k ~= N % The boundary condition for r44, do not rotate it anymore!
                    [r,d] = cordic_vector_mode_opt(M_(i,k), 0, iters_num,T);
                    M_(i,k) = r;
                end
            else
                [r,d] = cordic_vector_mode_opt(M_(i, k), M_(i+1, k), iters_num,T);
                % fprintf('Entry: (%d,%d)',i,k);
                M_(i,k) = r;
                M_(i+1,k) = 0;
            end

            % disp("After Vector mode");
            % M_


            % fprintf('Vector Mode:\n theta = %f , i = %d  \n',theta,i);

            for j = k+1:N
                % For R
                % This needed to be replaced with cordic, rotation mode
                % The boundary conditions.
                if i == N
                    if  k ~= N
                        [tmp1,tmp2] = cordic_rotation_mode_opt(M_(i,j), 0,d, iters_num,T);
                        M_(i, j) = tmp1;
                    end
                else
                    [tmp1,tmp2] = cordic_rotation_mode_opt(M_(i, j), M_(i+1, j),d, iters_num,T);
                    % fprintf('Entry: (%d,%d)',i,j);
                    % disp("Rotation mode Result x:");
                    % tmp1
                    % bin(tmp1)
                    M_(i, j) = tmp1;

                    % disp("Rotation mode Result y:");
                    % tmp2
                    % bin(tmp2)
                    M_(i+1, j) = tmp2;

                end

                % [tmp1, tmp2] = cordic_rotation_mode_opt(M_(i,j), M_( i+1,j ), d, iters_num,T);
                % fprintf('Rotation Mode:\n x = %f , y = %f , j = %d  \n',tmp1,tmp2,j);
            end

            % disp("After Rotations");
            % M_;

            if k == N
                break;
            end

            for j = 1:N
                if i == 4
                    d;
                    [tmp1,tmp2] = cordic_rotation_mode_opt(Q(i,j), 0,d, iters_num,S);
                    % tmp1
                    % bin(tmp1)
                    Q(i, j) = tmp1;
                else
                    [tmp1,tmp2] = cordic_rotation_mode_opt(Q(i, j), Q(i+1, j),d, iters_num,S);
                    Q(i, j) = tmp1;
                    Q(i+1, j) = tmp2;
                end

                % For Q, after calculation, take its transpose to get the correct Q, same for this portion.
                % [tmp1, tmp2] = cordic_rotation_mode_opt( Q(i-1,j), Q(i,j), d, iters_num,S);
            end
            M_;
            Q;
        end
    end
    Q;
    R(:) = M_;
    Q(:) = Q;
end
```

- Thus we can change the original QR algorithm into the QR Cordic Algorithm by simply replace the location where angle must be generated and where coordinates (x,y) must be calculated.
- Calculate angle, uses vector mode. Calculate vector uses Rotation mode.
- After we obtained the Correct Cordic Algorithm, we have to convert it into fix-point design.

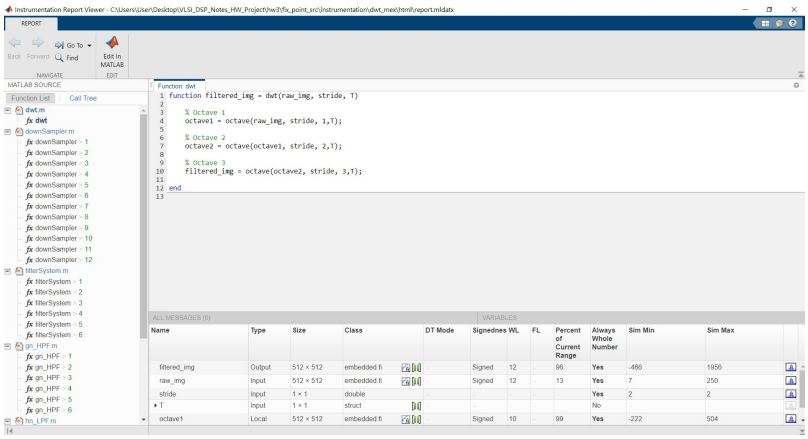
III. Fixed-point QR Cordic Design

Fixed point representation



- Fixed point representation is less expensive and easy to implement in hardware design. For more info please refer to HW3 DWT fixed point design.

Matlab Fixed-Point Designer



- Using the matlab fixed-point designer I can convert the original QR cordice algorithm into a fixed point one, then find the best fixed point word length and fraction length for my design to meet the criterion.

Criterion Delta

$$\delta = \frac{\sqrt{\sum (r_{ij} - \hat{r}_{ij})^2}}{\sqrt{\sum r_{ij}^2}} < 0.01$$

- The R matrix generated by fixed point algorithm must not be more than 1% error compared to the double precision R matrix. rij is the R matrix element calculated through double precision. rij_hat is the R matrix element calculate through fixed point algorithm.
- Since the input matrix is 8 bits integers, output matrices are 12 bits for both R & Q. We have 8 variables to search. The partial calculation word length and fraction length for both R & Q. Also the output fraction length for both R & Q. Also the word length and fraction length for K constant.

Tester for fixed point algorithm

```
%================================================================
%  QR Cordic checking for a random number of matrices
%================================================================
count = 0;
for i = 1:NUM_OF_MATRIX
    % Generating matrix
    matrix_i = randi([lower_bound,upper_bound],MATRIX_SIZE,MATRIX_SIZE);

    % Double calculation
    [q_double, r_double] = qr_cordic_opt(matrix_i, U,U);

    % Fixed point calculation
    matrix_i = cast(matrix_i, 'like', C);
    [q_f_opt, r_f_opt] = qr_cordic_opt_mex(matrix_i, T,S);

    % Convert back to double for verification
    q_double = double(q_double);
    r_double = double(r_double);

    q_f_opt = double(q_f_opt);
    r_f_opt = double(r_f_opt);

    % Should replace the lower triangle with 0, since it is not needed for metrics.
    r_double = r_double .* (1 - tril(ones(size(r_double))));
    r_f_opt = r_f_opt .* (1 - tril(ones(size(r_f_opt))));

    % Generate a bit string to check if met or not?
    [met_or_not(i), deltar_] = delta_calculation(r_double, r_f_opt, 0.01);
    [met_or_not(i), deltaq_] = delta_calculation(q_double, q_f_opt, 0.01);

    deltar(i) = deltar_;
    deltaq(i) = deltaq_;

end

disp("The largest delta of R within is:");
disp(max(deltar));
disp("Average delta for R:");
disp(mean(deltar,'all'));

disp("The largest delta of Q within is:");
disp(max(deltaq));
disp("Average delta Q:");
disp(mean(deltaq,'all'));
```

- This is the driver algorithm to test the average delta by running through a random number of matrices. To see if the proposed fixed point length meets the criteria or not.

Fixed point Word Length & Fraction Length Result

	R partial	Q partial	R output	Q output	K
Word Length	20	12	12	12	20
Fraction Length	10	10	3	10	10

```
The largest delta of R within is:
    0.0100

Average delta for R:
    0.0052

The largest delta of Q within is:
    0.1373

Average delta Q:
    0.0082
```

- The optimal word length and fraction length is found using the fixed point designer, and being tested using the delta error calculation.The result was tested by running through 1000 test cases for verification, notice that on average the result delta is within 1 percent, so this proposed fixed-point word length and fraction length meets the criteria.

IV. Testbench & Golden Model Generation

Matlab testbench generation

```
%============================================================
%  Write out golden pattern
%============================================================
matrix_golden = 'C:\Users\User\Desktop\VLSI_DSP_Notes_HW_Project\hw5\pattern\matrix_dat.txt';
q_golden = 'C:\Users\User\Desktop\VLSI_DSP_Notes_HW_Project\hw5\pattern\q_dat.txt';
r_golden = 'C:\Users\User\Desktop\VLSI_DSP_Notes_HW_Project\hw5\pattern\r_dat.txt';

% Writing  pattern
fileID_0 = fopen(matrix_golden, 'wt');
fileID_1 = fopen(q_golden, 'wt');
fileID_2 = fopen(r_golden, 'wt');

% fprintf(fileID_0, '%d \n\n', NUM_OF_MATRIX);
% Set the seed value
seedValue = 1234;

% Fix the seed for random number generation
rng(seedValue);

for i = 1:NUM_OF_MATRIX
    fprintf("Pat# %d",i);
    % Generating matrix
    matrix_i = randi([lower_bound, upper_bound], MATRIX_SIZE, MATRIX_SIZE);

