IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 11 December 2025, accepted 25 December 2025, date of publication 12 January 2026, date of current version 15 January 2026.

Digital Object Identifier 10.1109/ACCESS.2026.3652855

RESEARCH ARTICLE

SAPHO–Scalable-Architecture Processor for Hardware Optimization: An FPGA Customizable Implementation Approach

EDER BARBOZA KAPISCH, MELISSA SANTOS AGUIAR, LUCIANO MANHÃES DE ANDRADE FILHO, AND LEANDRO RODRIGUES MANSO SILVA
Departamento de Enhenharia Elétrica, Universidade Federal de Juiz de Fora, Juiz de Fora, Minas Gerais 36036-330, Brazil

Corresponding author: Leandro Rodrigues Manso Silva (leandro.manso@ufjf.br)

This work was supported in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) for the article processing fee under Grant ROR identifier: 00 × 0ma614; in part by the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq); and in part by the Fundação de Amparo à Pesquisa do Estado de Minas Gerais (FAPEMIG), for the promotion of scientific research.

ABSTRACT Soft-core processors are increasingly being used in Field Programmable Gate Arrays (FPGAs) to implement complex functions at critical points in embedded systems, where a pure firmware solution would demand excessive development effort or hardware resources. However, most existing processors rely on a fixed architecture, allocating the same amount of logic resources regardless of the application. This typically results in hardware over-provisioning, oversized data word lengths, and unnecessary power consumption. To overcome these limitations, this work presents SAPHO, a self-scalable processor that automatically adjusts its program and data memories as well as its arithmetic and logic circuits according to the software requirements. This design paradigm reduces hardware waste, lowers logic element utilization, and enables more energy-efficient execution compared to conventional soft-core processors. In addition, SAPHO incorporates a combinational Arithmetic Logic Unit (ALU), configurable for fixed or floating-point operations, that guarantees the execution of one instruction per clock cycle without pipeline stalls. Performance evaluation demonstrates that, when compared to a widely adopted commercial soft-core processor (NIOS II), SAPHO achieves lower hardware cost and execution time while maintaining scalability and flexibility for different embedded applications. Experimental results show that SAPHO executes all arithmetic operations in a fixed sequence of 13 clock cycles, whereas equivalent operations on the NIOS II require from tens to thousands of cycles depending on the processor configuration and data type. In an 8-point FFT benchmark, SAPHO reduces execution time by 77.36% compared to the fastest NIOS II configuration. Furthermore, SAPHO requires fewer logic elements (LEs) and exhibits lower dynamic core power consumption, showing a reduction in dynamic core power consumption ranging from 15% to 33% compared with the NIOS II variants. These results demonstrate that SAPHO achieves lower hardware cost and faster execution while maintaining scalability and flexibility for different embedded applications.

INDEX TERMS Embedded processor, floating-point arithmetic, NIOS II, soft-core.

I. INTRODUCTION

In modern embedded systems, critical parts are often difficult to implement due to their high computational load. This load may be present in the implementation of complex algorithms, such as those using mathematical transforms [1], [2], [3], stimulus-response feedback systems [4], cryptography [5], cybersecurity [6], parameters monitoring and control systems [7].

Recent works have underscored the growing importance of designing low-power, resource-efficient processors for edge and embedded computing, reflecting a trend toward energy-constrained FPGA-based systems for IoT, cyber-physical systems, and real-time applications [8], [9]. For

The associate editor coordinating the review of this manuscript and approving it for publication was Mohammad Hossein Moaiyeri.

example, the soft-processor architecture presented in [10] demonstrates a two-stage pipeline design that dynamically controls stage activation, achieving a 28.2% reduction in average power compared to single-cycle RV32I cores. Meanwhile, the evaluation of open-source soft-core processors shows that lightweight cores optimized for FPGA resources can provide competitive performance and maintain very low power consumption and resource usage, making them attractive for I/O-bound embedded tasks [11]. These efforts illustrate that carefully tuned soft-core CPU architectures can substantially reduce dynamic and total power, thus enabling energy-efficient designs for constrained systems without sacrificing flexibility.

For such constrained and energy-sensitive scenarios, simply mapping applications directly into hardware often leads to suboptimal resource utilization and power efficiency. Building upon the trend highlighted in the previous paragraph, where lightweight and power-aware soft-core CPUs enable more efficient embedded processing, recent studies show that employing Soft-Core Processors (SCPs) can further enhance design flexibility while minimizing hardware footprint and energy consumption. In particular, refinement techniques, customized datapaths, and domain-specific extensions have been shown to improve performance and reduce resource usage on FPGAs [11], [12], [13], [14]. Together, these works reinforce the role of SCP-based architectures as a compelling strategy for achieving optimized, low-power implementations in modern embedded systems.

There are several Soft-Core Processors (SCPs) available, both commercial and open-source [15]. MicroBlaze [16], for example, is a 7-stage pipeline processor [17], with a 32-bit bus and a maximum operating frequency of 250 MHz in Field Programmable Gate Array (FPGA) [18]. LEON3 [19] has 7 pipeline stages, a 32-bit bus and reaches a maximum frequency of about 180 MHz (in FPGA) and 400 MHz (in Application Specific Integrated Circuit (ASIC) [20]). OpenFire [21] has 3 pipeline stages, a 32-bit bus and a maximum frequency of around 200 MHz (in FPGA), being able to use only 4 kB for instruction memory and 4 kB for data memory. OpenRISC1200 [22] has 5 pipeline stages, 32 and a 64-bit buses, with a maximum frequency of 300 MHz (in ASIC) and 180 MHz (in FPGA). MB Lite [23] has 5 pipeline stages, a 32-bit bus and a maximum frequency of 230 MHz (in FPGA). However, it has a low usability due to the absence of an Assembler compiler [24]. AeMB [25] has 3 pipeline stages, being able to execute one instruction per clock cycle. It has a 32-bit bus and its operating frequency is the lowest among all analyzed: about 130 MHz (in FPGA).

Among the commercial SCPs, NIOS II [26] from Intel® has good support and user-friendly development tools [27]. It is a configurable SCP with numerous forms of configuration to meet different applications. It requires low hardware consumption while providing high performance. However, like several other soft-core processors, NIOS II still follows a fixed architectural structure for its datapath and memory organization: once synthesized, the processor allocates the same amount of logic resources regardless of the embedded program being executed. Its data word size is also fixed and typically chosen to accommodate worst-case application demands. This design strategy simplifies toolchain support and improves general-purpose usability, but it may lead to hardware over-provisioning and unnecessary power consumption when running lightweight or domain-specific applications.

In this work, an SCP named the Scalable-Architecture Processor for Hardware Optimization (SAPHO) is presented. SAPHO is an integrated development environment designed for the creation, parameterization, programming, compilation, synthesis, debugging, and analysis of soft-core processors on FPGAs for multiple applications. This environment is part of a project developed by the Signal Processing and Instrumentation Research Group (Núcleo de Instrumentação e Processamento de Sinais - NIPS) at the Federal University of Juiz de Fora (UFJF)[1]. The hardware resources required by SAPHO are allocated and automatically adjusted during the embedded software compilation process. Because the hardware adapts to the software, SAPHO generates a dedicated and optimized hardware description for each program, enabling the direct synthesis of a digital circuit from a software-level description. In this sense, SAPHO can be understood as a high-level synthesis (HLS) oriented system. Thanks to its simple I/O communication protocol, SAPHO has been used, mainly in a hardware architecture that explores a multi-core structure (multiple instances), in several consolidated systems [28], [29], [30], [31], [32].

SAPHO has two main compilers. One that performs the C language encoding into Assembly language (the C compiler), followed by the second one, which compiles the Assembly code into machine code (the assembler). The task of allocating the necessary hardware resources is done by the assembler, which generates a processor parameterization file in Verilog Hardware Description Language (HDL) [33]. In this way, the compiler initially identifies which instructions were generated by the user program, and creates only the hardware resources necessary for their execution. Most of the parameterization, such as the generation of the internal circuits of the Arithmetic and Logic Unit (ALU), is automatic. Other parameters can be adjusted by the programmer through compilation directives, such as the type of ALU (fixed-point or floating-point), data word size, number of input and output addresses, etc., so that designers can empirically tune the required numerical precision while minimizing logic consumption and avoiding unnecessary hardware resource utilization.

