CORDIC Co-Processor

Abhishek K, Akin Mary, Ashwin Rajesh, Harith Manoj

Guide : Mohamed Salih K. K

Government Engineering College, Thrissur

Vision

To be a Premier Institution of Excellence in Engineering Education and Research for Sustainable Development.

Mission

- Provide Quality Education in Engineering and Technology
- Foster Passion for Research
- Transform the Students into Committed Technical Personnel for the Social and Economic Wellbeing of the Nation

Department of Electronics and Communication Engineering

Vision

To become a nationally acclaimed Department of higher learning and research that will serve as a source of knowledge and expertise in Electronics & Communication Engineering.

Mission

- To provide quality education in the area of Electronics and Communication Engineering, for producing innovative and ethically driven professionals adept at dealing with a globally competitive environment, for the welfare of the nation.
- To inculcate inquisitiveness in young graduates thereby persuading them to undertake research in emerging areas of Electronics and Communication Engineering.

Abstract

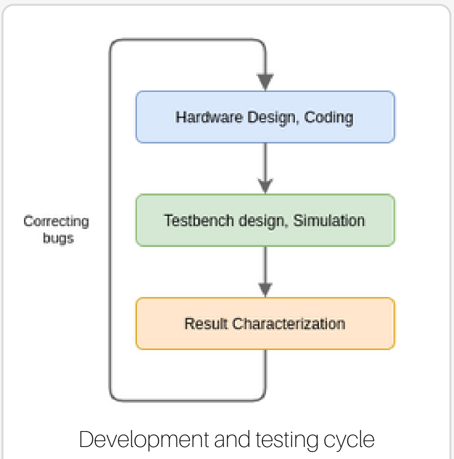
Transcendental functions are functions that cannot be represented by finite polynomials. Typically, they are calculated in computers using taylor series approximation, LUTs or other approximations. This is not fast enough for many use cases. With easily accessible reconfigurable platforms, there is a demand for custom hardware accelerators with hardware-software codesign techniques to improve system efficiency and speed. The wide use of transcendental functions warrants exploration of these techniques for computing such functions.

Our aim is to develop a distributable IP (Intellectual Property) that can be used to compute transcendental functions. It uses the hardware efficient CORDIC algorithm, which uses adders and shifters to perform circular and hyperbolic rotations.

Our IP is designed using a memory mapped register interface as a controller of the CORDIC algorithm. It has input registers to get the inputs to the algorithm, a control/flag register to configure the algorithm and output registers to retrieve the outputs. The IP has a state machine that implements the CORDIC algorithm. It will be used a slave in a bus system (like AXI4) with a main processor.

Development and Testing Methodology

Our development process started from the compute unit and then moved to the controller and then the bus manager. For each submodule, there were 3 steps in the development cycle.



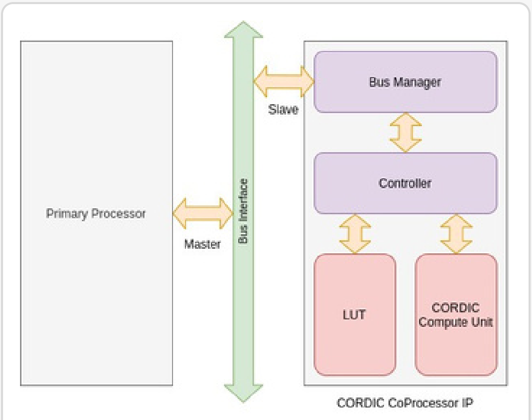
Development and testing cycle

- Hardware design and coding where the hardware specification was defined and implemented using System Verilog.
- Testbench design and simulation where the testbench was coded in System Verilog and connected to the DUT (Design Under Test) and simulated using a simulator like Vivado simulator or Synopsys VCS.
- Characterization where the logfiles from the simulator would be read and the error and overflow characteristics would be analyzed and visualized using python

System Architecture

The co-processor has 4 submodules.

- The CORDIC compute unit implements the arithmetic circuits to perform a single CORDIC iteration.
- The controller implements the state machine that drives the inputs and outputs of the compute unit to implement the CORDIC algorithm and is the heart of the IP.
- The LUT (Look Up Table) stores values of atan and atanh required for CORDIC..
- The Bus Manager is the interface between the main system bus and the controller.

