

Article

Hardware-Based WebAssembly Accelerator for Embedded System

Jinyeol Kim , Raehyeong Kim , Jongwon Oh  and Seung Eun Lee *

Department of Electronic Engineering, Seoul National University of Science and Technology,
Seoul 01811, Republic of Korea; kimjinyeol@seoultech.ac.kr (J.K.); kimraehyeong@seoultech.ac.kr (R.K.);
ohjongwon@seoultech.ac.kr (J.O.)

* Correspondence: seung.lee@seoultech.ac.kr; Tel.: +82-2-970-9021

Abstract: WebAssembly (WASM) has emerged as a novel standard aimed at enhancing the performance of web applications, developed to complement traditional JavaScript. By offering a platform-independent binary code format, WASM facilitates rapid and efficient execution within web browsers. This attribute is particularly advantageous for tasks demanding significant computational power. However, in resource-constrained environments such as embedded systems, the processing speed and memory requirements of WASM become prominent drawbacks. To address these challenges, this paper introduces the design and implementation of a hardware accelerator specifically for WASM. The proposed WASM accelerator achieves up to a 142-fold increase in computation speed for the selected algorithms compared to embedded systems. This advancement significantly enhances the execution efficiency and real-time processing capabilities of WASM in embedded systems. The paper analyzes the fundamentals of WebAssembly and provides a comprehensive description of the architecture of the accelerator designed to optimize WASM execution. Also, this paper includes the implementation details and the evaluation process, validating the utility and effectiveness of this methodology. This research makes a critical contribution to extending the applicability of WASM in embedded systems, offering a strategic direction for future technological advancements that ensure efficient execution of WASM in resource-limited environments.

Keywords: WebAssembly (WASM); JavaScript; embedded system; accelerator



Citation: Kim, J.; Kim, R.; Oh, J.; Lee, S.E. Hardware-Based WebAssembly Accelerator for Embedded System. *Electronics* **2024**, *13*, 3979. <https://doi.org/10.3390/electronics13203979>

Academic Editor: Lukáš Kohútka

Received: 30 July 2024

Revised: 7 October 2024

Accepted: 7 October 2024

Published: 10 October 2024



Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

In contemporary society, the internet has become indispensable, enabling functions from simple document transmission to large-scale data transfer, real-time communication, remote education, cloud computing, and the Internet of Things (IoT). It also powers advancements in autonomous vehicles, smart home systems, and telemedicine. The significance of the internet will continue to grow.

JavaScript dominates client-side code execution in the web ecosystem. Since its creation in 1995, JavaScript has been integral to web browsers, allowing for dynamic web pages. It excels in event handling, document object model (DOM) manipulation, and asynchronous JavaScript and XML (AJAX) [1]. Despite its versatility, JavaScript was initially designed for lightweight tasks. Today's internet, however, demands the execution of complex tasks such as gaming and image processing, highlighting JavaScript's performance limitations. Various optimizations have been implemented to improve JavaScript's performance, such as virtual machines and Just-in-Time (JIT) compilation. Nonetheless, significant overhead remains, particularly in complex computations or large-scale data processing [2].

To overcome these constraints, WebAssembly (WASM) was introduced in 2017 [3–5]. WASM is a platform-independent binary format, allowing code from languages like C, C++, and Rust to be compiled and executed on the web [6]. It provides much higher performance than JavaScript, achieving near-native execution speeds. With WASM, its fast load times,

consistent performance, and secure execution environment significantly enhance web application capabilities [7]. WASM is increasingly utilized in various applications, such as facial recognition [8], Google Maps, and Netflix, with its usage expected to expand. Ongoing research aims to further reduce the runtime of WASM and improve its performance [3].

Advancements in internet technology extend beyond traditional computing environments, profoundly impacting fields such as the IoT and the Internet of Vehicles (IoV). As the usage of the internet in embedded systems increases, these systems must operate efficiently even with limited resources, making fast runtime processing increasingly crucial. Consequently, there is active research into low-power design, optimized algorithm, and resource-efficient management techniques [9–11]. These studies aim to enhance the internet-based functionality of embedded systems, ensuring optimal performance under constrained conditions.

