YAC - Yet Another CORDIC Core

Christian Haettich [feddischson@gmail.com]

June 26, 2018

IP core and C-model documentation

Contents

Chapter 1

Organization and Introduction

1.1 History

Version	Date	Description	Author
0.02	26th June 2018	E-mail address update and diagrams de-drawn	Christian Haettich
0.01	4th March 2014	Initial document	Christian Haettich

Table 1.1: History

1.2 Folder and Files

```
/
├── c_octave
│   ├── cordic_iterative.c ........................ C-code for bit accurate model
│   ├── cordic_iterative_code.m ........ Script to auto-generate VHDL and C-code
│   └── cordic_iterative_test.m ................. YAC performance analysis script
├── doc ............................... Contains files to create this documentation
├── licenses
│   └── lgpl-3.0.txt ................................................. LGPL license
├── README.txt ........................................... A short read-me file
└── rtl
    └── vhdl
        ├── cordic_iterative_int.vhd ......................... Top-level VHDL file
        ├── cordic_iterative_pkg.vhd .......................... VHDL package file
        └── cordic_iterative_tb.vhd ............................ VHDL testbench
```

1.3 License

1.4 The CORDIC algorithm

The CORDIC algorithm is used to do trigonometric, hyperbolic, multiplication and division calculations in a hardware-efficient way. This means, that only bit-shift, addition and subtraction operations in combination with a lookup-table is required.
Good introductions are available in [1][2][4]. A description on wikibook.org [3] is also useful. For this reason, no more introduction is given here and the following chapters assume, that the reader is familiar with the CORDIC algorithm.

Chapter 2

IP-Core Description

The two files **cordic_iterative_int.vhd** and **cordic_iterative_pkg.vhd** implements an iterative CORDIC algorithm in fixed-point format. Iterative means, that the IP-core is started with a set of input data and after a specific amount of clock cycles, the result is available. No parallel data can be processed. The following six modes are supported:

- trigonometric rotation
- trigonometric vectoring
- linear rotation
- linear vectoring
- hyperbolic rotation
- hyperbolic vectoring

2.1 Port Description

Table 2.1 and Figure 2.1 gives an overview of the YAC ports. The three input ports start, done and mode and are used to interact with an external state machine or with a software driver. After setting `start` to high, the YAC starts processing. If all necessary rotations are done, the `done` output is set to high for one clock cycle. When starting the YAC, all other inputs must contain valid data. The mode input is used to select the CORDIC mode. Table 2.1 shows all input and output ports and Figure 2.1 shows a CORDIC block symbol: $x_i, y_i, a_i, x_o, y_o, a_o$ are data ports whereas start, done and mode are configuration signals.

2.2 Parameter Description

Table 2.2 shows the four parameter, which can be used to parameterize the YAC.

Name	Type	Direction	Size	Description
clk	std_logic	input	1	clock
rst	std_logic	input	1	synchronous reset
en	std_logic	input	1	clock-enable
start	std_logic	input	1	start trigger
done	std_logic	output	1	done indicator
mode	std_logic_vector	input	4	mode configuration
x_i	std_logic_vector	input	XY_WIDTH	X input vector
y_i	std_logic_vector	input	XY_WIDTH	Y input vector
a_i	std_logic_vector	input	A_WIDTH + 2	angular input vector
x_o	std_logic_vector	input	XY_WIDTH + GUARD_BITS	X output vector
y_o	std_logic_vector	input	XY_WIDTH + GUARD_BITS	Y output vector
a_o	std_logic_vector	input	A_WIDTH + 2	angular output vector