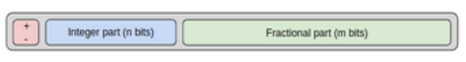


System Block Diagram

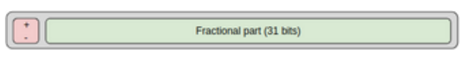
The primary processor interfaces with the system using a memory-mapped register interface. There are seven 32-bit registers as shown below, which are stored in the Bus Manager. The controller accesses the values of these registers and the bus manager manages manages reads and writes of these registers from the bus as well as from the controller. Each register has a format. The numerical input and result registers have fixed point Qn.m format and the control register is divided into read-only flag bits and read/write control bits.

Register Index	Register Name	Type	Description	Format
0	xInput	Write only	Input x value	Qn.m
1	yInput	Write only	Input y value	Qn.m
2	zInput	Write only	Input angle value	Q0.31
3	xResult	Read only	Output x value	Qn.m
4	yResult	Read only	Output y value	Qn.m
5	zResult	Read only	Output angle value	Q0.31
6	controlReg	Read/Write	Control and flag bits	ControlReg

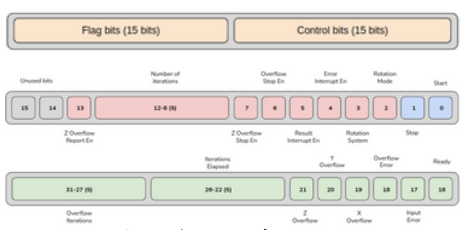
Registers in the IP



Qn.m format



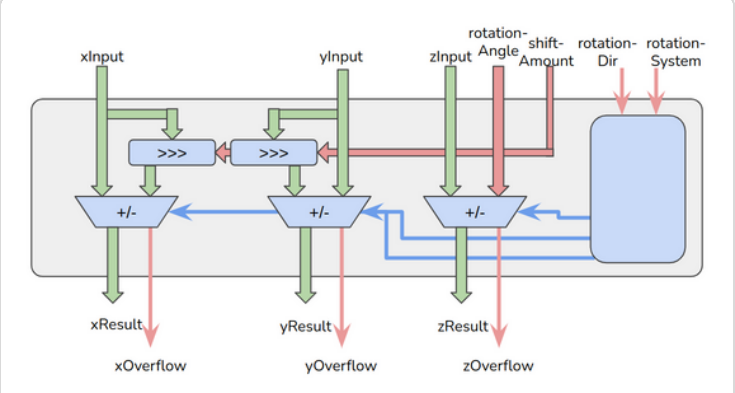
Q0.31 format



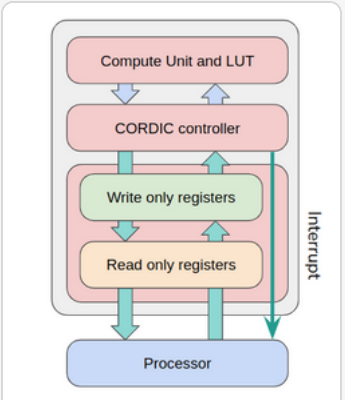
Control Register format

Hardware Design

The design of all submodules excluding the bus manager was done in RTL using system verilog. The bus manager was generated using the Xilinx IP pacakger tool with modifications made in ths source code to create read/write registers and read-only registers as per our register specification. The LUT was simply a ROM with values generated using a python script.