However, WASM faces significant limitations in embedded systems. Notably, there is considerable performance degradation compared to Desktop environments [12,13]. This reduction suggests that the potential performance benefits of WASM are not fully realized within the resource constraints of embedded systems. For instance, the Wasmer single-pass compiler [14], designed for fast compilation, may lack execution-time optimizations, leading to performance drawbacks. While various software-based solutions report performance improvements of up to 4–5 times [10,15–18], these gains still fall short compared to the execution speeds of WASM on desktop environments.

This paper aims to design a dedicated hardware accelerator to enhance the execution speed of WASM in embedded environments. This approach seeks to leverage its high performance and platform independence, making these benefits accessible within resource-constrained embedded systems. To the best of our knowledge, this research represents the first attempt to accelerate WASM with hardware, offering a significant departure from existing software-based optimization methods and promising fundamental performance improvements through hardware acceleration. The main contributions of this paper are as follows:

- **Design and Implementation of a WASM Hardware Accelerator:** We design a hardware accelerator for directly executing WASM instructions, optimizing performance in embedded systems.
- **Performance Enhancements:** The accelerator handles efficient encoding (LEB128) and advanced memory optimization to significantly improve WASM operation speeds.
- **Experimental Validation:** We validate the accelerator through experiments in embedded environments, demonstrating substantial performance gains over traditional methods.

This study focuses on improving WASM performance in embedded systems and explores potential applications in fields such as IoT and IoV. Overall, this research introduces a novel solution for enhancing performance in embedded systems, setting a foundation for future developments in these and other related areas. The remainder of this paper is organized as follows.

Section 2 provides a comprehensive overview of WASM, discussing its advantages, operational mechanisms, and usage in embedded systems. It also reviews research on WASM performance issues in resource-constrained environments, exploring software-based solutions to these challenges and investigating hardware-based solutions for similar challenges. Section 3 outlines the design and architecture of the proposed hardware accelerator for WASM, detailing its components, their interactions, and the strategies used for optimizing performance. Section 4 presents the experimental evaluation of the hardware accelerator using field programmable gate array (FPGA), including the setup, test cases, and analysis of how the accelerator enhances the execution speed of WASM in embedded contexts. Section 5 summarizes the findings, highlighting the performance improvements and discussing the advantages and limitations of the hardware accelerator. Finally, Section 6 discusses the limitations of the accelerator and suggests future improvements to enhance hardware-accelerated WASM in embedded systems.

2. Background and Related Work

This section provides an overview of WebAssembly's operation and its various run-times. Additionally, it discusses related research efforts focused on optimizing WASM and its applications.

2.1. WebAssembly

WASM is a binary instruction format designed to boost web application performance by enabling near-native execution speeds. This enhancement is achieved by compiling source code into a platform-independent binary format, facilitating efficient execution across various operating systems and browsers. The WASM sandbox environment enhances security by isolating code execution, preventing potential system vulnerabilities [19]. WASM also integrates well with JavaScript, making it easy to enhance existing web applications. Beyond the web, WASM is increasingly employed in embedded systems, server-side applications, and mobile apps due to its performance, security, and broad compatibility [20].

2.1.1. Operation

WASM operates primarily within web browsers and functions similarly to JavaScript. Utilizers write application code in high-level languages such as C, C++, or Rust, which is then compiled into WASM modules, or they have the option to directly write WASM code. WASM code can be represented in several forms: high-level source code, WASM text format (WAT), and WASM byte format. Figure 1 illustrates these transformations. High-level source code is the original code written by utilizers, which can be converted to WAT by compilers like Emscripten [21]. WAT is a human-readable text format that is useful for debugging, code inspection, or direct WASM programming [22].

When WAT is compiled into WASM binary, it produces a compact and efficient binary format suitable for fast parsing and execution. This binary format is highly suitable for network transmission and is typically employed in web applications due to its speed and small size. Tools are available to facilitate the conversion between WAT and WASM binary, allowing utilizers to write in the readable WAT format and convert it to executable WASM binary format. Once compiled into binary format, WASM modules are stored and transmitted to web browsers via web servers. Browsers load these modules and execute them in a runtime environment similar to JavaScript's. In this setup, JavaScript invokes the WASM modules, which enables data to flow between the two environments.