Table 2.1: Port description

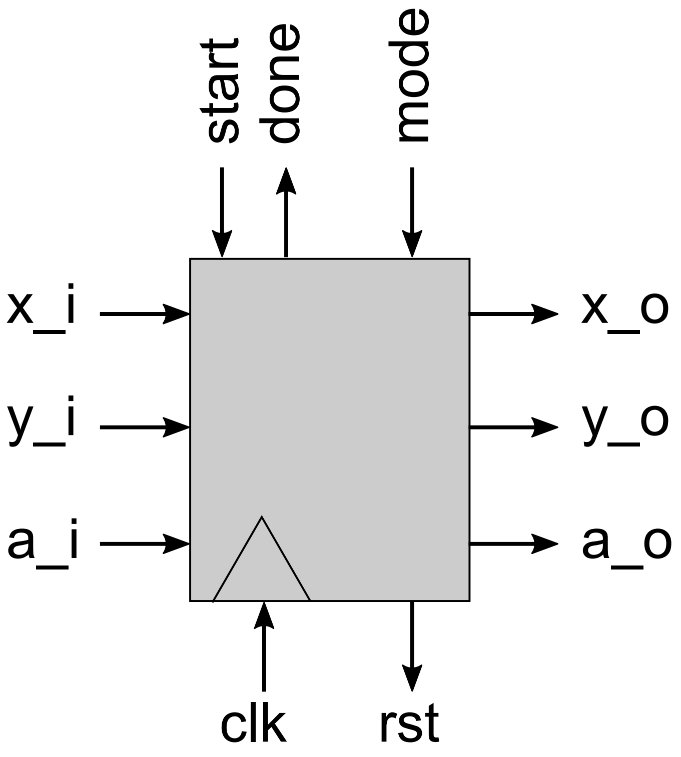

Figure 2.1: YAC block symbol

Name	type	size	Description
XY_WIDTH	natural	defines the size of x and y input and output vectors	
A_WIDTH	natural	defines the size of the angular input and output vectors	
GUARD_BITS	natural	defines the number of guard bits	
RM_GAIN	natural	defines the precision of the CORDIC gain removal	

Table 2.2: Parameter description

2.3 Mode description

With the mode-input, it is possible to select between circular, linear and hyperbolic operation as well as between rotation and vectoring operations:

- mode(1:0) = "00" selects circular operation
- mode(1:0) = "01" selects linear operation
- mode(1:0) = "10" selects hyperbolic operation
- mode(1:0) = "11" is reserved
- mode(3) = '0' selects rotation
- mode(3) = '1' selects vectoring

The package cordic_iterative_int.vhd contains the following defines to setup the mode:

- VAL_MODE_CIRC = "00"
- VAL_MODE_LIN = "01"
- VAL_MODE_HYP = "10"
- I_FLAG_VEC_ROT = 3 (bit index)

For example if a hyperbolic rotation mode is required, the mode input is "0010", and for the linear vector operation, the mode input is "1001". Please note, that bit number 2 is reserved for future implementations.
Table 2.3 defines the behaviour of the YAC for different mode configurations.

2.4 Data Range and Limitations

Both, the x_i and y_i inputs are defined with a size of XY_WIDTH and therefore, the maximum positive value in the two's complement format is

$$\alpha = 2^{XY\_WIDTH-1} - 1 \tag{2.1}$$

The angle input a_i is defined with a size of $A\_WIDTH + 2$. The value β is defined with

$$\beta = 2^{A\_WIDTH-1} - 1 \tag{2.2}$$

Setup	Description
mode =(FLAG_VEC_ROT = 1, VAL_MODE_CIR)	$a_o = \text{atan2}(y_i, x_i) \cdot \beta$, $x_o = \sqrt{x_i^2 + y_i^2}$
mode =(FLAG_VEC_ROT = 0, VAL_MODE_CIR)	$x_o = \cos(a_i/\beta) \cdot \alpha$ $y_o = \sin(a_i/\beta) \cdot \alpha$
mode =(FLAG_VEC_ROT = 1, VAL_MODE_LIN)	$a_o = z + y/x$
mode =(FLAG_VEC_ROT = 0, VAL_MODE_LIN)	$a_o = y + x \cdot z$
mode =(FLAG_VEC_ROT = 1, VAL_MODE_HYP)	$a_o = \tanh(y_i/x_i) \cdot \beta$, $x_o = \sqrt{x_i^2 - y_i^2}$
mode =(FLAG_VEC_ROT = 0, VAL_MODE_HYP)	$x_o = \cosh(a_i/\beta) \cdot \alpha$ $y_o = \sinh(a_i/\beta) \cdot \alpha$

Table 2.3: Mode description (see equation 2.1 and 2.2 for α and β)

2.4.1 Circular Vectoring Mode

The valid input range for x_i and y_i is $-\alpha ... + \alpha$. The angular input a_i is ignored.

2.4.2 Circular Rotation Mode

The valid input range for x_i and y_i is $-\alpha ... + \alpha$. For the angular input a_i, values between $-\beta\pi$ and $+\beta\pi$ are valid. For calculating a complex rotation of a complex vector, all three inputs are required. For calculating sin and cos, x_i is set to α and y_i to 0, the angle input a_i gets the angle.

2.4.3 Linear Vectoring Mode

The linear vectoring mode is used to calculate y_i/x_i. The limitation of this operation is the following:

$$\frac{y_i}{x_i} \leq 2 \tag{2.3}$$

The valid input values for x_i and y_i are $-\alpha ... + \alpha$.

2.4.4 Linear Rotation Mode

The two inputs x_i and y_i have a valid data range between $-\alpha$ and $+\alpha$.

2.4.5 Hyperbolic Vectoring Mode

The data-range for x_i and y_i is $-0.79 \cdot \alpha$ to $0.79 \cdot \alpha$.

2.4.6 Hyperbolic Rotation Mode

The data-range for a_i is $-\beta ... \beta$. Typically, x_i is set to α and y_i to 0.

