Fixed Point Atan2 Implementation Using CORDIC Algorithm

Gabriele Ara, *Student, ECS,* and Gabriele Serra, *Student, ECS,*

***Abstract*—In simple micro controllers or integrated circuits like FPGAs it is very unlikely to find an hardware multiplier. We can of course implement one, but even if we do, if we implement an algorithm in hardware performing many multiplications the overall system performance will slow down considerably.**

For these reasons, most times simpler algorithms are implemented in hardware using more efficient ad hoc algorithms that can execute multiplication-like operations in a more efficient way.

In this report we show how trigonometric functions can be implemented in hardware using an algorithm called CORDIC by implementing the Atan2 function.

I. Introduction

Trigonometric functions are very important in many application fields, like navigation, positioning systems, kinematic computations and so on. The arctangent function is used for example to determine angles from tangens-value obtained from sensors. A polynomial approximation of these trigonometric functions would require the usage of many multiplications which in general is an undesired condition when speed and system performance are a key factor like in automotive field and in general in real-time environments.

Simpler and more efficient implementations can be achieved using an algorithm conceived in 1956 by Jack E. Volder called CORDIC, which stands for Coordinate Rotation Digital Computer [1]. This algorithm allows the calculation of hyperbolic and trigonometric functions performing vector rotations via shift-and-add operations.

We will focus on the implementation of the arctangent function, but all the considerations we will do from now on can easily be re-formulated when applying the CORDIC algorithm to any other trigonometric function.

A. Implementing rotations efficiently

The arctangent function calculates the phase of a vector of two components, namely a and b, which can also be seen as the *real* and the *imaginary part* of the complex number $z = a+bj$. The CORDIC algorithm central idea is to consider the two arguments a and b as coordinates of a complex number z and perform rotations of this vector by a succession of constant values.

The rotation of the phase is realized multiplying z by a sequence of complex numbers. When multiplying two complex numbers, the phase of the sum is the sum of the phases of the two numbers and the magnitude is the product of the magnitudes, so given:

$$\begin{aligned} z_1 &= r_1 \cdot e^{\varphi_1} \\ z_2 &= r_2 \cdot e^{\varphi_2} \end{aligned} \tag{1}$$

we obtain

$$z_3 = z_1 \cdot z_2 = r_1 r_2 \cdot e^{\varphi_1+\varphi_2} \tag{2}$$

We hence can take advantage by this fact and implement a rotation by $+90°$ by performing:

$$\begin{aligned} R &= j \\ z' &= z \cdot R = -b + aj \end{aligned} \tag{3}$$

We can of course implement in the same way a rotation by $-90°$ by simply putting:

$$\begin{aligned} R &= -j \\ z' &= z \cdot R = b - aj \end{aligned} \tag{4}$$

In general, to rotate by an angle less than $90°$ we simply perform:

$$\begin{aligned} R &= 1 + kj \\ z' &= z \cdot R \\ &= (a + bj) \cdot (1 + kj) \\ &= (a - b \cdot k) + (a \cdot k + b)j \end{aligned} \tag{5}$$

Even if we still express the above operations using multiplications, choosing accordingly the value of k can lead to a much more efficient implementation. In fact, if k is expressed as a negative power of 2, these multiplications become indeed arithmetic shifting operations, which are much more efficient than multiplications:

$$\begin{aligned} k &= 2^{-i} \\ k \cdot a &= ShiftRightArithmetic(a, i) \end{aligned} \tag{6}$$

B. CORDIC Algorithm Rotations

The CORDIC algorithm can be expressed as it follows:

- First rotate the vector by $\pm 90°$, depending on the sign of the *imaginary part* of z.
- Then for each successive iteration i multiply the current vector by $R_i = 1 \pm j2^{-(i-1)}$.

The sign is chosen in such a way that the subsequent rotations will converge to a vector with zero phase, as stated in the CORDIC Convergence Theorem [2].

Of course, this series of multiplication has the side effect of incrementing the magnitude of the number z, because, as we have seen in equation (2), the magnitudes are multiplied together. Thus every rotation causes the magnitude to grow and the factor is dependent on the step of the CORDIC algorithm

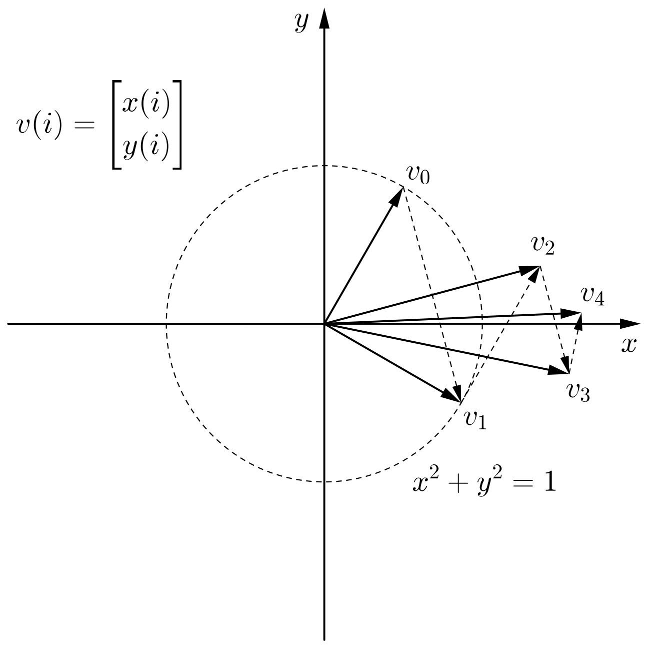


Fig. 1. Subsequent rotations of a vector performed according to the CORDIC algorithm.

TABLE I.
CORDIC ALGORITHM ROTATION VECTORS FOR EACH ITERATION i. THE FIRST ITERATION IS DIFFERENT FROM THE FOLLOWING ONES.

i	$k = 2^{-(i-1)}$	$R = 1 + kj$	φ_i (deg)	$\lvert R \rvert$	Gain
0	-	-	90	1	1
1	1.0	1+1.0j	45	1.41421356	1.41421356
2	0.5	1+0.5j	26.56505	1.11803399	1.58113883
3	0.25	1+0.25j	14.03624	1.03077641	1.62980060
4	0.125	1+0.125j	7.12502	1.00778222	1.64244841
5	0.0625	1+0.0625j	3.57633	1.00195122	1.64568892
6	0.03125	1+0.03125j	1.78991	1.00048816	1.64649228
7	0.015625	1+0.015625j	0.89517	1.00012206	1.64669325
8	0.007812	1+0.007813j	0.44761	1.00003052	1.64674351
⋮	⋮	⋮	⋮	⋮	⋮

i. Since the step count increases at each step, the overall gain with respect to the initial vector will increase at each iteration, but it can be proven that it also tends to about 1.6467. Fig. 1 shows how the CORDIC rotations affect an input vector phase and magnitude each step [3].

Table I contains the vectors used at each step by the CORDIC rotation algorithm and the magnitude gain of the result of the computation with respect to the initial value. From the table we can clearly see how the gain converges in a few steps to the limit value.

C. Calculating The Arctangent

From what we have seen, the CORDIC algorithm performs rotations using well known rotation vectors, summing or subtracting well known phases to the original vector. To calculate the arctangent we can hence take into account these well known phases and sum them together. Since the original vector phase converges to zero, also the calculated arctangent value will converge in the same way to the original phase of the vector, which is our goal:

$$\begin{gathered} \lim_{n\to\infty} \varphi - \sum_{i=0}^{n} \pm\varphi_i = 0 \\ \varphi = \tan^{-1}(z) = \lim_{n\to\infty} \sum_{i=0}^{n} \pm\varphi_i \end{gathered} \tag{7}$$

Hence for each step the algorithm will perform at the same time a rotation of the vector z and store the sum of the phases of the rotation vectors, until convergence is reached. The final result will be the requested value.

D. Algorithm abstract implementation

From what we have shown before, rotation vectors and corresponding phases can be stored in a table, in order to retrieve them at each step of the algorithm. Of course, if we limit the number of iteration steps n we won't reach perfect convergence, but the algorithm convergence is fast enough that a few number of iterations will lead to a small absolute error.

The following is the MATLAB implementation that we produced:

```matlab
function phi = ...
    cordicatan2(inY, inX, lutSize)

% Lookup table used to retrieve
% rotation phases
% NOTICE: matlab arrays are 1-based

cordicLUT = zeros(lutSize, 1);

cordicLUT(1) = pi/2;

for i = 0:lutSize-2
    cordicLUT(i+2) = atan(2^(-i));
end

% If both the input coordinates are
% zero, the result is zero
if(inX == 0 && inY == 0)
    phi = 0;
    return;
end

% We look at the sign of the y
% component to determine whether to
% rotate clockwise or counterclockwise
if inY >= 0
    % Rotate clockwise
    phi = cordicLut(1);

    tempX = + inY;
    tempY = - inX;
else
```

```
    % Rotate counterclockwise
    phi = -cordicLut(1);

    tempX = - inY;
    tempY = + inX;
end

% If the x coordinate is zero, the
% result is either +90 or -90
if inX == 0
    return;
end

inX = tempX;
inY = tempY;

for i = 1:lutSize-1
    if inY >= 0
        % Rotate clockwise
        phi = phi + cordicLut(i+1);

        % Instead of multiplications
        % we perform right arithmetic
        % shift operations
        tempX = inX + bitsra(inY, i-1);
        tempY = inY - bitsra(inX, i-1);
    else
        % Rotate counterclockwise
        phi = phi - cordicLut(i+1);

        % Instead of multiplications
        % we perform right arithmetic
        % shift operations
        tempX = inX - bitsra(inY, i-1);
        tempY = inY + bitsra(inX, i-1);
    end

    inX = tempX;
    inY = tempY;
end

end
```

II. Bit True Model Development

Designing a circuit that implements an algorithm is not an easy task and usually is composed by some intermediate steps.

1) A mathematical model of the problem we want to solve must be formally specified.
2) An algorithm that solves the problem has to be developed, as we did in the previous section.
3) The algorithm must be translated (*quantized*) into a *bit true model* which simulates the hardware behavior.
4) Finally, the *bit true model* can be actually implemented in hardware.

These steps are also shown in fig. 2. As shown, at each step we need to consider the requirements and check whether our implementation satisfies the given constraints:

- When implementing the algorithm, an *algorithmic error* is introduced; the obtained results differ from the theoretical expected ones, due to choices in the algorithm design, approximations, assumptions and so on.
- When implementing the *bit true model*, there can be *implementation losses* due to representation of real numbers on computer hardware; the *quantization* of the algorithm introduces some errors when performing arithmetic operations between represented real numbers.
- When realizing the hardware implementation, hardware *timing constraints* have to be satisfied to avoid race conditions and other electrical problems.

We have already performed the first steps of this process in the previous section. We will now evaluate the *algorithmic error* of the provided algorithm and then move on and with the *bit true model* realization.

A. Algorithmic Error Evaluation

First of all, we have to define the domain of the arctangent implementation we want to develop. We chose that the final hardware implementation will be on 12 bit signed integers in two's complement, so the input range for both the coordinates will be $[-2048, 2047]$.

The image of our arctangent implementation will be in the range $[-\pi, \pi]$ and encoded using 12 bit fixed-point numbers, using 9 bits as fraction part. With this configuration, the lookup table will consist of 11 entries, so 11 is also the number of iterations that our algorithm will perform.

Having a bigger lookup table (so executing a greater number of iterations) would result in no effect on the final result once the algorithm will be translated into fixed-point arithmetic, since the extra positions in the lookup table would be filled by zeros: in this case, the fixed-point system would continue to execute the algorithm but would obtain at the end the same result, so it's meaningless.

To evaluate the *algorithmic error* we need to give all the possible inputs to our previously defined MATLAB function and check the obtained arctangent against the `atan2` implementation that is found in MATLAB standard library.

For each obtained result, we evaluated the *algorithmic relative error* as:

$$\begin{matrix} algorithmic \\ relative \\ error \end{matrix} = \frac{|expected\ result - obtained\ result|}{|expected\ result|} \tag{8}$$

A graphical representation of the results of this analysis can be seen in fig. 3. In the figure, the two coordinates of the input have been named A and B. As we can clearly see, the relative error grows very fast as A approaches zero, when B is positive. This result depends partially from the finite arithmetic used to simulate the system, but mostly from the fact that our implementation executes only a finite number of

