Journal of
Low Power Electronics and Applications

MDPI

Article

CORDIC Hardware Acceleration Using DMA-Based ISA Extension

Erez Manor [1], Avrech Ben-David [2] and Shlomo Greenberg [1,3,*]

[1] Department of Electrical and Computer Engineering, Ben Gurion University, Beer-Sheva 8410501, Israel; erezmano@post.bgu.ac.il
[2] Electrical & Computer Engineering Faculty, Technion Institute, Haifa 3200003, Israel; avrech@campus.technion.ac.il
[3] Department of Electrical Engineering, Sami Shamoon College of Engineering, Beer-Sheva 8410802, Israel
* Correspondence: shlomo.greenberg@gmail.com or shlomog@bgu.ac.il or shlomgr@sce.ac.il

Abstract: The use of RISC-based embedded processors aimed at low cost and low power is becoming an increasingly popular ecosystem for both hardware and software development. High-performance yet low-power embedded processors may be attained via the use of hardware acceleration and Instruction Set Architecture (ISA) extension. Recent publications of AI have demonstrated the use of Coordinate Rotation Digital Computer (CORDIC) as a dedicated low-power solution for solving nonlinear equations applied to Neural Networks (NN). This paper proposes ISA extension to support floating-point CORDIC, providing efficient hardware acceleration for mathematical functions. A new DMA-based ISA extension approach integrated with a pipeline CORDIC accelerator is proposed. The CORDIC ISA extension is directly interfaced with a standard processor data path, allowing efficient implementation of new trigonometric ALU-based custom instructions. The proposed DMA-based CORDIC accelerator can also be used to perform repeated array calculations, offering a significant speedup over software implementations. The proposed accelerator is evaluated on Intel Cyclone-IV FPGA as an extension to Nios processor. Experimental results show a significant speedup of over three orders of magnitude compared with software implementation, while applied to trigonometric arrays, and outperforms the existing commercial CORDIC hardware accelerator.

Keywords: RISC; CORDIC; ISA-Extension; low-power; hardware accelerator; FPGA; neural networks

Citation: Manor, E.; Ben-David, A.; Greenberg, S. CORDIC Hardware Acceleration Using DMA-Based ISA Extension. *J. Low Power Electron. Appl.* **2022**, *12*, 4. https://doi.org/10.3390/jlpea12010004

Academic Editor: Aatmesh Shrivastava

Received: 10 November 2021
Accepted: 14 January 2022
Published: 15 January 2022

1. Introduction

In the last years, the complexity of embedded platform, such as Internet of Things (IoT) devices, has been increasing steadily with the conflicting requirements for high performance and real-time capabilities versus minimal amount of power and size. Typically, while using a general-purpose RISC processor it is difficult to face all the application-specific requirements in real-time. Extending the RISC ISA by using a specific custom instruction allows flexible and efficient implementation in hardware. The custom ISA extension allows for precise implementation of the instruction groups that the application needs as optimized hardware, maximizing performance while minimizing power. Examples for such processors that support custom ISA extension can be found in Tensilica Xtensa [1], Intel Nios [2], and RISC-V with its open-source ISA extensions [3]. As part of the current trend of using custom instructions in low-power processors, ARM Cortex-M33 and M55 also support the possibility of custom instructions [4]. X. Wang et al. [5] presented efficient usage of custom instructions targeted at neural network processing. They introduced an energy-efficient neural network running on the ARM Cortex-M series and the novel RISC-V-based Parallel Ultra-Low-Power (PULP) platform.

The Coordinate Rotation Digital Computer (CORDIC) algorithm is a well-known algorithm used for computing a wide range of mathematical functions and is applied to computer vision and DSP that require heavy computational functions [6]. CORDIC is also

used as an efficient NN computation engine for implementing multiply-and-accumulate (MAC) and nonlinear neuron activation function [7,8]. Efficient implementations of the CORDIC algorithm to calculate the differential equations of the neurons in Spiking Neural Network (SNN) are presented in [9,10].

This paper proposes a floating-point CORDIC accelerator aimed at extending the arithmetic logic unit (ALU) instruction set for a range of extensive computation transcendental functions, particularly the trigonometric family. Accelerating these functions to perform repeated calculations on an array of values greatly contributes to meeting real-time constraints. An analysis of several mathematical applications using the synchronous data flow graph (SDF) shows that most of the heavy computations can be mapped into hardware using CORDIC as ISA extensions allowing significant speedup computation while still keeping low power [11].

The CORDIC ISA extensions are evaluated on the Nios-II/f Intel processor, which is a general-purpose RISC processor core, implemented as a soft core in the FPGA. The Nios core has built-in feature supporting custom instruction (CI) logic in the arithmetic logic unit (ALU) to form ISA extensions.

The rest of this paper is organized as follows: Section 2 outlines the Nios-II processor and the CORDIC algorithm, Section 3 presents a brief overview of related works, while Section 4 provides the main methodology of this study. Finally, Section 5 shows implementation and experimental results, and Section 6 concludes the paper.

2. Background

This section provides a background of the Nios-II ISA extension feature and the usage of hardware custom instruction to accelerate time-critical software algorithms. A brief introduction to the CORDIC algorithm and Altera hardware CORDIC module aimed at computing a diverse range of trigonometric functions is given below.

2.1. Nios-II Custom Instruction

There are three kinds of custom instruction: combinational, multicycle, and parameterized mode. In a combinational custom instruction mode, an instruction is completed in a single clock cycle. This mode requires a result output port and may have two optional input ports (*dataa* and *datab*). The custom instructions take values from up to two source registers and optionally write back a result to a destination register (Figure 1). Multicycle custom instructions consist of a logic block that requires two or more clock cycles to complete an operation, in either a fixed or variable number of clock cycles. The *start* and *done* ports participate in a handshaking scheme to determine when the custom instruction execution is complete. The multicycle custom instruction allows to add an interface to communicate with logic outside of the processor's data path. An extended custom instruction allows a single custom logic block to implement several different operations. Extended custom instructions use an extension index *n* to specify which operation the logic block performs. Ports *a*, *b*, and *c* specify the internal registers from which to read or to which to write. This parameterization option can be enabled in both combinatorial and multicycle custom instructions.