2.5 Internal Operation

Five states are used do to the CORDIC algorithm. The five states are visualized in a state diagram in Figure 2.2. After reset, the YAC goes into the `ST_IDLE` state. Only in this state, the start signal is accepted. After starting the YAC, the state switches from the `ST_IDLE` state to the `ST_INIT` state. The init state does some initial rotations/flipping. After initialization, the main rotation starts with the `ST_ROTATION` state (there are a few cases, where no further rotations are required, see next sections). Every rotation step is done within two clock cycles: in the first cycle, the angular value is loaded from the lookup-table and shifted. In addition, the x and y values are shifted. In the second clock cycle, the results are added according to the CORDIC algorithm. After all required iterations are done, the state switches to the `RM_GAIN` state, where the CORDIC gain is removed (depending on the mode). The last state `ST_DONE` is only used to set the done flag to '1' for one clock cycle. After this, the YAC returns to the `ST_IDLE` state.

2.5.1 Initialization

During the initialization state ST_INIT, pre-rotations are done, depending on the selected mode.

Circular Vectoring Mode

Because we support atan2 and not atan, we do the following pre-rotations. Some specific input values don't need a further processing, for example $atan(0, 0)$.

Input	**Description**
$x_i = 0, y_i = 0$	Fixed result: $x_o = 0, a_o = 0$, to support cartesian to polar coordinate transformations, further processing is skipped
$x_i = 0, y_i > 0$	Fixed result: $x_o = y_i, a_o = \pi/2$, further processing is skipped
$x_i = 0, y_i < 0$	Fixed result: $x_o = -y_i, a_o = -\pi/2$, further processing is skipped
$x_i < 0, y_i < 0$	Pre-rotation from the third to the first quadrant, the angular register is initialized with $-\pi$ and the sign of the x and y register is flipped.
$x_i < 0, y_i \geq 0$	Pre-rotation from the second to the fourth quadrant, the angular register is initialized with π and the sign of the x and y register is flipped.
$x_i = +\alpha, y_i = +\alpha$	Fixed result: $x_o = \sqrt{2} \cdot \alpha, a_o = \frac{\pi}{4} \cdot \beta$, further processing is skipped
$x_i = +\alpha, y_i = -\alpha$	Fixed result: $x_o = \sqrt{2} \cdot \alpha, a_o = -\frac{\pi}{4} \cdot \beta$, further processing is skipped
$x_i = -\alpha, y_i = +\alpha$	Fixed result: $x_o = \sqrt{2} \cdot \alpha, a_o = \frac{3\pi}{4} \cdot \beta$, further processing is skipped
$x_i = -\alpha, y_i = -\alpha$	Fixed result: $x_o = \sqrt{2} \cdot \alpha, a_o = -\frac{3\pi}{4} \cdot \beta$, further processing is skipped

Circular Rotation Mode

The following pre-rotations are done:

Input	Description
$a_i < -\frac{\pi}{2}$	Pre-rotation from the third to the first quadrant, the sign of the x and y register is flipped, the angular register is initialized with π
$a_i > \frac{\pi}{2}$	Pre-rotation from the second to the fourth quadrant, the sign of the x and y register is flipped, the angular register is initialized with $-\pi$

Linear Vectoring Mode

The following pre-rotations are done:

Input	Description
$x_i < 0$	Pre-rotation from the left half to the right half, thus the sign of the x and y register is flipped

2.5.2 Rotation

Hyperbolic Repetitions

The following iteration steps are repeated for convergence of the hyperbolic mode:

$$4, 13, 40, 121, ..., 3i + 1 \tag{2.4}$$

2.6 Testbench

A VHDL testbench (cordic_iterative_tb.vhd) is available which reads from an input file data. This data is fed into the YAC. In addition, the input file must provide the output data, which is expected from the YAC processing. The following format is used in the testbench input-file:

```
x_i     y_i    a_i   x_o   y_o   a_o   mode

1234   1234   1234  1234  1234  1234    111  \
2345   2345   2345  2345  2345  2345    222  |  7 columns:
3456   3456   3456  3456  3456  3456    333  |  3 input, 3 output and one mode
...    ...                                   |
...                                          |
...
```

The data must be in the two's complement with a base of 10. After processing all lines, the testbench prints some info, how much tests failed.
For generating the test-patterns, the C-model with octave is used. By running the script **cordic_iterative_test.m**, an entry in the testbech data file is done for every cordic calculation.

