CSE565M: Acceleration of Algorithms in Reconfigurable Logic

Learn by Doing: CORDIC (Pt. 2)

Anthony Cabrera

FL24::L06

Washington University in St. Louis

Table of contents

Recall the CORDIC Algorithm

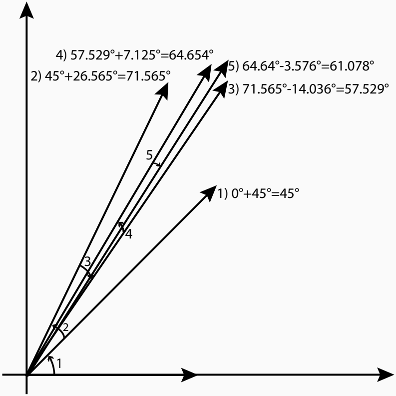


Figure 1: Calculating $\cos 60^\circ$ and $\sin 60^\circ$ using the CORDIC algorithm. Five rotations are performed using incrementally larger i values (0,1,2,3,4). The result is a vector with an angle of 61.078°. The corresponding x and y values of that vector give the approximate desired cosine and sine values.

Sequential Version of CORDIC

```
#include "cordic.h"
// The cordic_phase array holds the angle for the current rotation
// cordic_phase[0] =~ 0.785, cordic_phase[1] =~ 0.463, etc.
void cordic(THETA_TYPE theta, COS_SIN_TYPE &s, COS_SIN_TYPE &c) {
  // Set the initial vector that we will rotate
  COS_SIN_TYPE current_cos = 0.60735; // start at x = 1, y = 0, phi = 0
  COS_SIN_TYPE current_sin = 0.0;

  COS_SIN_TYPE factor = 1.0;
  // This loop iteratively rotates the initial vector to find the
  // sine and cosine values corresponding to the input theta angle
  for (int j = 0; j < NUM_ITERATIONS; j++) {
    // Determine if we are rotating by a positive or negative angle
    int sigma = (theta < 0) ? -1 : 1;

    // Multiply previous iteration by 2^(-j)
    COS_SIN_TYPE cos_shift = current_cos * sigma * factor;
    COS_SIN_TYPE sin_shift = current_sin * sigma * factor;

    // Perform the rotation
    current_cos = current_cos - sin_shift;
    current_sin = current_sin + cos_shift;

    // Determine the new theta
    theta = theta - sigma * cordic_phase[j];

    factor = factor / 2;
  }
  // Set the final sine and cosine values
  s = current_sin;
  c = current_cos;
}
```

CORDIC :: Exercise 4

Additonal rotations

Is it possible to get worse accuracy by performing more rotations? Provide an example when this would occur.

Cartesian to Polar Conversion

Cartesian to Polar Coordinates

With some modifications, the CORDIC can perform other functions. For example, it can convert between Cartesian and polar representations

$$(x, y) \longleftrightarrow (r, \theta)$$

Recall the relationship between these coordinates:

$$x = r\cos\theta$$

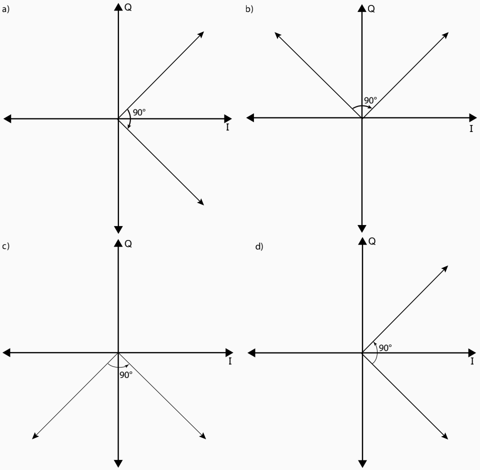
$$y = r\sin\theta$$

$$r = \sqrt{x^2 + y^2}$$

$$\theta = \mathrm{atan2}(y, x)$$

Using CORDIC for Cartesian2Polar

First, rotate the vector into quandrant I or IV



Using CORDIC for Cartesian2Polar

Subsequent rotations will allow the vector to reach a final angle of 0°. At this point, the radial value of the initial vector is the x value of the final rotated vector and the phase of the initial vector is the summation of all the angles that the CORDIC performed.

Number Representation

Representing Arbitrary Precision Integers in C/C++

- *standard* types like `int` and `long` are implementation defined
- In the C99 standard, the `inttypes.h` header introduced the types
  - `int8_t`
  - `int16_t`
  - `int32_t`
  - `int64_t`
  - `uint64_t`
- These can stil be awkward to use. Gets worse for fixed-point math

Representing Arbitrary Precision Integers in HLS

For these reasons, it's usually preferable to use C++ and the HLS template classes `ap_int<>`, `ap_uint<>`, `ap_fixed<>`, and `ap_ufixed<>` to represent arbitrary precision numbers.

- The `ap_int<>` and `ap_uint<>` template classes require a single integer template parameter that defines their width.
- Only if the result is assigned to a narrower bitwidth does overflow or underflow occur.

```
#include "ap_int.h"
ap_uint<15> a =0x4000;
ap_uint<15> b = 0x4000;
// p is assigned to 0x10000000.
ap_uint<30> p = a*b;
```

Representing Arbitrary Precision Integers in HLS

The `ap_fixed<>` and `ap_ufixed<>` template classes are similar, except that they require two integer template arguments that define the overall width (the total number of bits) and the number of integer bits.

```
#include "ap_fixed.h"
// 4.0 represented with 12 integer bits.
ap_ufixed<15,12> a = 4.0;
// 4.0 represented with 12 integer bits.
ap_ufixed<15,12> b = 4.0;
// p is assigned to 16.0 represented with 12 integer bits
ap_ufixed<18,12> p = a*b;

```

Floating Point

Floating point numbers provide a large amount of precision, but this comes at a cost;

- it requires significant amount of computation which in turn translates to a large amount of resource usage and many cycles of latency.
- Thus, floating point numbers should be avoided unless absolutely necessary as dictated by the accuracy requirements application.
    - Unfortunately, this is often a non-trivial task and there are not many good standard methods to automatically perform this translation. This is partially due to the fact that moving to fixed point will reduce the accuracy of the application and this tradeoff is best left to the designer.

CORDIC Optimizations

Updated CORDIC Code Listing

```
#include "cordic.h"
void cordic(THETA_TYPE theta, COS_SIN_TYPE &s, COS_SIN_TYPE &c) {
  COS_SIN_TYPE current_cos = 0.60735;
  COS_SIN_TYPE current_sin = 0.0;
  for (int j = 0; j < NUM_ITERATIONS; j++) {
    // Multiply previous iteration by 2^(-j).
    COS_SIN_TYPE cos_shift = current_cos >> j;
    COS_SIN_TYPE sin_shift = current_sin >> j;
    if (theta >= 0) {
      // Perform the rotation
      current_cos = current_cos - sin_shift;
      current_sin = current_sin + cos_shift;

      theta = theta - cordic_phase[j];
    } else {
      // Perform the rotation
      current_cos = current_cos + sin_shift;
      current_sin = current_sin - cos_shift;

      theta = theta + cordic_phase[j];
    }
  }
  s = current_sin;
  c = current_cos;
```

CORDIC :: Exercise 5

Varying the Data Type

How do you think the area, throughput, and precision of the sine and cosine results change as you vary the data type? Do you expect to see a significant difference when `THETA_TYPE` and `COS_SIN_TYPE` are floating point types vs. `ap_fixed<>` types? What about using the code?

References i