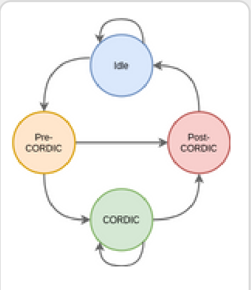
CORDIC Compute Unit design



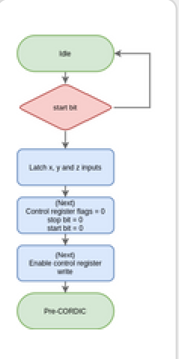
Bus Manager Architecture

Controller state machine

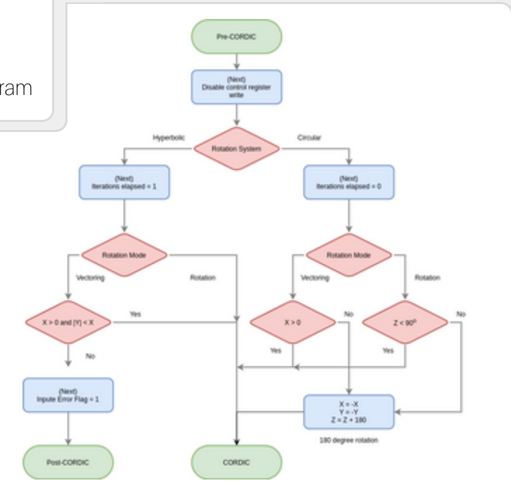
The controller is an FSM with 4 states. The idle state waits for start command, pre-cordic corrects the inputs and cordic state performs the algorithm. The post-cordic state simply reports errors and raises the interrupt



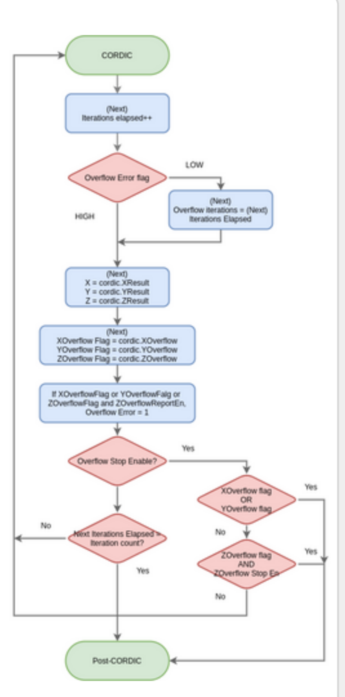
Controller state diagram



Idle state flowchart



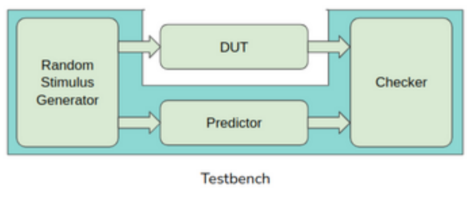
Pre-CORDIC state flowchart



CORDIC state flowchart

Verification and characterization

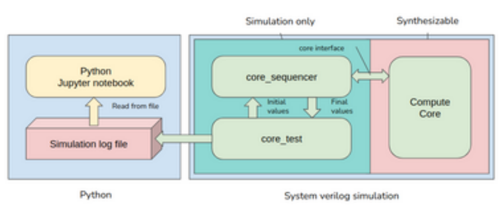
Our methodology was based on the constrained random verification methodology. We generated random inputs for a given configuration of CORDIC and compared the output from the simulated hardware design with the reference values that should be observed from the ideal CORDIC equations for infinite iteration counts.



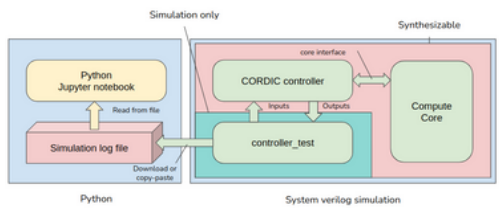
Constrained Random Verification Methodology

The error was recorded in a log file which was analyzed using python. There were 4 phases of testing and development as can be seen below.

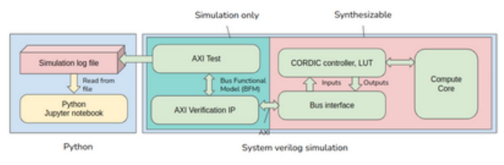
Verification and Characterization Phases