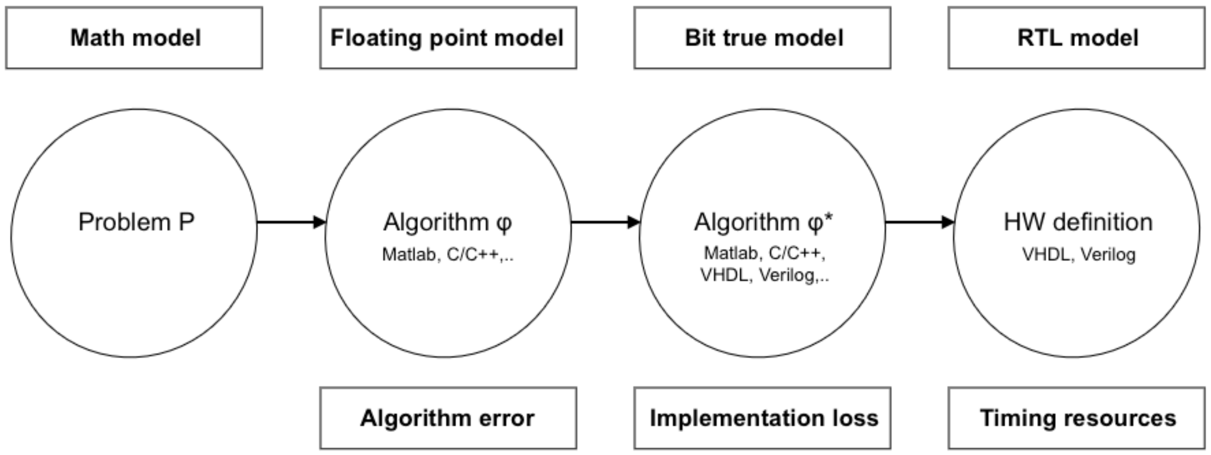
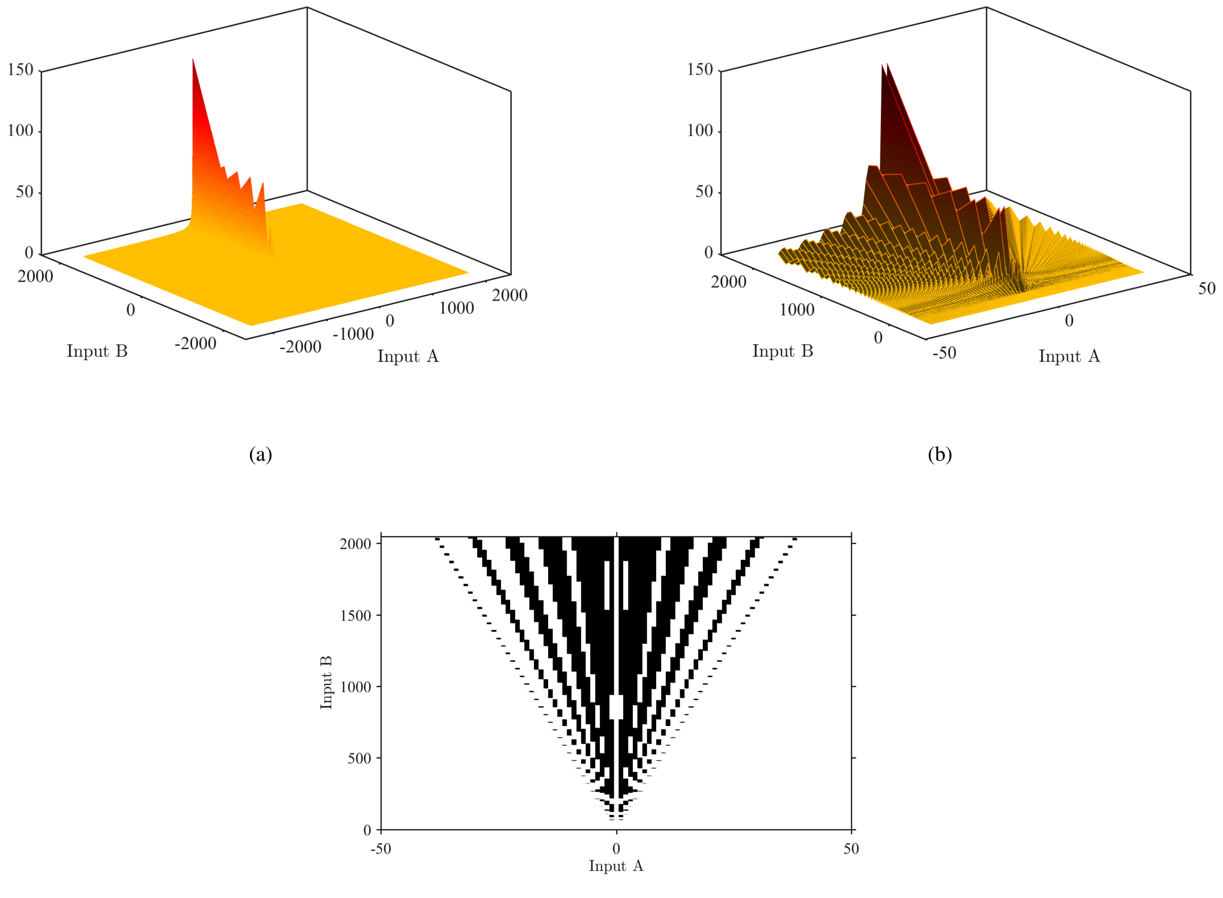


Fig. 2. Development process of a hardware algorithm implementation. After each step of the development process an analysis has to be performed to check whether the implementation satisfies the constraints.



(c)

Fig. 3. *Algorithmic relative error* evaluation of the relative error. (a) Full scale representation (%). (b) Zoom on the area with the most significant *relative error* (%). (c) Areas in which the *algorithmic relative error* is greater than 10%.

iterations (given by the size of the lookup table) instead than performing an infinite amount of iterations. Even if we actually implemented the algorithm in an Infinite Precision Arithmetic (IPA), a certain amount of *absolute error* cannot be avoided if we perform only a finite amount of iterations [4].

As we reduce the magnitude of A (for positive values of B), the expected output becomes closer and closer to zero, so, even if the *algorithmic absolute error* is limited to about 0.00195 rad in our domain[1], the *relative error* can only grow with it. This behavior is intrinsic of the nature of the problem and cannot be avoided increasing the precision of the Finite Precision Arithmetic (FPA) used.

The only solution to limit this behavior is to increment the accuracy of the algorithm using a greater number of iterations (so a bigger lookup table), but as we said this would lead to better results only in IPA or floating-point arithmetic, while in fixed-point it would have no effect.

B. Bit True Model Implementation in MATLAB

The next step in the development process is the derivation of a *bit true model* that can be used to simulate how our system performs once quantized in floating-point arithmetic and implemented in hardware.

To do so, we first performed this analysis in MATLAB, using the same function defined in the previous section, with little changes that let us force it to execute all operations using fixed-point arithmetic.

The lookup table used now is the fixed-point equivalent of the previous one, which, as we described before, has 12 bits elements with 9 bits of fraction part.

The two input coordinates are still 12 bits long, however an extension to 14 bits has been necessary to avoid overflow errors in fixed-point operations inside the function. In fact, as we execute each step the magnitude of the input vector increases with each rotation, until the maximum gain is reached. This means that both the components can exceed the maximum representable numbers in two's complement 12 bit words.

Since, as we said, the maximum gain is not greater than about 1.6467, 14 bits are more than sufficient to represent internally the input numbers avoiding any overflow problem.

The result of the *error analysis* is shown in fig. 4. Again, while the *absolute error* is limited to a maximum of about 0.01 rad, the relative error grows in certain areas of the domain, as shown in fig. 4c. Usually (but not always), the total obtained error is more than the algorithmic one due to intrinsic limits of fixed-point arithmetic and quantization.

If we however neglect the problems that we encounter in the aforementioned areas, the *total relative error* is not only smaller than 10%, but it is in general much smaller than 1%, which seems us a pretty good result.

III. CORDIC Hardware Architectures

After the MATLAB *bit true model* implementation we now need to move to a more low level description of the actual hardware implementation of the algorithm. There are some different ways to implement the CORDIC algorithm in hardware [5], [6]. The ideal architecture depends on the speed vs area trade-offs in the intended application. We will now show the two main architectures before adopting one for our design.

A. Iterative Architecture

An *iterative* CORDIC architecture can be obtained simply realizing the iterative function presented in section I using hardware components. The resulting architecture [7] is shown in fig. 5.

In this implementation, the sign of y is used to control the sum/subtraction operations, hence performing clockwise or counterclockwise rotations. The initial coordinates are loaded via multiplexers into the two registers, then for each of the next N clock cycles, the values from the registers shifted and added/subtracted to perform the rotations. The results are then placed back into the registers.

The shifting components have to be able to accept a variable shift size, because for each iteration the number of bits that need to be shifted increases. In the same way, the LUT address is incremented so that the appropriate elementary angle value is given to the result adder/subtractor. On the last iteration, the result is read directly from the adder/subtractor. Obviously, it is needed to keep track of the current iteration number in order to check if the process is completed, to select the degree of shifting and to select the LUT address for each iteration.

B. Unrolled Architecture

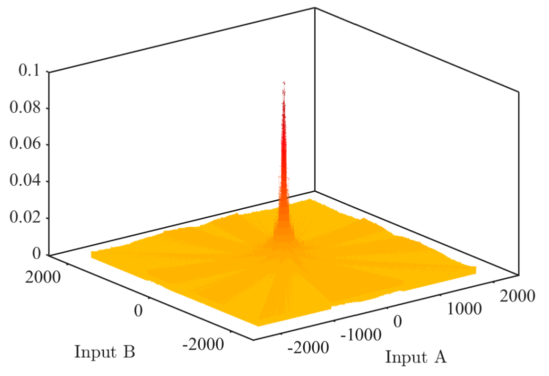
The CORDIC implementation discussed above is *iterative*, which means the developed processing unit has to perform N iterations to calculate the result. The *iterative* process can be *unrolled*, which means that replicationg N times the components needed by the *iterative* process we can calculate the same result with only one iteration. Each of the N replicated components performs at each iteration the same operation, as in a pipelined scheme. The resulting architecture [7] scheme is shown in fig. 6.

C. Comparison Between The Two Architectures

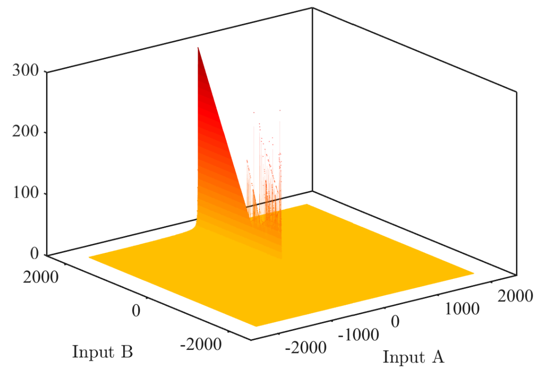
Unrolling the processor results in three significant simplifications:

1) Each shifter performs now a shift of the same number of bits each iteration, which means that they can be implemented easily using only wires.
2) The lookup table values for each phase accumulator are distributed as constants to each phase accumulator in the chain. These constants can be hardwired instead of requiring storage space, resulting in the entire CORDIC processor being implemented as an array of interconnected adder/subtractor components.
3) The need for registers is also eliminated, as the CORDIC implementation becomes a combinatorial logic network, resulting in an overall faster solution than the *iterative* one.

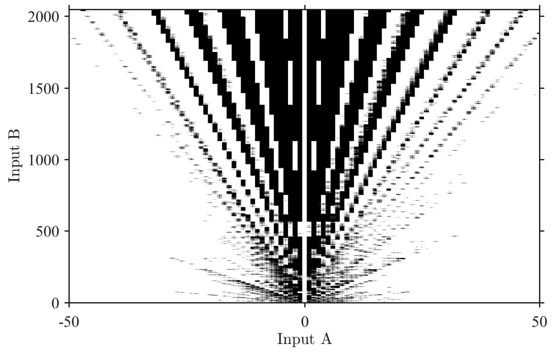
[1]We neglect the fact that this error has been calculated using MATLAB double precision, which obviously is not infinite.



(a)



(b)



(c)

Fig. 4. System *absolute and relative error* evaluation. (a) Total *absolute error* evaluation (rad). (b) Total *relative error* evaluation (%). (c) Areas in which the total *relative error* is greater than 10%.

However, it brings also some disadvantages:

1) The delay of the combinatorial logic network could be pretty big, considering its size, so some registers and a pipeline organization could be necessary to meet the clock frequency constraints of the implementation. The *iterative* architecture can operate at higher clock frequencies than the *unrolled* one, even if it requires multiple clock cycles.
2) It requires a greater number of components and much more space than the *iterative* scheme, which leads to a bigger power consumption.
3) The *iterative* scheme can be implemented in an hardware description language, such as Verilog or VHDL, to be scalable[2], while the *unrolled* one does not scale at all. Thus, if the number of iteration must be increased, the overall implementation has to be modified.

[2]As long as the lookup table is updated when changing the number of bits of the implementation.

In conclusion we chose to implement the *iterative* architecture scheme of our algorithm, over the *unrolled* one, because of its scalability and because of the speed vs area trade-off when implementing this scheme on an FPGA [6].

IV. VHDL Description

The complete list of hardware schemes and VHDL source files can be found in Appendix A. In this section we will illustrate some of the key characteristics of our implementation.

First of all, the complete scheme has been divided into 4 main components:

- A counter, which gives to the other components the current iteration count of the algorithm.
- A component which executes the CORDIC rotational algorithm on the two components A and B, performing at each step the arithmetic operations needed on such components.
- A Special Case Detector, which is responsible for handling the following two special cases:

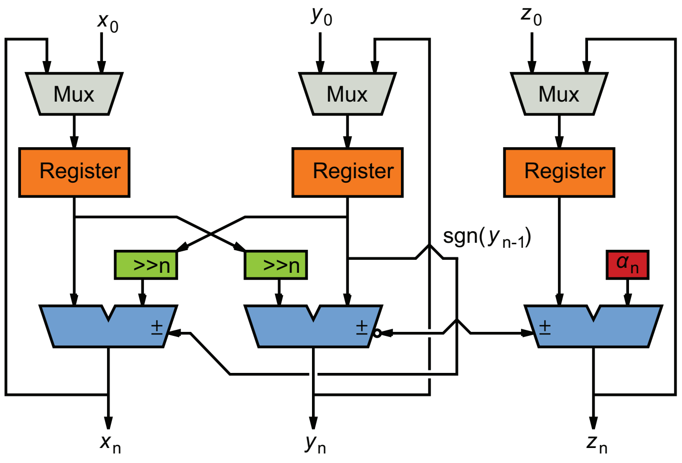


Fig. 5. CORDIC Iterative Rotating Architecture.

1) $A = 0$, $B = 0$, where the output shall be 0;
2) $A = 0$, $B \neq 0$, where the output shall be $\pm 90°$, depending on the sign of B;

- The Atan Accumulator, which at each step accumulates phase values taken from the lookup table and that will provide the final result at the last iteration.

This complete scheme is shown in fig. 24. Of course, to develop these main components many simpler building blocks were needed, like multiplexers, adders, subtractors, shifters and so on. However, we will not go down into details of such implementations here, for further information take as reference the listings in Appendix A.

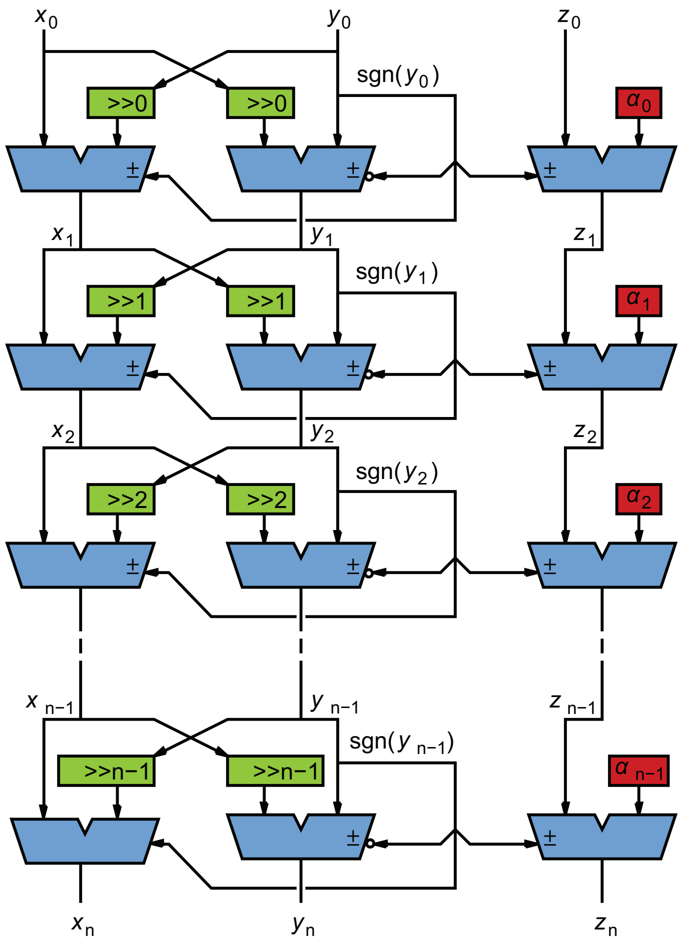


Fig. 6. CORDIC Unrolled Rotating Architecture.