    % Fixed point calculation
    matrix_i = cast(matrix_i, 'like', C);
    [q_f_opt, r_f_opt] = qr_cordic_opt_mex(matrix_i, T, S);

    for k = 1:MATRIX_SIZE
        for j = MATRIX_SIZE:-1:1
            matrix_fix_val = matrix_i(j, k);
            matrix_fix_bin = bin(matrix_i(j, k));
            % Append the pattern number at the start of every pattern.
            if j == 4 && k == 1
                fprintf(fileID_0, '%s  //  %f pat# %d (%d,%d) \n', matrix_fix_bin, matrix_fix_val, i, j, k);
            else
                fprintf(fileID_0, '%s  //  %f (%d,%d)\n', matrix_fix_bin, matrix_fix_val, j, k);
            end
        end
    end
    for j = 1:MATRIX_SIZE
        for k = 1:MATRIX_SIZE
            q_fix_val = q_f_opt(j, k);
            q_fix_bin = bin(q_f_opt(j, k));

            r_fix_val = r_f_opt(j, k)
            r_fix_bin = bin(r_f_opt(j, k));

            % the lower half of R should be regarded as zeroes, so instantiate a fixed point zero for them.
            zero_d = 0;
            zero = cast(zero_d, 'like', T.x_output);
            zero = bin(zero);

            % Append the pattern number at the start of every pattern.
            if j == 1 && k == 1
                fprintf(fileID_1, '%s  //  %f pat# %d (%d,%d)\n', q_fix_bin, q_fix_val, i, j, k);
            else
                fprintf(fileID_1, '%s  //  %f (%d,%d)\n', q_fix_bin, q_fix_val, j, k);
            end

            % R matrices, the lower part should all be zeroes.
            if j == 1 && k == 1
                fprintf(fileID_2, '%s  //  %f pat# %d (%d,%d)\n', r_fix_bin, r_fix_val, i, j, k);
            elseif j > k
                fprintf(fileID_2, '%s  //  %f (%d,%d)\n', zero, zero_d, j, k);
            else
                fprintf(fileID_2, '%s  //  %f (%d,%d)\n', r_fix_bin, r_fix_val, j, k);
            end
        end
    end
end

fclose(fileID_0);
fclose(fileID_1);
fclose(fileID_2);
```

- The following main loop generates A,Q,R patterns in matlab. Consist of qr_cordic_opt that is the fixed point algorithm we just derived. Send random matrix in to get the desired Q & R. Later write those result in binary format into the data.txt file. So that we can later read these data into the pseudoDRAM within the testbench.

Generation of A

```
01001001  //  73.000000 pat# 1 (4,1)
11110000  //  -16.000000 (3,1)
00011111  //  31.000000 (2,1)
10110001  //  -79.000000 (1,1)
01001101  //  77.000000 (4,2)
11000110  //  -58.000000 (3,2)
11000101  //  -59.000000 (2,2)
01000111  //  71.000000 (1,2)
00000000  //  0.000000 (4,3)
11011011  //  -37.000000 (3,3)
01100000  //  96.000000 (2,3)
01110101  //  117.000000 (1,3)
00001111  //  15.000000 (4,4)
11011110  //  -34.000000 (3,4)
00110110  //  54.000000 (2,4)
00101110  //  46.000000 (1,4)
01100001  //  97.000000 pat# 2 (4,1)
01000101  //  69.000000 (3,1)
10000011  //  -125.000000 (2,1)
00000000  //  0.000000 (1,1)
11011110  //  -34.000000 (4,2)
10010011  //  -109.000000 (3,2)
00011101  //  29.000000 (2,2)
11011101  //  -35.000000 (1,2)
01001001  //  73.000000 (4,3)
11100101  //  -27.000000 (3,3)
00100110  //  38.000000 (2,3)
01101110  //  110.000000 (1,3)
11101111  //  -17.000000 (4,4)
01011110  //  94.000000 (3,4)
00010001  //  17.000000 (2,4)
11010001  //  -47.000000 (1,4)
00110100  //  52.000000 pat# 3 (4,1)
00110100  //  52.000000 (3,1)
10100100  //  -92.000000 (2,1)
01001101  //  77.000000 (1,1)
01101000  //  104.000000 (4,2)
```

- The input matrix A is written out in an column-major reversed scanned order, such that the design can receive the data in a much more efficient manner.

Generation of R, Q

```
110001111010  //  -112.750000 pat# 1 (1,1)
000001000010  //  8.250000 (1,2)
000110010010  //  50.250000 (1,3)
000000010111  //  2.875000 (1,4)
000000000000  //  0.000000 (2,1)
010000100100  //  132.500000 (2,2)
000100000110  //  32.750000 (2,3)
000010111110  //  23.750000 (2,4)
000000000000  //  0.000000 (3,1)
000000000000  //  0.000000 (3,2)
010001111000  //  143.000000 (3,3)
001001011011  //  75.375000 (3,4)
000000000000  //  0.000000 (4,1)
000000000000  //  0.000000 (4,2)
000000000000  //  0.000000 (4,3)
111110110011  //  -9.625000 (4,4)
101010100000  //  -172.000000 pat# 2 (1,1)
001010011010  //  83.250000 (1,2)
111111101011  //  -2.625000 (1,3)
111110000010  //  -15.750000 (1,4)
000000000000  //  0.000000 (2,1)
001011001100  //  89.500000 (2,2)
111101000111  //  -23.125000 (2,3)
110111011011  //  -68.625000 (2,4)
000000000000  //  0.000000 (3,1)
000000000000  //  0.000000 (3,2)
101110110101  //  -137.375000 (3,3)
001000111100  //  71.500000 (3,4)
000000000000  //  0.000000 (4,1)
000000000000  //  0.000000 (4,2)
000000000000  //  0.000000 (4,3)
000100101101  //  37.625000 (4,4)
010001100010  //  140.250000 pat# 3 (1,1)
110110010110  //  -77.250000 (1,2)
111011100011  //  -35.625000 (1,3)
111100010011  //  -29.625000 (1,4)
000000000000  //  0.000000 (2,1)
```

- Both R matrix & Q matrix are written out to the dat file for comparision in testbench.

Testbench

```
//================================================================
  //  initial
  //================================================================
  initial
  begin
    //Reading in A
    $readmemb(`A_GOLDEN ,A_GOLDEN);
    //Put golden model in pseudoRAM
    $readmemb(`Q_GOLDEN ,Q_GOLDEN);
    $readmemb(`R_GOLDEN ,R_GOLDEN);

    in_valid = 0 ;
    in = 8'bx ;
    rst_n = 1 ;

    reset_task;
    total_cycles = 0 ;
    total_pat = 0 ;
    errors    = 0 ;

    @(negedge clk);
    for( patcount=0 ; patcount<PATNUM ; patcount=patcount+1 )
    begin

      qr_feed_data_task;

      total_pat = total_pat + 1 ;

      wait_outvalid;
      check_ans;
      delay_task;

      if(color < 100)
        $display("/033[38;5;-mPASS PATTERN NO.M/033[00m", color, patcount+1);
      else
        $display("/033[38;5;=mPASS PATTERN NO.M/033[00m", color, patcount+1);
    end