2.2. Altera Hardware CORDIC Unit

The Coordinate Rotation Digital Computer (CORDIC) algorithm [12] is a well-known shift-add iterative algorithm, for computing a wide range of functions: trigonometric, hyperbolic, linear, and logarithmic. The CORDIC has a simple and hardware-efficient mapping that uses simple shift, add, subtract, and table look-up operations, instead of using Calculus-based methods such as polynomial or rational functional approximation. The CORDIC core provides a solution to implement and solve a diverse range of trigonometric functions (e.g., sine and cosine) faster than software functions [13], which use the taylor approximation or emulation of the CORDIC algorithm to compute these functions, thus saving significant processing time and power.

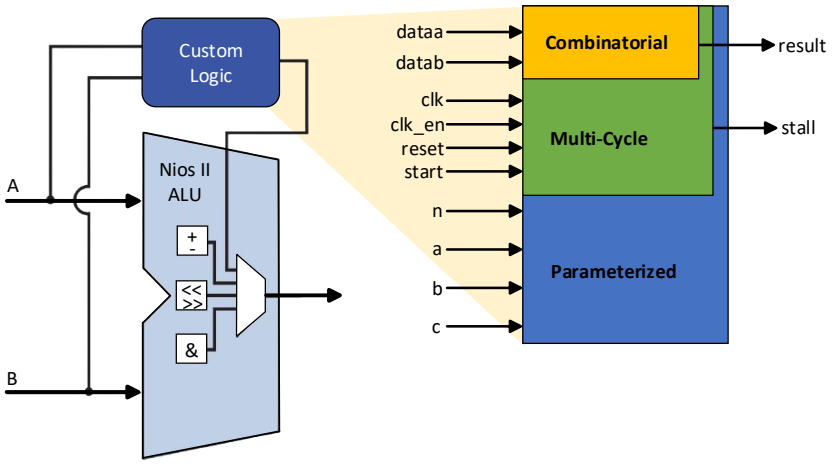

Figure 1. Custom instruction architecture. Intel's website [2].

The CORDIC can be employed in two different modes: rotation mode and vectoring mode. In the rotation mode, the coordinate components of a vector and a rotation angle are given, and the coordinates of the original vector, after rotation, are computed. In the vectoring mode, the coordinate components of a vector are given and the magnitude and angular argument of the original vector are computed.

The equations of the CORDIC algorithm are given in Equations (1)–(3). Equations (1) and (2) depict the iterative equations for the rotation mode, where i stands for the iteration index, N is the number of iterations, and the rotation direction is $d_i = sign(z_i)$ [14]. The coordinates of the rotated vector are given in Equation (2), where K_n is the scale factor.

$$\begin{aligned} x_{i+1} &= x_i - y_i * d_i * 2^{-i} \\ y_{i+1} &= y_i - x_i * d_i * 2^{-i} \\ z_{i+1} &= z_i - d_i * atan(2^{-i}) \end{aligned} \tag{1}$$

$$\begin{aligned} x_N &= K_N(x_0 cosz_0 - y_0 sinz_0) \\ y_N &= K_N(y_0 cosz_0 + x_0 sinz_0) \\ z_N &= 0 \end{aligned} \tag{2}$$

Equation (3) describes the unified CORDIC algorithm [15]. The algorithm merges both rotation and vectoring modes, as well as the phase type (circular, linear, and hyperbolic), where μ determines the sign bit and $e(i)$ selects the rotation function.

$$\begin{aligned} x_{i+1} &= x_i - \mu\sigma_i * y_i * 2^{-i} \\ y_{i+1} &= y_i - \sigma_i * x_i * 2^{-i} \\ z_{i+1} &= z_i - \sigma_i * e(i) \end{aligned} \tag{3}$$

A simplified top-level block diagram of the Altera CORDIC block is shown in Figure 2. The x, y, and z inputs and outputs are twos-complement signed numbers, while *mode* enforces vectoring mode ($mode = 0$) or rotation mode ($mode = 1$) The bit widths and number of iterations are parameterizable. This implementation is based on ALTERA_CORDIC IP Core. Table 1 presents a list of the 12 supported mathematical functions. When using the CORDIC core, the user must select the required function.

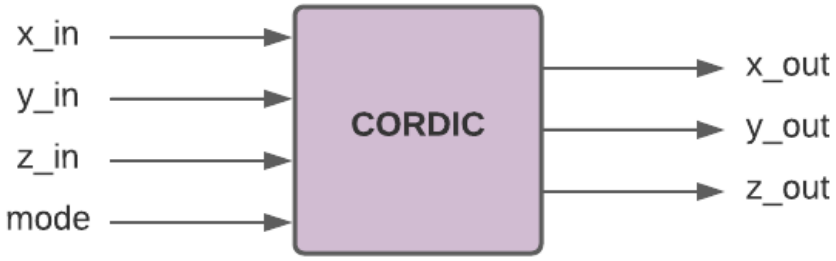

Figure 2. The CORDIC Component Port Map.

Table 1. CORDIC Available Functions.