A. Testing

Testing the developed components in VHDL is an important phase of the development process. That's why for some of our components we developed some test benches to check if they actually have the intended behavior.

The most important test that we needed to perform however is the system test bench, in which we check that the overall VHDL-specified system provides the same behavior of the MATLAB *bit true model*. To check completely the system behavior, all the possible inputs have to be provided one by one, to check if the system gives for each of them the correct result.

However, since our input domain is $[-2048, 2047] \times [-2048, 2047]$, the overall number of test cases that we should provide to the system is over 16 million, which is an unreasonable number for our processing ability.

Hence, making a test bench that explores all possible combinations of input is not possible. Given this fact, to test our system we provided to it 1000 different pairs of coordinates (A, B) generated randomly from MATLAB and we checked for the obtained results. To be more precise:

- We first selected 1000 randomly generated pairs[3].
- We produced the expected results using a trusted algorithm, which in our case is the MATLAB *bit true model* of our system.
- We gave the same inputs to the system and compared the obtained results with the expected ones.

To provide the pairs to the system and check the obtained results, we exported them from MATLAB in two's complement binary representation and put them inside a test bench developed in VHDL language. The test bench listing is shown in Appendix A-P. The execution of this test did not show any error for any of the test cases, including the special ones.

V. FPGA Implementation

The final step of the development process consists in implementing the CORDIC on a Xilinx Zybo FPGA. The ZYBO (Zynq Board) is an entry-level embedded software and digital circuit development platform built around the smallest member of the Xilinx Zynq-7000 family, the Z-7010. The Z-7010 tightly integrates a dual-core ARM Cortex-A9 processor with Xilinx 7-series Field Programmable Gate Array (FPGA) logic. The tool used for the synthesis is Vivado, which is

[3]To check also the behavior of the system in the special cases, we actually provided to it 998 random test cases and the two special cases that we described before.

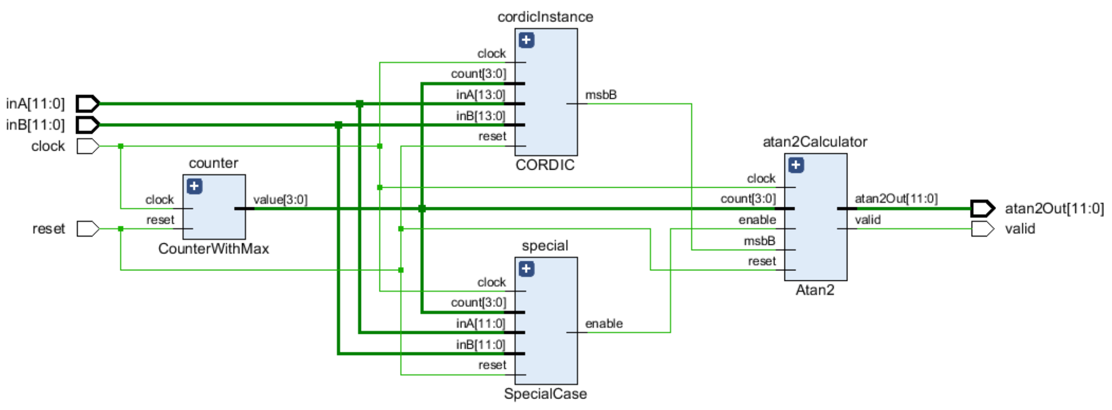


Fig. 7. Vivado RTL design schematic, generated automatically using the RTL elaboration tool. Notice the similarity with fig. 24.

directly provided by Xilinx. To complete with success the implementation on the board, the tool provide a step-by-step mechanism.

A. RTL elaboration

The first step is *RTL analysis*. In this phase the tool analyzes the VHDL source codes provided and generates the complete schematic of the system, which can be expanded to explore the hierarchy of system components. The design elaborated by the tool allows to check whether the VHDL description of the system is actually what the designer was describing in VHDL language. The obtained schematic is shown in fig. 7.

B. Synthesis

The second step is the *synthesis* phase. In this phase, many constraints can be provided to the tool to execute a design optimization and check whether these constraints are indeed satisfied or not. The constraints that we have to check before getting further in the development process are obviously *timing constraints*. As we know, there are two inequalities that need to be satisfied in order to obtain a reliable electronic component:

$$\begin{gathered} T_{clock} \geq t_{cq} + t_{p_{logic}} + t_{su} \\ t_{hold} \leq t_{cd_{logic}} + t_{cd_{reg}} \end{gathered} \tag{9}$$

In particular, the first equation determines an upper bound to the clock frequency we can provide to the system: in fact, it requires that the clock period is greater than the delay of the "longest" path (the one with a bigger delay) between a register and another register, also known as *critical path*. If this constraint is not met we would have problems when updating registers' content at each clock cycle.

These constraints are mandatory and they need to be verified on the *critical path* to be hence verified on each register-logic-register path in the system. When executing the *synthesis* using Vivado tools, this operation is performed automatically and we obtain as a result a timing report.

On the Xilinx Zync Board, there is an obscillator that works at 50 MHz frequency and we can obtain a clock source up to 125 MHz using a Phase-Locked Loop (PLL) component connected to the FPGA. So we would like the system to be able to run at least at 125 MHz.

When providing as *timing constraint* a 125 MHz clock we indeed find out that our system could work properly even at a higher frequency, since the obtained *slack* parameters are all positive. The *slack* parameters represent the margin that we have with respect with the inequalities shown in (9).

The result of the analysis with a 125 MHz clock is shown in fig. 8. In this report, the *Worst Negative Slack* refers to the first inequality in (9), while the *Worst Hold Slack* to the second one. A further inspection of the paths showed that the critical path is between the Accumulator instances inside each Single Dimension Rotator and its counterpart in the opposite Rotator in the CORDIC component, due to the high number of logic components between such registers (adder/subtractors, shifters and so on).

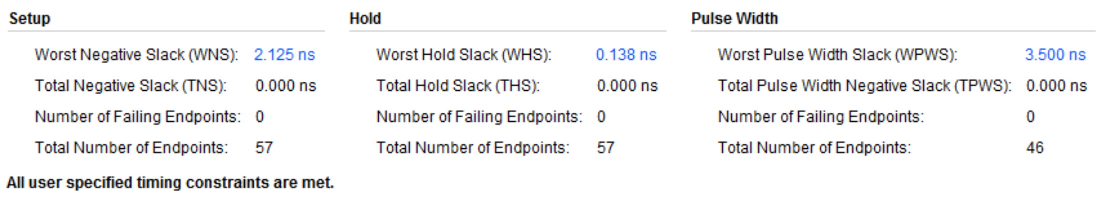
If we keep performing these analyses, we find out actually that the maximum clock frequency at which the system still verifies the constraints is about 170 MHz. However it should be avoided, if possible, to use this design with such clock frequency, since the *Worst Negative Slack* becomes zero, hence problems like *clock jitter* and *clock skew* may compromise the correct behavior.

C. Implementation

The last step before the actual flashing of the system on a physical board is the *implementation* phase. In this phase, the tool performs the placing of the required components in a simulation of the specified FPGA, in order to check once more all the requirements before flashing the system on the device.

The simulations and analyses that can be performed on the system in this phase are the closest to the real execution of the system on the FPGA. However, to do so we also need to set for all the I/O ports in our schematic the actual port of the board that will be connected to it. Since we cannot perform a simulation on a real board, we stopped at this step the implementation process.

However, we can show the Utilization of the FPGA as reported by Vivado, shown in Table II, and perform a Power



Setup

Worst Negative Slack (WNS):	2.125 ns
Total Negative Slack (TNS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	57

Hold

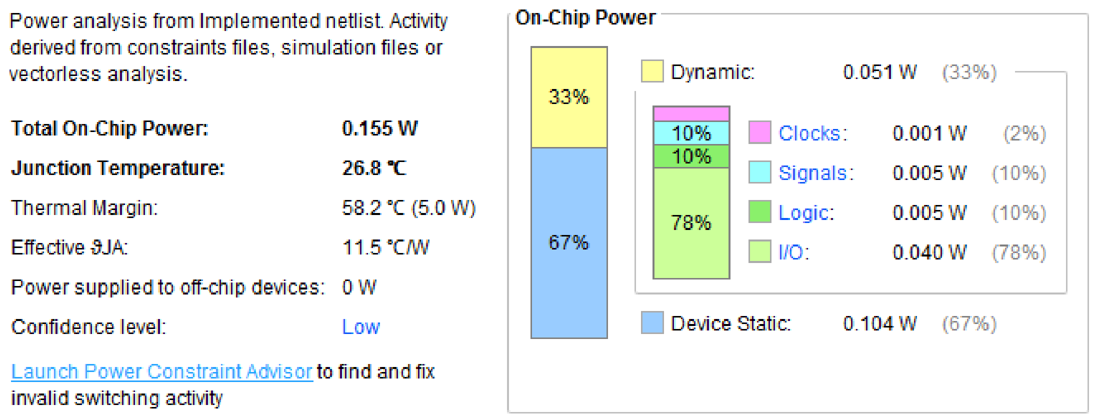
Worst Hold Slack (WHS):	0.138 ns
Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	57

Pulse Width

Worst Pulse Width Slack (WPWS):	3.500 ns
Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	46

All user specified timing constraints are met.

Fig. 8. Vivado Synthesis Timing Summary executed with a 125 MHz clock.



Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.155 W**
Junction Temperature:	**26.8 ℃**
Thermal Margin:	58.2 ℃ (5.0 W)
Effective ϑJA:	11.5 ℃/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:	0.051 W	(33%)
Clocks:	0.001 W	(2%)
Signals:	0.005 W	(10%)
Logic:	0.005 W	(10%)
I/O:	0.040 W	(78%)
Device Static:	0.104 W	(67%)

Fig. 9. Vivado Implementation Power Consumption. The ”Low” Confidence level is due to the fact that we didn’t map the actual physical ports into the I/O ports of the design. Performing again this analysis with this mapping would give a higher Confidence level.

Consumption Estimation of the device, shown in fig. 9. For that estimation we set the I/O ports working voltage to 3.3V to perform a closest-to-real analysis.

TABLE II.
UTILIZATION FACTOR

Resource	Utilization	Available	Utilization %
LUT	221	17600	1.26
FF	45	35200	0.13
IO	39	100	39.00
BUFG	1	32	3.13

D. Final remarks

In conclusion, both synthesis and the implementation phase concluded with no error. The overall process however contains some warnings. These warnings are shown in *Design Rule Checking* violations section of the Vivado project and they are due to the fact that:

1) The ARM Cortex micro controller available on the board is not used;
2) The I/O ports are not mapped onto the FPGA physical ones.

The first warning can be ignored, as we only need to flash the system on the FPGA and we don’t need the micro controller. Before generating the *bitstream* for the device, the second warning must be solved performing the port mapping, thus allowing the device to communicate with the external environment.

VI. CONCLUSION

The CORDIC algorithm was conceived in 1956, thus it is well known in research electronics areas. However, with the overhead introduced by a more flexible design such as FPGA, in order to remain competitive, it is important to implement it as a fast and low power-demanding algorithm. This work showed that adopting the CORDIC algorithm efficient and scalable implementations of trigonometric function can be easily implemented on FPGA-based computing machines, which maybe will the basis for the next generation of DSP systems.

REFERENCES

[1] J. E. Volder, “The CORDIC trigonometric computing technique,” *IRE Transactions on electronic computers*, no. 3, pp. 330–334, 1959.

[2] J. S. Walther, “A unified algorithm for elementary functions,” in *Proceedings of the May 18-20, 1971, spring joint computer conference*. ACM, 1971, pp. 379–385.

[3] B. Lakshmi and A. Dhar, “CORDIC architectures: A survey,” *VLSI design*, vol. 2010, p. 2, 2010.

[4] L. Pilato, L. Fanucci, and S. Saponara, “Real-time and high-accuracy arctangent computation using CORDIC and fast magnitude estimation,” *Electronics*, vol. 6, no. 1, p. 22, 2017.

[5] P. K. Meher, J. Valls, T.-B. Juang, K. Sridharan, and K. Maharatna, “50 years of CORDIC: Algorithms, architectures, and applications,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 56, no. 9, pp. 1893–1907, 2009.

[6] R. Andraka, “A survey of CORDIC algorithms for FPGA based computers,” in *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*. ACM, 1998, pp. 191–200.