In addition to web applications, WASM is increasingly adopted as a standalone technology in various other domains. Its light weight and portability make it suitable for server-side applications, desktop applications, and embedded systems [23]. In these settings, WASM modules operate independently, offering the performance and security benefits typical of native applications. Independence from traditional web contexts allows WASM to be integrated into diverse application ecosystems, expanding its utility and impact beyond the web browser [24].

High-level source code (C code)	WebAssembly text format (.wat)	WebAssembly byte format (.wasm)
<pre>int factorial (int n) { if (n == 0) return 1; else return n * factorial(n-1); }</pre>	<pre>(func(param i64) (result i64) local.get 0 i64.eqz if (result i64) i64.const 1 else local.get 0 local.get 0 i64.const 1 i64.sub 0 call i64.mul end)</pre>	<pre>00 61 73 6D 01 00 00 00 01 06 01 60 01 7E 01 7E 03 02 01 00 0A 17 01 15 00 20 00 50 04 7E 42 01 05 20 00 20 00 42 01 7D 10 00 7E 0B 0B</pre>

Figure 1. WebAssembly compilation process diagram.

Figure 2 illustrates some of the instructions in WASM. WASM supports four fundamental value types: i32, i64, f32, and f64. These basic types can be combined to form more complex data structures. Each WASM instruction is represented by an 8-bit value ranging from 00 to FF, with the interpretation of each subsequent instruction potentially varying based on the preceding one. WASM instructions are categorized into five main groups:

- **Numeric Instructions:** These instructions perform arithmetic, logical, and comparison operations on the basic data types. They include both unary and binary operators, such as addition and subtraction.
- **Parametric Instructions:** These instructions operate independently of specific data types and primarily facilitate type conversion, allowing operations across different data types.
- **Variable Instructions:** These instructions handle the reading and writing of local and global variable values. Examples include *get_local*, *set_local*, *get_global*, and *set_global*. Variables are accessed by their index in a table, allowing efficient manipulation.
- **Memory Instructions:** These instructions are related to memory access, responsible for loading and storing data. They enable reading from and writing to the memory allocated to WASM modules.
- **Control Instructions:** These instructions manage the flow of the program, including function calls, conditional branches, and loops. Instructions like *call*, *if*, *loop*, and *branch* fall into this category. WASM supports calling functions, including recursive calls, to control execution flow effectively.

The WASM instruction set is designed to enable high-performance execution across various platforms. The WASM virtual machine operates as a stack-based machine, similar to the Java VM. In this model, operands are pushed onto the stack, and operations are performed by popping operands from the stack and pushing the results back. This approach simplifies the instruction set compared to register-based machines, resulting in smaller binary encodings and faster code verification. The stack-based model also facilitates straightforward implementation and consistent performance across different environments.

Numeric Instructions	Variable Instructions	Control Instructions
$nn, mm ::= 32 64$ $sx ::= u s$ $iunop ::= clz ctz popcnt$ $ibinop ::= add sub mul div_{sx} rem_{sx}$ $\quad \quad \quad and or xor shl shr_{sx} rotr rotr$ $funop ::= abs neg sqrt ceil floor trunc nearest$ $fbinop ::= add sub mul div min max copysign$ $itestop ::= eqz$ $irelop ::= eq ne lt_{sx} gt_{sx} le_{sx} ge_{sx}$ $frellop ::= eq ne lt gt le ge$	$instr ::= \dots$ $\quad \quad \quad local.get localidx$ $\quad \quad \quad local.get localidx$ $\quad \quad \quad local.tee localidx$ $\quad \quad \quad global.get globalidx$ $\quad \quad \quad global.set globalidx$	$blocktype ::= typeidx valtype$ $instr ::= \dots$ $\quad \quad \quad nop$ $\quad \quad \quad unreachable$ $\quad \quad \quad block blocktype instr^* end$ $\quad \quad \quad loop blocktype instr^* end$ $\quad \quad \quad if blocktype instr^* else instr^* end$ $\quad \quad \quad br labelidx$ $\quad \quad \quad br_if labelidx$ $\quad \quad \quad br_table vec(labelidx) labelidx$ $\quad \quad \quad return$ $\quad \quad \quad call funcidx$ $\quad \quad \quad call_indirect funcidx$
Parametric Instructions	Memory Instructions	
$instr ::= \dots$ $\quad \quad \quad drop$ $\quad \quad \quad select$	$memarg ::= \{offset\ u32, align\ u32\}$ $instr ::= \dots$ $\quad \quad \quad inn.load memarg fmn.load memarg$ $\quad \quad \quad inn.store memarg fmn.store memarg$ $\quad \quad \quad inn.load8_{sx} memarg inn.load16_{sx} memarg i64.load32_{sx} memarg$ $\quad \quad \quad inn.store8_{sx} memarg inn.store16_{sx} memarg i64.store32_{sx} memarg$ $\quad \quad \quad memory.size$ $\quad \quad \quad memory.grow$	