In order to be able to modify its internal structure according to the implemented program, SAPHO loses the capability of modifying or loading new programs at runtime. This occurs because the compiler generates a customized RTL

[1] URLhttps://www.nipscern.com/

architecture in which instruction and data memories are physically dimensioned and connected during synthesis, leaving no hardware support for dynamic reprogramming. This is an intrinsic consequence of its compilation model: the program is structurally embedded in the generated hardware. In practical terms, runtime reprogramming would require full resynthesis and reconfiguration of the FPGA. Although this prevents runtime programmability, the resulting reduction in logic usage and memory footprint justifies this trade-off in most practical embedded applications, addressing the central limitation of fixed-architecture SCPs.

Although several soft-core processors could serve as a reference, this work adopts NIOS II as baseline for comparison because it is one of the most widely used commercial SCPs, provides strong toolchain support, is fully integrated into Intel FPGA environments, and represents a mature and well-documented industrial solution. As such, it offers a realistic benchmark for evaluating SAPHO in terms of resource usage, performance, and usability.

The text of this paper is organized as follows. Section II presents the proposed processor architecture, as well as auxiliary software tools. The circuits developed for floating-point arithmetic are detailed in Section III. Section IV shows the results and comparisons with the SCP NIOS II. Finally, some conclusions are outlined in Section V.

II. SAPHO DESCRIPTION

SAPHO is based on the Harvard architecture [34] with a reduced set of instructions (Reduced Instruction Set Computer (RISC)) [35], that can be parameterized according to the developed code. Their Data Memory (DM) and Program Memory (PM) have a self-adjustable number of addresses and the ALU can be configured for fixed-point or floating-point operations. The hardware of the processor was developed in Verilog HDL, so it can be synthesized in any FPGA device.

In addition to the processor, the necessary programming tools were also developed: a compiler that uses a subset of the C language [36], an assembler, and a Integrated Development Environment (IDE).

Some advantages of SAPHO can be highlighted:

- Self-scaling hardware resources allocation at design time, according to the software to be executed.
- Three-stage pipeline architecture with combinational ALU, which allows execution of one instruction per clock cycle without pipeline breaking, even in conditional jump routines.
- Possibility of using ALU in fixed-point or floating-point, with configurable word size.

A. HARDWARE ARCHITECTURE

The main blocks of a SAPHO processor and their interconnections are shown in Fig. 1. In this figure, the solid black interconnections represent the data path, green continuous lines represent address signals, and the dashed interconnections represent control signals.

It is possible to notice that there are four blocks filled in a lighter tone: I → F, F → I, Stack Pointer (Instr.), and Index Register. These blocks are instantiated only when the code requires their use. All other blocks are always instantiated since they are necessary for the processor to function, independently of its configuration. Besides, the ALU instantiates the internal Logic and Arithmetic circuits only when their respective instructions are found in the Assembly code. In this way, SAPHO optimizes hardware resources, customizing the implementation to the specific operations needed to execute the program. The function of each block is explained further ahead.

As the memories are synchronous, the processor needs three pipeline stages: i) instruction fetch (**F**) ii) instruction decode (**D**), and iii) instruction execution (**E**).

Another important aspect to be highlighted about the architecture is that it was designed to be able to execute any program without breaking the pipeline chain (F → D → E), even if the program has conditional statements or function calls.

Because the ALU is combinational, it is possible to accumulate a logical-arithmetic operation in the data register (ACC) at each clock cycle. All Logical-arithmetic instructions contain only one operand coming from memory, the other operand is the own ACC.

SAPHO has stack pointers for instructions and data separately. The former is used to manage subroutine calls while the latter is for logic-arithmetic data-flow. The stacks themselves are allocated at the highest addresses lines of the two memories. The size of each stack can be configured by the user.

1) INT TO FLOAT (I → F) AND FLOAT TO INT (F → I)

The user can choose between two types of ALU: either fixed-point or floating-point. For the latter, even the integer arithmetic is performed using the floating-point circuitry to avoid allocating extra resources.

If the ALU is configured as floating-point, transformation circuits between fixed-point and floating-point representation are automatically instantiated on the input and output buses, so that the processor's communication with I/O devices is performed using fixed-point integer numbers in two's complement representation [37]. The block I → F performs the Int to float conversion while the block F → I does the opposite. SAPHO uses a custom floating-point representation developed to optimize the hardware implementation in FPGA. More details about this representation and its advantages are described in Section III.

2) DATA AND PROGRAM MEMORIES

As previously mentioned, the data memory and program memories have lengths dependent on the developed code. The assembler can determine the exact number of addresses

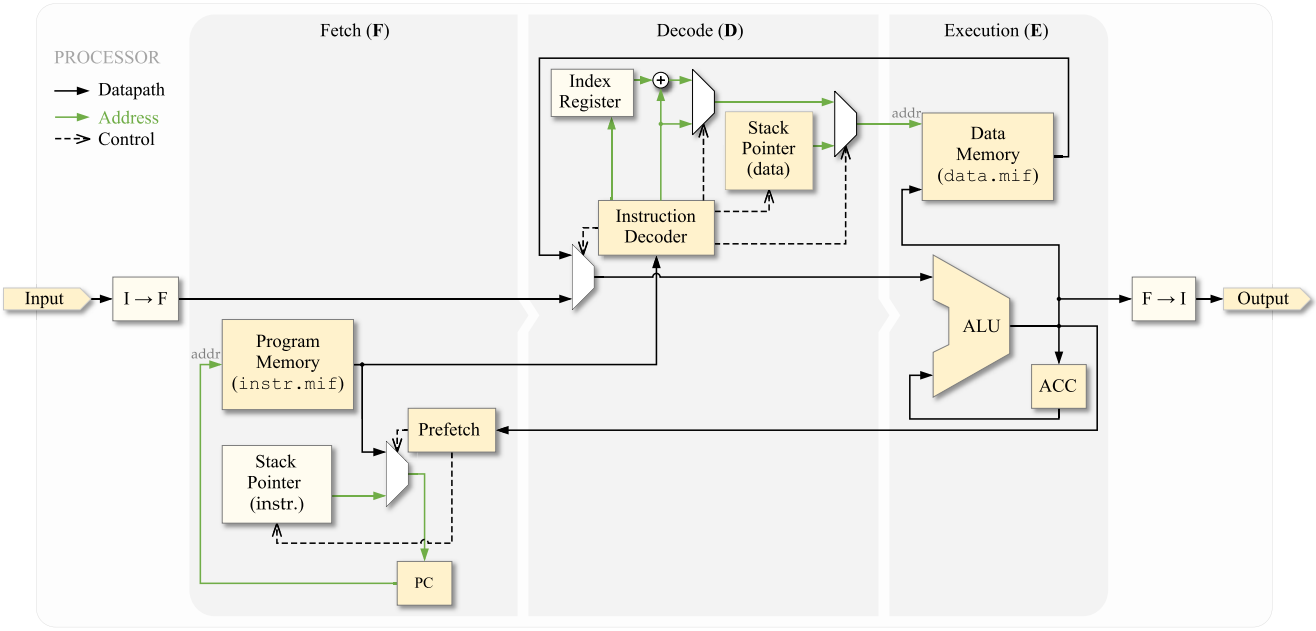

FIGURE 1. SAPHO processor and its main blocks.

needed to store all variables and instructions of the program code and parameterize the memories accordingly. The contents of these memories are also generated by the assembler and saved in memory initialization files (.mif), which are instantiated in the FPGA, along with the processor hardware.

3) PROGRAM COUNTER-PC

This block is responsible for pointing at the address of the current instruction to be read in the program memory. During normal program execution, it is incremented to every instruction. When a jump instruction is detected, it is loaded with a specific value, relative to the next instruction. The PC register number of bits is set by the assembler compiler according to the program memory size.

4) PREFETCH

It fetches from program memory for the next instruction while the processor is executing the current instruction. It is responsible for separating the opcode from the operand and controlling the Instruction Decoder and the PC. This avoids idle cycles in the execution flow and ensures that the instruction register is always updated without introducing stalls. The prefetch mechanism anticipates the next instruction, reducing latency and contributing to the execution of one instruction per clock cycle.

The control signals generated by the Prefetch determine the source of the next instruction to be loaded. Specifically, these signals indicate whether the upcoming instruction must be fetched from program memory or retrieved from the instruction stack, used during subroutine calls and returns. By managing this selection, the Prefetch block ensures correct sequencing across normal execution flow, jumps, and function calls, maintaining a consistent and predictable instruction stream.