    #(1000);
    YOU_PASS_task;
    $finish;
  end
```

- The following is the testbench main loop for testing my circuit for a random number of test patterns. The reset_task resets the system, qr_feed_data_task feeds the value into circuit for consecutive 16 cycles. Later the testbench would wait for the out_valid signal to be raised. Checks the answer using the pseudoDRAM Q_GOLDEN & R_GOLDEN then delay in a random period of cycles, continue sending in new data into the circuits.

V. QR Cordic Hardware Design

Single Assignment Form

Single assignment form

For k from 1 to $N-1$

For i from N -1 to k

For j from k to N

$$ox(i,j,k) \leftarrow ny(i,j,k-1)$$

$$oy(i,j,k) \leftarrow \begin{cases} ny(i,j,k-1) & \text{if } i = N-1 \\ nx(i+1,j,k) & \text{if } i \neq N-1 \end{cases}$$

$$\theta(i,j,k) \leftarrow \begin{cases} \tan^{-1}(oy(i,j,k)/ox(i,j,k)) & \text{if } j = k \\ \theta(i,j-1,k) & \text{if } j \neq k \end{cases}$$

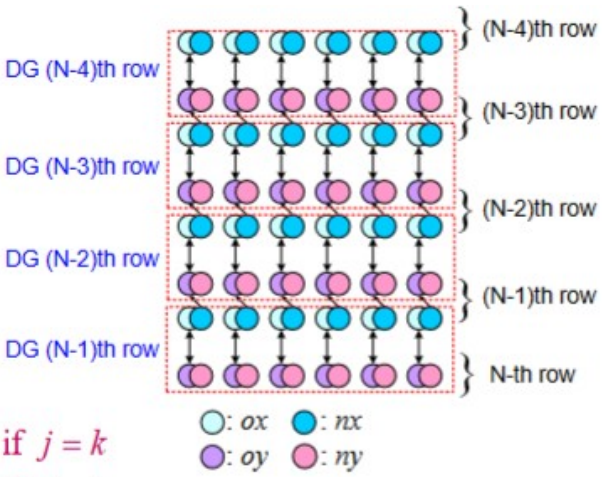
$$nx(i,j,k) \leftarrow ox(i,j,k)\cos(\theta(i,j,k)) + oy(i,j,k)\sin(\theta(i,j,k))$$

$$ny(i,j,k) \leftarrow -ox(i,j,k)\sin(\theta(i,j,k)) + oy(i,j,k)\cos(\theta(i,j,k))$$

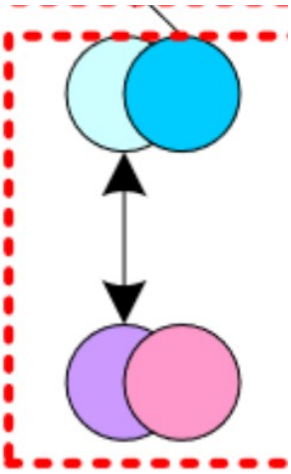
where

$nx(i,j,0) = a_{i,j}$: input matrix

$ny(N-1,j,0) = a(N,j)$: last row of input matrix

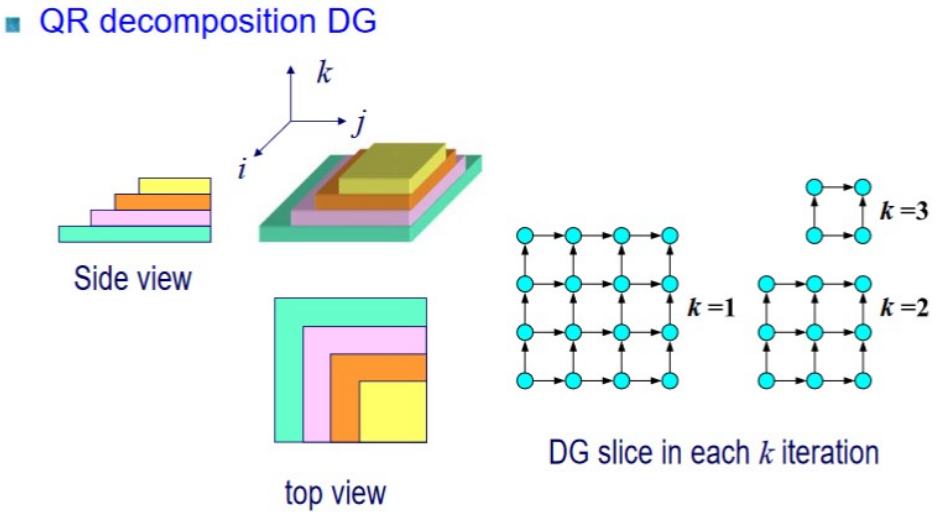


- We must first understand the Single Assignment form in the handout to understand how the suggested design is derived. We can notice that each nodes is a rotation of x,y. The result is then stored into nx,ny.

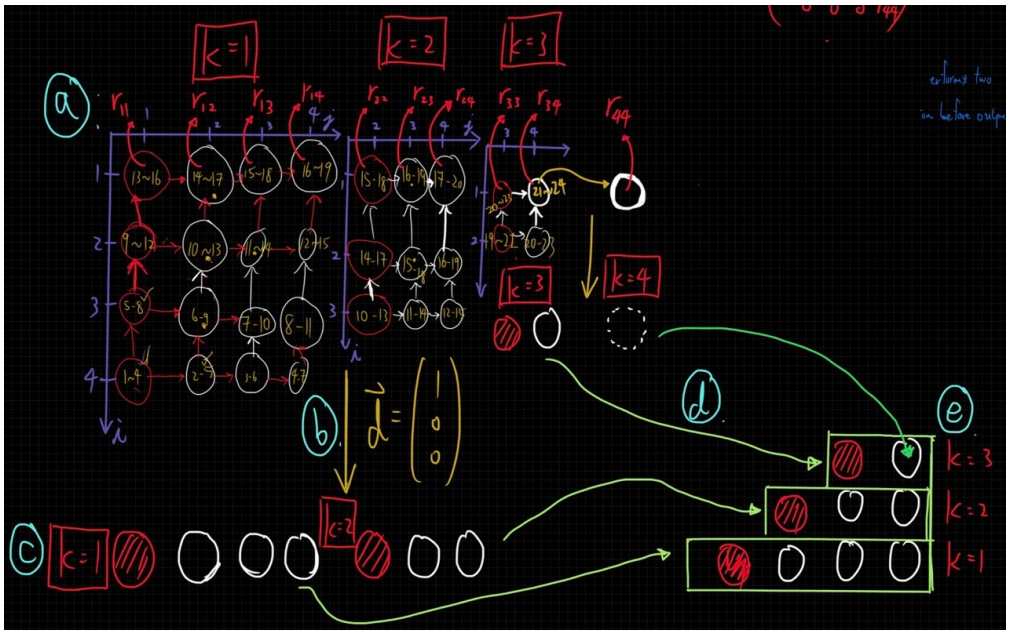


- Each of this nodes, represents a calculation nodes, which forms the fundamental calculation for our 3 dimensional DG construction.
- We can draw out the correspondent dependence graph using this SA form. Thus perform further derivation for our systolic array.

QR Dependence graph



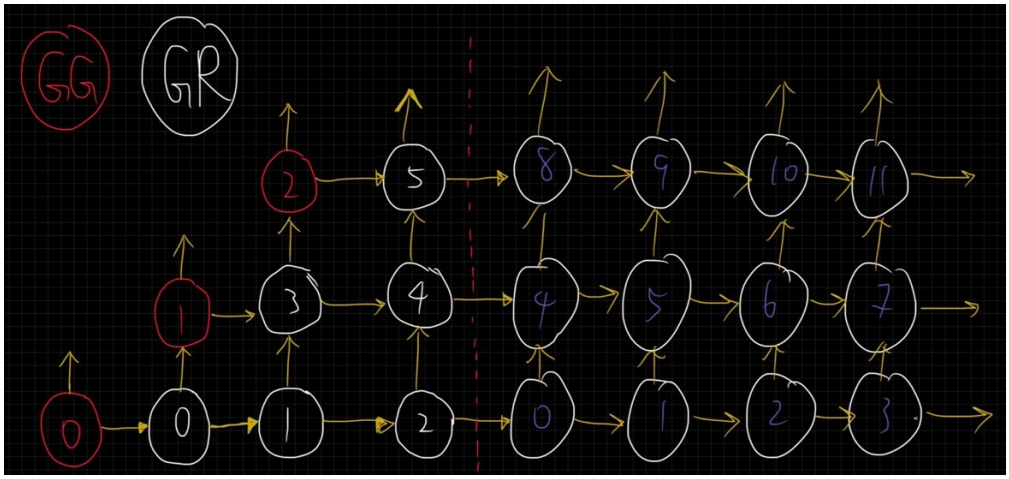
- The handout suggest a dependence graph which you can see is a pyramid like structure; however, this dependence graph is actually incorrect in a sense that the stacked k=2,3,4 should not be moving away from the i,j axes. The stacked k=2,3,4 should be move toward j axes. Otherwise, an incorrect DG might be created.