Function	Mode	x_in	y_in	z_in	Result	Phase Type
A*cos(B)	1	A	0	B	x_out	Circular
A*sin(B)	1	A	0	B	y_out	Circular
asin(A)	1	Unit	0	A	z_out	Circular
atan(B/A)	0	A	B	0	z_out	Circular
Mag(A,B)	0	A	B	0	x_out	Linear
C-B/A	0	A	B	C	z_out	Hyperbolic
A*cosh(B)	1	A	0	B	x_out	Hyperbolic
A*sinh(B)	1	0	A	B	x_out	Hyperbolic
A*exp(B)	1	A	A	B	y_out	Hyperbolic
atanh(B/A)	0	A	B	0	z_out	Hyperbolic
0.5*ln(A)	0	$A+1$	$A-1$	0	z_out	Hyperbolic
sqrt(A)	0	$A+\frac{1}{4}$	$A-\frac{1}{4}$	0	x_out	Hyperbolic

The CORDIC algorithm can be implemented in two different manners: (a) basic sequential structure or (b) pipelined parallel structure [16]. The sequential implementation assumes that only one iteration is performed per clock cycle. Figure 3 depicts the block schematic of the main hardware elements of the CORDIC data path. The data path consists of n-bit adders/subtractors, sign extending shifters, and a lookup table (LUT) used to store the phase constants. The pipelined implementation is based on a parallel structure consisting of N cascaded calculation blocks producing a new result in each clock cycle (with a latency of N cycles).

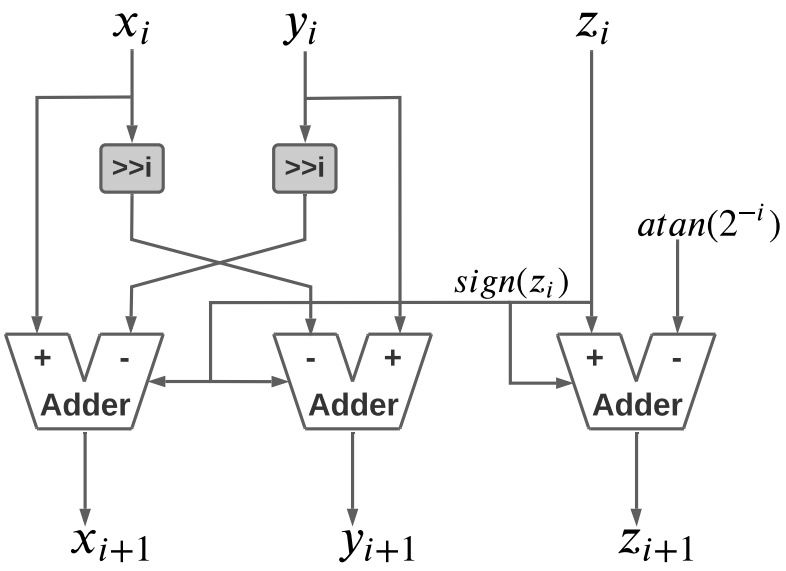

Figure 3. CORDIC calculation block.

3. Related Work

Several approaches for hardware computation acceleration of trigonometric functions have been presented [17,18]. The well-known CORDIC algorithm offers a variable precision approach for varied transcendental functions. R. Andraka et al. [19] described in detail how to drive the CORDIC core to evaluate trigonometric, exponential, logarithmic, and other functions in fixed-point representation.

Previous work have shown the benefit of using CORDIC for AI algorithms in terms of power and area. G. Raut et al. [7,8] presented an optimized CORDIC-based architecture to enable computations required in the neural networks. The CORDIC engine was employed in hyperbolic rotation mode to realize both tanh and sigmoid activation functions. M. Heidarpur et al. [9] presented a spiking neuron model (Izhikevich neuron) utilizing spike timing-dependent plasticity (STDP) learning based on the CORDIC algorithms to calculate Izhikevich neuron differential equations. X. Hao et al. [10] presented an implementation of a cerebellar Purkinje spiking neuron model using the CORDIC algorithm.

Accelerating the CORDIC algorithm for software–hardware acceleration was addressed by E. Liventsev et al. [18]. Liventsev presented several approaches for CORDIC core extension of MIPS FPGA processor instruction set for the acceleration of fixed-point calculations. The proposed accelerator contains two CORDIC modules and a fast multiplication core. A Uniform Driver Interface (UDI) is used to decode custom instructions and access the processor register file. A speedup factor of 6.24 compared with software implementation is demonstrated for matrix transform operations. The use of radix-4 CORDIC accelerator was suggested by S. Nolting et al. [20] illustrating that the processing performance can be increased by a factor of 28.1.

A floating-point CORDIC coprocessor interfacing with a Nios processor was presented by [21]; the authors suggested using an argument reduction algorithm as the preprocessing and iterative CORDIC calculation followed by scaling and normalization, and demonstrated a speedup factor of around 100. J. Zhou et al. [22] presented a compact SAR processor composed of four FFT processing elements and a floating-point pipeline hybrid-mode CORDIC coprocessor implemented on FPGA. A reconfigurable pipeline CORDIC architecture that can be configured to operate in different modes to achieve single precision floating-point arithmetic operations was presented in [23]. The CORDIC-based floating-point arithmetic processor has been efficiently implemented on Xilinx FPGA. A pipeline-based implementation of a CORDIC accelerator was presented by K. Nguyen et al. [24]. The CORDIC was coupled to a RISC-V microprocessor execution path, performing 414 times faster than software.

A. Buzdar et al. [25] presented a novel CORDIC accelerator integrated with an embedded processor data path to improve processor performance in terms of execution time and energy efficiency. The authors showed that a modified CORDIC data path, using custom instruction, is 14.5 times more cycle efficient than a data path lacking a modified CORDIC accelerator. The novelty of this design is that it takes a single iteration to compute sine and cosine compared with the standard CORDIC algorithm, which requires N iterations. This provides effective usage of the accelerator in which a series of values of sine and cosine are required to be computed.