5) INSTRUCTION DECODER

This block decodes the current instruction and generates the control signals to all the blocks, according to the instruction function. Some of these control signals are shown in Fig. 1. Since this processor executes one instruction per clock, this block is simply a look-up table mapping the opcode instruction index to all the control values.

Regarding the control signals generated by the Instruction Decoder, they define both the data-flow and addressing behavior of the processor. These signals determine whether the ALU operands originate from the external data input or from the data memory. Additionally, they specify the addressing mode to be used for memory access: immediate/direct addressing (when the address field is encoded in the instruction itself), stack-based addressing (when the address is obtained from the data stack), or indexed addressing (when the effective address is formed by adding an offset to the Index Register during vector or array manipulation). By orchestrating these selections, the Instruction Decoder ensures correct operand routing and addressing semantics for every instruction.

6) STACK POINTERS-SP

There are two stack pointers, one for data and one for instructions. They point to the top position of the stack which is allocated the highest addresses of the data and program memories, respectively. The data stack is accessed with assembly instructions PUSH and POP. The instruction stack is filled with the return address for a function call

TABLE 1. Instructions and respective circuits created automatically.

Instruction	Circuit	ALU	Fix. P.	Float. P.
DIV	Division	X	X	X
OR	Bitwise OR	X	X	
LOR	Logic OR	X	X	X
GRE	Greater than	X	X	X
MOD	Module	X	X	
MLT	Multiplication	X	X	X
LES	Less than	X	X	X
EQU	Equal to	X	X	X
AND	Bitwise AND	X	X	
LAN	Logic AND	X	X	X
INV	Bitwise Inverter	X	X	
LIN	Logic inverter	X	X	X
SHR	Right shift	X	X	
SHL	Left shift	X	X	
SRS	Shift with sign	X	X	
CALL	Instruction stack		X	X
SRF	Indirect addressing		X	X

made with the assembly instruction CALL. The instruction stack, and its respective SP, is generated only if the use of subroutines (assembly instructions CALL and RETURN) is identified.

7) INDEX REGISTER

This block works as an address offset to data memory, allowing the use of arrays in the program code. It is instantiated only when the compiler identifies the use of arrays.

8) ALU

An important feature of the ALU of this SCP is that it has great flexibility in automatic parameterization. This allows the processor to have its structure adapted to the function for which it was assigned for. Most part of the hardware resources comes from the logic and arithmetic circuitry inside the ALU. However, those circuits are automatically instantiated depending on the instructions generated by the C compiler. Other parameters are chosen by the user at design time, such as floating-point or fixed-point arithmetic and the size of the data word.

Table 1 shows the circuits that are automatically created by the assembler, among which the ALU internal circuits are identified.

9) ACC

The ACC register is an accumulator that stores the result of an instruction at the end of the third pipeline stage. Fig.2 shows the pipeline stages executed by a SAPHO processor and the pipeline table of the C-code example shown at the top of the same figure.

Since the ULA is purely combinational, its result is available already in the second pipeline stage. This allows the Prefetch block to decide, in advance, for the new address in a conditional jump, avoiding pipeline breaking.

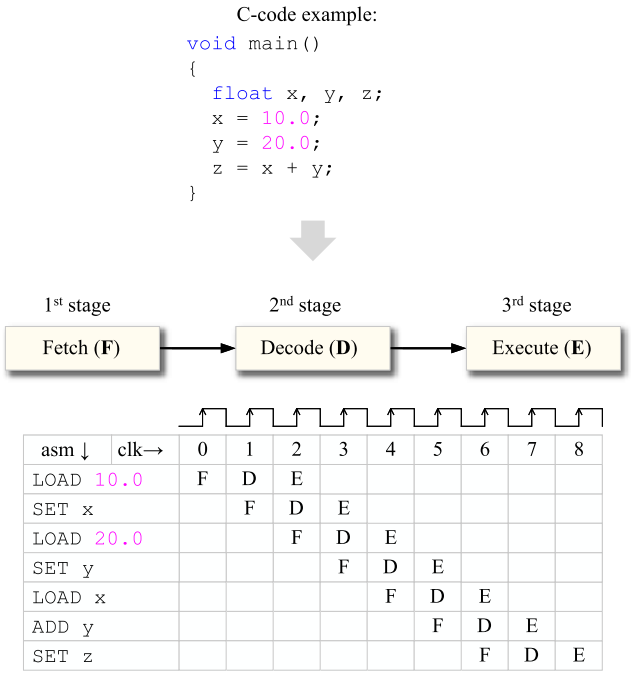

FIGURE 2. Pipeline stages.

TABLE 2. Fixed instructions regardless of the application program.

Instruction	Description
LOAD	Loads a value from data memory into the accumulator (ACC).
PLD	Pushes the current ACC value onto the data stack and loads a value from data memory into the ACC.
SET	Stores the ACC value into data memory.
SETP	Stores the ACC value into data memory and loads the top of the data stack into the ACC.
PUSH	Pushes the ACC value onto the data stack.
JZ	Conditional jump executed if the ACC value is zero.
JMP	Unconditional jump to the specified address.
RETURN	Returns from a function or subroutine.
IN	Reads an input value into the ACC.
OUT	Sends the ACC value to the output interface.

In addition to the instructions presented in Table 1, SAPHO provides a reduced set of fixed instructions responsible for memory access, control flow, and input/output operations. These instructions implement the essential mechanisms required for program execution, and their descriptions are summarized in Table 2.

B. SOFTWARE INFRASTRUCTURE

SAPHO generates the processor's Verilog code automatically from a specially developed IDE to call the C compiler and the assembler. The C Compiler takes the C program code and generates an assembly code, which is then used by the assembler to generate the hardware description code, with the correct contents in the program and data memories.

1) SAPHO'S IDE

SAPHO's IDE was developed in C# [38] language and has some tools to assist in the parameterization and development

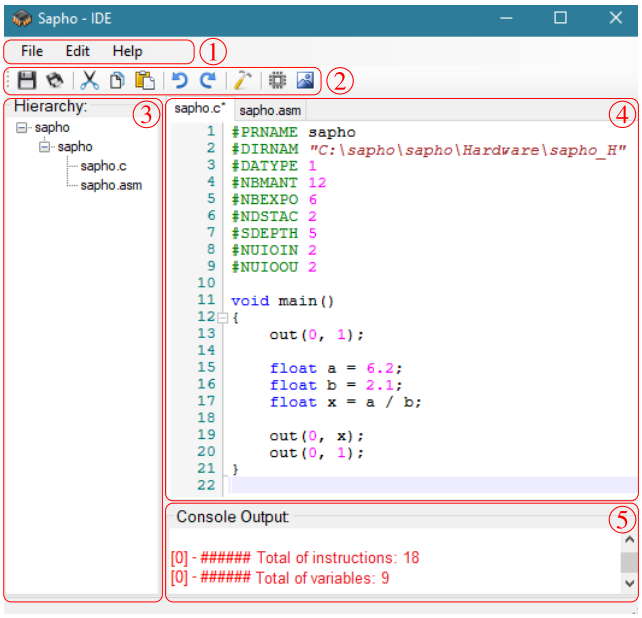

FIGURE 3. SAPHO IDE main screen.

of processors. It is possible to handle multiple processors instances in a project. Fig. 3 shows the main screen of the SAPHO graphical interface, highlighting the 5 main areas: the menu bar ①, the toolbar ②, the project hierarchy tree ③, the programming window ④, and the output console ⑤.

① The menu bar allows the creation of a new project, as well as the management and edition of the current project, or exit the application.

② The toolbar is located right below the menu bar. It contains the shortcuts icons to some basic commands, such as adding a new processor to the project ( ) and the compilation command ( ).

③ In the project hierarchy tree, it is possible to see all processors in the project, as well as access the C (.c) and assembly (.asm) files for each of them.

④ In the programming window, one can program each processor present in the project, as well as view any previous code in both C and assembly languages through the open tabs.

⑤ The console window displays the messages resulting from the compilation processes, such as errors and warnings. If everything is correct, the number of instructions and the number of variables are displayed at the end of the compilation.

When adding a new processor to the project (File menu > New Project), the window named "Configuration Wizzard", shown in Fig. 4, is opened to provide the processor parameterization.

It is possible to customize the following settings:

- Processor Name: A name to identify the processor as well as its C and assembly code.
- Type: fixed-point or floating-point.