[7] Unknown, “Digital circuits / CORDIC — Wikibooks, the free textbook project,” 2017. [Online]. Available: https://en.wikibooks.org/w/index.php?title=Digital_Circuits/CORDIC&oldid=3231681

APPENDIX A
VHDL IMPLEMENTATION OF THE CORDIC ITERATIVE ROTATION ARCHITECTURE

A. *Generic Register*

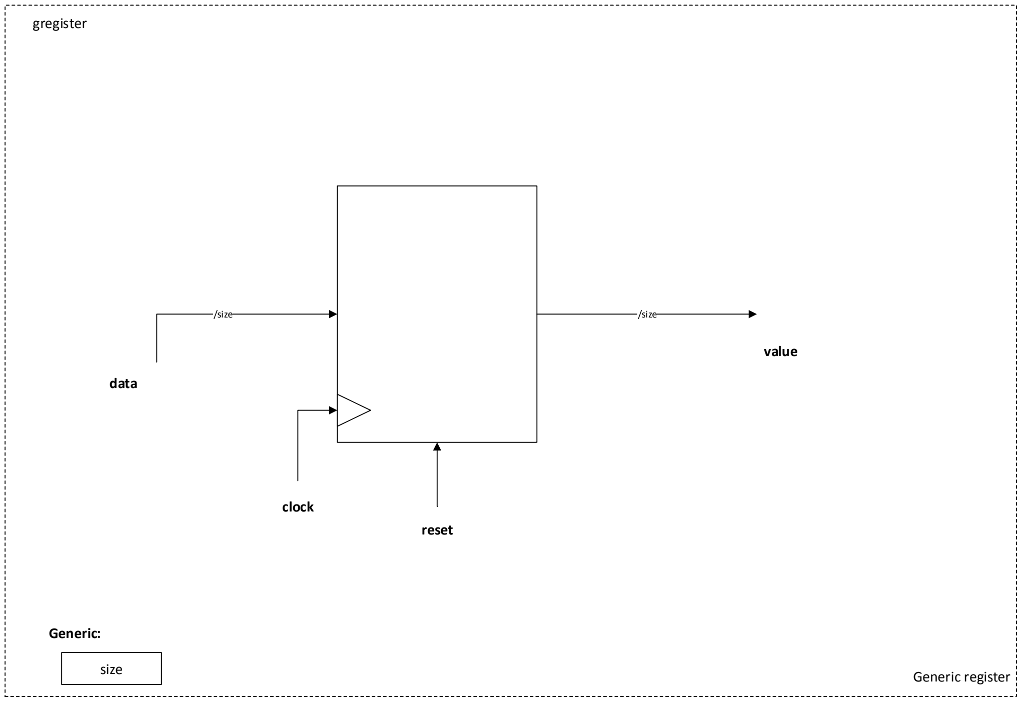


Fig. 10. Generic Register

```
library IEEE;
use IEEE.std_logic_1164.all;

--------------------------------------------------------------------------------
-- Generic Register
--
-- This component defines a register of a generic size. The reset signal is
-- active when it is set to 1.
--------------------------------------------------------------------------------

entity GRegister is
    generic (size : positive := 8);
    port (
        clock   : in    std_ulogic;
        reset   : in    std_ulogic;
        data    : in    std_ulogic_vector(size-1 downto 0);
        value   : out   std_ulogic_vector(size-1 downto 0)
    );
end GRegister;

architecture GRegister_Arch of GRegister is
begin
    assignment: process(clock, reset)
    begin
        if reset = '1' then
            value <= (others => '0');
        elsif(clock'event and clock = '1') then
            value <= data;
        end if;
    end process;
end GRegister_Arch;
```

B. Generic Register with Enable

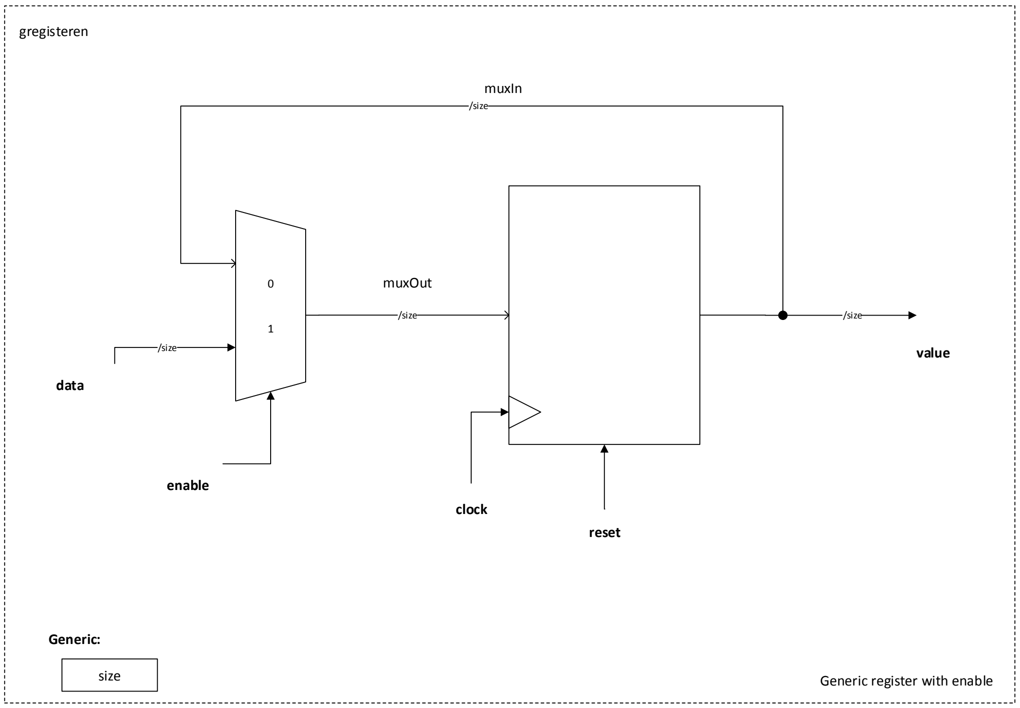


Fig. 11. Generic Register with Enable

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

--------------------------------------------------------------------------------
-- Generic Register with Enable signal
--
-- This component defines a register of a generic size which can be enabled or
-- disabled using a special input:
--
-- - when the enable input is high, the register acts like a normal Generic
--   Register.
--
-- - when the enable input is low, the register enters a holding state and it is
--   unsensitive to its input data.
--
--------------------------------------------------------------------------------

entity GRegisterEn is
    generic (size : positive := 8);
    port (
        clock   : in    std_ulogic;
        reset   : in    std_ulogic;
        enable  : in    std_ulogic;
        data    : in    std_ulogic_vector(size-1 downto 0);
        value   : out   std_ulogic_vector(size-1 downto 0)
    );
end GRegisterEn;

architecture GRegisterEn_Arch of GRegisterEn is

    -- The sensitivity of the register is regulated using a multiplexer, the
    -- inputs of the multiplexer are the data from outside and the value
    -- contained in the register. When the enable signal is zero, the loopback
    -- value is sampled again, hence the value of the register does not change.
    signal dataMuxIn : std_ulogic_vector(size-1 downto 0);
    signal dataMuxOut : std_ulogic_vector(size-1 downto 0);

begin
    assignment: process(clock, reset)
    begin
        if reset = '1' then
            dataMuxIn <= (others => '0');
        elsif(clock'event and clock = '1') then
            dataMuxIn <= dataMuxOut;
        end if;
    end process;

    dataMuxOut <= data when enable = '1' else dataMuxIn;
    value <= dataMuxIn;

end GRegisterEn_Arch;
```

C. Full Adder

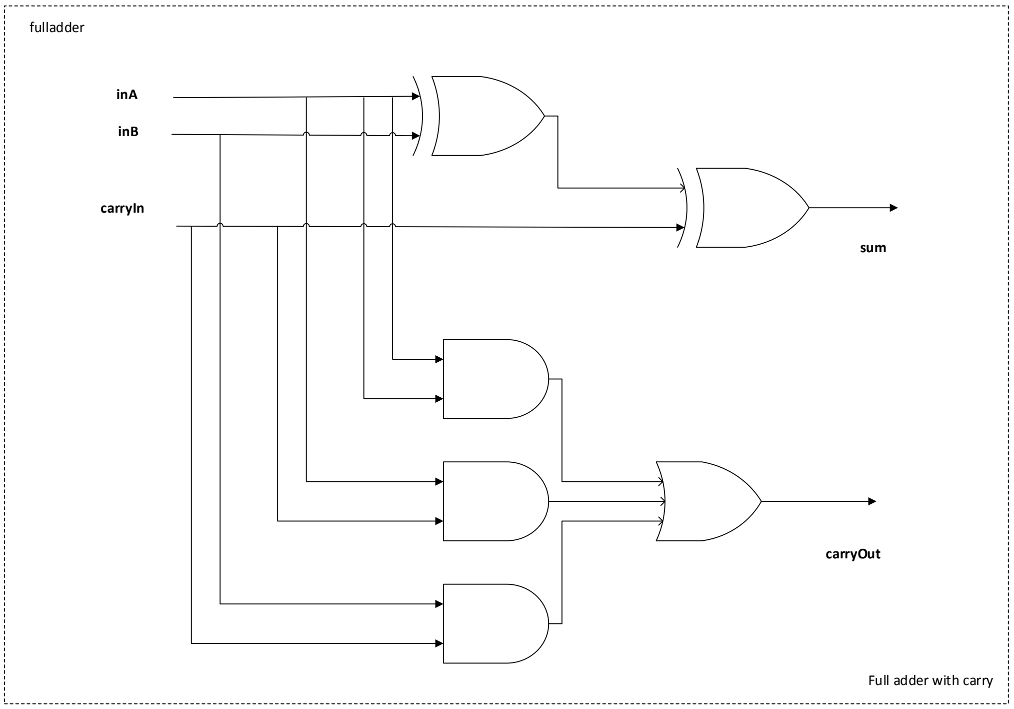


Fig. 12. Full Adder

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

--------------------------------------------------------------------------------
-- Full Adder
--
-- This component defines a standard full adder with one bit.
--------------------------------------------------------------------------------

entity FullAdder is
    port (
        inB         : in    std_ulogic;
        inA         : in    std_ulogic;
        carryIn     : in    std_ulogic;
        carryOut    : out   std_ulogic;
        sum         : out   std_ulogic
    );
end FullAdder;

architecture FullAdder_Arch of FullAdder is
begin
    sum         <= inA xor inB xor carryIn;
    carryOut    <= (inA and inB) or (inA and carryIn) or (inB and carryIn);
end FullAdder_Arch;
```

D. Adder/Subtractor

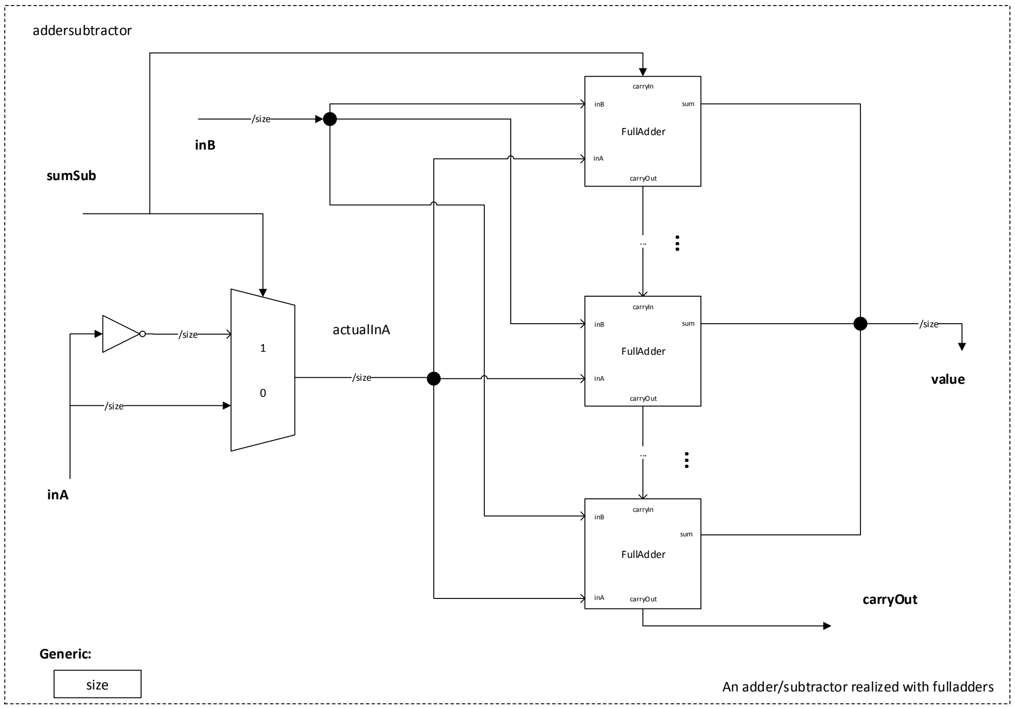


Fig. 13. Adder/Subtractor

```
library IEEE;
use IEEE.std_logic_1164.all;

--------------------------------------------------------------------------------
-- Adder Subtractor
--
-- This component defines a combinatoral logic that is able to perform basic
-- sums, subtractions and change of sign in two's complement between numbers
-- expressed with a generic number of bits.
--
-- The sumSub input is responsible to decide whether the output of the component
-- will be the sum A+B or the subtractions B-A of the two inputs:
--
-- - sumSub = 0 then the output is A+B
--
-- - sumSub = 1 then the output is B-A
--
-- To perform the inverse of a single number, the number must be put in the A
-- input and the B input must be zero, so basically the output is 0-A.
--
--------------------------------------------------------------------------------

entity AdderSubtractor is
    generic (size : positive := 8);
    port (
        inA          : in    std_ulogic_vector(size-1 downto 0);
        inB          : in    std_ulogic_vector(size-1 downto 0);
        sumSub       : in    std_ulogic;
        carryOut     : out   std_ulogic;
        value        : out   std_ulogic_vector(size-1 downto 0)
    );
end AdderSubtractor;

architecture AdderSubtractor_Arch of AdderSubtractor is

    component FullAdder
        port (
            inA          : in    std_ulogic;
            inB          : in    std_ulogic;
            carryIn      : in    std_ulogic;
            carryOut     : out   std_ulogic;
            sum          : out   std_ulogic
        );
    end component FullAdder;

    signal carryWires   : std_ulogic_vector(size-2 downto 0);
    signal actualInA    : std_ulogic_vector(size-1 downto 0);

begin

    -- To perform subtractions, the input A must be converted in its inverse in
    -- two's complement and then added to the input B; to do so, we first
    -- calculate the one's complement of A
    actualInA <= inA when sumSub = '0' else not(inA);

    -- The Adder inside is obtained chaining together Full Adders, in a Ripple
    -- Carry Adder configuration
    generateFullAdders: for i in 0 to size-1 generate

        -- The first Full Adder has carryIn connected to the sumSub of the
        -- design, in order to add 1 to perform the two's complement on A when
        -- B-A operation is requested
        first: if i = 0 generate
                fullAdderFirst: FullAdder
                    port map (
                        inA          => actualInA(i),
                        inB          => inB(i),
                        carryIn      => sumSub,
                        carryOut     => carryWires(i),
                        sum          => value(i)
                    );
            end generate first;

        -- Internal Full Adders have carries connected in chain
        internal: if i > 0 and i < size-1 generate
                fullAdderInternal: FullAdder
                    port map (
                        inA          => actualInA(i),
                        inB          => inB(i),
                        carryIn      => carryWires(i-1),
                        carryOut     => carryWires(i),
```

```vhdl
                          sum         => value(i)
                      );
              end generate internal;

          --The last Full Adder has carryOut connected to the carryOut of the
          -- design
          last: if i = size-1 generate
                  fullAdderLast: FullAdder
                      port map (
                          inA         => actualInA(i),
                          inB         => inB(i),
                          carryIn     => carryWires(i-1),
                          carryOut    => carryOut,
                          sum         => value(i)
                      );
              end generate last;
      end generate generateFullAdders;
  end AdderSubtractor_Arch;
```

