Embedded Signal Processing Systems SS2020

Lab: Implementation of CORDIC based DDFS System

Report submitted by:

Aadarsh Kumar Singh
Matrikel Number : 766499

Hari Krishna Yelchuri
Matrikel Number : 766518

30.06.2020

Table of Contents

1 Aim

- The requirements of the lab are as follows:
  1. Create a SW-based reference implementation for a full-range CORDIC algorithm operating in rotation mode and verify its correct operation for input test vectors residing in all four quadrants.
  2. Implement the same cordic system using SystemVerilog and verify its functionality by creating a dedicated testbench and simulating it in Modelsim.
  3. Implement a CORDIC based DDFS system using the same CORDIC core developed above and verify its functionality of the CORDIC based DDFS system creating a dedicated testbench and simulating it in Modelsim.

2 Implementation

2.1 Software Used

- SW- based reference implementation of the cordic algorithm was done in C because we wanted to make the implementation closer to the actual hardware, so that we are able to do decide on bit-width, number representations in HW.
- To find the the number of iterations necessary we used python script, we analyzed by changing the number of iterations how the accuracy and precision were affected, we found greater the number of iterations, higher the precision.
- ModelSim software was used for System Verilog Implementation of CORDIC based DDFS system and for simulation

2.2 Analysis and theory behind our implementation

- **Usage of Fixed-Point Representation:**
  - CORDIC algorithm can be executed using additions, subtractions and shift operations. To implement the fractional arithmetic involved with this algorithm we focused on fixed point representation.
  - The key reason was to avoid usage of floating-point numbers in an algorithm which is aimed to be implemented on hardware.

- **Format of Fixed-Point Representation used and the Idea behind the format:**
  - We converted the specified CORDIC angles (45, 26….) from degrees to radians.
  - Then we converted these angles as fixed-point representations. We chose to have 32 bits in total, out of which 28 least significant bits form the fractional part, next three represent the whole number and the most significant bit decides the sign. This can also be called as <32,28> fixed point representation. (<1 bit(sign): 3 bits (decimal part): 28 bits (fractional part) >)
  - The idea behind choosing <32,28>, was that when maximum possible angle 360 degrees is converted to radians it results in 6.28391. The decimal part of this angle i.e **6** can be represented using 3 bits in binary, and we have one bit reserved for the sign, then we are left with 28 bits for the fractional part. Using this fixed-point representation, with 28 fractional bits has helped us in getting very precise values

- Then, after this we converted the value of Pi from radians to <32,28> fixed point representation and we got 843314856.
  .

- **Deciding the Number of iterations**
  - When Number of iterations is more (tends to infinity) the gain factor K becomes a constant.
  - Now for determining the value of K-CONST we first had to fix the number of iterations. From our observation using trial and error method in the python script, we observed that we were getting good precision at 12 or more iterations. So, we decided to go with 12 iterations. Now, to calculate K-CONST we derived its value for 12 iterations and converted it to <32,28> which gave us the value 163008218.

- **Pre-Scaling to compute angles**
  - As all the values were being pre-scaled in the fixed-point implementation format, input values x and y also had to modified accordingly.
  - The new value of x = x * 2^28 and y = y * 2^28. This pre-processing of all the variables was important to compute the angles out of the region of convergence.
  - In the end to get the exact fractional value of cos … and sin … we divided the respective values in fixed point representation by 2^28.

2.3 Implementation in c

In our implementation in C, we started with creating macros for K-Const, Pi and the total number of iterations. Then we have a table of restricted angles of data type int32_t. All these values have been scaled to our selected fixed-point representation as discussed in our theory above. Two arrays have been created to store the values of x and y after every iteration. The last value of these array is taken as the final value. A small function convertIntoRadian (float angle) converts an angle in degrees to radians. This function is used to calculate the target angle in radians.

Now before calling the main function, we have to adjust the input according to the angle of rotation. When the angle of rotation is less than 90 degrees input values to the algorithm are (x, y). Now when the angle of rotation is greater than 90 and less than 180, target angle is the difference between input angle and 90 degree, and the input to algorithm is (-y, x). The same continues for 3rd quadrant angle, target angle is difference of input angle and 180 degree. For 4th quadrant angle, target angle is difference of input and 270 degree and input to the algorithm is (y, -x). This pre-processing of the angle and input values is important to fit with the restrictions of cordic algorithm.

The main function, takes as parameters the starting values of x and y and also the angle with which the rotation has to be done. A local variable sigma has been defined to check if the coming iteration will be clockwise or anti-clockwise. Here we also do the pre-scaling of input x and y to our fixed-point representation before starting with our iterations. A while loop runs for the defined number of iterations (12), and calculates the next values of x, y and z (angle). The three equations for these are as follows,

1. x_next = (x_current – (sigma * (y_current >> i)));
2. y_next = (y_current – (sigma * (x_current >> i)));
3. z_next = (x_current – (sigma * restrictedAngles[i]));

Here, the sign of z-next decides the value of sigma to be 1 or -1. After each iteration the value of x and y are fed into the array. Now when all the 12 iterations are over, we go to display the final value. Here, the last value from the arrays are picked and down-scaled to compute a value in float. This computed value is displayed on the console.

2.4 Implementation in SystemVerilog

For the implementation of cordic algorithm in System Verilog we have, cordic_algorithm.sv file containing the module cordic_algorithm with output parameters cosine and sine, and input parameters are the pre-scaled input values of x any y and the angle of rotation. Firstly, we create a atan table consisting of 12 restricted angles of rotation. Then we have three arrays of 12 elements, two for the inputs x and y and one for the angle. The data width of all of them is 32 bits.

An always_comb block starts by observing the quadrant of the angle of rotation and computes the new values of x and y accordingly. This is done in the exact same way as in c implementation. After this comes a for loop running for 12 times and calculating and storing the new values of x and y after each iteration. Variables, x_shiftRight and y_shiftRight consist of the right shifted values of x and y according to the current iteration going on. Again, the next values of x, y and z are calculated with the same equations as shown in the C implementation. Here, we did not go for a separate variable to determine the direction of rotation, instead we observe the MSB of z-next and determine the direction.

A testbench cordic_algorithm_tb is created to test this module. Here, we define the pre-scaled values of the input. Angle of rotation is specified and two variables for sine and cosine output are created and fed to the module. In this testbench we observe the output of the module at five different angles, 30, 60, 120, 210 and 300. The values of all the sine and cosine outputs are printed after performing the down scaling operation.

This cordic_algorithm module is used by the module ddfs_using_cordic to generate DDFS. This module has input parameters clock and reset and we get sine and cosine values at the output. Here we declare the value of angle with respect to 1 degree. The input values x and y are also pre-scaled here. A variable count has been declared which can reach a max of 360 (max possible angle), after which it is reset. A module timeBaseGeneration has been implemented to get an enable signal with a frequency of 10Hz. This module is used in ddfs_using_cordic to get an enable, which when triggered the angle is incremented by a value which increases the degree by 1. As soon as the angle is modified and fed to the cordic_algorithm module, the always_comb is executed and we obtain the new values of sine and cosine at that angle.