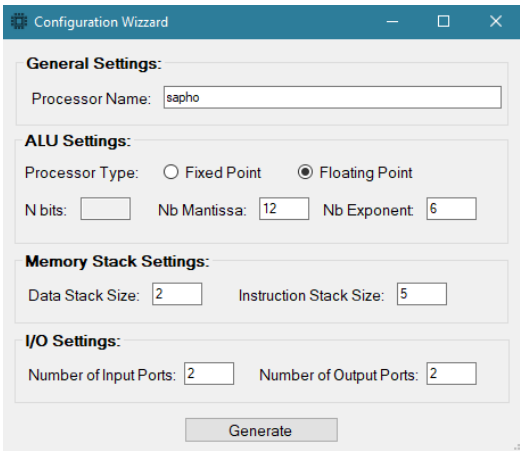

FIGURE 4. Processor configuration wizard screen of SAPHO's IDE.

- The number of bits (N bits) for representation, if fixed-point is chosen, and the number of bits for mantissa (Nb Mantissa) and exponent (Nb Exponent), if floating-point is chosen [39].
- Number of positions of the data stack (Data Stack Size) to be allocated into the Data memory.
- Number of positions of the instruction stack (Instruction Stack Size), if it is used.
- Number of input and output addresses.

This configuration stage is fundamental, as it bridges software-level parameter selection with automatic RTL generation, ensuring that each processor instance is synthesized with only the hardware resources required by the program.

The customized parameters are written as compilation directives to the header of the. c file, as shown in Fig. 3, area ④, lines 1 to 9, and will impact the hardware resources used by the processor. When compiling the project, the memory initialization files *data.mif* and *inst.mif* are generated, which will be synthesized as contents of the data and program memories, respectively. In addition to these files, a Verilog file is generated, which is used to instantiate the processors in the project hardware, which contain all the parameterization done in the SAPHO IDE.

The project development in SAPHO is based on multiple processors, which facilitates the development of multi-core systems since it is possible to easily connect the processors through the I/O buses.

2) C COMPILER

This compiler was developed using the GNU tools *flex* and *bison* [40], to detect keywords and text patterns, respectively. For a friendly syntax, we chose to use a subset of the C language. Table 3 displays the keywords (mnemonics) and logic-arithmetic operators that the compiler recognizes. The program must be written in a single file and accept subroutines. If they are detected, the stack of instructions will be instantiated in the hardware, to enable the automatic return

TABLE 3. Adopted mnemonics and logic-arithmetic operators of the C language.

Mnemonics	float int void return while if else
Operators	- + * / < > ! % & | >> << >= <= == != && ||

of functions, thus allowing the implementation of recursive algorithms.

The use of pointers is only allowed to index fixed-size one-dimensional arrays, so that, the previous calculation of the data memory size is possible. Dynamic memory allocation is not allowed, since the purpose of the processor is to optimize the hardware resources usage. If the compiler detects the array declaration, an indirect addressing circuit (Index Register in Fig. 1) is automatically created in the hardware.

3) ASSEMBLER

The assembler has 47 instructions and it is responsible for generating the Verilog files with the description of the processor's hardware. It was developed using the GNU program *flex* for recognizing the opcode and operands of each instruction. The assembler recognizes the final number of program instructions and the number of variables needed to create the program and data memory with the correct size. It is also responsible to parameterize the processor and to instantiate only the necessary circuitry to run the current application.

C. SAPHO IDE FLOWCHART

The steps necessary to develop a project in the SAPHO IDE are described in the flowchart of Fig. 5. When a new project is created, the SAPHO IDE tool creates a hierarchy of directories where the project files will be stored. From there, it is possible to add processors to the project.

SAPHO is open source, hence all necessary hardware description files are added to the folders. Each processor is programmed individually, using a file in C language or Assembly. After compilation, the *proc.v*, *instr.mif* and *data.mif* files are generated, which are the processor instantiation file and instruction and data memory initialization, respectively.

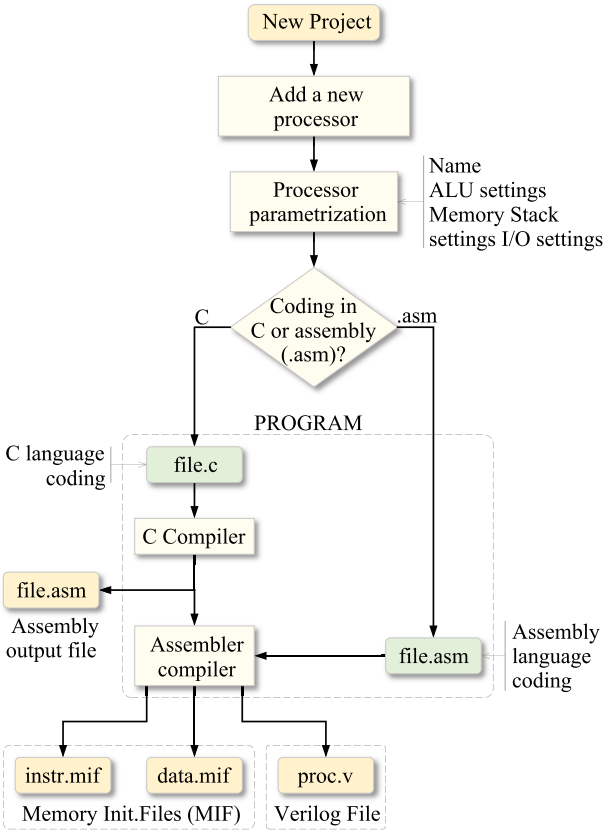

FIGURE 5. SAPHO IDE flowchart.

III. PROPOSED FLOATING-POINT FORMAT

In digital circuits, the floating-point number representation is generally done using the IEEE 754 standard [41], which uses 32 bits in the single format and 64 bits in the double format. This standard also defines some representations of special numbers and the way operations between numbers should be performed.

Single-precision floating-point format (32 bits) consists of 1 sign bit (S), 8 bit for the exponent (E), and a 23 bit for the mantissa (M). The exponent field corresponds to the sum of 127 with the exponent at the base 2 of the represented number. The mantissa is always considered normalized between 1 and 2 in decimal. In this way, its integer part is always a bit equal to 1 (one) and it is omitted in the binary representation. The floating-point representation for 32 bits, according to IEEE 754, is described as:

$$(-1)^S \times 1.M \times 2^{(E-127)} \tag{1}$$

Digital circuits that perform arithmetic operations according to the IEEE 754 standard consume a lot of hardware resources and generally operate in sequential circuits, taking a few clock cycles to generate their responses. In order to work with floating-point numbers in a more efficient way from the point of view of hardware resources and data flow, SAPHO uses a simplified floating-point representation, proposed by the authors of this work in [42].

The proposed floating-point format has a similar structure to the IEEE 754 standard, in which the number is divided into sign (S), mantissa (M) and exponent (E). The mantissa is represented in modulus, as in the IEEE standard (positive values only). To represent the exponent E, a direct representation in twos complement is proposed, to facilitate the hardware description. One advantage of the proposed format is that it can be used for any combination of mantissa and exponent numbers. The normalized mantissa between 1 and 2 is always considered. In this way, its integer part is always a bit equal to 1 (one) which in this case, its representation is necessary.

The proposed floating-point representation is described as:

$$(-1)^S \times M \times 2^E \tag{2}$$

One of the factors that makes the proposed representation lighter, compared to the IEEE standard, is the non-use of extra resources for the representation of special numbers and exception handling. However, in most practical cases, such exceptions are not necessary when the program is carefully designed. Therefore, we chose to prioritize the hardware resources optimization over the exception handling.

Exhaustive tests have shown that, for most of the practical applications, $M = 12$ and $E = 6$ (i.e. total length of 19 bits considering the sign bit) are enough to provide satisfactory results accuracy [42].

In the results section, some comparison results of the proposed floating point format and the IEEE 754 format will be shown, attesting the effectiveness of the proposed format in reducing the hardware usage.

IV. RESULTS

This section presents the performance evaluation of the proposed floating point format and then, the performance of SAPHO is compared to that of NIOS II [26]. The lates was chosen because it is configurable, thus encompassing features found in the various SCPs currently available. Furthermore, NIOS II has a user-friendly IDE and accepts the same C code written in SAPHO.