E. Accumulator

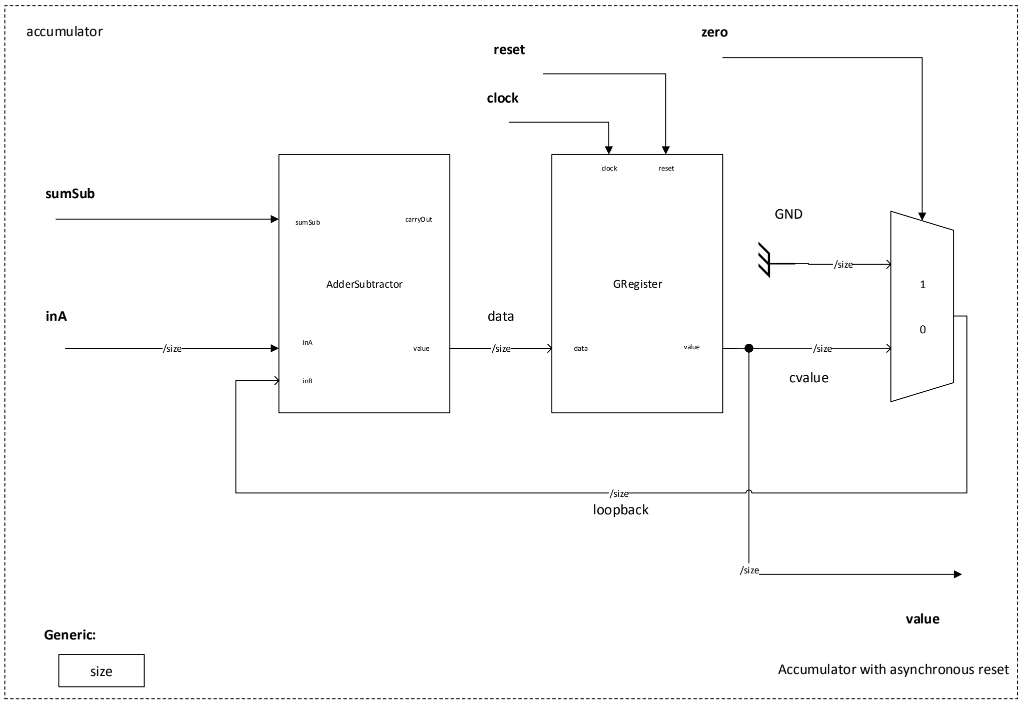


Fig. 14. Accumulator

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

-----------------------------------------------------------------------------------
-- Accumulator
--
-- This component defines an Accumulator, which is basically like a Counter, but
-- it is able to perform both sums or subtractions of the memorized value each
-- iteration.
--
-- The zero input is used to reset synchronously the value of the Accumulator,
-- so the value of the next iteration will be the one provided as input to the
-- Accumulator, since it would be 0+A or 0-A, depending on the sumSub value.
--
-----------------------------------------------------------------------------------

entity Accumulator is
    generic (size : positive := 8);
    port (
        clock   : in    std_ulogic;
        reset   : in    std_ulogic;
        zero    : in    std_ulogic;
        inA     : in    std_ulogic_vector(size-1 downto 0);
        sumSub  : in    std_ulogic;
        value   : out   std_ulogic_vector(size-1 downto 0)
    );
end Accumulator;

architecture Accumulator_Arch of Accumulator is
    component AdderSubtractor is
        generic (size : positive := 8);
        port (
            inA         : in    std_ulogic_vector(size-1 downto 0);
            inB         : in    std_ulogic_vector(size-1 downto 0);
            sumSub      : in    std_ulogic;
            carryOut    : out   std_ulogic;
            value       : out   std_ulogic_vector(size-1 downto 0)
        );
    end component;

    component GRegister
        generic (size : positive := 8);
        port (
            clock   : in    std_ulogic;
            reset   : in    std_ulogic;
            data    : in    std_ulogic_vector(size-1 downto 0);
            value   : out   std_ulogic_vector(size-1 downto 0)
        );
    end component GRegister;

    -- Wire between the output of the adder and the input of the register
    signal data     : std_ulogic_vector(size-1 downto 0);

    -- Wires used to loopback the output of the register into the input of the
    -- adder and to connect it to the output of the design
    signal cvalue   : std_ulogic_vector(size-1 downto 0);
    signal loopback : std_ulogic_vector(size-1 downto 0);

begin

    addsubInstance : AdderSubtractor
        generic map(size => size)
        port map (
            inA         => inA,
            inB         => loopback,
            sumSub      => sumSub,
            carryOut    => open,
            value       => data
        );

    registerInstance : GRegister
        generic map(size => size)
        port map (
            clock   => clock,
            reset   => reset,
            data    => data,
            value   => cvalue
        );

    -- The output is the output value of the register
    value <= cvalue;

    -- The loopback is the output value of the register, unless the synchronous
    -- reset "zero" is set to 1
    loopback <= cvalue when zero = '0' else (others => '0');

end Accumulator_Arch;
```

F. Accumulator with Enable signal

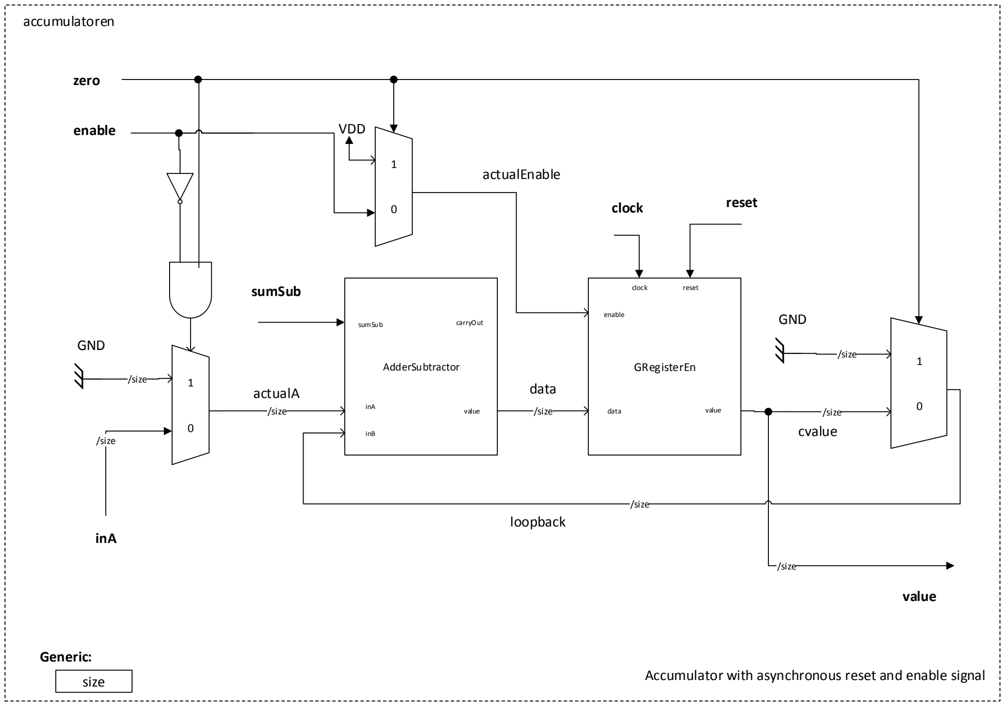


Fig. 15. Accumulator with Enable signal

```
library IEEE;
use IEEE.std_logic_1164.all;

--------------------------------------------------------------------------------
-- Accumulator with Enable signal
--
-- This component defines a variation of the Accumulator which uses a Generic
-- Register with Enable as memory.
--
-- Its behavior depends on both the values of the zero and the enable signals;
-- the exact behavior of the component is the following:

-- +------------+------------+---------------------------------------------+
-- |    Zero    |   Enable   |                   Behavior                  |
-- +============+============+=============================================+
-- |     0      |     1      | The same exact behavior of the Accumulator  |
-- +------------+------------+---------------------------------------------+
-- |     1      |     1      | The same exact behavior of the Accumulator  |
-- +------------+------------+---------------------------------------------+
-- |     0      |     0      | The component becomes insensitive to the input |
-- |            |            | values, like the Register with Enable signal |
-- +------------+------------+---------------------------------------------+
-- |     1      |     0      | The component stored value becomes zero, for |
-- |            |            | whatever input value is provided            |
-- +------------+------------+---------------------------------------------+
--
--------------------------------------------------------------------------------

entity AccumulatorEn is
    generic (size : positive := 8);
    port (
        clock   : in    std_ulogic;
        reset   : in    std_ulogic;
        zero    : in    std_ulogic;
        enable  : in    std_ulogic;
        inA     : in    std_ulogic_vector(size-1 downto 0);
        sumSub  : in    std_ulogic;
        value   : out   std_ulogic_vector(size-1 downto 0)
    );
end AccumulatorEn;

architecture AccumulatorEn_Arch of AccumulatorEn is
    component AdderSubtractor is
        generic (size : positive := 8);
        port (
            inA          : in    std_ulogic_vector(size-1 downto 0);
            inB          : in    std_ulogic_vector(size-1 downto 0);
            sumSub       : in    std_ulogic;
            carryOut     : out   std_ulogic;
            value        : out   std_ulogic_vector(size-1 downto 0)
        );
    end component;

    component GRegisterEn
        generic (size : positive := 8);
        port (
            clock   : in    std_ulogic;
            reset   : in    std_ulogic;
            enable  : in    std_ulogic;
            data    : in    std_ulogic_vector(size-1 downto 0);
            value   : out   std_ulogic_vector(size-1 downto 0)
        );
    end component GRegisterEn;

    -- Wire between the output of the adder and the input of the register
    signal data     : std_ulogic_vector(size-1 downto 0);

    -- Wires used to loopback the output of the register into the input of the
    -- adder and to connect it to the output of the design
    signal cvalue       : std_ulogic_vector(size-1 downto 0);
    signal loopback     : std_ulogic_vector(size-1 downto 0);

    -- Wires used to ensure the behavior of the component described in the table
    signal actualA      : std_ulogic_vector(size-1 downto 0);
    signal actualEnable : std_ulogic;

begin

    addsubInstance : AdderSubtractor
        generic map(size => size)
        port map (
            inA         => actualA,
            inB         => loopback,
            sumSub      => sumSub,
            carryOut    => open,
            value       => data
        );

    registerInstance : GRegisterEn
        generic map(size => size)
        port map (
            clock   => clock,
            reset   => reset,
            enable  => actualEnable,
            data    => data,
            value   => cvalue
        );

    -- The output is the output value of the register
    value <= cvalue;

    -- The loopback is the output value of the register, unless the synchronous
    -- reset "zero" is set to 1
    loopback <= cvalue when zero = '0' else (others => '0');

    actualA <= (others => '0') when zero = '1' and enable = '0' else inA;
    actualEnable <= enable when zero = '0' else '1';

end AccumulatorEn_Arch;
```

G. Counter with a Maximum value

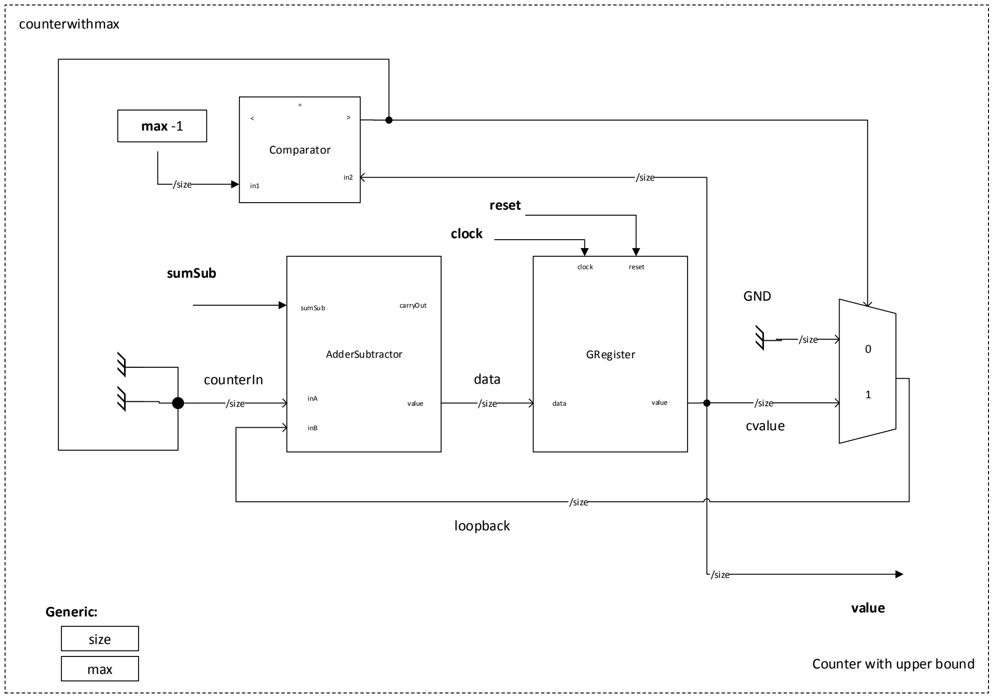


Fig. 16. Counter with a Maximum value

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

----------------------------------------------------------------------------------
-- Counter with a Maximum value
--
-- This component defines a variation of the Accumulator which can count only
-- by steps of 1 each time and that is automatically resetted when the internal
-- value reaches a maximum value.
--
-- Basically, the output of the Counter each clock is a value between 0 and
-- max-1.
--
----------------------------------------------------------------------------------

entity CounterWithMax is
    generic (
        size    : positive := 8;
        max     : positive := 16
    );
    port (
        clock   : in    std_ulogic;
        reset   : in    std_ulogic;
        value   : out   std_ulogic_vector(size-1 downto 0)
    );
end CounterWithMax;

architecture CounterWithMax_Arch of CounterWithMax is
    component AdderSubtractor is
        generic (size : positive := 8);
        port (
            inA          : in    std_ulogic_vector(size-1 downto 0);
            inB          : in    std_ulogic_vector(size-1 downto 0);
            sumSub       : in    std_ulogic;
            carryOut     : out   std_ulogic;
            value        : out   std_ulogic_vector(size-1 downto 0)
        );
    end component;

    component GRegister
        generic (size : positive := 8);
        port (
            clock    : in    std_ulogic;
            reset    : in    std_ulogic;
            data     : in    std_ulogic_vector(size-1 downto 0);
            value    : out   std_ulogic_vector(size-1 downto 0)
        );
    end component GRegister;

    -- Wire between the output of the adder and the input of the register
    signal data          : std_ulogic_vector(size-1 downto 0);

    -- Wire used to loopback the output of the register into the input of the
    -- adder and to connect it to the output of the design
    signal cvalue        : std_ulogic_vector(size-1 downto 0);
    signal loopback      : std_ulogic_vector(size-1 downto 0);
    signal counterIn     : std_ulogic_vector(size-1 downto 0);

begin

    addsubInstance : AdderSubtractor
        generic map(size => size)
        port map (
            inA          => counterIn,
            inB          => loopback,
            sumSub       => '0',
            carryOut     => open,
            value        => data
        );

    registerInstance : GRegister
        generic map(size => size)
        port map (
            clock    => clock,
            reset    => reset,
            data     => data,
            value    => cvalue
        );

    -- The output is the output value of the register
    value <= cvalue;

    -- The input of the Adder Subtractor is 1, unless the value has reached the
    -- maximum permitted value, in that case the input is 0
    counterIn <= std_ulogic_vector(to_unsigned(1, size))
                when ((max-1) > to_integer(unsigned(cvalue)))
                else std_ulogic_vector(to_unsigned(0, size));

    -- The loopback is the output value of the register, unless the value
    -- has reached the maximum permitted value
    loopback <= cvalue
                when ((max-1) > to_integer(unsigned(cvalue)))
                else (others => '0');

end CounterWithMax_Arch;
```