F. Sun et al. [26] proposed a hybrid approach integrating both coprocessors and custom instructions as two different forms of hardware acceleration that can be applicable at different levels of granularity and offer differing trade-offs. This methodology builds upon the basic observations that coprocessors are usually good for coarse-grained tasks and require minimal intervention or support from the processor, while custom instructions are usually suited to fine-grained operations that are best integrated into a processor pipeline.

Contrarily to other existing CORDIC accelerators, we propose a DMA-based ISA extension integrated with a pipeline CORDIC accelerator. The CORDIC ISA extension is directly interfaced with a standard processor data path allowing efficient implementation of new trigonometric ALU-based custom instructions. The proposed DMA-based CORDIC accelerator can also be used to perform repeated calculations on an array of values offering

significant speedup with respect to doing the calculations in software, which is very effective for calculating a long array of output layers in NN applications.

4. Methodology

This paper proposes and evaluates a floating-point CORDIC accelerator aimed at extending the ALU instruction set to support a range of trigonometric functions using multicycle custom instruction with 15-bit CORDIC. Dedicated pre and post hardware processing units are used to convert the standard IEEE-754 floating-point input format into a fixed-point format and vice versa. We evaluate three different hardware accelerator implementations for the proposed CORDIC custom instruction: (a) basic approach, (b) pipelined mode, and (c) DMA-based approach.

4.1. ISA Extension Basic Implementation

The CORDIC custom instruction implements the C standard library trigonometric functions. The custom instruction interface has 2 input operands and one return value. The basic function call operation will stall the software until the results are available, thus, driving each software variable latency is equal to the latency of the hardware core. The CORDIC core in this implementation is not pipelined since it is not needed, thus saving logic resources. Figure 4 schematically describes the CORDIC custom instruction block diagram.

The CORDIC custom instruction data path supports IEEE-754 floating-point conversion with variable precision. The CORDIC has a configurable accuracy with linear complexity for the latency and polynomial for the area. The CORDIC multiplies the output amplitude with an inherent gain factor, which should be compensated somewhere in the algorithm or ignored (if amplitude is not an issue). The CORDIC core resolution is determined by the input/output signals' width and the number of effective iterations. The number of effective iterations is limited by phase word width. The flexibility of the CORDIC core enables us to configure a nonstandard word width, for example, if only few additional LSB bits are required to satisfy the resolution requirements.

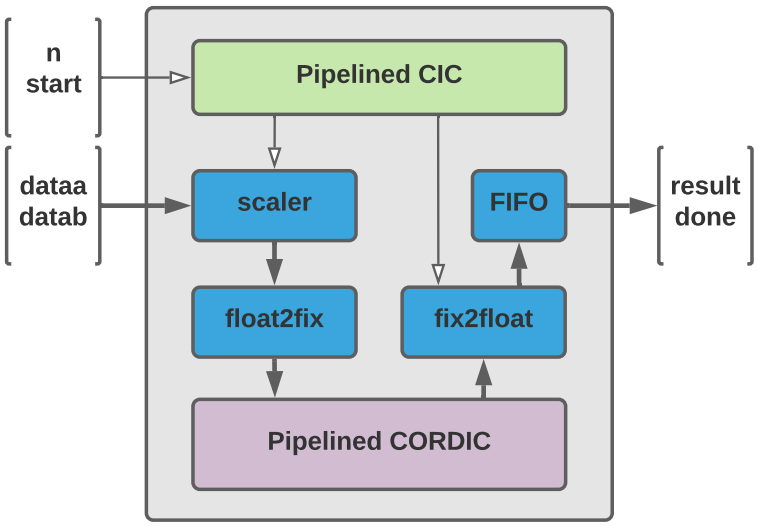

Figure 4. CORDIC Custom Instruction Basic Block Diagram.

The CORDIC operands are fixed-point words, with a predefined width. The z_in word—a twos-complement signed integer—represents the continuous segment $[-\pi, \pi)$ with a resolution of $\pi \times 2^{-B}$, where B is the integer width. The scaler block scales the phase periodical range $[-\pi, \pi)$ to the integer width, multiplying z_in by $2^{B-1}/\pi$. The scaler can also be configured to compensate the CORDIC gain factor. The z_in word width also

determines the latency of the CORDIC unit. The fixed-point conversion range of x_in and y_in, using the float2fix and fix2float blocks, cover the range of the input vector. In this work, we used $Q2.13$ binary fixed-point number format with 2 integer bits and 13 fractional bits for both (x_in, y_in) to represent the real segment $[-2, 2-2^{-13}]$ with quantization of 2^{-13}. This representation enables common fractional part for using the ALTERA_CORDIC IP Core that dictates a Q2.13 presentation for the rotation, and Q3.13 for vectoring.

The scaling operation adds 5 cycles to the latency, the float2fix conversion adds another 6 cycles latency in each direction. The total latency of this component is 32 cycles. Every additional bit to the phase word will add 1 cycle of latency to this chain. For a fixed-point-based application, such as typical DSP applications, the extra cost due to floating-point conversions can be saved.

A custom instruction control (CIC) block handles special instructions, for example, the rotate function, which perform both sine and cosine function with the same input arguments, to produce a single calculation for the (x_out, y_out) pair; so, although this couple is computed by two sequential processing, the function call occurs only once, thus saving valuable clock cycles. Table 2 describes the set of CORDIC functions that have been chosen for acceleration using this ISA extension. The *sigmoid* function is realized using the *tanh* CORDIC built-in function as shown in Equation (4).

$$sigmoid(z) = \frac{1}{1+e^{-z}} = \frac{1+tanh(z/2)}{2} \tag{4}$$

The CORDIC core is configured with 15 bit for phase width, adding 15 cycles to the latency of the processing chain. The CORDIC custom instruction latency extends 32 clock cycles, including the conversion and scaling operations. We tested the Nios using Tightly Coupled Memory (TCM). The Nios requires approximately 22 clock cycles per operation call. Ultimately, the throughput of cosine operation is one per 54 clock cycles. This is 90 times faster than the C Standard Library Function (using floating-point hardware component).