A. FLOATING POINT RESULTS

Through the proposed floating point representation, arithmetic operations (addition, multiplication and division) were performed, in order to compare the results with operations using the IEEE 754 standard. For sake of comparison, the SAPHO project was configured to have the same number of bits for the Mantissa (23 bits) and the Exponent (8 bits) than the IEEE 754 (32 bits). It is important to highlight that both the IEEE standard and the proposed representation have extremes values of numerical representations, and that the operations presented in Table 4 are operations that comprises both numbers in the middle range and in the extremes.

With these results, it is possible to note that for numbers outside the extremes of the representation scale, the proposed floating point format presents the same result of the IEEE 754 format. For the operations with extreme values or when a division or multiplication operation is performed with one very large number and one very small number (in modulus) the proposed floating point format shows inconsistency.

For some of this cases, the IEEE 754 format uses the Inf and NaN special values. The proposed floating point representation does not have these special representation, for sake of circuit simplicity. Besides, in most applications they are not necessary and the accuracy of the results can be adjusted according to the user application, since the number of bits for Mantissa and Exponent representation is flexible.

The resources used by each arithmetic circuit in both representation formats are shown in Table 5, where LE represents the Logical Elements of the FPGA, REG the registers, and MULT the dedicated multipliers circuits.

TABLE 4. Floating point operations results comparison.

IN1	IN2	OP	IEEE 754	Proposed FP
2.5	3.14	/	0.796178	0.796178
2.5	3.14	-	-0.640000	-0.6400003
-192.27	15.38	×	-2957,112	-2957,112
1.70E+38	1.70E+38	+	3.40E+38	2.94E-39
1.70E+38	2	×	3.40E+38	0
3.55E-06	9.76E-08	-	3.45E-06	3.45E-06
3.55E-06	9.76E-08	×	3.46E-13	3.46E-13
1.70E+38	0.00E+00	/	Inf	0
1,70E+38	0,00E+00	+	1,70E+38	1,70E+38
1,00E+37	-2,00E-20	/	NaN	0
1,00E+37	-2,00E-20	×	-2,00E+16	0

TABLE 5. Hardware usage comparison.

OP	IEEE 754			Proposed FP		
	LE	REG	MULT	LE	REG	MULT
+	258	119	144	635	0	0
×	227	267	7	83	0	7
/	263	316	16	1675	0	7
-	258	119	144	635	0	0

Observing Table 5, it is clear that the proposed floating point representation format presents a great advantage in relation to the IEEE 754 regarding the resources usage. The hardware implementation of the proposed floating point format is fully combinational unlike the IEEE standard. This is crucial to keep the unbroken three stage pileline structure of the proposed processor.

Only in multiplication and division operations, multipliers are used in the proposed format, while in the IEEE 754 they are also used in addition/subtraction, in large quantities. The higher number of LE in the proposed format for addition and subtraction operations is due to the fact that both require combinational mantissa normalization and denormalization circuits. The addition circuit itself spends 79 LE, while the normalization and denormalization circuit spends 155 LE and 401 LE respectively.

B. SAPHO PERFORMANCE

To make the performance test, we used a Terasic development kit, the DE2-115 [43], which contains an FPGA from the Cyclone IV E Family, from Intel® [44]. Three models of the NIOS II processor were used [45]. Among these models, the simplest is the NIOS II/e, which has 32 bit buses, only 1 pipeline stage, and a maximum operating frequency of 200 MHz, focusing on resource optimization.

The intermediate model is the NIOS II/s, which has 32 bit buses, 5 pipeline stages, maximum operating frequency of 165 MHz, as well as multiplication and division hardware. Finally, the NIOS II/f model was also tested, which has 32 bit buses, 6 pipeline stages, maximum operating frequency of 185 MHz, and multiplication and division hardware, focusing on performance optimization. For each of these three models, it was possible to choose between either using or not additional floating-point hardware, totaling 6 possible processor configurations. The generated processors were submitted to performance tests, using the ModelSim-Altera® simulator and Quartus II® hardware synthesis environment.

Several metrics were employed to evaluate the performance of SAPHO in comparison with the NIOS processors, including the utilization of logic elements, memory resources, DSP blocks, maximum operating frequency, and power consumption. In addition, the total execution time for basic arithmetic operations and for the computation of an FFT was also measured.

First, a program was written in C to perform a simple arithmetic operation (sum, product, and division), combined with memory reading and writing instructions. With this test, it is possible to verify the minimum amount of resources and the number of clock cycles needed to perform basic operations, both in fixed-point and floating-point.

A 8-Point Fast Fourier Transform (FFT) [46] algorithm was also implemented. It was chosen because it is one of the most used algorithms in digital signal processing [47]. To identify the beginning and end of the code under test in the ModelSim® software, an I/O instruction is executed immediately before and right after the operation. Thus, it is possible to count the number of clock cycles during the execution of this block of instructions.

Tables 6 and 7 show the number of clock cycles and logical resources required for SAPHO and the 6 NIOS II configurations (with and without the additional hardware for floating-point). For SAPHO processors, the floating-point configuration is $M = 12$ and $E = 6$, since its configuration has proved to reach satisfactory accuracy.

Unlike conventional processors, aiming at optimizing resources, SAPHO does not perform integer arithmetic operations when the instantiated ALU is floating-point and vice versa. From a practical point of view, all variables, even those used for indexing or counters, can be computed in floating-point format when this type of ALU is used. Thus, the results for SAPHO are presented only for the data type referring to the respective ALU type. In the case of fixed-point ALU, SAPHO was set to 32 bits, to provide a fair comparison with NIOS II.

Regarding the implementation of the FFT, SAPHO processor was configured using only a floating-point representation ALU. The results are also presented in Tables 6 and 7. The main part of the FFT code is shown in SAPHO IDE (Fig. 6).

It is worth noting that all arithmetic operations in SAPHO present the same execution time of 13 clock cycles in Table 6. SAPHO uses a simplified control unit based on a fixed sequence of 13 micro-operations that govern the entire instruction cycle, including instruction fetch, decoding, operand selection, ALU activation, memory access, and program counter update. Because the ALU is fully combinational, each arithmetic operation completes in a single micro-operation, and the remaining micro-operations correspond to control, data routing, and synchronization tasks required to ensure correct timing across the datapath. As a consequence, all instructions—regardless of opcode or operand type—follow the same 13-cycle control flow. The only exception is the FFT benchmark, whose total clock count reflects the number of instructions in the program rather than the cost of each arithmetic operation.

When generating a NIOS II processor, the number of logic elementss (LEs), memory bits, and multipliers (Digital Signal Processors (DSPs)) are fixed, regardless of the program being executed. One can see that in Table 7, where for all basic operations and FFT, it is used the same hardware and maximum frequency for each NIOS II. On the other hand, in SAPHO these parameters are auto-scalable according to

TABLE 6. Number of clocks for each operation. Hardware configurations for floating-point are indicated by *.

Modelsim Simulation	Sum		Product		Division		FFT
	int	float	int	float	int	float	
NIOS II/e	117	2703	263	8246	401	3822	813538
NIOS II/e*	117	1454	263	1456	489	1481	161041
NIOS II/s	41	911	45	1040	183	1214	118378
NIOS II/s*	41	529	45	537	186	554	40605
NIOS II/f	28	771	28	872	161	1008	80654
NIOS II/f*	34	465	40	467	162	490	27884
SAPHO	13	13	13	13	13	13	1981

TABLE 7. Logical cost and frequency for each operation. Hardware configurations for floating-point are indicated by *.

FPGA Implementation			Logical Resources	Memory Bits	DSP (9 bits)	Freq. MHz
NIOS II/e			1440	141312	0	136.28
NIOS II/e*			7969	142393	7	205.34
NIOS II/s			2225	175616	4	173.25
NIOS II/s*			8828	176697	11	182.88
NIOS II/f			3024	193216	4	132.52
NIOS II/f*			9615	194297	11	158.08
SAPHO	Sum	int	186	288	0	120.66
		float	617	171	0	54.08
	Product	int	191	288	2	77.47
		float	447	171	2	71.62
	Division	int	1.424	288	0	8.55
		float	912	171	0	15.19
	FFT*		884	9.025	2	49.61

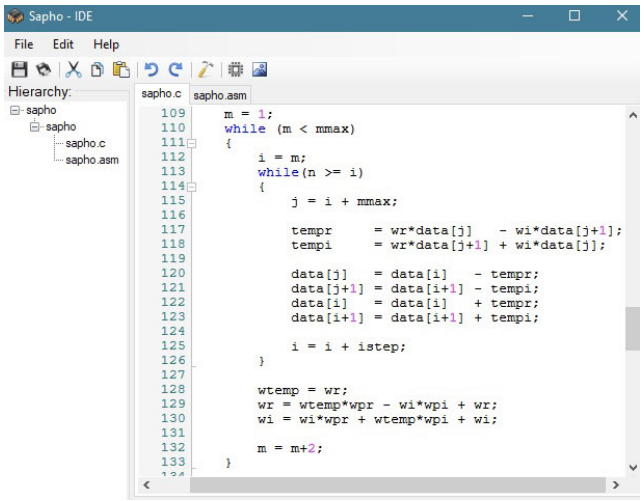

FIGURE 6. SAPHO IDE with the main part of the FFT code.