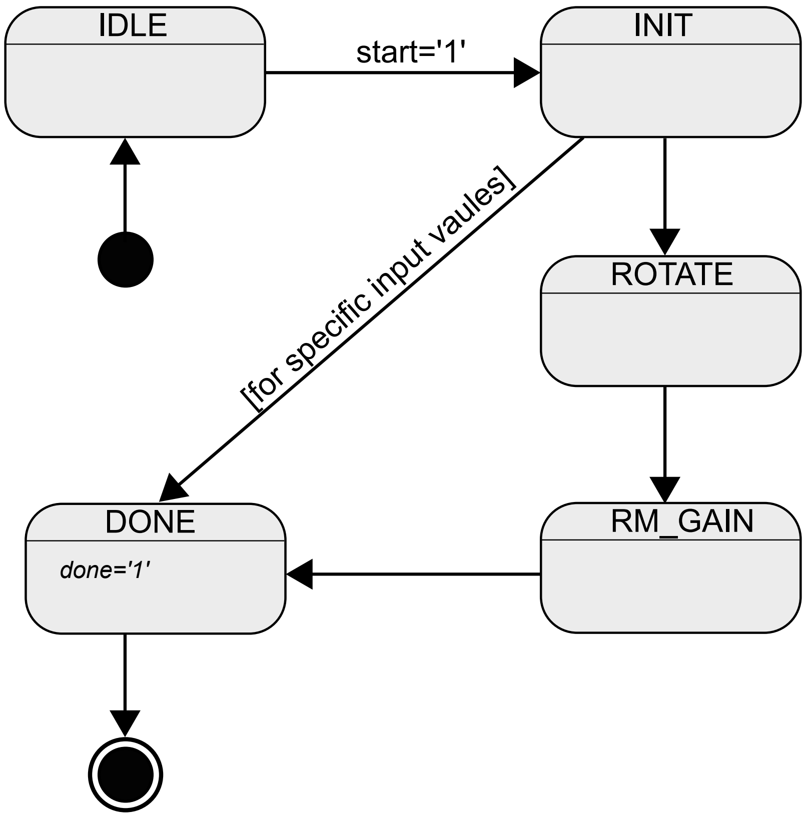

Figure 2.2: State diagram

Chapter 3

Using the C-Model with Octave or Matlab

For using the C-model, open octave or Matlab and compile the C-Model with
mex cordic_iterative.c Afterwards, the C-model can be used in Octave/Matlab with the function call

```
[ x_o, y_o, a_o, it ] = cordic_iterative( x_i, y_i, a_i, mode, XY_WIDTH, ANGLE_WIDTH,
GUARD_BITS, RM_GAIN ).
```

The input and output arguments reflect almost the ports and parameters of the VHDL toplevel implementation, except that there is no done, start clk and rst port in the C-model. In addition, the last argument provides the number of iterations, which are done.

The input arguments x_i, y_i, and a_i can be 1xN vectors with N elements. In this case, this three input arguments must have the same size. All output arguments will also have the same size 1xN.

3.1 Guard-Bits – Input and output sizes

It is possible to define MSB guard bits. The need of MSB guard bits have several reasons. On the one hand, they are required to handle the data growth of the cordic gain within the rotation state. (which is compensated later). A nother reason is the data growth caused by the choise of input data. For example

$$\sqrt{\alpha^2 + \alpha^2} = \sqrt{2}\alpha \tag{3.1}$$

and therefore, a growth by the factor $\sqrt{2}$ happens, if an absolute calculation of the two maximum input values is processed.

Another point is the input and output size of the angular values (a_i and a_o) Because within this YAC, β represents the angle 1, the input and output with of a_i and a_o is A_WIDTH+2, which allows to process angle values within $-\pi ... \pi$

3.1.1 Input data width versus iterations

The number of iterations depends on the width of the input data. Input data with X bit require $X + 1$ rotations [4]. This requires, that the number of iterations are adapted. One solution might be to detect the size of the input data and use this information for the CORDIC processing. In YAC, a different sheme was chosen: the rotations are done until

- the angular register is 0 or doesn't change (in case of the rotation mode)
- the y register is 0 or doesn't change (in case of the vecotring mode).

This results in a dynamic iterations adaption, depending the the input data width.

Chapter 4

Performance

The performance is evaluated through the matlab script **cordic_iterative_test.m**.

Chapter 5

Open issues and future work

5.1 Next steps

- Add YAC to a FPGA based System-on-Chip to prove FPGA feasibility
- Performance optimization
- circuit optimizations
- numerical optimizations

5.2 Future plans

- Hyperbolic range extension
- Floating point CORDIC

Bibliography

[1] Andraka, Ray; *A survey of CORDIC algorithms for FPGA based computers*

[2] Hu, Yu Hen; *CORDIC-Based VLSI Architectures for Digital Signal Processing*

[3] CORDIC on wikibook: `http://en.wikibooks.org/wiki/Digital_Circuits/CORDIC`

[4] David, Herbert; Meyr, Heinricht; *CORDIC Algorithms and Architectures* `http://www.eecs.berkeley.edu/newton/Classes/EE290sp99/lectures/ee290aSp996_1/cordic_chap24.pdf`