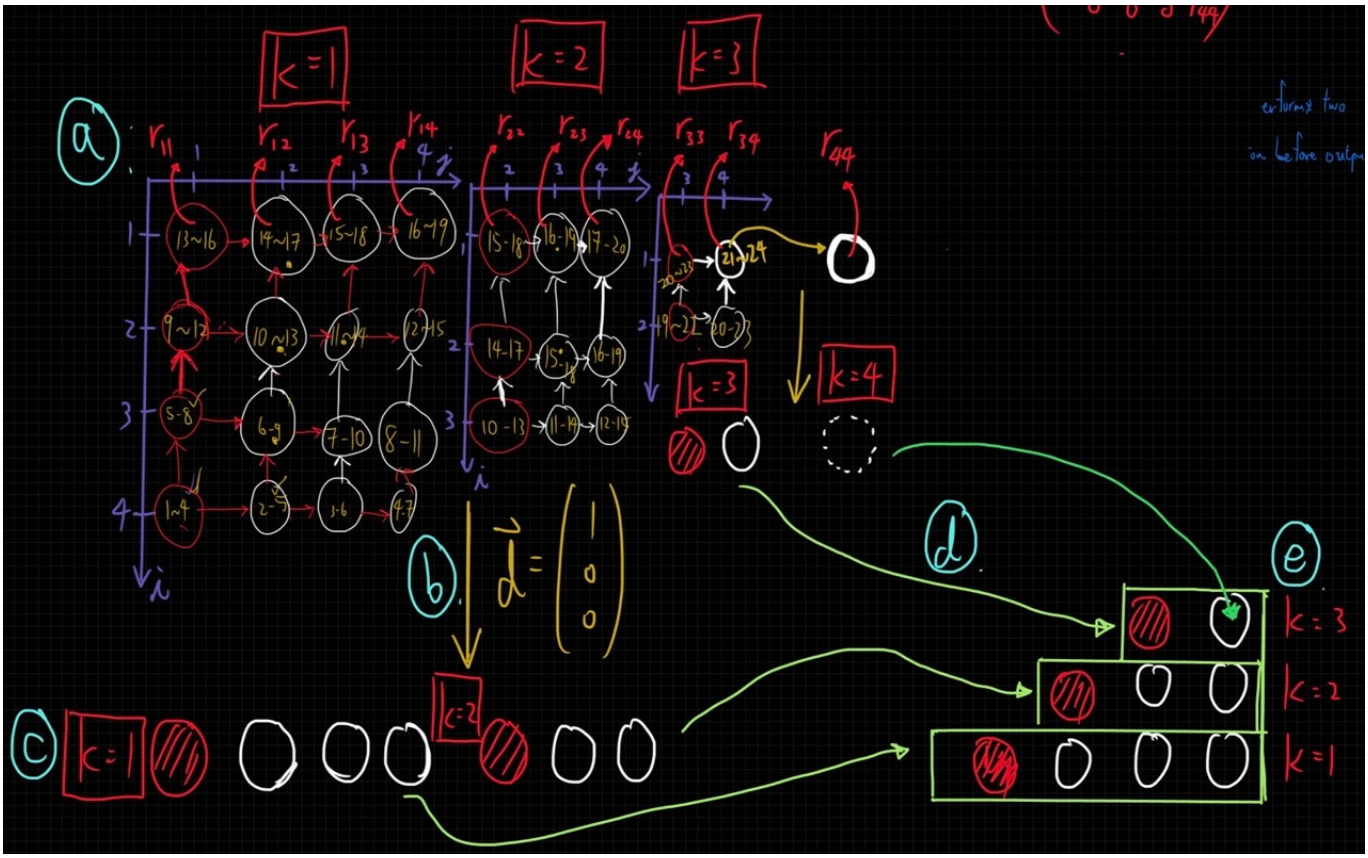
- a) is the Correct dependence graph, notice that the stacked k=2 is different than the stacked k=2 suggested in the handout, however, only by doing so we can created a reasonable DG.
- b) After performing the projection through d, we can get the triangular systolic array.
- c) Corresponent to k=1, which is the lowest part of our triangular systolic array DFG design.
- d) k=3 maps to the highest portion of the triangular systolic array DFG.
- e) Beware of the dg of k=4, since it is r44 in the R matrix, from the algorithm we know that, r44 is already created alongside with r34 in DG graph (1,4,3)=(i,j,k). Thus r44 would actually resides in the y value of DG node(1,4,3). As a result, k=4 is actually a redundant node thus can be removed.

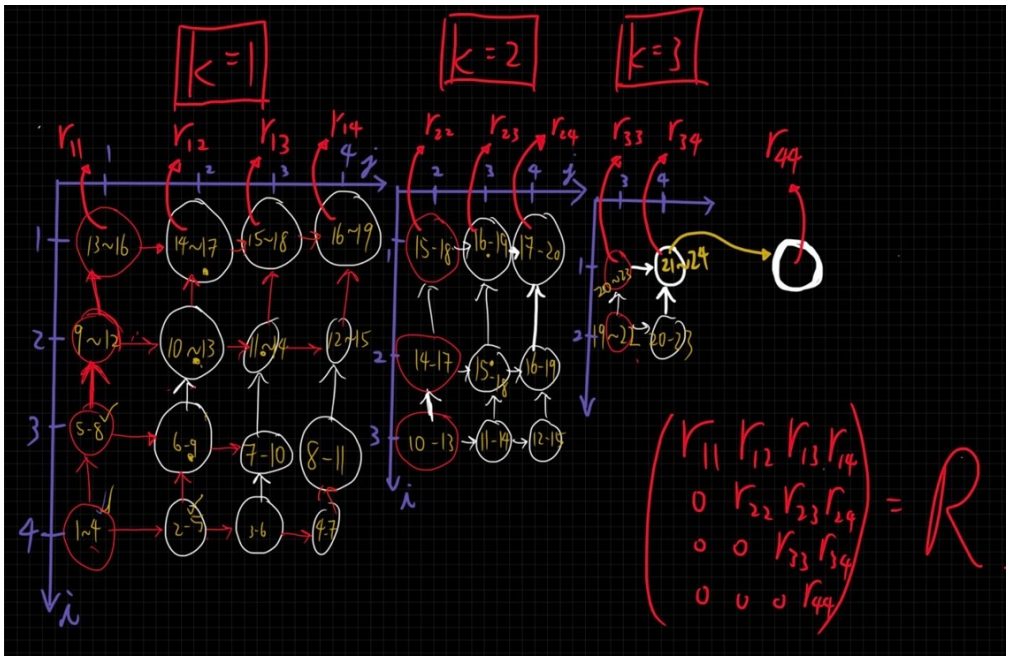
- Note that Q matrices dependence graph can be derived in a similiar manner,thus trivial. Notice that k=4 is also redundent for Q because no rotation is needed to be performed for k=4 DG of Q. The derivation of dependence graph is left as an exercise for the reader.

Scheduling & Correspondence of DG & DFG



- This is the DFG for our design. GG node is the Givens Generation nodes, generating the direction vector then the direction vector get propogated into the GR node which is the Givens Rotation nodes.

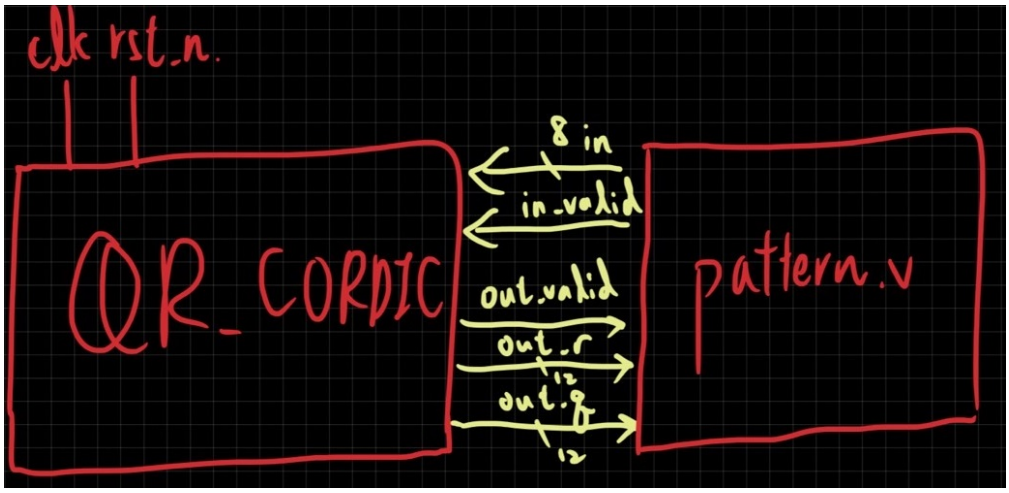




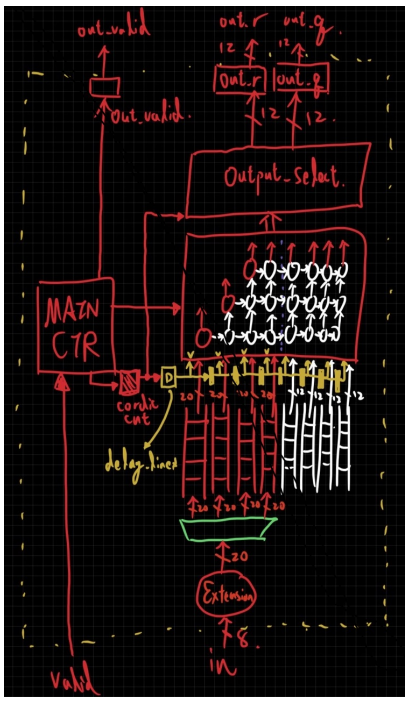
- The red r11 means that the result r11 would get produced at (1,1,1) node in the cycle 17 since its computation time is 13~16 cycles.
- The yellow number marked within the DG node specified when this node get executed. And after projection you can know exactly when and where this particular computation would take place within your systolic array.
- To mark the scheduling onto the DG, one must first obtained a systolic array design using the scheduling vector s=(-1 1 1).
- Then perform folding to extends the execution cycles of every node. Also at the same time, beware of the dependency when you try to perform folding.
- Specially noted that, each GR would execute for 2 cycles for the first time before they send their first value to the next layer. After sending their first value, they send their value once every clock cycle. This can be spotted within the dependence graph. This is important for the later design of our PE units.