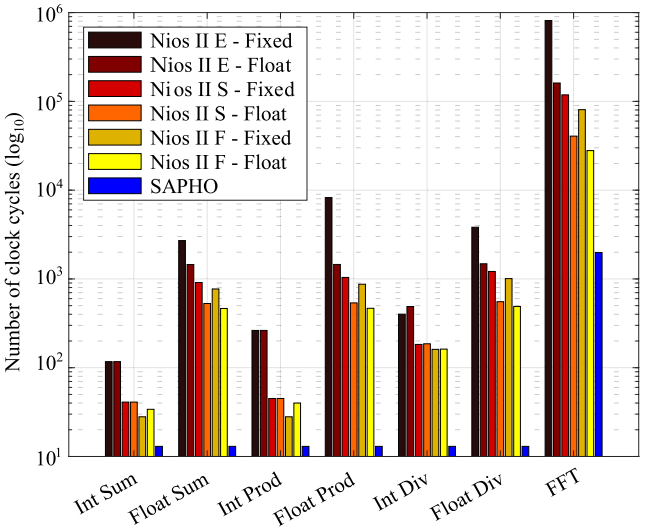

FIGURE 7. Graphical representation of the total number of clocks for each operation from Table 6 (the lower the better).

the written program code. In the case of arithmetic operations, the largest number of LEs for SAPHO occurs when it detects the division operation, which in this example is 32 bits for a fixed-point ALU and 19 bits for floating-point configuration. In programs where this operation is not used, SAPHO self-scalability is evident, showing one of its most important features, the optimization of hardware resources.

According to Table 7, NIOS II/e, which is the simplest configuration among the 6 NIOS II variations, uses 1440 LEs to perform both basic operations and FFT code. This number is greater than the number of LEs used by SAPHO in its higher consumption version (fixed-point division with 1424 LE) and considerably higher than SAPHO configured to perform FFT (884 elements). Also, the NIOS II/e needs 813,583 clock cycles to calculate FFT in floating-point, while SAPHO executes the FFT in 1,981 clock cycles. Even the highest performing version of NIOS II (NIOS II/f) requires 27,884 clock cycles to calculate the FFT, using 9,615 logic elements, much more expensive quantities than those presented by SAPHO (see Tables 6 and 7).

The amount of memory bits and the number of DSP blocks is also significantly higher in NIOS II variants. Furthermore, these numbers are auto-scalable in SAPHO, avoiding the waste of these two very important resources in most FPGA hardware applications.

Despite SAPHO presenting a lower maximum operating frequency compared to the NIOS II configurations, this difference does not directly translate to poorer performance. The execution time of a given task depends jointly on the maximum clock frequency and on the number of clock cycles required to complete each instruction. While NIOS II relies on multi-stage pipelines and microcoded operations that often require hundreds or even thousands of cycles to complete an arithmetic instruction, SAPHO executes every instruction in a fixed sequence of 13 cycles due to its simplified control structure and fully combinational ALU. Therefore, the total execution time T_{exec} can be expressed as:

$$T_{exec} = \frac{N_{cycles}}{f_{max}},$$

where N_{cycles} is the number of cycles needed to complete the execution, and f_{max} is the maximum operation frequency. For SAPHO, the reduction in cycle count largely compensates for the lower operating frequency.

To better visualize the results from Tables 6 and 7, Figures 7 and 8 were created, respectively. In Fig. 7, the total number of clock cycles per mathematical operation is plotted. The lower the number of clocks, the faster the operation is performed. We can see that SCPs SAPHO (blue color) displays the shorter bars in all operations. It is important to observe that it was necessary to use the Y axis on a logarithmic scale ($\log_{10}$) so that the bars relating to SAPHO could be seen, due to the difference compared to NIOS II.

Fig. 8 shows the hardware resource consumption and maximum operating frequencies for mathematical operations. We can see that the hardware consumption optimization of SAPHO outperforms the hardware consumption of all NIOS II configurations while maintaining a close maximum operating frequency. Again, the Y axis was used on a logarithmic scale for better visualization.

Table 8 shows the execution time of the basic operations sum, product, and division, as well as the execution time of the FFT for the six different NIOS II configurations and SAPHO. For this table, each processor was executed at the highest allowed frequency, as shown in Table 7. Among the NIOS II configurations, the one with the highest performance is the NIOS II/f with floating-point hardware, named NIOS II/f* in the table. This configuration runs the FFT code faster than other NIOS II configurations, totaling 176.39 µs. However, SAPHO was able to execute the same

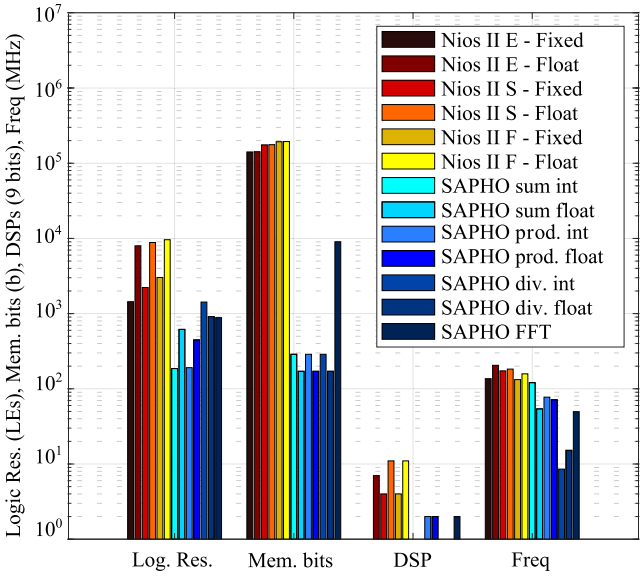

FIGURE 8. Graphical representation of the hardware resource consumption and frequency for each operation from Table 7 (the lower the better).

TABLE 8. Execution time of each operation. Hardware configurations for floating-point are indicated by *.

Time (μs)	Sum		Product		Division		FFT
	int	float	int	float	int	float	
NIOS II/e	0.85	19.83	1.93	60.51	2.94	28.05	5969.61
NIOS II/e*	0.57	7.08	1.28	7.09	2.38	7.21	784.26
NIOS II/s	0.23	5.26	0.26	6.00	1.06	7.01	683.28
NIOS II/s*	0.22	2.89	0.25	2.94	1.02	3.03	222.03
NIOS II/f	0.21	5.37	0.21	6.59	1.21	7.61	608.62
NIOS II/f*	0.22	2.94	0.25	2.95	1.02	3.10	176.39
SAPHO*	0.11	0.24	0.17	0.18	1.52	0.86	39.93

code in an even shorter time, that is, 39.93 µs, representing a processing time reduction of approximately 77.36%.

At the same time, SAPHO presented a considerably reduced hardware consumption, both in terms of LE, memory bits, and embedded multipliers in the FPGA, represented by the number of 9-bit DSP blocks used in the implementation.

As for the maximum frequency, due to the fact that the ALU is combinational in SAPHO, it operates, on average, three times slower than the NIOS II. However, the combinational ALU allows a three-stage pipeline architecture without pipeline breaking, which makes a sequence of operations to be predictable and performed with less clock cycles.