Figure 2. WebAssembly instruction set overview.

2.1.2. Compilation Method

WASM has established itself as a flexible bytecode-based virtual machine that can be used across a variety of target platforms including web browsers, servers, and embedded systems. This bytecode-based virtual machine typically supports three execution methods: interpreter, ahead-of-time (AOT) compilation [25], and JIT compilation [18]. Firstly, the interpreter executes bytecode instructions one at a time, applying operations to a virtual stack machine. Secondly, AOT compilation translates bytecode into executable machine code specific to the target machine, either at load time on the device or beforehand, externally. Thirdly, JIT compilation dynamically translates bytecode during execution time. These execution methods often work together to enhance performance.

WASM execution relies on a runtime environment that manages program operations and provides services like memory management and exception handling [26]. Figure 3 illustrates the Wasm3 [27] runtime architecture, which includes a profiler, compiler, and interpreter. The profiler gathers performance data, the compiler generates optimized native code, and the interpreter executes this code, enhancing WASM application performance. These methods and components enable WASM to deliver efficient and adaptable performance across different computing environments.

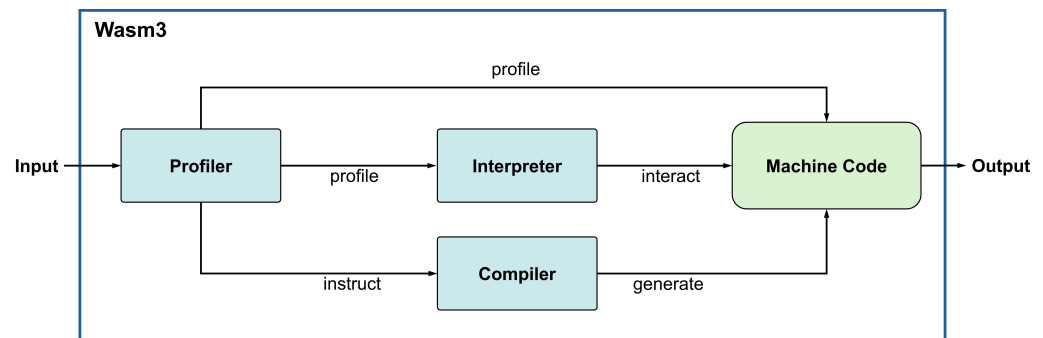


Figure 3. Wasm3 runtime transformation flow [18].

Prominent WASM runtimes include V8 [28], Wasmtime [29], Wasm3 [27], and Wasmer [14]. As illustrated in Table 1, each WASM runtime possesses unique characteristics and advantages. For instance, V8 [28] is a high-performance JavaScript and WASM engine applied in Google’s Chrome browser and Node.js, boasting fast execution speeds through JIT compilation. V8 excels particularly in web applications and server-side applications. Wasmtime [29] is a fast and secure WASM runtime that is popular in serverless computing environments. It supports both AOT and JIT compilation, providing a balance between initial execution speed and runtime performance. Additionally, it is designed to operate across various operating systems and platforms, favoring cross-platform application development. Wasm3 [27], a lightweight WASM interpreter suitable for low-resource environments, offers very low memory usage and quick start-up times, making it effective for embedded systems and IoT devices. Although it does not support JIT compilation, it can still deliver sufficient performance through an interpreter approach. Wasmer [14] enables WASM modules to be executed like native applications and supports various execution modes, particularly enhancing performance through JIT compilation while also speeding up initial execution through AOT compilation. Wasmer can be utilized in various fields such as serverless and edge computing, helping utilizers to easily deploy and execute WASM modules [22].