H. Shift Selection Adjuster

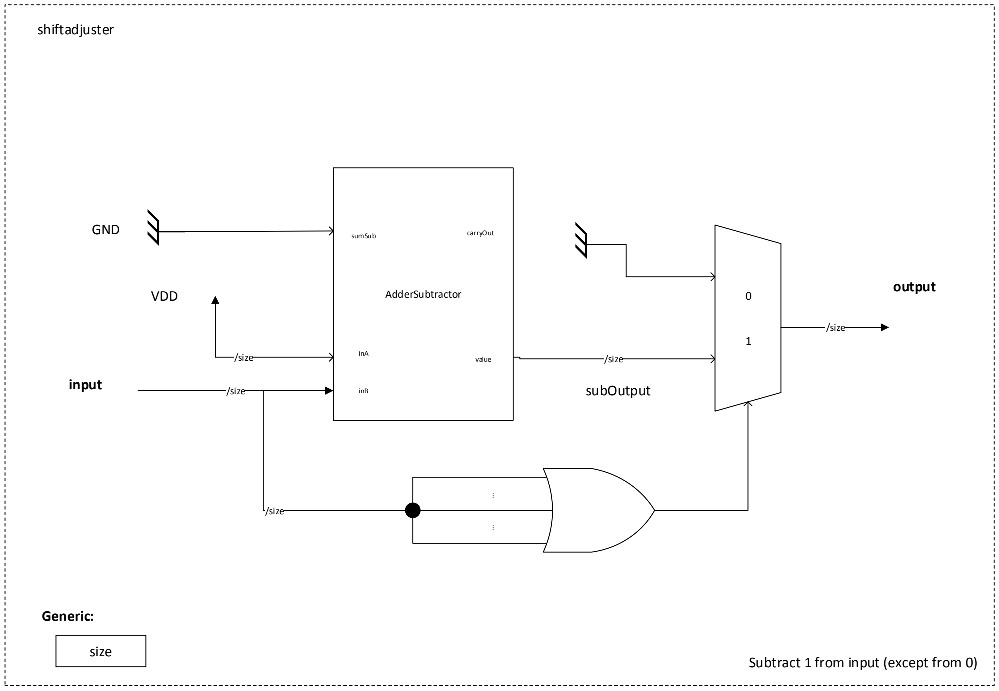


Fig. 17. Shift Selection Adjuster

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

--------------------------------------------------------------------------------
-- Shift Selection Adjuster
--
-- This component defines a combinatoral logic that adjusts the select value of
-- the shifter inside the CORDIC component to satisfy the needs of the CORDIC
-- algorithm.
--
-- Basically, this component takes as input the current iteration of the CORDIC
-- algorithm and provides as output the number of bits that need to be shifted
-- in that iteration, that is equal to:
--
-- - count = 0
--              no shift is needed, the output is 0
-- - count > 0
--              the input word must be shifted of count-1 positions
--
--------------------------------------------------------------------------------

entity ShiftAdjuster is
    generic (
        size    : positive := 8
```

```vhdl
    );
    port (
        input   : in    std_ulogic_vector(size-1 downto 0);
        output  : out   std_ulogic_vector(size-1 downto 0)
    );
end ShiftAdjuster;

architecture ShiftAdjuster_Arch of ShiftAdjuster is
    component AdderSubtractor is
        generic (size : positive := 8);
        port (
            inA         : in    std_ulogic_vector(size-1 downto 0);
            inB         : in    std_ulogic_vector(size-1 downto 0);
            sumSub      : in    std_ulogic;
            carryOut    : out   std_ulogic;
            value       : out   std_ulogic_vector(size-1 downto 0)
        );
    end component;

    signal subOutput : std_ulogic_vector(size-1 downto 0);
begin

    output <= (others => '0') when unsigned(input) = 0 else subOutput;

    -- The input value of the subtractor is -1 in two's complement, so the
    -- resulting operation in two's complement is count + (-1) = count-1
    subtractor : AdderSubtractor
        generic map (size => size)
        port map (
            inA         => (others => '1'),
            inB         => input,
            sumSub      => '0',
            carryOut    => open,
            value       => subOutput
        );
end ShiftAdjuster_Arch;
```

I. Right Arithmetic Shifter

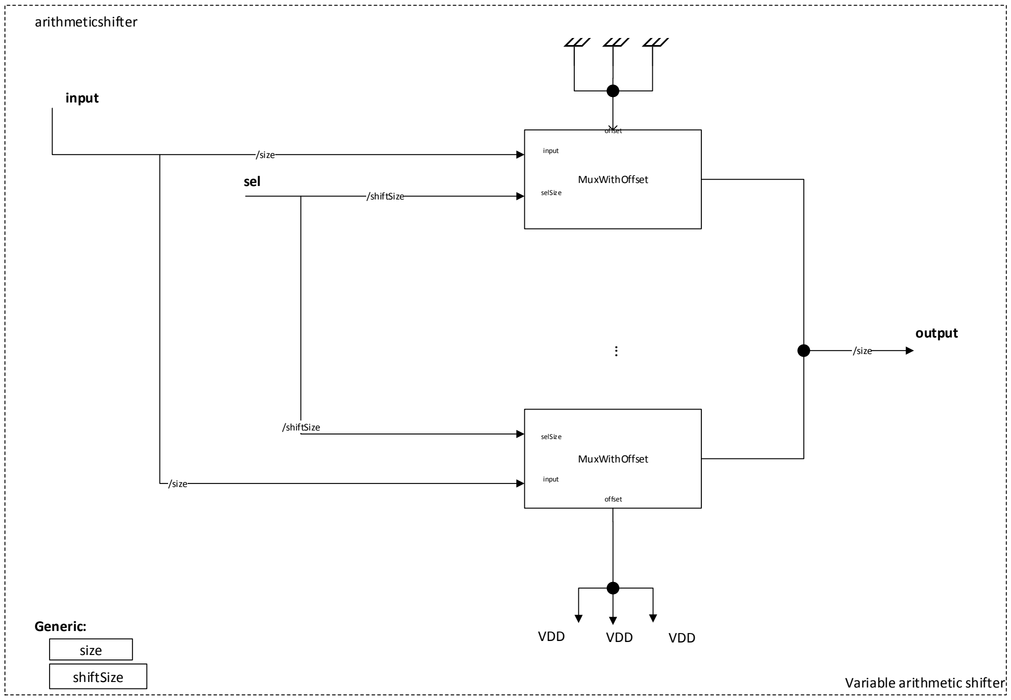


Fig. 18. Right Arithmetic Shifter

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

--------------------------------------------------------------------------------
-- Arithmetic Shifter
--
-- This component defines a combinatoral logic that executes the Arithmetic
-- Shift operation of the input word.
--
--------------------------------------------------------------------------------

entity ArithmeticShifter is
    generic (
        size        : positive := 8;
        shiftSize   : positive := 3
    );
    port (
        shift    : in    std_ulogic_vector(shiftSize-1 downto 0);
        input    : in    std_ulogic_vector(size-1 downto 0);
        output   : out   std_ulogic_vector(size-1 downto 0)
    );
end ArithmeticShifter;

architecture ArithmeticShifter_Arch of ArithmeticShifter is
```

```
    component MuxWithOffset is
        generic (
            size    : positive  := 8;
            selSize : positive  := 3;
            offset  : natural   := 0
            );
        port (
            sel     : in    std_ulogic_vector(selSize-1 downto 0);
            input   : in    std_ulogic_vector(size-1 downto 0);
            output  : out   std_ulogic
        );
    end component;
begin

    -- Each Multiplexer with Offset used inside is used to select the proper
    -- value for the output bit in the position equal to the given offset
    generateMuxes: for i in 0 to size-1 generate
        mux: MuxWithOffset
            generic map (
                size        => size,
                selSize     => shiftSize,
                offset      => i
                )
            port map (
                sel         => shift,
                input       => input,
                output      => output(i)
            );
    end generate generateMuxes;

end ArithmeticShifter_Arch;
```

J. *Multiplexer with static Offset*

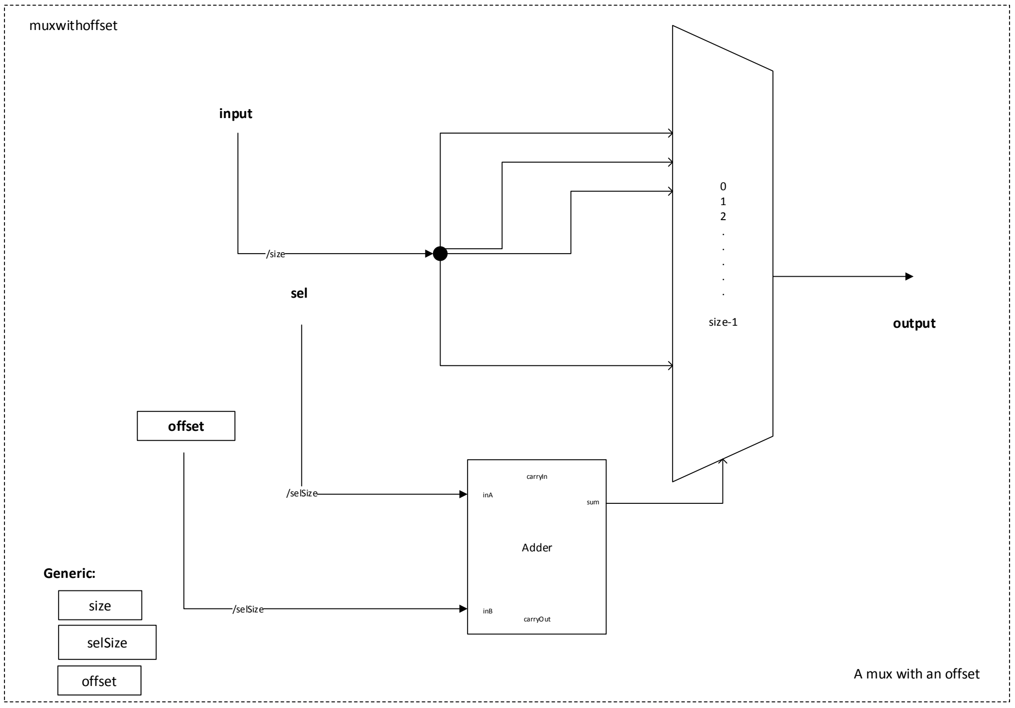


Fig. 19. Multiplexer with static Offset

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

--------------------------------------------------------------------------------
-- Multiplexer with static Offset
--
-- This component defines a combinatoral logic that is equivalent to a
-- multiplexer, but the multiplexer selector has a fixed offset value.
--
-- This means that when the sel input is zero, the bit in the output is the one
-- in the position corresponding with the offset value. If the sel is 1, the
-- output is the one in the position next to the offset value one and so on.
--
-- When the requested bit position is not possible, the most significant bit of
-- the input word is provided as output.
--
-- This component has a proper meaning only when used to generate an Arithmetic
-- Shifter, so please see its usage in the Arithmetic Shifter architecture.
--
--------------------------------------------------------------------------------

entity MuxWithOffset is
    generic (
        size    : positive  := 8;
        selSize : positive  := 3;
        offset  : natural   := 0
        );
    port (
        sel     : in    std_ulogic_vector(selSize-1 downto 0);
        input   : in    std_ulogic_vector(size-1 downto 0);
        output  : out   std_ulogic
    );
end MuxWithOffset;

architecture MuxWithOffset_Arch of MuxWithOffset is
begin

    selection: process(input, sel)
        variable selValue : natural := 0;
    begin
        -- The actual select value of the Multiplexer is the sum between the
        -- input sel and the fixed offset
        selValue := to_integer(
                        resize(unsigned(sel), selSize+1) +
                        to_unsigned(offset, selSize+1)
                    );