4.2. ISA Extension Using Pipelined Approach

The proposed pipeline mode is characterized by a pipeline flow that enables providing the next input arguments concurrently with the calculation of the current arguments. Therefore, new arguments can be fetched in parallel with the current calculation. We suggest implementing a pipelined CORDIC architecture within the custom instruction module. The pipeline mode can significantly speed up calculations of arrays of arguments compared with the basic CORDIC ISA extension of the Nios core, which calculates a single argument.

The implementation of the pipelined CORDIC custom instruction is based on the basic architecture depicted in Figure 4. The implementation of the CORDIC accelerator is based on a unique pipeline architecture, and an additional FIFO is required for temporary storing the resulted arguments until they are fetched by the CPU. All the CORDIC functions described in Table 2 are supported in this pipeline mode. The CIC unit includes support for pipeline control and synchronization. The average execution time for computing a sine array is 16 cycles per array element. This pipelined implementation outperforms the basic CORDIC by a factor of about $\times 3$. However, processing of larger arrays that are located in an external memory results in approximately 45 cycles per array element due to the high memory latency. Moreover, the software loop overhead of a single-issue RISC core [27] requires an additional five cycles to update the data pointers, although this can be improved by a zero overhead loop controller [28].

Table 2. CORDIC Direct Addressing Custom Instruction.

Macro	C Equivalent Function
CORDIC_COS(phase)	cos(phase)
CORDIC_SIN(phase)	sin(phase)
CORDIC_AMPCOS(amp,phase)	amp*cos(phase)
CORDIC_AMPSIN(amp,phase)	amp*sin(phase)
CORDIC_ROTATEX(x,y,phase)	x*cos(phase) − y*sin(phase)
CORDIC_ROTATEY(x,y,phase)	y*cos(phase) + x*sin(phase)
CORDIC_SIGMOID(x)	sigmoid(x)

4.3. ISA Extension Using DMA Approach

We propose to integrate a DMA controller within the custom instruction module for efficient external data access. The DMA controller handles all the data transfers between memory and the CORDIC using two DMA channels (READ and WRITE). The CORDIC generates a DMA request on its read channel, upon which the DMA controller fetches an argument from memory. When the CORDIC finishes the first calculation, it generates a DMA request on the write channel.

Figure 5 depicts the block diagram of the DMA-based CORDIC custom instruction, which consists of three main components: (a) DMA CIC—This unit implements the interface to the CPU via the Altera Custom Instruction Slave (CIS) interface. The CIC is responsible for the DMA configuration with the required transfer parameters (i.e., address pointers, input/output array size, and operation type). The CIC is connected to the AVM through a custom status registers (CSR) standard interface. The CIC monitors the DMA operation and returns done to the CPU when the transaction finished. (b) Pipelined CORDIC Data Path—This acceleration unit adapts the pipeline data path as described in Section 4.2, and performs the CORDIC trigonometric functions. (c) Avalon-based Interface—The Avalon-based interface has multiple DMA channels using the Avalon Memory-Mapped interface (AVM) to access external memory device. The Avalon stream interface handles the data input (ASI) and output (ASO) to and from the pipelined CORDIC. The DMA-based custom instruction approach can perform functions based on both direct and indirect addressing (using pointers).

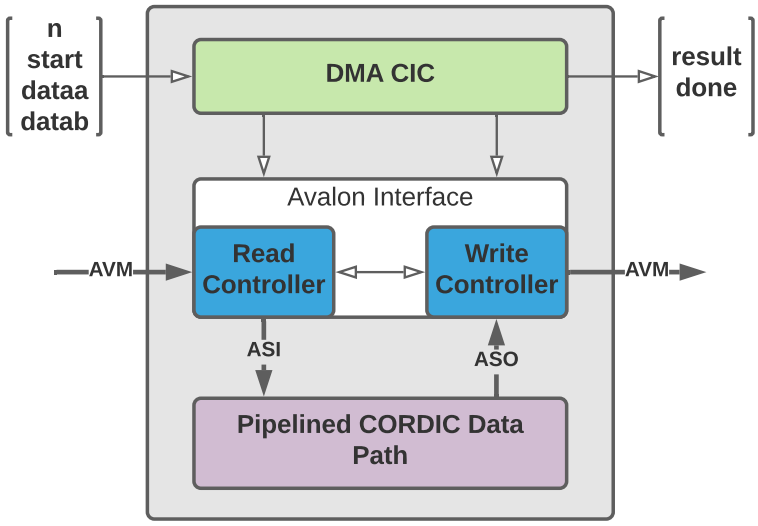

Figure 5. DMA-Based CORDIC Custom Instruction.

The functions described in Table 3 use indirect addressing mode, thus transferring a pointer through the ISA I/F. The DMA CORDIC accesses the arguments using this pointer, or by direct addressing as in Table 2. The DMA module is customized to handle 3 read and

2 write addresses with a relatively small latency. The DMA design is scalable and can be extended and adapted to other hardware acceleration using ISA extensions; for example, the unique DMA_CORDIC_NEURON function implements the neuron computational unit including the MAC operation and a sigmoid activation function, as shown in Figure 6. Performance simulations show that the DMA-based custom instruction utilizes the full memory bandwidth. The DMA approach enables a throughput of two cycles per array element for sine operation, and five cycles for rotate and a neuron operation.