VI. Hardware Implementation

System Architecture



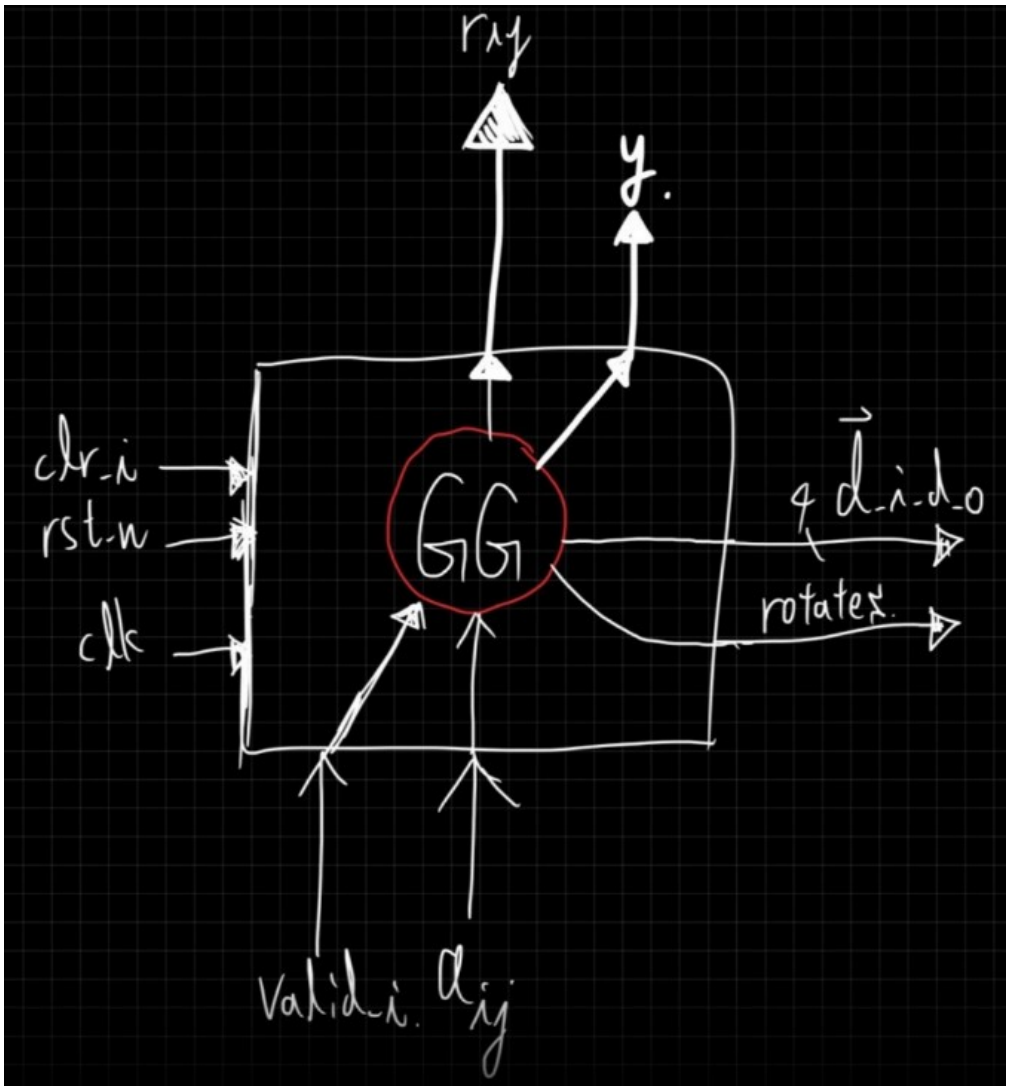
- The graph shows the system of Testbed, input matrix would be sent out from pattern.v to QR CORDIC. The QR CORDIC would send out_r & out_q data after they finish their computation. The testbench would compare the value with golden model during the output phase of the circuit.



- The following is the detailed system architecture of QR CORDIC module. The input buffer is implemented for sending in the data into the Cordic systolic array architecture in the correct moment and the correct PEs. FIFO is implemented by pure shift registers. Later after the computation of QR, output would be send out to pattern through out_r and out_q registers.

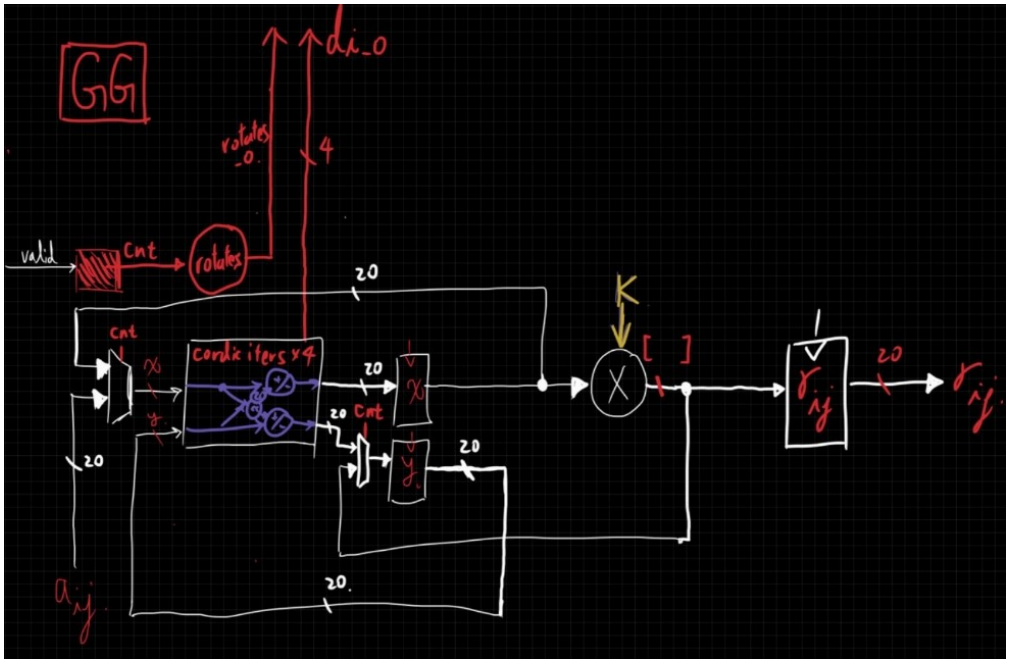
Design of GG

GG PE Block diagran



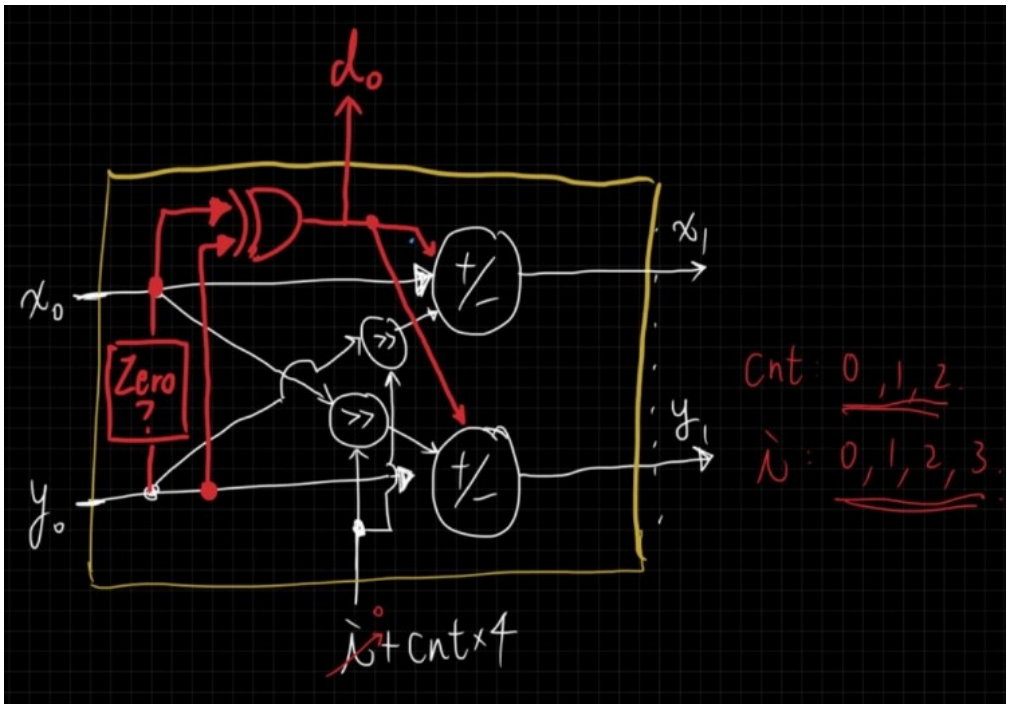
- This is the block diagram of Givens Generation Unit. valid_i signal is used to indicate when the GG starts its calculation. At the same time, one should send data through port aij. While calculating, GG generates the one cycle delayed rotation vector di alongside with the one cycle delayed rotates signal. Both of these signals get sent to GR units.

Structure of GG



- Details of GG contains a sub-control counter indicating whether GG is rotating, working or multiplying K. When rotating, GG generates di from the cordic iters block. The final (r,0) value after rotation would replace the old (x,y), the result would be (0,r) = (x,y).

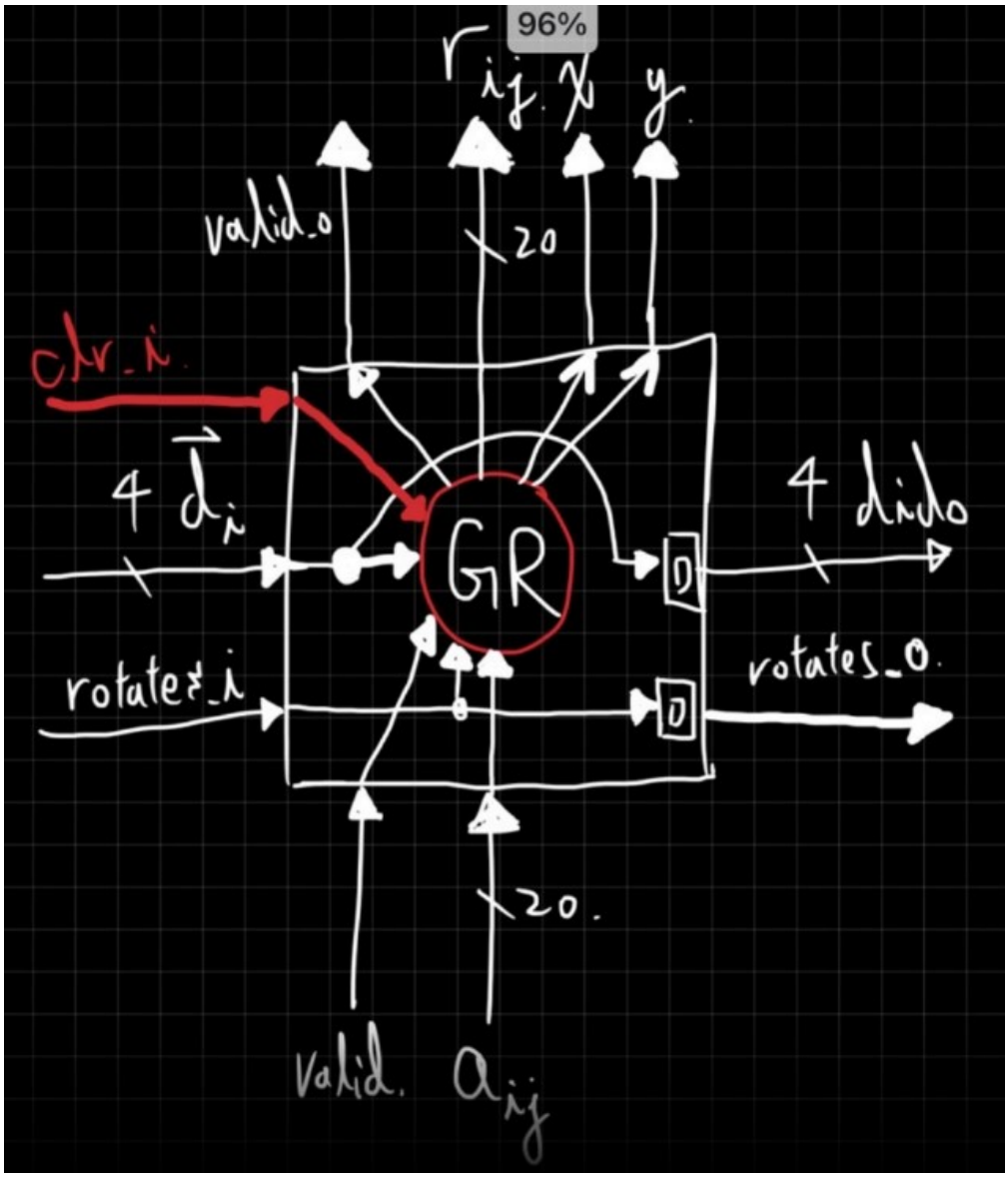
Structure of iteration



- Each of this sub-block represents an iteration in cordic. The boundary condition of determining whether the input value is 0 must be handled. Each iter sub-block generates a correspondent di signal during rotation. And the shift amount for both shifter is determined by the number of block also the value of counter.

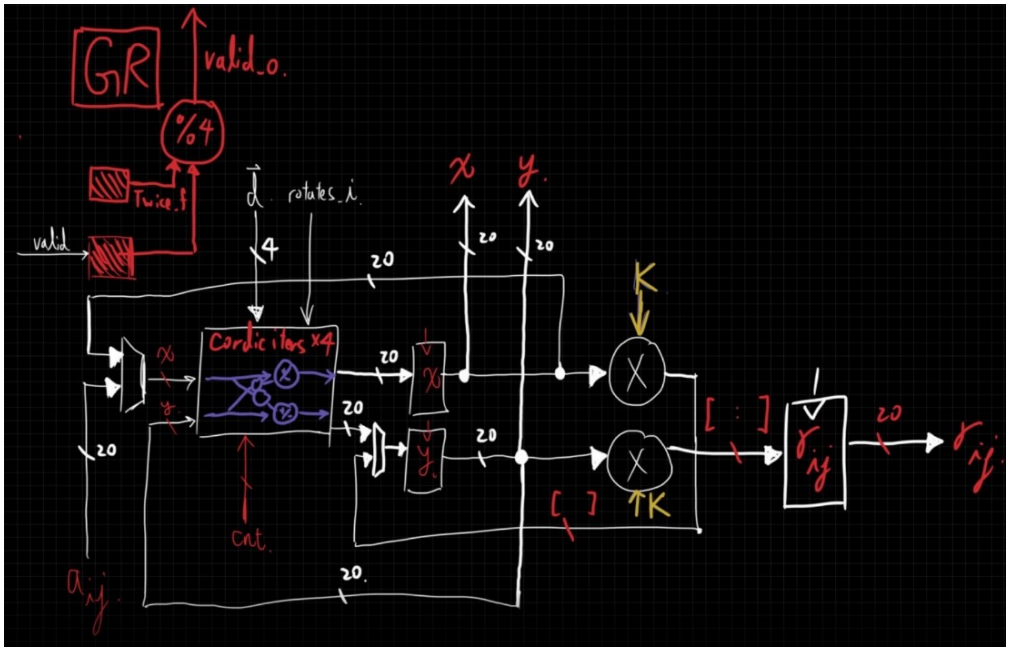
Design of GR

GR PE block diagram



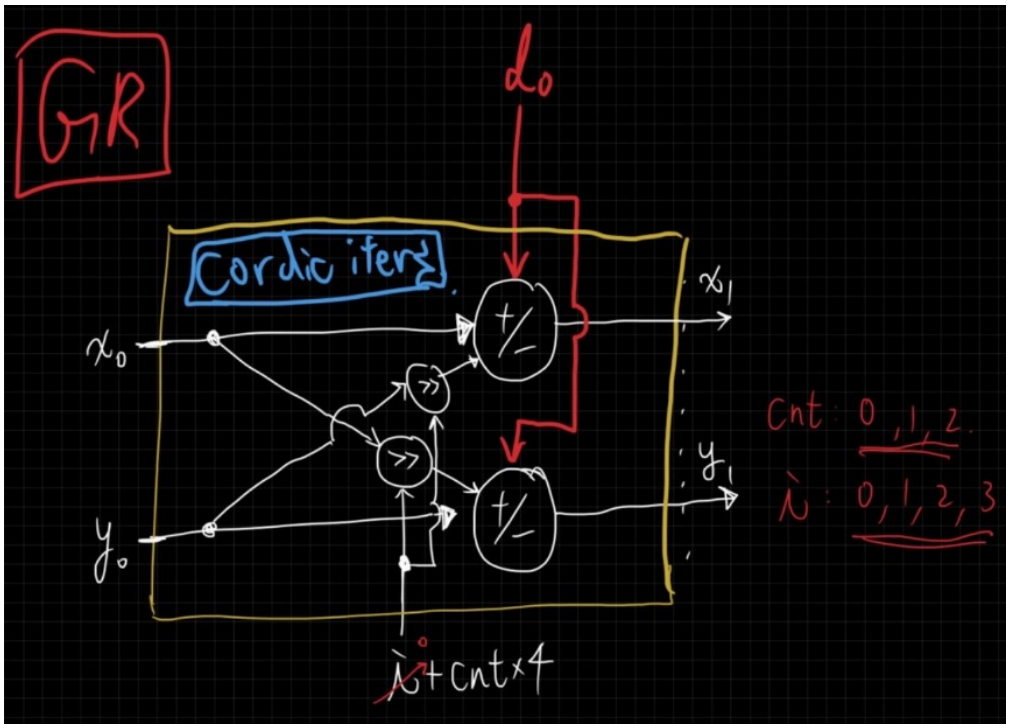
- Vector di is sent in alongside with rotates signal to start the rotation.
- di & rotation signal must also get propogate to the next GR for pipelined rotation.