        -- If the position excedes the input word length, the msb is provided as
        -- output
        if (selValue > size-1) then
            selValue := size-1;
        end if;

        output <= input(selValue);

    end process selection;

end MuxWithOffset_Arch;
```

K. Special Case Detector

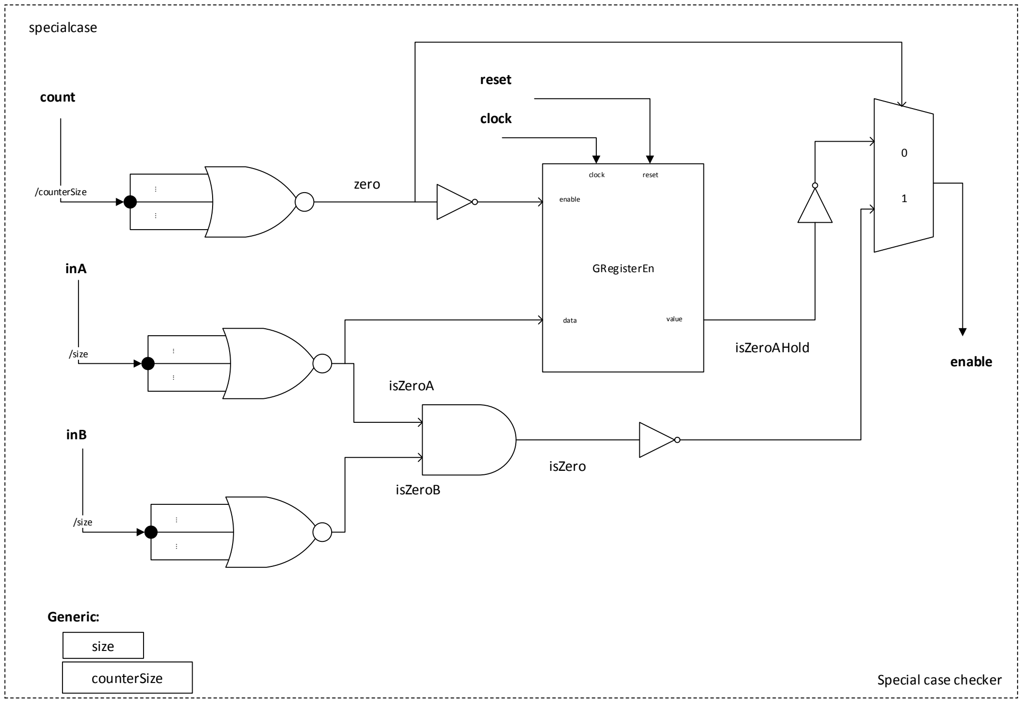


Fig. 20. Special Case Detector

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

--------------------------------------------------------------------------------
-- Special Case detector
--
-- This component checks for the special cases specified by the CORDICAtan2
-- algorithm and pilots the Atan2 component via the enable signal.
--
-- The basic idea behind this component is that there are two special cases:
--
-- - A = 0, B <> 0
--                   in this case, the result of the algorithm must be either pi/2
--                   if B > 0 and -pi/2 otherwise; this result is obtained making
--                   Atan2 component run only for the first iteration and then go
--                   idle after, until a new input is provided.
--
-- - A = 0, B = 0
--                   in this case, the result of the algorithm must be 0 by
--                   convention, so the Atan2 component must not consider any
--                   value coming from the LUT that it has inside, until a new
--                   input is provided.
--------------------------------------------------------------------------------

```

```
entity SpecialCase is
    generic (
        size        : positive := 8;
        counterSize : positive := 8
    );
    port (
        clock       : in    std_ulogic;
        reset       : in    std_ulogic;
        count       : in    std_ulogic_vector(counterSize-1 downto 0);
        inA         : in    std_ulogic_vector(size-1 downto 0);
        inB         : in    std_ulogic_vector(size-1 downto 0);
        enable      : out   std_ulogic
    );
end SpecialCase;

architecture SpecialCase_Arch of SpecialCase is
    component GRegisterEn is
        generic (size : positive := 8);
        port (
            clock   : in    std_ulogic;
            reset   : in    std_ulogic;
            enable  : in    std_ulogic;
            data    : in    std_ulogic_vector(size-1 downto 0);
            value   : out   std_ulogic_vector(size-1 downto 0)
        );
    end component;

    signal zero          : std_ulogic;
    signal enableReg     : std_ulogic;

    signal isZero        : std_ulogic;

    signal isZeroA       : std_ulogic_vector(0 downto 0);
    signal isZeroB       : std_ulogic;

    signal isZeroAHold   : std_ulogic_vector(0 downto 0);

begin

    -- The register inside this component is active only at the first iteration
    -- and for the following ones it will hold its value in order to enable or
    -- not the Atan2 component.
    zero <= '1' when unsigned(count)  = 0 else '0';
    isZeroA <= "1" when unsigned(inA)  = 0 else "0";
    isZeroB <= '1' when unsigned(inB)  = 0 else '0';

    isZero <= isZeroA(0) and isZeroB;

    isZeroAReg: GRegisterEn
        generic map (size => 1)
        port map (
            clock   => clock,
            reset   => reset,
            enable  => zero,
            data    => isZeroA,
            value   => isZeroAHold
        );

    -- At the first iteration, the regster is bypassed and if the two input
    -- values are both zero the Atan2 component is disabled; then we are only
    -- interested if the input A was zero or not at the first iteration
    enable <= not isZero when zero = '1'
                else not isZeroAHold(0);

end SpecialCase_Arch;
```

L. Single Dimension Rotator

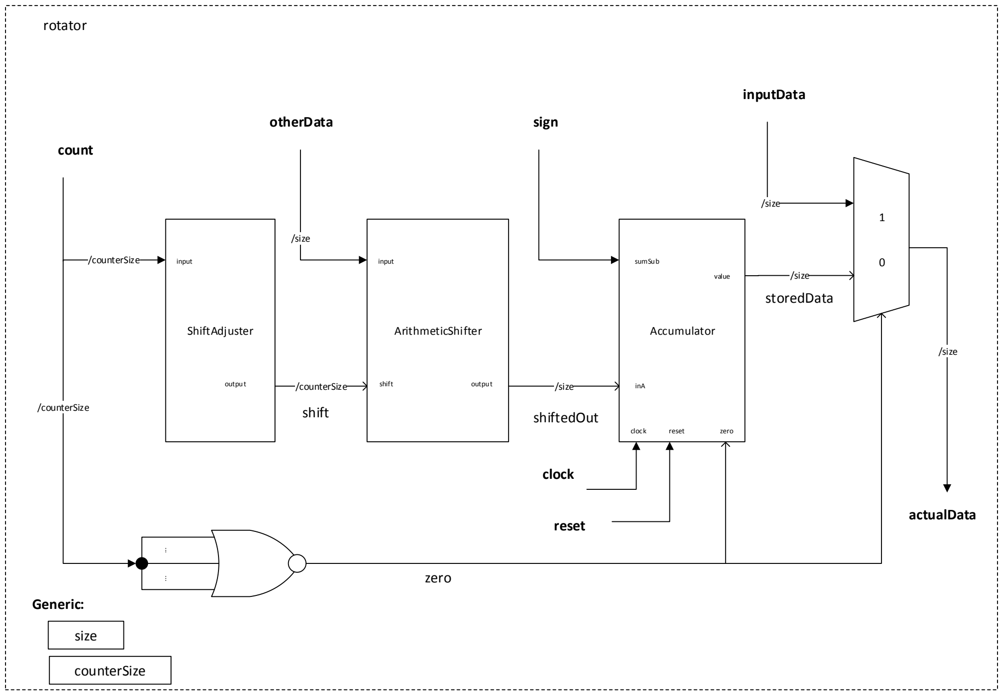


Fig. 21. Single Dimension Rotator

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

---------------------------------------------------------------------------------
-- Single Dimension Rotator
--
-- This component performs the rotations needed by the CORDIC algorithm on one
-- of the dimensions of the input vector.
--
-- Each iteration, the component updates its value depending on the output of
-- another Single Dimension Rotator and its currently stored value, as required
-- by the CORDIC algorithm.
--
-- See the algorithm description for further information.
--
-- The sign input specifies if the given value has to be summed or subtracted to
-- the currently stored value.
--
-- The output value of the rotator is its currently stored value, the other
-- Rotator will shift this value by the amount of bits required by the CORDIC
-- algorithm at the current iteration.
--
-- The inputData signal has a meaning only at the first iteration and then it is
-- ignored for the following ones.
```

```
--
----------------------------------------------------------------------------------

entity Rotator is
    generic (
        size        : positive := 8;
        counterSize : positive := 8
    );
    port (
        clock       : in    std_ulogic;
        reset       : in    std_ulogic;
        sign        : in    std_ulogic;
        count       : in    std_ulogic_vector(counterSize-1 downto 0);
        inputData   : in    std_ulogic_vector(size-1 downto 0);
        otherData   : in    std_ulogic_vector(size-1 downto 0);
        actualData  : out   std_ulogic_vector(size-1 downto 0)
    );
end Rotator;

architecture Rotator_Arch of Rotator is
    component Accumulator is
        generic (size : positive := 8);
        port (
            clock   : in    std_ulogic;
            reset   : in    std_ulogic;
            zero    : in    std_ulogic; -- synchronous reset of the accumulated value
            inA     : in    std_ulogic_vector(size-1 downto 0);
            sumSub  : in    std_ulogic;
            value   : out   std_ulogic_vector(size-1 downto 0)
        );
    end component;

    component ArithmeticShifter is
        generic (
            size        : positive := 8;
            shiftSize   : positive := 3
        );
        port (
            shift   : in    std_ulogic_vector(shiftSize-1 downto 0);
            input   : in    std_ulogic_vector(size-1 downto 0);
            output  : out   std_ulogic_vector(size-1 downto 0)
        );
    end component;

    component ShiftAdjuster is
        generic (
            size    : positive := 8
        );
        port (
            input   : in    std_ulogic_vector(size-1 downto 0);
            output  : out   std_ulogic_vector(size-1 downto 0)
        );
    end component;

    signal zero         : std_ulogic;
    signal shiftedOut   : std_ulogic_vector(size-1 downto 0);
```

```vhdl
    signal storedData   : std_ulogic_vector(size-1 downto 0);
    signal shift        : std_ulogic_vector(counterSize-1 downto 0);

begin

    zero <= '1' when unsigned(count) = 0 else '0';

    actualData <= inputData when zero = '1' else storedData;

    adjusterInstance : ShiftAdjuster
        generic map (
            size    => counterSize
        )
        port map (
            input   => count,
            output  => shift
        );

    accumulatorInstance: Accumulator
        generic map (size => size)
        port map (
            clock   => clock,
            reset   => reset,
            zero    => zero,
            inA     => shiftedOut,
            sumSub  => sign,
            value   => storedData
        );

    shifter: ArithmeticShifter
        generic map (
            size        => size,
            shiftSize   => counterSize
        )
        port map (
            shift       => shift,
            input       => otherData,
            output      => shiftedOut
        );

end Rotator_Arch;
```

M. Cordic Rotational Algorithm component

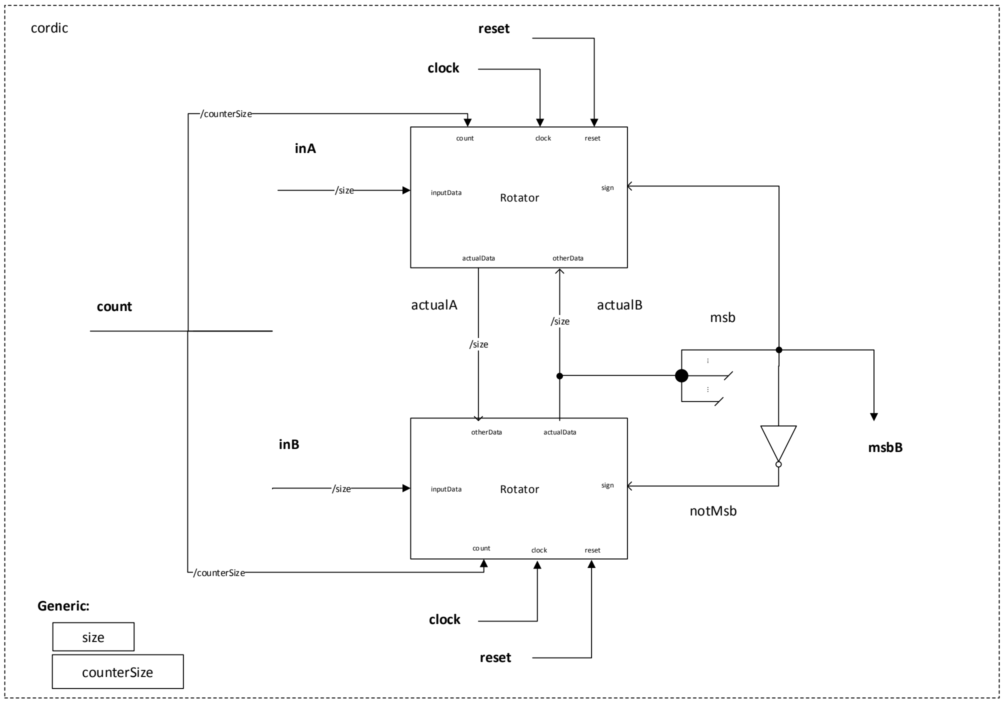


Fig. 22. Cordic Rotational Algorithm component

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

--------------------------------------------------------------------------------
-- CORDIC Algorithm
--
-- This component implements the CORDIC rotational algorithm for a vector of two
-- components A and B.
--
-- The reuslt is obtained connecting together accordignly two Single Dimension
-- Rotators.
--
--------------------------------------------------------------------------------

entity CORDIC is
    generic (
        size        : positive := 8;
        counterSize : positive := 8
    );
    port (
        clock       : in    std_ulogic;
        reset       : in    std_ulogic;
        inA         : in    std_ulogic_vector(size-1 downto 0);
        inB         : in    std_ulogic_vector(size-1 downto 0);
        count       : in    std_ulogic_vector(counterSize-1 downto 0);
        msbB        : out   std_ulogic
    );
end CORDIC;

architecture CORDIC_Arch of CORDIC is
    component Rotator is
        generic (
            size        : positive := 8;
            counterSize : positive := 8
            );
        port (
            clock       : in    std_ulogic;
            reset       : in    std_ulogic;
            sign        : in    std_ulogic;
            count       : in    std_ulogic_vector(counterSize-1 downto 0);
            inputData   : in    std_ulogic_vector(size-1 downto 0);
            otherData   : in    std_ulogic_vector(size-1 downto 0);
            actualData  : out   std_ulogic_vector(size-1 downto 0)
        );
    end component;

    signal msb          : std_ulogic;
    signal notMsb       : std_ulogic;
    signal actualA      : std_ulogic_vector(size-1 downto 0);
    signal actualB      : std_ulogic_vector(size-1 downto 0);
begin

    msb <= actualB(size-1);
    msbB <= msb;
    notMsb <= not actualB(size-1);

    -- The sign in input of rotatorA is the same as the MSB of B, so 1 for B < 0
    -- and 1 otherwise
    rotatorA: Rotator
        generic map (
            size        => size,
            counterSize => counterSize
        )
        port map (
            clock       => clock,
            reset       => reset,
            sign        => msb,
            count       => count,
            inputData   => inA,
            otherData   => actualB,
            actualData  => actualA
        );

    -- The sign in input of rotatorB is the negation of the MSB of B itself
    rotatorB: Rotator
        generic map (
            size        => size,
            counterSize => counterSize
        )
        port map (
            clock       => clock,
            reset       => reset,
            sign        => notMsb,
            count       => count,
            inputData   => inB,
            otherData   => actualA,
            actualData  => actualB
        );
end CORDIC_Arch;
```