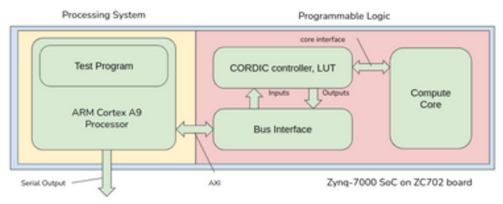
Phase I : Compute Unit characterization



Phase II : Controller and LUT characterization



Phase III : IP level characterization with VIP

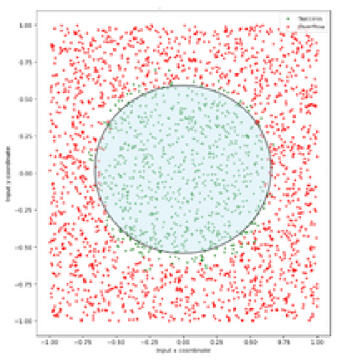


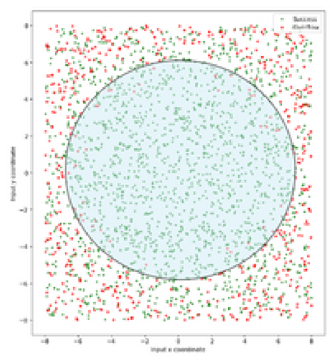
Phase IV : Testing on Zynq7000 SoC

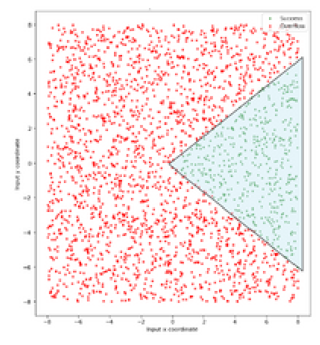
Characterization Results

By performing an analysis of overflows in calculations compared with the input X and Y coordinates, we can conclude that the input point coordinates must have a magnitude less than the reciprocal of the scaling factor in CORDIC. This can be seen as a circle inside which no overflows occur. The exception is hyperbolic vectoring where the area without overflows is a triangle. Outside this, the value of atanh is not a defined real number, causing the overflows.

Overflow(color) v/s Input coordinates (X,Y)

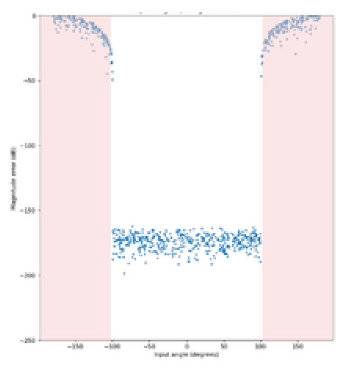

Circular rotation and vectoring

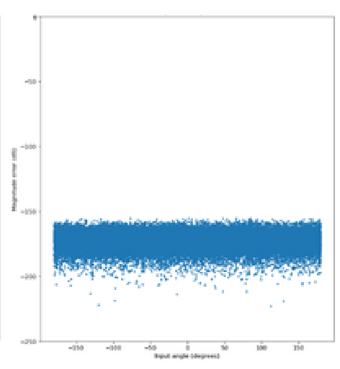

Hyperbolic rotation

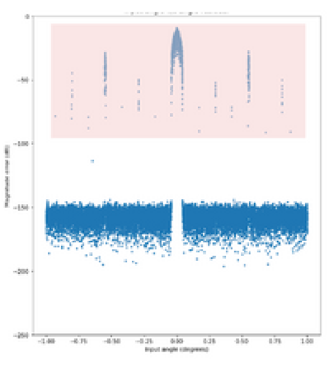

Hyperbolic vectoring

By comparing output error with the total CORDIC rotation angle, we can see that the error explodes for circular mode for angle > 100 degrees because of the CORDIC convergence limit. This is solved by rotating such inputs by 180 degrees before CORDIC. This is implemented in the pre-cordic state of the controller. In the hyperbolic mode, certain angle values have a high error value. This is because of the nature of the atanh(2^-i) series.

Output error (dBfs) v/s rotation angle

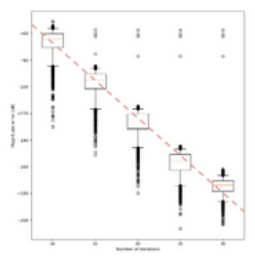

Circular mode without pre-rotation

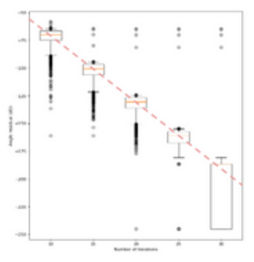

Circular mode with pre-rotation

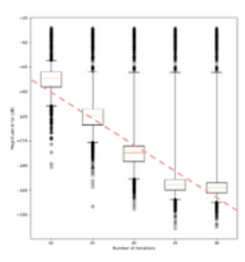

Hyperbolic mode

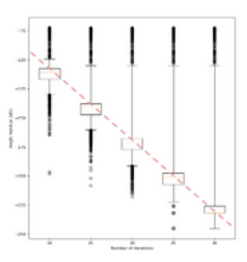
Comparing output error with iteration count, we can see a steady fall on the logarithmic scale indicating an exponential error decrease. The error does not reduce beyond 31 iterations because of the limits of the number system itself. For 31 iterations, maximum output magnitude error in circular rotation mode is of the 10^-9 order of magnitude.