- valid_i signal is sent in alongside with aij to tell GR when to start the computation.
- rij is sent out alongside with valid_o to allow the computation of higher dimension PEs from k=1 ~ k=2.
- Additional x,y ports allows debugging and easier value extraction after the whole computation.

Structure of GR



- GR is different than GG in some aspects, first it receive each direction vector from the previous GG or GR. Secondly, it needs two multipliers for the update of x,y due to the fact that both (x,y) is needed for the next full rotation.
- The valid_o signal can only be sent after 2 full rotation, thus it is controlled by the Twice_f register. Indicating whether the PE has already rotates two full iterations or not.

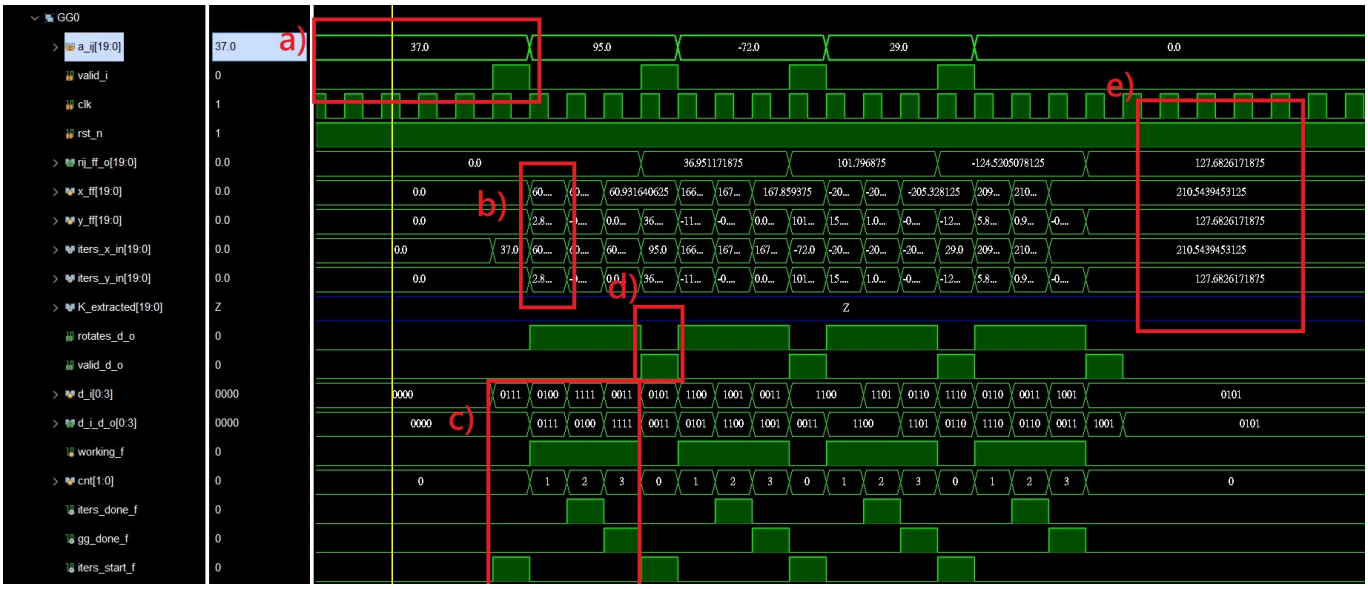
Structure of Iteration



- GR receive the direction vector from other PEs to rotates to its desired direction. Compare to GG, it does not need to generate the direction vector.

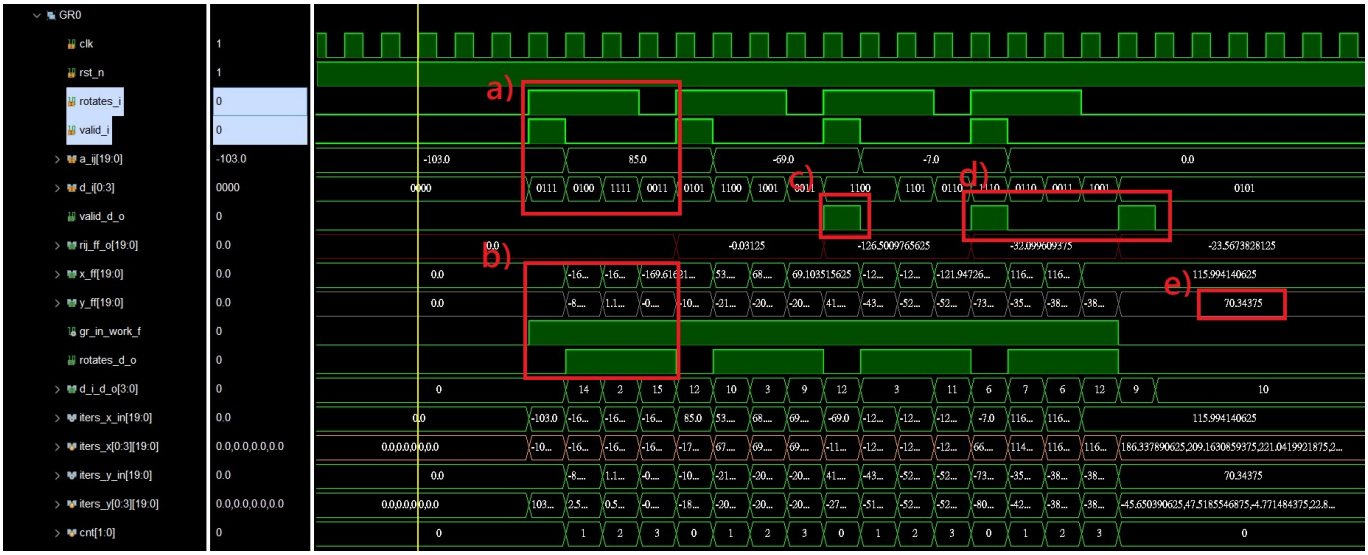
VII. Simulation Result

GG timing waveform



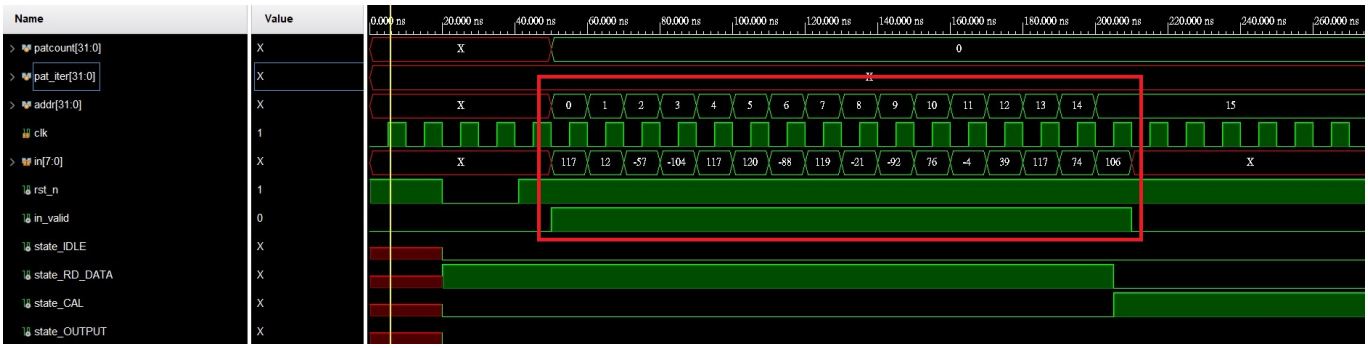
- a) Sends in the valid_i signal & value to tell GG starts working.
- b) 4 iterations of givens rotation result.
- c) Works for the next 4 consecutive cycles generating rotation direction d & the rotates signal to GR.
- d) This is actually not needed, simply used for debugging wire.
- e) The output final result of this PE.

GR Timing Waveform

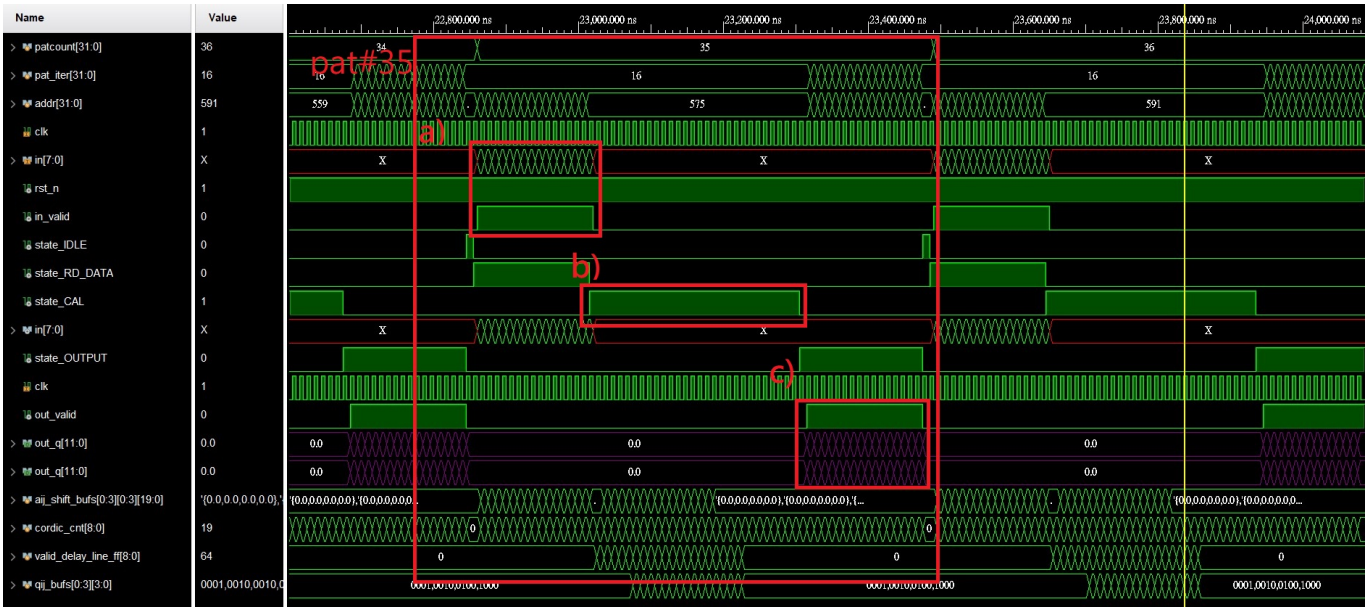


- a) Sends in the valid_i signal & value to tell GG starts working.
- b) 4 cycles of computation, 12 iterations for generating the new x,y
- c) Generate valid_o only after two full rotations for the first time.
- d) After the first time generate valid_o after each full rotation.
- e) The output final result of this PE.

Execution Cycle of a Pattern



- Data of Matrix A is sent in through 16 consecutive cycles.



- This is the waveform of pattern No.35.
- a) The input stage
- b) The calculation stage where computation has been taken place. After the computation is done, output is being sent out to the output stage.
- c) Output R & Q is being sent out the the testbench and compare with the golden model in the next 16 consecutive clock cycles.