To provide a clearer comparison of the execution times presented in Table 8, Fig. 9 illustrates the total execution time T_{exec} for each processor configuration. In this bar plot, shorter bars indicate lower execution time and, consequently, faster performance. As in the previous figures, a logarithmic scale ($\log_{10}$) was used on the Y-axis to improve visualization, given the significant differences between the evaluated

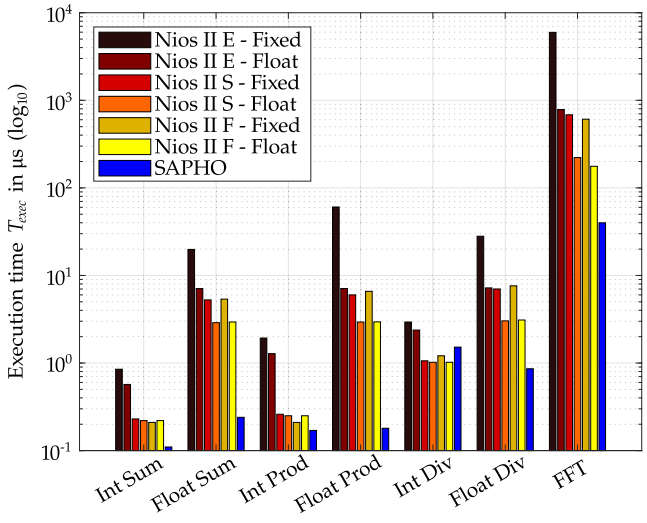

FIGURE 9. Graphical representation of the execution time (T_{exec}) for each operation from Table 8. Shorter bars indicate faster execution.

TABLE 9. Thermal power dissipation of each configuration. Hardware configurations for floating-point are indicated by *.

Power Dissipation (mW)	Core Static	Core Dynamic	Total
NIOS II/e	93.97	0.51	94.48
NIOS II/e*	94.05	0.55	94.60
NIOS II/s	94.01	0.66	94.67
NIOS II/s*	94.09	0.61	94.70
NIOS II/f	94.04	0.57	94.61
NIOS II/f*	94.11	0.58	94.69
SAPHO*	93.92	0.44	94.36

architectures. The results show that SAPHO achieves the lowest execution time in all tested operations, with the exception of the integer division, for which SAPHO still ranks among the fastest configurations. These results confirm that the reduction in clock cycles per instruction strongly compensates for SAPHO's lower maximum operating frequency.

Table 9 shows that static power dominates the total power consumption for all processor configurations. The core static power remains nearly constant across all implementations, varying only from 93.92 mW to 94.11 mW, a fluctuation of less than 0.2%. This behavior confirms that static power is essentially determined by the FPGA device characteristics rather than by the complexity of the implemented processor.

Dynamic power, on the other hand, shows noticeable variation across architectures, ranging from 0.51 to 0.66 mW in the NIOS II processors while still remaining below 0.7% of the total power. The NIOS II core presents the highest dynamic power due to its more complex architecture, and enabling hardware floating-point support produces only minor changes. SAPHO exhibits the lowest dynamic power (0.44 mW, 15–33% lower than the NIOS II variants), consistent with its streamlined, application-specific design,

resulting in the lowest total thermal dissipation among all evaluated processors.

V. CONCLUSION

To achieve versatility, conventional SCPs use an excessive amount of hardware resources, where most of these resources are not even used. To optimize the use of these resources, this work presented a self-scaling SCP, where only the resources detected by the assembler compiler are instantiated. To do that, the assembler compiler is responsible for creating the processor description code.

This new programming paradigm, where the hardware description code is created at software compilation time, opens paths for new research in the field of optimization of embedded circuits in configurable devices, such as FPGAs. In this type of integrated circuit, whose reprogrammability is an universal feature, the use of static SCPs is characterized by enormous waste of resources.

Since SCPs are generally used at critical points in a system, for example, to perform complex mathematical calculations, it is important to develop simple circuits for floating-point arithmetic calculation. The IEEE 754 standard, although robust, is not suitable for FPGA implementation. The SCP proposed in this work uses a combinational floating-point ALU specially developed to perform feedback operations in a single clock cycle. This circuit has been extensively tested in several projects in the field of electrical power quality, where a data word of only 19 bits has been shown to be sufficiently efficient.

As future work, we intend to extend SAPHO by introducing additional arithmetic modules that can be instantiated on demand, such as trigonometric and logarithmic units, work with 2-D arrays, complex numbers, among others, further reducing the need for software-level approximations. Also, support for programmable interrupt sources and basic exception handling is planned for future versions in addition to broader experimental evaluations using larger benchmark suites and real-world embedded applications to reinforce the generality and scalability of SAPHO's architecture.

Finally, at the initial stage of development in which the contribution of this work is placed, it should be noted that the objective is not to overcome well-established commercial SCPs, such as NIOS II, in complex management applications, such as operating systems execution, different data communication buses, and code debug interface. These complex features are outside the scope of this work. The purpose of this processor is the application in the execution of specific tasks in critical points of a digital system and this task is becoming increasingly important and present in modern embedded systems.

ACKNOWLEDGMENT

This work utilized the ChatGPT artificial intelligence system, developed by OpenAI, for grammatical review and text enhancement. The technical content and ideas presented are solely the authors' responsibility.

REFERENCES

[1] V. M. Ribeiro, N. D. S. M. D. Santos, E. B. Kapisch, L. R. M. Silva, and C. A. Duque, "Real-time implementation of stockwell transform in FPGA platform using soft-core processor applied to novelty detection in power quality signals," *J. Control, Autom. Electr. Syst.*, vol. 35, no. 3, pp. 509–521, Jun. 2024.

[2] S. K. Mitra, *Digital Signal Processing: A Computer-Based Approach*. New York, NY, USA: McGraw-Hill, 2001.

[3] D. Sundararajan, *Discrete Wavelet Transform: A Signal Processing Approach*. Hoboken, NJ, USA: Wiley, 2015.

[4] S. Haykin, *Kalman Filtering and Neural Networks*. Hoboken, NJ, USA: Wiley, 2004.

[5] A. C. Mert, E. Karabulut, E. Öztürk, E. Savas, and A. Aysu, "An extensive study of flexible design methods for the number theoretic transform," *IEEE Trans. Comput.*, vol. 71, no. 11, pp. 2829–2843, Nov. 2022.

[6] E. M. Benhani, L. Bossuet, and A. Aubert, "The security of ARM TrustZone in a FPGA-based SoC," *IEEE Trans. Comput.*, vol. 68, no. 8, pp. 1238–1248, Aug. 2019.

[7] G. C. Goodwin, S. F. Graebe, and M. E. Salgado, *Control System Design*. Upper Saddle River, NJ, USA: Prentice-Hall, 2000.

[8] I. Zagan and V. G. G itan, "System verification and FPGA implementation of hardware preemptive scheduler for RISC-V processor," *IEEE Access*, vol. 13, pp. 103019–103032, 2025.

[9] I. Zagan and V. G. Gãitan, "Custom soft-core RISC processor validation based on real-time event handling scheduler FPGA implementation," *IEEE Access*, vol. 11, pp. 36264–36280, 2023.

[10] S. Kumar and K. C. Ray, "An energy-efficient adaptive two stage pipeline RISC-V based soft-processor for edge-computing applications," *Int. J. Parallel, Emergent Distrib. Syst.*, vol. 41, no. 1, pp. 1–15, Jan. 2026.

[11] M. Gazziro, J. M. D. A. Junior, O. H. A. Junior, M. R. Cavallari, and J. P. Carmo, "Design and evaluation of open-source soft-core processors," *Electronics*, vol. 13, no. 4, p. 781, Feb. 2024. [Online]. Available: https://www.mdpi.com/2079-9292/13/4/781

[12] G. S. Chae, "Refinement of a soft-core processor implementation on FPGA," *J. Theor. Appl. Inf. Technol.*, vol. 103, no. 13, pp. 4693–4704, 2025.

[13] I. Zagan and V. G. G itan, "Soft-core processor integration based on different instruction set architectures and field programmable gate array custom datapath implementation," *PeerJ Comput. Sci.*, vol. 9, p. e1300, Apr. 2023.

[14] H. Sriraman and A. Ravikumar, "Customized FPGA design and analysis of soft-core processor for DNN," *Proc. Comput. Sci.*, vol. 218, pp. 469–478, Jan. 2023.

[15] M. Makni, M. Baklouti, S. Niar, M. W. Jmal, and M. Abid, "A comparison and performance evaluation of FPGA soft-cores for embedded multi-core systems," in *Proc. 11th Int. Design Test Symp. (IDT)*, Dec. 2016, pp. 154–159.

[16] *Microblaze Processor Reference Guide Embedded Development Kit Edk 7.1I*, Xilinx, San Jose, CA, USA, 2012.