Output error (dBfs) v/s iteration count


Circular rotation

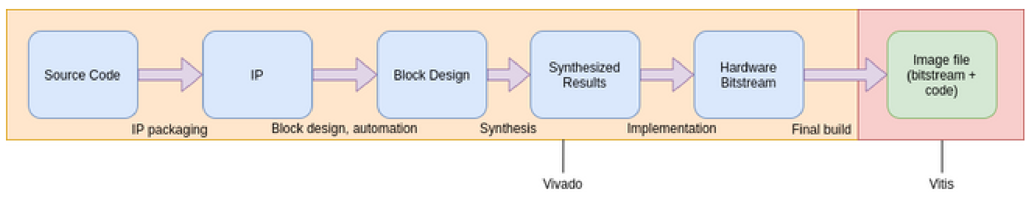

Circular vectoring


Hyperbolic rotation


Hyperbolic vectoring

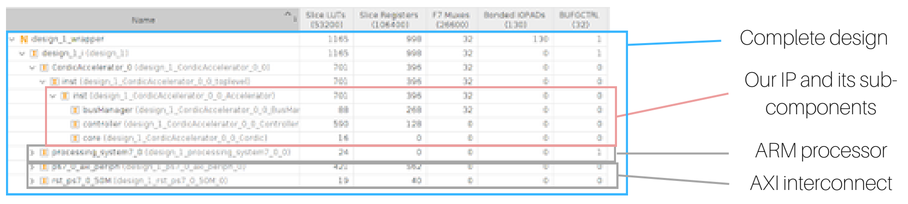
System Integration and Testing

The IP had to be integrated into a processing system and tested. We chose the Zynq7000 SoC on the ZC702 board. The following flow was followed to convert our source code into the final image file to be uploaded into the Zynq7000 SoC.



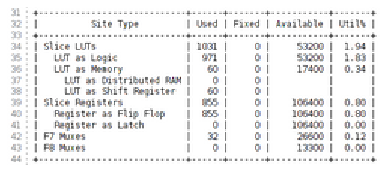
Xilinx tool flow

After packaging the IP, it was connected to the Zynq7000 SoC on board the ZC702 board synthesized and implemented using Xilinx Vivado. The results are as follows. Only about 2% of total LUTs and less than 1% of distributed memory resources were used. In the timing analysis, we can see a worst negative slack of 9.1ns.



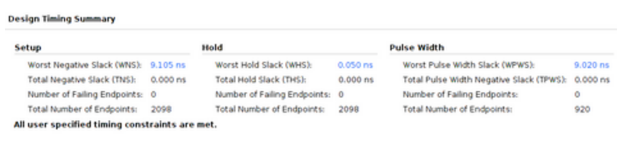
Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	Bonded IOPADs (130)	BUFGCTRL (32)
design_1_wrapper	1165	998	32	130	1
design_1_i (design_1)	1165	998	32	0	1
CordicAccelerator_0 (design_1_CordicAccelerator_0_0)	701	396	32	0	0
inst (design_1_CordicAccelerator_0_0_toplevel)	701	396	32	0	0
inst (design_1_CordicAccelerator_0_0_Accelerator)	701	396	32	0	0
busManager (design_1_CordicAccelerator_0_0_BusMar	88	268	32	0	0
controller (design_1_CordicAccelerator_0_0_Controller	593	128	0	0	0
core (design_1_CordicAccelerator_0_0_Cordic)	16	0	0	0	0
processing_system7_0 (design_1_processing_system7_0_0)	24	0	0	0	1
ps7_0_axi_periph (design_1_ps7_0_axi_periph_0)	421	562	0	0	0
rst_ps7_0_50M (design_1_rst_ps7_0_50M_0)	19	40	0	0	0