```
/033[38;5;26mPASS PATTERN NO. 984/033[00m
/033[38;5;21mPASS PATTERN NO. 985/033[00m
/033[38;5;56mPASS PATTERN NO. 986/033[00m
/033[38;5;91mPASS PATTERN NO. 987/033[00m
/033[38;5;126mPASS PATTERN NO. 988/033[00m
/033[38;5;161mPASS PATTERN NO. 989/033[00m
/033[38;5;196mPASS PATTERN NO. 990/033[00m
/033[38;5;166mPASS PATTERN NO. 991/033[00m
/033[38;5;136mPASS PATTERN NO. 992/033[00m
/033[38;5;106mPASS PATTERN NO. 993/033[00m
/033[38;5;76mPASS PATTERN NO. 994/033[00m
/033[38;5;46mPASS PATTERN NO. 995/033[00m
/033[38;5;41mPASS PATTERN NO. 996/033[00m
/033[38;5;36mPASS PATTERN NO. 997/033[00m
/033[38;5;31mPASS PATTERN NO. 998/033[00m
/033[38;5;26mPASS PATTERN NO. 999/033[00m
/033[38;5;21mPASS PATTERN NO.1000/033[00m
--------------------------------------------------------------------------------------------------------------
                                        Congratulations!
                                You have passed all patterns!