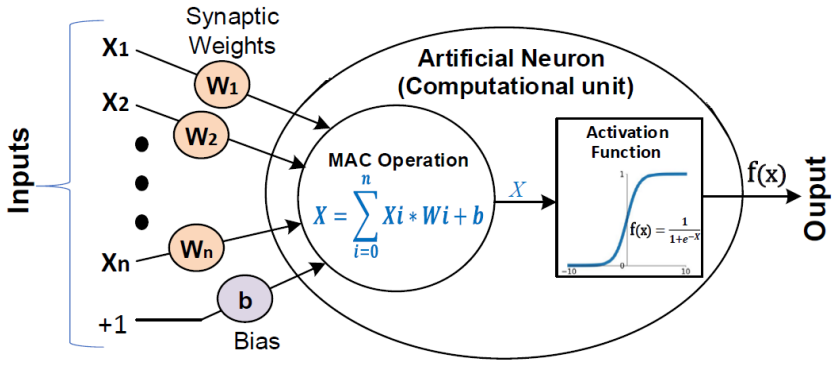

Figure 6. Single neuron with multiple inputs followed by activation function [7].

Table 3. CORDIC Indirect Addressing Custom Instruction.

Macro	Functionality
DMA_CORDIC_COS(z_in*,x_out*,size)	x_out = cos(z_in)
DMA_CORDIC_SIN(z_in*,y_out*,size)	y_out = sin(z_in)
DMA_CORDIC_AMPCOS (z_in*,x_in*,x_out*,size)	x_out = x_in*cos(z_in)
DMA_CORDIC_AMPSIN (z_in*,x_in*,y_out*,size)	y_out = x_in*sin(z_in)
DMA_CORDIC_ROTATEXY (z_in*,x_in*,y_in*,x_out*,y_out*,size)	x_out = x_in*cos(z_in) − y_in*sin(z_in) y_out = y_in*cos(z_in) + x_in*sin(z_in)
DMA_CORDIC_NEURON (x_in*,y_in*,y_out*,size)	y_out = sigmoid($\sum_{i=1}^{n} x\_in_i * y\_in_i$)

5. Results

Performance evaluation of the proposed ISA extension approach using custom instruction design was carried out for the CORDIC trigonometric functions. The three suggested ISA accelerators approaches was tested and compared with a commercial CORDIC hardware accelerator (STM32). To accurately evaluate the program execution time using the proposed hardware custom instruction module, the ALTERA performance counter profiler was used. The seven CORDIC functions described in Table 2 were examined with and without ISA extension acceleration.

Figure 7 depicts the average execution time for *sine* and *rotate* functions, for the three ISA hardware accelerations, and for two software implementations using math library and CORDIC emulation. Results show that the DMA approach outperforms the pipeline and the basic ISA extension approaches, allowing efficient memory interfacing. For example, for the sine function, the DMA requires average execution time of only 2 clock cycles (for array size of 2048 elements) compared with 54 and 21 clock cycles for the basic and pipeline implementation, respectively. This demonstrates a speedup factor of 112, 289, and 3037 for the basic, pipeline, and DMA implementations, respectively, compared with the basic math library software implementation. A significant speedup of up to a factor of 466 is also demonstrated compared with the CORDIC software emulator. It can be seen that the superiority of the DMA-based ISA accelerator is more significant as the array size increases. Similar results are demonstrated for the *rotate* function. The average

execution time using DMA is only 5 clock cycles compared with 122 clock cycles for the basic implementation. A speedup factor of 5148 is demonstrated compared with the math library software implementation. The *sigmoid* and *neuron* operations shows similar execution time to the *rotate* function; although a similar work [21] showed slightly better acceleration for the pipeline approach, our proposed DMA approach outperforms the related method by a factor of about $\times 10$.

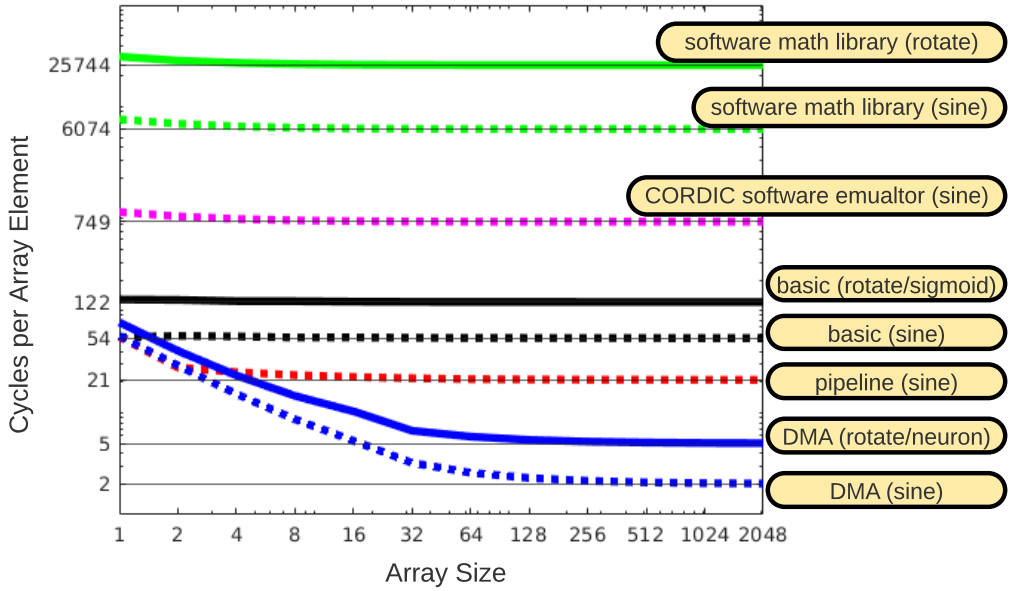

Figure 7. Comparison of sine and rotate Average Execution Time.