Resource utilization after synthesis



```
+----------------------------+------+-------+-----------+-------+
|          Site Type         | Used | Fixed | Available | Util% |
+----------------------------+------+-------+-----------+-------+
| Slice LUTs                 | 1031 |     0 |     53200 |  1.94 |
|   LUT as Logic             |  971 |     0 |     53200 |  1.83 |
|   LUT as Memory            |   60 |     0 |     17400 |  0.34 |
|     LUT as Distributed RAM |    0 |     0 |           |       |
|     LUT as Shift Register  |   60 |     0 |           |       |
| Slice Registers            |  855 |     0 |    106400 |  0.80 |
|   Register as Flip Flop    |  855 |     0 |    106400 |  0.80 |
|   Register as Latch        |    0 |     0 |    106400 |  0.00 |
| F7 Muxes                   |   32 |     0 |     26600 |  0.12 |
| F8 Muxes                   |    0 |     0 |     13300 |  0.00 |
+----------------------------+------+-------+-----------+-------+
```

Resource utilization after implementation



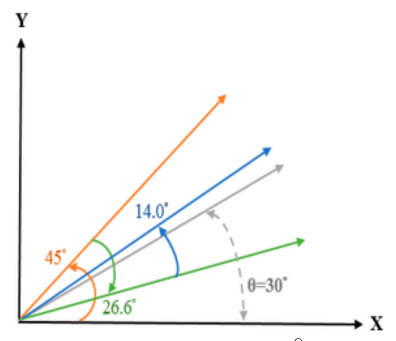
```
Design Timing Summary

Setup                                      Hold                                        Pulse Width
  Worst Negative Slack (WNS):   9.105 ns     Worst Hold Slack (WHS):          0.050 ns     Worst Pulse Width Slack (WPWS):            9.020 ns
  Total Negative Slack (TNS):   0.000 ns     Total Hold Slack (THS):          0.000 ns     Total Pulse Width Negative Slack (TPWS):   0.000 ns
  Number of Failing Endpoints:  0            Number of Failing Endpoints:     0            Number of Failing Endpoints:               0
  Total Number of Endpoints:    2098         Total Number of Endpoints:       2098         Total Number of Endpoints:                 920
All user specified timing constraints are met.
```

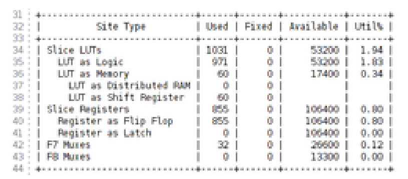
Timing Summary

The CORDIC algorithm

The CORDIC algorithm does a 2D coordinate rotation. It does this using solely rotations and additions making a fixed-point implementation very efficient. It does a series of "micro-rotations" which are rotations by a series of specially chosen angles (45, 26.6, 14,...) for which the 2D rotation matrix includes terms of the power of 2, which means multiplications turn into shifting. Multiple such "micro-rotations" are done, and the direction of these rotations is controlled to make the total rotation angle equal to the desired angle. A vectoring mode also exists which can find the angle of a particular coordinate from the X axis. A hyperbolic counterpart also exists.



Example : First 3 iterations of 30° rotation



```
+----------------------------+------+-------+-----------+-------+
|          Site Type         | Used | Fixed | Available | Util% |
+----------------------------+------+-------+-----------+-------+
| Slice LUTs                 | 1031 |     0 |     53200 |  1.94 |
|   LUT as Logic             |  971 |     0 |     53200 |  1.83 |
|   LUT as Memory            |   60 |     0 |     17400 |  0.34 |
|     LUT as Distributed RAM |    0 |     0 |           |       |
|     LUT as Shift Register  |   60 |     0 |           |       |
| Slice Registers            |  855 |     0 |    106400 |  0.80 |
|   Register as Flip Flop    |  855 |     0 |    106400 |  0.80 |
|   Register as Latch        |    0 |     0 |    106400 |  0.00 |
| F7 Muxes                   |   32 |     0 |     26600 |  0.12 |
| F8 Muxes                   |    0 |     0 |     13300 |  0.00 |
+----------------------------+------+-------+-----------+-------+
```

Resource utilization after implementation

Links

Github project repository

Download final report PDF