[17] P. M. Kogge, "The microprogramming of pipelined processors," *ACM SIGARCH Comput. Archit. News*, vol. 5, no. 7, pp. 63–69, Mar. 1977.

[18] B. U. Meyer, *Digital Signal Processing With Field Programmable Gate Arrays*. Berlin, Germany: Springer, 2007.

[19] *Processor Users Manual XT Edition*, LEON3, Gothenburg, Sweden, Oct. 2008.

[20] R. Hossain, *High Performance ASIC Design: Using Synthesizable Domino Logic in an ASIC Flow*. Cambridge, U.K.: Cambridge Univ. Press, 2008.

[21] (2023). *Openfire*. [Online]. Available: https://opencores.org/projects/openfire2

[22] D. Lampret, "OpenRISC1200 IP core specification," OpenCores, Ljubljana, Slovenia, Tech. Rep. Rev. 0.7, 2001.

[23] (2023). *MB-Lite*. [Online]. Available: https://opencores.org/projects/mblite

[24] R. Hyde, *The Art of Assembly Language*. San Francisco, CA, USA: No Starch Press, 2010.

[25] (2023). *AeMB*. [Online]. Available: https://opencores.org/projects/aemb

[26] *NIOS II Processor Reference Guide*, Intel, Santa Clara, CA, USA, 2019.

[27] Y. Elangovan, M. Saraf, B. Satyanarayana, S. S. Upadhya, N. Panyam, R. Shinde, G. Majumder, D. Sil, Pathaleswar, S. Thoi Thoi, and K. C. Ravindran, "NIOS II soft-core processor and Ethernet controller solution for RPC-DAQ in INO ICAL," *J. Instrum.*, vol. 20, no. 1, Jan. 2025, Art. no. P01021.

[28] E. B. Kapisch, L. R. M. Silva, C. H. N. Martins, A. S. Barbosa, L. M. A. Filho, C. A. Duque, A. E. Tavil, and L. A. R. de Souza, "An implementation of a power system smart waveform recorder using FPGA and ARM cores," *Measurement*, vol. 90, pp. 372–381, Aug. 2016.
[29] E. B. Kapisch, L. R. M. Silva, A. S. Cerqueira, L. M. de Andrade Filho, C. A. Duque, and P. F. Ribeiro, "A gapless waveform recorder for monitoring smart grids," in *Proc. 17th Int. Conf. Harmon. Quality Power (ICHQP)*, Oct. 2016, pp. 130–136.
[30] E. B. Kapisch, L. R. M. Silva, C. H. N. Martins, A. S. Barbosa, C. A. Duque, L. M. de Andrade Filho, and A. S. Cerqueira, "An electrical signal disturbance detector and compressor based on FPGA platform," in *Proc. 16th Int. Conf. Harmon. Quality Power (ICHQP)*, May 2014, pp. 278–282.
[31] C. H. N. Martins, H. L. M. Monteiro, M. M. D. Oliveira, L. R. M. Silva, C. A. Duque, and P. F. Ribeiro, "A virtual instrument for time varying harmonic analysis," in *Proc. IEEE Power Energy Soc. Gen. Meeting (PESGM)* Jul. 2016, pp. 1–5.
[32] M. M. de Oliveira, L. R. M. Silva, C. A. Duque, L. M. de Andrade Filho, and P. F. Ribeiro, "Implementation of an electrical signal compression system using sparce representation," in *Proc. 18th Int. Conf. Harmon. Quality Power (ICHQP)*, May 2018, pp. 1–5.
[33] M. D. Ciletti, *Advanced Digital Design With the Verilog HDL*, vol. 1. Upper Saddle River, NJ, USA: Prentice-Hall, 2003.
[34] D. Harris and S. Harris, *Digital Design and Computer Architecture*. San Mateo, CA, USA: Morgan Kaufmann, 2010.
[35] D. A. Patterson, "Reduced instruction set computers," *Commun. ACM*, vol. 28, no. 1, pp. 8–21, Jan. 1985.
[36] D. Ritchie and B. Kernighan, *The C Programming Language*. Englewood Cliffs, CA, USA: Bell Laboratories, 1988.
[37] R. Tocci, *Digital Systems: Principles and Applications*. Upper Saddle River, NJ, USA: Prentice-Hall, 1991.
[38] M. Skinner, *Beginning Visual C# 2010*. IN, USA: Wrox, 2010.
[39] S. Boldo, C.-P. Jeannerod, G. Melquiond, and J.-M. Müller, "Floating-point arithmetic," *Acta Numer.*, vol. 32, pp. 203–290, May 2023.
[40] J. Levine, *Flex & Bison*. Sebastopol, CA, USA: O'Reilly Media, 2009.
[41] *IEEE Standard for Floating-Point Arithmetic*, Standard IEEE Std 754-2019, 2019.
[42] V. A. M. Santos, E. B. Kapisch, L. R. M. Silva, and L. M. A. Filho, "Implementation of arithmetic circuits in floating point, using format with configurable number of bits," in *Proc. 22th Brazilian Autom. Congr.*, 2018.
[43] *DE2-115 User Manual-world Leading FPGA Based Products and Design Services*, Terasic Technologies Inc, Taiwan, 2013.
[44] Altera, *Cyclone IV Device Handbook*, vol. 3. San Jose, CA, USA: Altera Corporation, 2016.
[45] *Nios II Core Implementation Details*, vol. 3, Altera Corporation, San Jose, CA, USA, 2015.
[46] W. H. Press, *Numerical Recipes*. Cambridge, U.K.: Cambridge Univ. Press, 1992.
[47] T. Ayhan, W. Dehaene, and M. Verhelst, "A 128:2048/1536 point FFT hardware implementation with output pruning," in *Proc. 22nd Eur. Signal Process. Conf. (EUSIPCO)*, Sep. 2014, pp. 266–270.

EDER BARBOZA KAPISCH was born in Rio de Janeiro, Brazil, in July 1989. He received the B.S., M.Sc., and Ph.D. degrees in electrical engineering from the Federal University of Juiz de Fora (UFJF), Brazil, in 2013, 2015, and 2019, respectively. From 2017 to 2018, he was a Guest Ph.D. Researcher with Eindhoven University of Technology (TU/e), The Netherlands. Since 2019, he has been an Adjunct Professor with UFJF. His field of acting is electronic systems, electronic instrumentation, and digital signal processing applied to power systems. Some of his works involve digital signal processing algorithms development and implementation using field programmable gate array (FPGA) platform applied to detection and compression of power quality electrical disturbances.

MELISSA SANTOS AGUIAR was born in Minas Gerais, Brazil, in May 1996. She received the B.S. and M.Sc. degrees in electrical engineering from the Federal University of Juiz de Fora (UFJF), Brazil, in 2020 and 2023, respectively. She is currently pursuing the Ph.D. degree. She works with the ATLAS Experiment, European Organization for Nuclear Research (CERN), Geneva, Switzerland. Her area of work is digital signal processing and embedded systems based on FPGA.

LUCIANO MANHÃES DE ANDRADE FILHO received the degree in electrical engineering from UERJ, in 2001, the master's degree from Brazilian Center for Physics Research, in 2004, and the Ph.D. degree in electrical engineering from UFRJ/COPPE, in 2009, with a sandwich period with the Center Européenne pour la Recherche Nucléaire (CERN), Geneva. He was a Post-Doctorate in data communication in electrical networks from the Federal University of Juiz de Fora (UFJF), in 2009, and in nuclear instrumentation from CERN, in 2021. He is currently an Associate Professor IV with UFJF. He has experience in the area of nuclear instrumentation and particle accelerators, with an emphasis on electronic circuits, working mainly on the following topics: signal processing for calorimetry, digital processing in FPGA, and computational intelligence.

LEANDRO RODRIGUES MANSO SILVA was born in Valença, Brazil, in August 1985. He received the B.S., M.Sc., and Ph.D. degrees in electrical engineering from the Federal University of Juiz de Fora (UFJF), Brazil, in 2009, 2012, and 2016, respectively. He is currently an Associate Professor with UFJF. His field of working is real-time digital signal processing algorithms developing and implementation on dedicated platforms, such as DSP and FPGA, and applied to power systems. He also works with computational intelligence, load modeling, time-varying harmonic estimation, power quality signals compression, and embedded systems.

Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) - ROR identifier: 00x0ma614