Nate Mosk

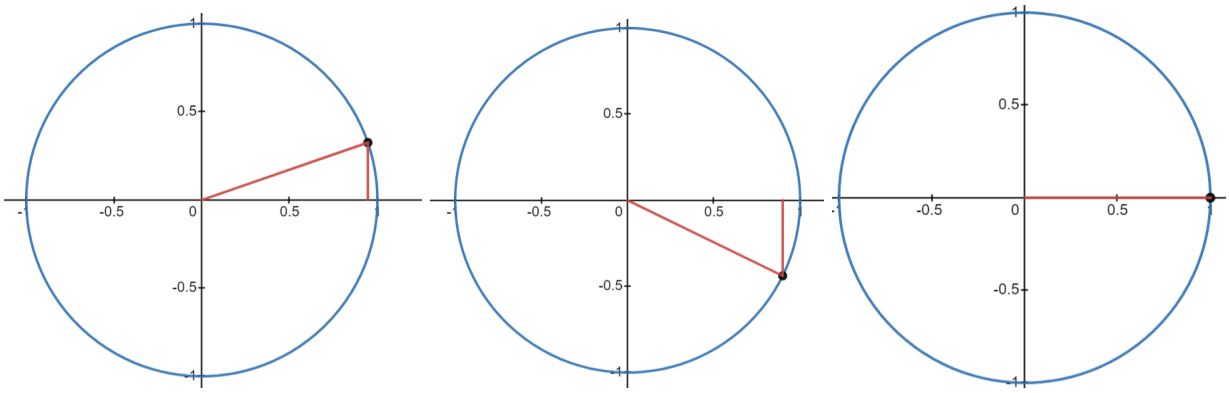
WES 237C Cordic

What is a cordic?

A cordic is a coordinate digital computer that was developed for use in avionic devices to calculate bearings. We take for granted the fact that calculators can find sine and cosine of any angle at any time. Here we are building such a tool from scratch.

Our particular project will take (x, y) coordinates as inputs and determine the polar coordinates. It's easy enough to find radius with the pythagorean theorem, r = sqrt(x^2 + y^2). To find the angle from the origin, theta, we will take iterative steps towards the origin. By adding up the length of steps to the origin and taking into account the direction, we can know the starting angle.

For example if we start at 19deg, and take a 45deg step towards the origin, we'll go to -26deg. Then if we take a 26 deg step towards the origin, we end at 0 deg. We can add up our steps 45 + (-26) to find our starting point was 19deg.



Step 1: 45deg clockwise, Step 2: 26deg counterclockwise, End point: 0deg

There is math backing the scaling of each step. Since we are using binary bits, we want to take steps that are linked to powers of 2. So that's why we do 45deg, then 26deg, etc. because arctan(2^0) is 45, arctan(2^-1) is 26, arctan(2^-2) is 14, and so on.

That's essentially it! We just program our IP block on our pynq board to do the work for us, adding and subtracting angles.

Baseline build

Here's the crux of the code. Remember, given an (x, y), we want to find the angle of the vector.

1. We check if the point is above or below the x axis.
2. If it's above the x axis, then we know to rotate clockwise. If it's below, we rotate counter clockwise. The shift values of our x and y are calculated accordingly.

3. We take our iteration from the current xy spot to the next xy spot which will be closer to the origin.

```
for (int j = 0 ; j < NO_ITER ; j++ ){

    // determine direction of rotation
    coef_t sigma = ( cur_int_y < 0 ) ? 1:-1;  // if y is not negative,
    if (cur_int_y < 0) {
        // determine magnitude of xy vector change
        cos_shift = cur_int_x >> j;
        sin_shift = cur_int_y >> j;
        printf("cos_shift: %d\n",cos_shift);
        printf("sin_shift: %d\n",sin_shift);
    }
    else
    {
        // determine magnitude of xy vector change
        cos_shift = ~(cur_int_x-1) >> j;
        sin_shift = ~(cur_int_y-1) >> j;
        printf("cos_shift: %d\n",cos_shift);
        printf("sin_shift: %d\n",sin_shift);
        }

    // rotate values and add to angle tracker
    cur_int_x = cur_int_x - sin_shift;
    cur_int_y = cur_int_y + cos_shift;
    current_theta = current_theta - sigma*angles[j];
}
```

In the baseline, we are taking 16 steps, and achieve a throughput of 1.22 MHz.

<u>Design considerations</u>

1. Simple Operations

By avoiding multiplications and divisions we can save time computing. Simple instructions like shifting bits and adding are quicker than multiplication.

That's why rather than dividing by 2, the bits are shifted down by one.