The proposed DMA-based ISA acceleration was also compared to a commercial STM32G4 MCU [29] integrating a CORDIC hardware accelerator with DMA channels. Our proposed CORDIC acceleration outperformed the STM32G4 CORDIC, demonstrating 54 cycles for a single-element sine calculation and 2 cycles on average for large sine arrays (over 128 elements), compared with 79 cycles and 8 cycles for the STM CORDIC, respectively. The main reason for the performance improvement compared with the STM32G4 is due to our efficient pipeline implementation of the custom instruction acceleration. The STM32G4 CORDIC can process a value in 6 clock cycles results in an average process time of 8 cycles per value (including one cycle for write contents into the CORDIC and another cycle to transfer the result into a data register), while the proposed DMA-based pipeline can process a value in only 2 clock cycles.

Table 4 describes the detailed hardware cost for each of the following system implementations: Nios with FPU, Basic implementation, pipeline, and DMA approaches. Both the maximum frequency and resource utilization (in terms of required memory, LUT, registers, and DSP) are demonstrated using Altera Cyclone-IV FPGA. The three proposed ISA accelerators implementation achieve a high synthesizable frequency of around 150 MHz compared with the 115 MHz achieved with Nios FPU combinational custom instruction. The DMA custom instruction power estimation is 35 mW for 115 MHz and doubles at 150 MHz. The extra logic elements required for the implementation of the proposed ISA accelerators are minor in regards to the demonstrated speedup, and power overhead should be saved by the reduced Nios toggle rate.

Table 4. Synthesis Results on Altera FPGA device.

	LUTs	Registers	Memory	DSP	Speed (MHz)
Component					
Nios-II/f Core	1290	376	10,240	6	-
FPU Accelerator	415	198	144	7	-
CORDIC	1099	170	0	0	-
scaler	126	148	0	0	-
float2fix	261	257	0	0	-
fix2float	192	238	0	0	-
Basic CIC	634	1039	768	4	-
Pipelined CIC	658	1071	1280	6	-
DMA CIC (+Avalon I/F)	1790	2382	4096	6	-
System					
Nios with FPU	1819	717	10,453	13	114
Basic Implementation	4933	2463	12,032	10	145
Pipelined Approach	4955	2495	12,544	12	143
DMA Approach	6089	3796	15,360	12	146

6. Summary and Conclusions

This paper demonstrates an efficient hardware implementation of CORDIC acceleration using the ISA extension approach for a RISC architecture. The proposed CORDIC ISA extension can directly interface any standard processor data path, allowing efficient implementation of new trigonometric ALU-based custom instructions. Contrarily to other existing CORDIC accelerators, we propose a DMA-based ISA extension integrated with a pipeline CORDIC accelerator. Performance evaluation of the proposed DMA-based ISA extension approach demonstrates significant speedup while keeping low power. The proposed ISA acceleration also outperforms some existing commercial CORDIC hardware accelerators. A speedup of three orders of magnitude is presented compared with software implementation. The proposed CORDIC accelerator can also be efficiently applied to DSP and Deep Neural Network applications requiring the support of large data arrays and repeated calculations.

Author Contributions: All authors (E.M., A.B.-D. and S.G.) contributed equally to this work. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Not applicable.

Acknowledgments: This work was supported by the High-tech scholarship award and the Israel Innovation Authority GenPro Consortium.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Cadence. Tensilica Customizable Processors. Available online: https://ip.cadence.com (accessed on 9 November 2021).
2. Intel. Nios II Processors. Available online: https://www.intel.com (accessed on 9 November 2021).
3. Davide Schiavone, P.; Conti, F.; Rossi, D.; Gautschi, M.; Pullini, A.; Flamand, E.; Benini, L. Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications. In Proceedings of the 2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS), Thessaloniki, Greece, 25–27 September 2017; pp. 1–8. [CrossRef]
4. Joseph Yiu, A. Innovate by Customized Instructions, but without Fragmenting the Ecosystem. Available online: https://armkeil.blob.core.windows.net/developer/Files/pdf/white-paper/arm-custom-instructions-without-fragmentation-whitepaper.pdf (accessed on 9 November 2021).
5. Wang, X.; Magno, M.; Cavigelli, L.; Benini, L. FANN-on-MCU: An Open-Source Toolkit for Energy-Efficient Neural Network Inference at the Edge of the Internet of Things. *arXiv* **2019**, arXiv:1911.03314.

