Messages of the week

- Projects
  - Team formation pretty much done – two people looking for a team to join
  - Time to meet and discuss project plans
  - Time available next week to prepare for Week 5 presentations

- Seminars
  - Please complete your selection this week
  - Meet with group, read and discuss papers suggested for your topic

- Labs
  - Discussion

UNSW SYDNEY

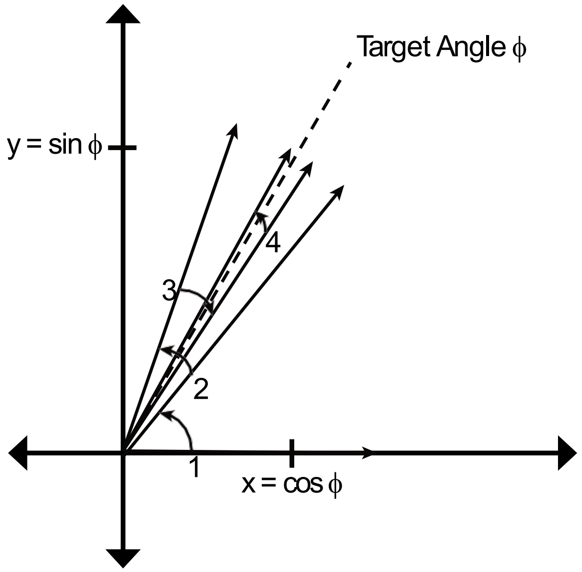
Ch 3. CORDIC

Overview

- CORDIC (Coordinate Rotation Digital Computer) is an efficient technique to evaluate trigonometric, hyperbolic, and other mathematical functions
  - It is a digit-by-digit algorithm that produces one additional digit of precision per iteration
  - We can therefore tune the *accuracy* of the algorithm to the application requirements, which is another common design evaluation metric alongside *performance* and *resource usage*
  - CORDIC performs simple computations using only addition, subtraction, bit shifting, and table lookups, which are efficient to implement in hardware
  - CORDIC has been used in maths co-processors, digital signal processing, Fourier transforms, and provided as IP cores to calculate trigonometric functions in FPGAs
- In this chapter we create an optimized CORDIC core using HLS
- The main HLS optimization that is highlighted in this chapter is choosing the correct number representation for the variables
- See pages 55-66 of the text for an introduction to and background on the CORDIC algorithm

CORDIC rotation

- The idea is to approach a target angle ϕ by rotating vector v_i



- At each step, the rotated vector v_i is given by:

$$v_i = K_i \begin{bmatrix} 1 & -\sigma_i 2^{-i} \\ \sigma_i 2^{-i} & 1 \end{bmatrix} \begin{bmatrix} x_{i-1} \\ y_{i-1} \end{bmatrix}$$

where

$$\sigma_i = \begin{cases} 1, \arg(v_{i-1}) < \phi \\ -1, \arg(v_{i-1}) \geq \phi \end{cases}$$

$$K_i = \frac{1}{\sqrt{1 + 2^{-2i}}}$$

and

$$K = \lim_{n \to \infty} \prod_{i=0}^{n-1} K_i \approx 0.607252935$$

CORDIC algorithm

```
// The file cordic.h holds definitions for the data types and constant values
#include "cordic.h"

// The cordic_phase array holds the angle for the current rotation
// cordic_phase[0] =~ 0.785 =~ arctan(2^-0) = arctan(1) in radians
// cordic_phase[1] =~ 0.463 =~ arctan (2^-1) = arctan(0.5) in radians

void cordic(THETA_TYPE theta, COS_SIN_TYPE &s, COS_SIN_TYPE &c)
{
  // Set the initial vector that we will rotate
  // current_cos = I; current_sin = Q
  COS_SIN_TYPE current_cos = 0.60725; // set magnitude of initial
  COS_SIN_TYPE current_sin = 0.0;     // vector to the reciprocal
                                      // of infinite CORDIC gain, K
  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;
}
```

Listing of cordic.h

```
#ifndef CORDIC_H
#define CORDIC_H
#include "ap_fixed.h"

typedef float THETA_TYPE;
typedef float COS_SIN_TYPE;

const int NUM_ITERATIONS=32;

static THETA_TYPE
cordic_phase[64]={0.78539816339744828000,0.46364760900080609000,
0.24497866312686414000,…,0.00000000000000000011};// arctan(2^-i)

void cordic(THETA_TYPE theta, COS_SIN_TYPE &s, COS_SIN_TYPE &c);
#endif
```


UNSW SYDNEY

Arbitrary precision numbers

- Rather than being restricted to using data types that are 8, 16, 32 or 64 bits wide, Vivado HLS provides several C++ template classes to represent arbitrary precision numbers (with specifically chosen bit widths)
  - The `ap_int<>` and `ap_uint<>` integer template classes require a single integer template parameter to define their width
  - The `ap_fixed<>` and `ap_ufixed<>` fixed point template classes require two integer template arguments that define (1) the overall width (total number of bits) and (2) the number of integer bits
  - For example:

```
#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;

#include "ap_fixed.h"
// 4.0 represented with 12 integer bits
ap_ufixed<15,12> a = 4.0; // a = 0b000000000100000
// 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; // p = 0b000000010000000000
```