                                Your execution cycles   = 30000 cycles
                                Your clock period       = 10.0 ns
                                Average Cycle Per pattern = 30.0 ns
                                Total latency           = 310000.0 ns
--------------------------------------------------------------------------------------------------------------
```

- After passing through 1000 patterns, total latency and average pattern latency is calculated. Notice that each QR takes 30 cycles to compute. While each consecutive QR algorithm takes 16 cycles due to the I/O limit. That is if we only have a single port, we must first read in and order the data for 16 cycles. Thus requires a 16 cyles latency for data reordering. This might be solved if we can have more bandwidth.

Note

- During debug, a mysterious 1 bit error drove me crazy, I eventually found out that it is caused by the boundary condition when generating the direction vector.
- While debugging, expecting which value would be produced beforehead is a powerful method while debugging, this can only be achieved with the correct algorithm and the support of high level language.
- DG is important for systolic archtiecture derivation, one must trust math!
- Note when tuning for matlab fimath fixed point computation method, one should remember to use Floor and Full Precision calculation. Otherwise simulation mismatch might occurs.

VIII. Acknowledgement

- I would like to ackowledge Prof. Huang for the design of the HW; also EECS undergraduate Chao Hsin-Tsai , EECS undergraduate Kuan-Ting Du, EECS graduate Shuan Yu Lin and my EECS graduate seniors Mo Shuan Kuma, Hun Rei Chang for their assistent throughout the research, design and implementation of this homework. Without them, this design cannot become a reality.

IX. References

[1] ECE 4760, Adams/Land, Fixed Point arithmetic , Spring 2021

[2] Best Practices for Converting MATLAB Code to Fixed Point Using Fixed-Point Designer

[3] VLSI DSP 2023, Lecture Handouts, Ch5 Mapping Algorithms Onto Array Structures 5-54~5-60, Y.T Hwang

[4] EE 5324 – VLSI Design II , Kia Bazargan , University of Minnesota , Part IX: CORDIC Algorithms

[5] UMN EE-5329 VLSI Signal Processing Lecture-15 (Spring 2019),Systolic Architecture Design , Prof. Keshab Parhi

[6] 2021_Spring_NCTU_ICLAB Lab03,Sudoku, Testbench and Pattern, mirkat1206

[7] Rounding Numbers without Adding a Bias, Gisselquist Technology,LLC 2017