6. Sharma, N.K.; Rathore, S.; Khan, M.R. A Comparative Analysis on Coordinate Rotation Digital Computer (CORDIC) Algorithm and Its use on Computer Vision Technology. In Proceedings of the 2020 First International Conference on Power, Control and Computing Technologies (ICPC2T), Raipur, India, 3–5 January 2020; pp. 106–110. [CrossRef]
7. Raut, G.; Rai, S.; Vishvakarma, S.K.; Kumar, A. A CORDIC Based Configurable Activation Function for ANN Applications. In Proceedings of the 2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), Limassol, Cyprus, 6–8 July 2020; pp. 78–83. [CrossRef]
8. Raut, G.; Rai, S.; Vishvakarma, S.K.; Kumar, A. RECON: Resource-Efficient CORDIC-Based Neuron Architecture. *IEEE Open J. Circuits Syst.* **2021**, *2*, 170–181. [CrossRef]
9. Heidarpur, M.; Ahmadi, A.; Ahmadi, M.; Rahimi Azghadi, M. CORDIC-SNN: On-FPGA STDP Learning with Izhikevich Neurons. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2019**, *66*, 2651–2661. [CrossRef]
10. Hao, X.; Yang, S.; Wang, J.; Deng, B.; Wei, X.; Yi, G. Efficient Implementation of Cerebellar Purkinje Cell With the CORDIC Algorithm on LaCSNN. *Front. Neurosci.* **2019**, *13*, 1078. [CrossRef] [PubMed]
11. Manor, E.; Greenberg, S. Efficient Hardware/Software partitioning for Heterogeneous Embedded Systems. In Proceedings of the 2018 IEEE International Conference on the Science of Electrical Engineering in Israel (ICSEE), Eilat, Israel, 12–14 December 2018; pp. 1–4. [CrossRef]
12. Volder, J.E. The Birth of Cordic. *J. VLSI Signal Process. Syst.* **2000**, *25*, 101–105. [CrossRef]
13. Lin, K.J.; Hou, C.C. Implementation of trigonometric custom functions hardware on embedded processor. In Proceedings of the 2013 IEEE 2nd Global Conference on Consumer Electronics (GCCE), Tokyo, Japan, 14 November 2013; pp. 155–157.
14. Walther, J.S. A Unified Algorithm for Elementary Functions. In Proceedings of the Spring Joint Computer Conference—AFIPS '71 (Spring), Atlantic City, NJ, USA, 18–20 May 1971; Association for Computing Machinery: New York, NY, USA, 1971; pp. 379–385. [CrossRef]
15. Walther, J.S. *A Unified Algorithm for Elementary Functions*; AFIPS '71; Spring: Berlin/Heidelberg, Germany, 1971.
16. Nguyen, H.; Nguyen, X.; Pham, C.; Hoang, T.; Le, D. A parallel pipeline CORDIC based on adaptive angle selection. In Proceedings of the 2016 International Conference on Electronics, Information, and Communications (ICEIC), Danang, Vietnam, 27–30 January 2016; pp. 1–4. [CrossRef]
17. Detrey, J.; de Dinechin, F. Floating-Point Trigonometric Functions for FPGAs. In Proceedings of the 2007 International Conference on Field Programmable Logic and Applications, Amsterdam, The Netherlands, 27–29 August 2007; pp. 29–34.
18. Liventsev, E.; Silantiev, A.; Primakov, E.; Telminov, O. Extending MIPSfpga instruction set for navigation data processing. In Proceedings of the 2017 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus), St. Petersburg and Moscow, Russia, 1–3 February 2017; pp. 480–484. [CrossRef]
19. Andraka, R. A Survey of CORDIC Algorithms for FPGA Based Computers. In Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays—FPGA '98, Monterey, CA, USA, 22–24 February 1998; Association for Computing Machinery: New York, NY, USA, 1998; pp. 191–200. [CrossRef]
20. Nolting, S.; Payá-Vayá, G.; Schmeadecke, I.; Blume, H. Evaluation of a Generic Radix-4 CORDIC Coprocessor Tightly Coupled with a Generic VLIW-SIMD ASIP Architecture. 2012. Available online: https://www.researchgate.net/profile/Guillermo-Paya-Vaya/publication/260614052_Evaluation_of_a_Generic_Radix-4_CORDIC_Coprocessor_Tightly_Coupled_with_a_Generic_VLIW-SIMD_ASIP_Architecture/links/570121bb08aea6b7746a78b1/Evaluation-of-a-Generic-Radix-4-CORDIC-Coprocessor-Tightly-Coupled-with-a-Generic-VLIW-SIMD-ASIP-Architecture.pdf (accessed on 9 November 2021).
21. Ibrahim, M.; Chen, K.T.; Idroas, M.; Yahya, Z. The implementation of a pipelined floating-point CORDIC coprocessor on NIOS II soft processor. *Int. J. Electr. Electron. Data Commun.* **2015**, *3*, 15–20.
22. Zhou, J.; Dong, Y.; Dou, Y.; Lei, Y. Dynamic Configurable Floating-Point FFT Pipelines and Hybrid-Mode CORDIC on FPGA. In Proceedings of the 2008 International Conference on Embedded Software and Systems, Chengdu, China, 29–31 July 2008; pp. 616–620.
23. Li, B.; Fang, L.; Xie, Y.; Chen, H.; Chen, L. A unified reconfigurable floating-point arithmetic architecture based on CORDIC algorithm. In Proceedings of the 2017 International Conference on Field Programmable Technology (ICFPT), Melbourne, VIC, Australia, 11–13 December 2017; pp. 301–302. [CrossRef]
24. Nguyen, K.D.; Kiet, D.T.; Hoang, T.T.; Quynh, N.Q.N.; Tran, X.T.; Pham, C.K. A trigonometric hardware acceleration in 32-bit RISC-V microcontroller with custom instruction. *IEICE Electron. Express* **2021**, *18*, 20210266. [CrossRef]
25. Buzdar, A.; Sun, L.; Khan, S.; Buzdar, A. Area and Energy efficient CORDIC Accelerator for Embedded Processor Datapaths. *Inf. Midem Ljubl.* **2016**, *46*, 197–208.
26. Sun, F.; Ravi, S.; Raghunathan, A.; Jha, N.K. A Synthesis Methodology for Hybrid Custom Instruction and Coprocessor Generation for Extensible Processors. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2007**, *26*, 2035–2045. [CrossRef]
27. Wolf, M. Chapter 2 -Instruction Sets. In *Computers as Components*, 3rd ed.; Wolf, M., Ed.; The Morgan Kaufmann Series in Computer Architecture and Design; Morgan Kaufmann: Boston, MA, USA, 2012; pp. 51–93. [CrossRef]
28. Kavvadias, N.; Masselos, K. Efficient Hardware Looping Units for FPGAs. In Proceedings of the 2010 IEEE Computer Society Annual Symposium on VLSI, Lixouri, Greece, 5–7 July 2010; pp. 35–40. [CrossRef]
29. STMicroelectronics. AN5325 Getting Started with the CORDIC Accelerator Using STM32CubeG4 MCU Package. Available online: https://www.st.com/en/embedded-software/stm32cubeg4.html#documentation (accessed on 9 November 2021).