Overflow and underflow

- Overflow occurs when a number is larger than the largest number that can be represented in a given number of bits
- Underflow occurs when a number is smaller than the smallest number that can be represented
- Both are commonly handled by dropping the most significant bits of the original number in a process often termed wrapping
  - Beware that wrapping can cause +ve numbers to become –ve and vice-versa

2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	
0	0	1	0	1	1	0	1	0	0	= 11.25
	0	1	0	1	1	0	1	0	0	= 11.25
		1	0	1	1	0	1	0	0	= 11.25
			0	1	1	0	1	0	0	= 3.25

Rounding

- When a number cannot be represented precisely in a given number of fractional bits, rounding is necessary
  - There are several ways of doing this – see quantization modes in Xilinx Vivado HLS User Guide, UG902, p556 for more details

Just drop the extra fractional bits
Called rounding down
Corresponds to `floor()`

0b0100.00	= 4.0		0b0100.0	= 4.0
0b0011.11	= 3.75		0b0011.1	= 3.5
0b0011.10	= 3.5		0b0011.1	= 3.5
0b0011.01	= 3.25		0b0011.0	= 3.0
0b0011.00	= 3.0	→ Round to Negative Infinity →	0b0011.0	= 3.0
0b1100.00	= −4.0		0b1100.0	= −4.0
0b1011.11	= −4.25		0b1011.1	= −4.5
0b1011.10	= −4.5		0b1011.1	= −4.5
0b1011.01	= −4.75		0b1011.0	= −5.0
0b1011.00	= −5.0		0b1011.0	= −5.0

Also could round up, as in `ceil()`
Or round to zero, as in `trunc()`
Or round to infinity, as in `round()`

A better way may be to round to the nearest even number, if the number can't be represented exactly, as implemented by `lrint()`

0b0100.00	= 4.0		0b0100.0	= 4.0
0b0011.11	= 3.75		0b0100.0	= 4.0
0b0011.10	= 3.5		0b0011.1	= 3.5
0b0011.01	= 3.25		0b0011.0	= 3.0
0b0011.00	= 3.0	→ Round to Nearest Even →	0b0011.0	= 3.0
0b1100.00	= −4.0		0b1100.0	= −4.0
0b1011.11	= −4.25		0b1100.0	= −4.0
0b1011.10	= −4.5		0b1011.1	= −4.5
0b1011.01	= −4.75		0b1011.0	= −5.0
0b1011.00	= −5.0		0b1011.0	= −5.0

UNSW SYDNEY

Floating point

- Vivado HLS can synthesize `float` data types, BUT they will consume considerable resources and have a high latency.
- When targetting FPGAs, it is FAR BETTER to use fixed point arithmetic to represent fractional numbers
- Given that the desired precision, performance and utilization are application/designer dependent, <u>there is no standard approach as to which fixed point representation should be used</u>
- A standard approach is to start with a floating point representation to obtain a functionally correct implementation. THEN optimize the number representation to reduce resource usage and increase performance

CORDIC optimizations

- The original code works using either floating or fixed point data types
  - It contains several multiplications involving `sigma` and `factor`
- Since a common aim is to eliminate the multiplications, the code can be restructured using shift operations and alternative code branches to update the angle of the rotated vector

Fixed-point and optimized CORDIC

```
// The file cordic.h holds definitions for the data types and
// constant values
#include "cordic.h"

// The cordic_phase array holds the angle for the current rotation
// cordic_phase[0] =~ 0.785
// cordic_phase[1] =~ 0.463

void cordic(THETA_TYPE theta, COS_SIN_TYPE &s, COS_SIN_TYPE &c)
{
  // Set the initial vector that we will rotate
  // current_cos = I; current_sin = Q
  COS_SIN_TYPE current_cos = 0.60725;
  COS_SIN_TYPE current_sin = 0.0;

  // 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++) {
      // Multiply previous iteration by 2^(-j).  This is equivalent
      // to a right shift by j on a fixed-point number.
      COS_SIN_TYPE cos_shift = current_cos >> j;
      COS_SIN_TYPE sin_shift = current_sin >> j;

    // Determine if we are rotating by a positive
    // or negative angle
    if(theta >= 0) {
        // Perform the rotation
        current_cos = current_cos - sin_shift;
        current_sin = current_sin + cos_shift;

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

        // Determine the new theta
        theta = theta + cordic_phase[j];
    } // end if
  }  // end for

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

CORDIC optimizations

- The original code works using either floating or fixed point data types
  - It contains several multiplications involving `sigma` and `factor`
- Since a common aim is to eliminate the multiplications, the code can be restructured using shift operations and alternative code branches to update the angle of the rotated vector
- We can also tune the accuracy of the algorithm by varying the number of iterations of the main loop, however, not without impacting on loop latency
- In the lab, I ask you to take a look not just at these optimizations, but also to examine the impact of loop unrolling and pipelining on the performance and utilization as well