A slow model with multiplication was built to see just how much improvement is seen in the baseline by removing the multiplications from the main loop. There is a 2.65x speedup between the slow model and the baseline model once multiplications are removed.

Nate Mosk

2. Rotations

We can always make choices to improve our performance or optimize for a desired trait in our design.

If we want more precision and are willing to pay for it with slower speed, we can add iterations (i.e. more rotations) to our algorithm.

Every progressively smaller step we take towards the origin gives us higher resolution of the starting angle, but we have to do another computation cycle to take the step, therefore making our program slower.

Testing out a range of iterations, the throughput almost doubles when we only take 5 steps, but we lose a lot of precision. The error of our output theta is 534x larger when we take 5 steps than the baseline 16 steps we take.

Conversely, if we take 20 steps, the throughput is 15% slower than the baseline 16 steps, but the error is about 100x smaller.

At least 11 steps are needed to meet our threshold of < 0.001 total error between our 4 test points.

The accuracy stops improving so much towards the later steps. This could be avoided by using long long ints, because we could continue shifting the bits down, dividing by two to scale down our steps.

Model	Theta Accuracy	R accuracy	Throughput [MHz]
Baseline (16 iter)	4.36719E-05	3.47985E-08	1.22
20 iteration	5.97976E-07	3.47985E-08	1.04
5 iteration	0.023345545	3.47985E-08	2.38
11 iterations ints	0.000598536	3.47985E-08	1.57
11 iteration shorts	0.000598536	3.47985E-08	1.57

3. Data Type

Using different types to store data is another way we can optimize the code. We need enough bits in our type to hold enough information so that we can get enough steps in and find the angle with enough accuracy and precision. However, we likely can get higher throughput by not having too many extra bits.

See in the table above, there was not much difference in the throughput between a design using ints versus a design using shorts. Using long long type with 64 bits may have shown more of a difference in performance.

The shortest data type that was able to get the computations was short. Because we know we need at least 11 steps to get an answer with low enough error to pass, we need at least 2^11 bits. 2^11 is 2048 and a signed char goes from -128 to 127.

There's a tool called <ap_int.h> that allows us to use arbitrary length data types. We can specify ap_int<11> if we want a type with 11 bits.

I couldn't figure out how to make it work !! But I know it's possible. At most we would save a few bits. Given that there was minimal improvement from 32 bit ints to 16 bit shorts, I don't think the juice is worth the squeeze to switch to the ap_int.

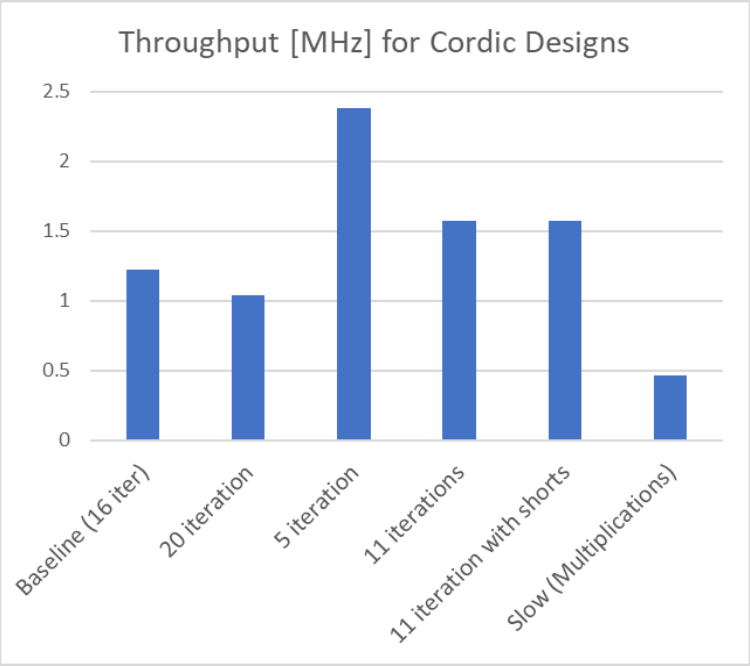
Results

The bar graph below allows for quick comparison of the different designs.

There are a couple of outlier cases. On the quick side, we can see huge throughput if we only have 5 iterations, but recall that design was not accurate enough. On the slow side we have the design that uses multiplication instead of bit shifting.

The ideal design would likely use ap_int<13> or some similar size type that has enough bits to make the 11+ steps needed to get the accuracy required for the application. Yet we'd save operations by having a smaller data type to handle.

Nate Mosk


Throughput [MHz] for Cordic Designs
2.5
2
1.5
1
0.5
0
Baseline (16 iter)
20 iteration
5 iteration
11 iterations
11 iteration with shorts
Slow (Multiplications)