N. Atan Accumulator with lookup table

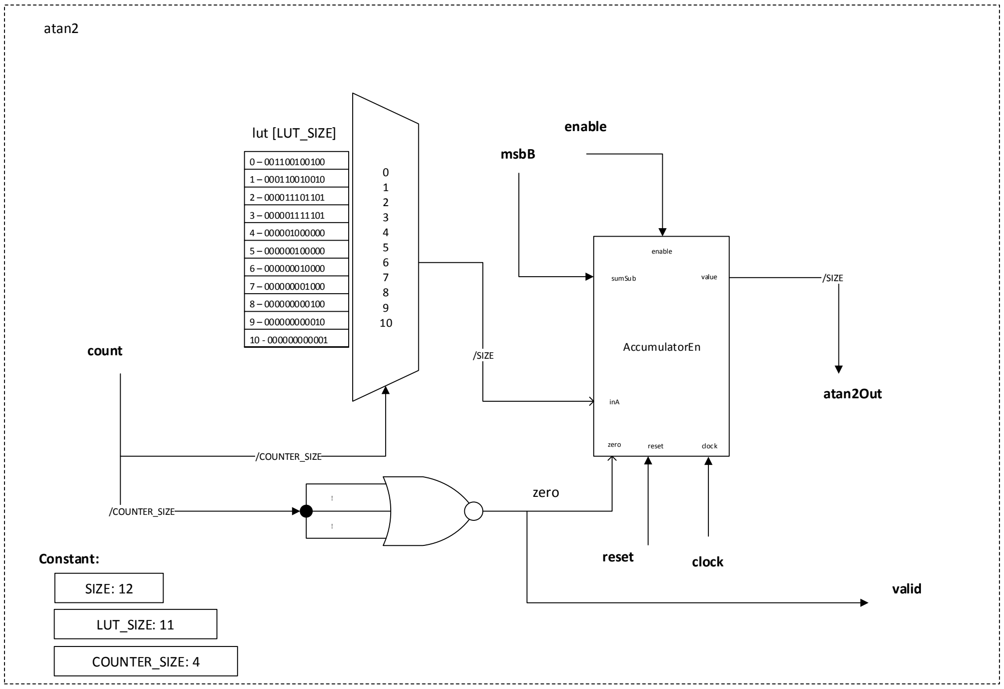


Fig. 23. Atan Accumulator with lookup table

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

--------------------------------------------------------------------------------
-- Atan2 calculator
--
-- This component takes as input the iteration count of the CORDIC algorithm and
-- the sign (so the MSB) of the B operand of the CORDIC algorithm and calculates
-- the corresponding Atan2 value by repeatedly summing known Atan2 values,
-- contained in a look up table (LUT).
--
-- The validity of the output is specified using an additional signal; the idea
-- behind this is that the output of the component is valid for only one clock
-- cycle, at the end of all the needed CORDIC iterations.
--
--------------------------------------------------------------------------------

entity Atan2 is
        port (
                clock       : in    std_ulogic;
                reset       : in    std_ulogic;
                msbB        : in    std_ulogic;
                enable      : in    std_ulogic;
                count       : in    std_ulogic_vector(4-1 downto 0);
                valid       : out   std_ulogic;
                atan2Out    : out   std_ulogic_vector(12-1 downto 0)
        );
end Atan2;

architecture Atan2_Arch of Atan2 is

        -- Due to lut generation and other factors, this component is not generic,
        -- hence it can be used only if the CORDIC component matches these constants
        constant SIZE           : positive := 12;
        constant COUNTER_SIZE   : positive := 4;
        constant LUT_SIZE       : positive := 11;

        type bin_array is array (natural range <>)
            of std_ulogic_vector(SIZE-1 downto 0);

        constant lut : bin_array := ("001100100100",
                                     "000110010010",
                                     "000011101101",
                                     "000001111101",
                                     "000001000000",
                                     "000000100000",
                                     "000000010000",
                                     "000000001000",
                                     "000000000100",
                                     "000000000010",
                                     "000000000001");

        component AccumulatorEn is
                generic (size : positive := 8);
                port (
                        clock   : in    std_ulogic;
                        reset   : in    std_ulogic;
                        zero    : in    std_ulogic; -- synchronous reset
                        enable  : in    std_ulogic; -- register enable
                        inA     : in    std_ulogic_vector(size-1 downto 0);
                        sumSub  : in    std_ulogic;
                        value   : out   std_ulogic_vector(size-1 downto 0)
                );
        end component;

        signal atanFromLut      : std_ulogic_vector(SIZE-1 downto 0);
        signal zero             : std_ulogic;
begin

        zero <= '1' when unsigned(count) = 0 else '0';
        atanFromLut <= lut(to_integer(unsigned(count)));
        valid <= zero;

        atanAccumulator : AccumulatorEn
                generic map (size => SIZE)
                port map (
                        clock   => clock,
                        reset   => reset,
                        zero    => zero,
                        enable  => enable,
                        inA      => atanFromLut,
                        sumSub   => msbB,
                        value    => atan2Out
                );
end Atan2_Arch;
```

O. Complete System

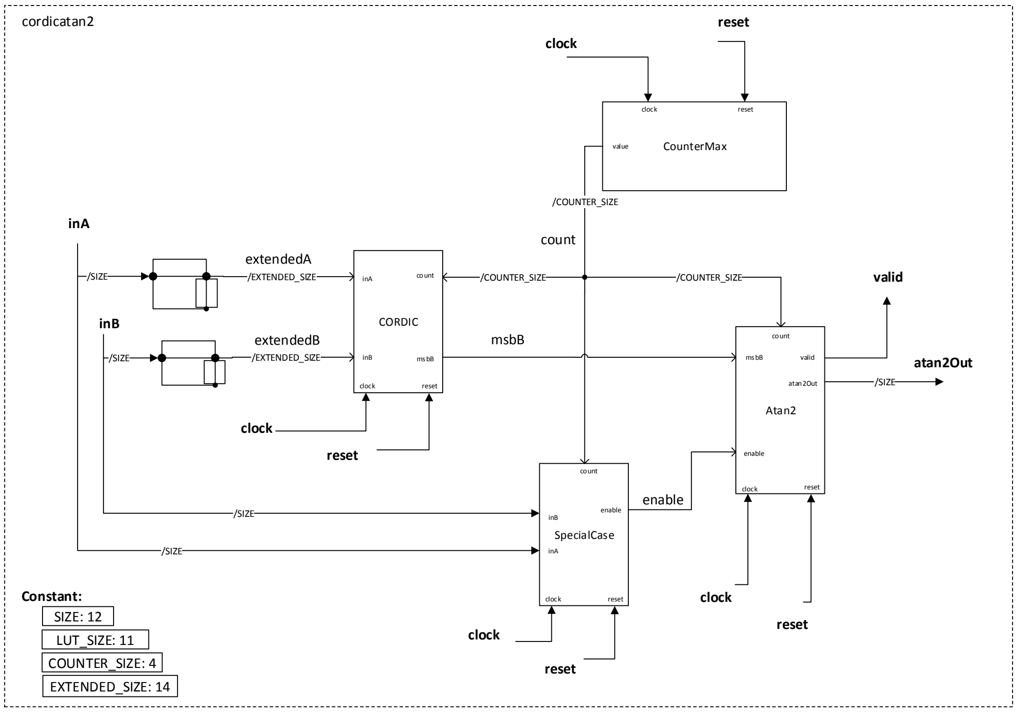


Fig. 24. Complete System

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

--------------------------------------------------------------------------------
-- CORDIC Atan2 Algorithm
--
-- This component implements the CORDIC algorithm for calculating the Atan2 of
-- two inputs A and B.
--
-- The component accepts an input each LUT_SIZE clock cycles.
--
-- The validity of the output is specified using an additional signal, the
-- "valid" one. It shall be ignored otherwise.
--
-- When the "valid" signal is high, the output of the system is the result of
-- the previously requested computation and a new values for A and B are sampled
-- for the next computation.
--
--------------------------------------------------------------------------------

entity CORDICAtan2 is
    port (
        clock        : in    std_ulogic;
        reset        : in    std_ulogic;
```

```vhdl
        inA         : in    std_ulogic_vector(12-1 downto 0);
        inB         : in    std_ulogic_vector(12-1 downto 0);
        valid       : out   std_ulogic;
        atan2Out    : out   std_ulogic_vector(12-1 downto 0)
    );

end CORDICAtan2;

architecture CORDICAtan2_Arch of CORDICAtan2 is

    constant SIZE           : positive := 12;
    constant EXTENDED_SIZE  : positive := SIZE+2;
    constant COUNTER_SIZE   : positive := 4;
    constant LUT_SIZE       : positive := 11;

    component Atan2 is
        port (
            clock       : in    std_ulogic;
            reset       : in    std_ulogic;
            msbB        : in    std_ulogic;
            enable      : in    std_ulogic;
            count       : in    std_ulogic_vector(4-1 downto 0);
            valid       : out   std_ulogic;
            atan2Out    : out   std_ulogic_vector(12-1 downto 0)
        );
    end component;

    component CORDIC is
        generic (
            size        : positive := 8;
            counterSize : positive := 8
        );
        port (
            clock       : in    std_ulogic;
            reset       : in    std_ulogic;
            inA         : in    std_ulogic_vector(size-1 downto 0);
            inB         : in    std_ulogic_vector(size-1 downto 0);
            count       : in    std_ulogic_vector(counterSize-1 downto 0);
            msbB        : out   std_ulogic

        );
    end component;

    component CounterWithMax is
        generic (
            size    : positive := 8;
            max     : positive := 16
        );
        port (
            clock   : in    std_ulogic;
            reset   : in    std_ulogic;
            value   : out   std_ulogic_vector(size-1 downto 0)
        );
    end component;

    component SpecialCase is
```

```
        generic (
            size        : positive := 8;
            counterSize : positive := 8
        );
        port (
            clock       : in    std_ulogic;
            reset       : in    std_ulogic;
            count       : in    std_ulogic_vector(counterSize-1 downto 0);
            inA         : in    std_ulogic_vector(size-1 downto 0);
            inB         : in    std_ulogic_vector(size-1 downto 0);
            enable      : out   std_ulogic
        );
    end component;

    signal msbB         : std_ulogic;
    signal enable       : std_ulogic;
    signal count        : std_ulogic_vector(COUNTER_SIZE-1 downto 0);

    signal extendedA    : std_ulogic_vector(EXTENDED_SIZE-1 downto 0);
    signal extendedB    : std_ulogic_vector(EXTENDED_SIZE-1 downto 0);
begin

    extendedA <= inA(size-1) & inA(size-1) & inA;
    extendedB <= inB(size-1) & inB(size-1) & inB;

    atan2Calculator : Atan2
        port map (
            clock       => clock,
            reset       => reset,
            msbB        => msbB,
            enable      => enable,
            count       => count,
            valid       => valid,
            atan2Out    => atan2Out
        );

    cordicInstance : CORDIC
        generic map (
            size        => EXTENDED_SIZE,
            counterSize => COUNTER_SIZE
        )
        port map (
            clock       => clock,
            reset       => reset,
            inA         => extendedA,
            inB         => extendedB,
            count       => count,
            msbB        => msbB
        );

    counter : CounterWithMax
        generic map (
            size    => COUNTER_SIZE,
            max     => LUT_SIZE
        )
        port map (
            clock   => clock,
            reset   => reset,
            value   => count
        );

    special : SpecialCase
        generic map (
            size        => SIZE,
            counterSize => COUNTER_SIZE
        )
        port map (
            clock       => clock,
            reset       => reset,
            count       => count,
            inA         => inA,
            inB         => inB,
            enable      => enable
        );

end CORDICAtan2_Arch;
```

P. System Test Bench

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

--------------------------------------------------------------------------------
-- System Test Bench
--
-- This test bench has been developed to test the correct system behavior with
-- respect to the MATLAB bit true model implementation.
--
-- Values of inputs and expected results have been randomically generated using
-- MATLAB (checking first that both the special cases were included in the test
-- set).
--
-- If one of the tests fails, an error is shown both in console and in the
-- simulation, since the signal test_correct is put to zero.
--
--------------------------------------------------------------------------------


entity System_TB is
end System_TB;

architecture System_TB_Arch of System_TB is

    component CORDICAtan2 is
        port (
            clock           : in    std_ulogic;
            reset           : in    std_ulogic;
            inA             : in    std_ulogic_vector(12-1 downto 0);
            inB             : in    std_ulogic_vector(12-1 downto 0);
            valid           : out   std_ulogic;
            atan2Out        : out   std_ulogic_vector(12-1 downto 0)
        );
    end component;

    constant SIZE           : positive := 12;
    constant LUT_SIZE       : positive := 11;
    constant TEST_SIZE      : positive := 1000;

    type word_array         is array (natural range <>) of
        std_ulogic_vector(SIZE-1 downto 0);

    signal test_inA         : std_ulogic_vector(SIZE-1 downto 0);
    signal test_inB         : std_ulogic_vector(SIZE-1 downto 0);
    signal test_output      : std_ulogic_vector(SIZE-1 downto 0);
    signal test_valid       : std_ulogic;

    signal test_expected    : std_ulogic_vector(SIZE-1 downto 0);

    signal test_correct     : std_ulogic := '1';
    signal test_reset       : std_ulogic := '1';
    signal test_clock       : std_ulogic := '0';
    signal test_end         : std_ulogic := '0';

    -- TODO: fill these
    constant test_inputA        : word_array :=
        ("110001100111","000101111001","011101100100","111010110111", ... );
    constant test_inputB        : word_array :=
        ("001011110100","000111000000","000110110100","100110010100", ... );
    constant expected_output    : word_array :=
        ("010011100111","000110111111","000001110101","110001110101", ... );

begin

    cordicAtan2UnderTest : CORDICAtan2
        port map (
            clock       => test_clock,
            reset       => test_reset,
            inA         => test_inA,
            inB         => test_inB,
            valid       => test_valid,
            atan2Out    => test_output
        );

    test_clock <= (not test_end) and (not test_clock) after 50ns;

    -- stimuli
    driveProcess : process
    begin
        wait for 60ns;

        test_reset <= '0';

        for i in 0 to TEST_SIZE-1 loop
            test_inA <= test_inputA(i);
            test_inB <= test_inputB(i);

            for j in 0 to LUT_SIZE-1 loop
                wait until rising_edge(test_clock);
            end loop;

            test_expected <= expected_output(i);

            wait for 25ns;

            if test_valid = '1' and test_output = expected_output(i) then
                test_correct <= '1';
            else
                test_correct <= '0';
            end if;

            assert (test_valid = '1' and test_output = expected_output(i))
            report "ERROR: detected for index i = " & integer'image(i)
                & "; A = " & integer'image(to_integer(signed(test_inA)))
                & "; B = " & integer'image(to_integer(signed(test_inB)))
                & "."
            severity error;

        end loop;

        wait for 100ns;

        test_end <= '1';

        wait;
    end process;

end System_TB_Arch;
```