Table 1. Comparative analysis of WebAssembly runtimes [13].

Runtime	Language	AOT	JIT	Embedded	Feature
V8 [28]	C++	-	O	O	A high-performance engine that offers platform independence and is suitable for embedded systems.
Wasmtime [29]	Rust	O	O	O	A WebAssembly runtime implemented in Rust, providing fast execution speeds and safety.
Wasm3 [27]	C	O	-	-	A lightweight runtime capable of quickly loading and executing WebAssembly modules.
Wasmer [14]	Rust	O	O	O	Compiles and executes WebAssembly applications as native code.

These various WASM runtimes can be selected according to different environments and requirements [30]. For example, V8 is optimal for web applications, Wasmtime for

serverless computing environments, Wasm3 for low-resource devices, and Wasmer for executing native applications. The choice of runtime should be made by comprehensively considering factors such as application performance, resource usage, and deployment environments. In conclusion, WASM and its supporting runtimes enable efficient and flexible application execution across modern computing environments. Utilizers can employ these runtimes to maximize performance and resource efficiency, providing a consistent execution environment across various platforms [31]. As the WASM ecosystem continues to evolve, more runtimes and tools will emerge, offering utilizers more choices and possibilities.

2.2. WASM in Embedded Systems

2.2.1. Embedded Systems

Embedded systems are composed of specialized computers designed for specific tasks, found in devices like appliances, cars, industrial machines, and IoT devices [32]. They must efficiently operate with limited resources and often demand high reliability and real-time performance. WASM is becoming prominent in embedded systems due to its platform independence and security features. It is especially beneficial for IoT devices, enabling efficient data processing and remote updates [33]. The ability of WASM to run code from various languages simplifies development and deployment, proving to be a valuable tool for embedded applications. In essence, its resource efficiency, platform independence, and security make it a crucial technology in the realm of embedded systems [34].

2.2.2. Performance Issues and Limitations

Although WASM offers many advantages for embedded systems, it does not always achieve optimal performance in embedded systems. Research shows that WASM experiences significant slowdowns in embedded systems compared to JavaScript. For instance, ref. [12] reports that tasks executed by WASM on a Raspberry Pi take roughly five times longer than on a typical CPU, with memory usage tripling. This slowdown is attributed to the WASM single-pass compiler, which prioritizes quick compilation over runtime optimization. And, ref. [13] found that WASM achieves only about 50% of desktop performance when executed on the ESP32-C3, a RISC-V 32-bit based embedded system. This performance gap is linked to the portability and isolation features of WASM, which introduce additional overhead.

Several factors contribute to these performance challenges. The platform independence of WASM often requires execution through JIT compilers or interpreters, which can add significant overhead. Additionally, the WASM approach to memory management might not be suitable for the limited memory resources typical of embedded systems. Furthermore, the current debugging and profiling tools for WASM are not yet fully developed, making it difficult for utilizers to optimize performance effectively. In summary, while WASM provides resource efficiency, platform independence, and security benefits in embedded systems, it also faces significant challenges related to performance degradation, memory management, and tool support [19]. These limitations need to be addressed to fully leverage the potential of WASM in embedded environments.

2.3. The Need for Hardware Acceleration

2.3.1. Software-Based WebAssembly Acceleration Efforts

Many studies have been conducted to address the performance limitations of WASM in embedded environments. Table 2 summarizes these related works, highlighting various approaches to enhance the performance of WASM in such constrained settings.

Table 2. Analysis of related works.

Source	Proposed Approach	Feature	Acceleration
Fabian Scheidl, 2020 [10]	Streaming-like single-pass compiler	Utilization of WebAssembly module structure, deferred machine code emission, command merging, and peephole optimization	Only 21% slower in embedded systems (improved from 87%)
Gregor Peach et al., 2020 [15]	eWASM software sandbox	Emporal and spatial isolation for sandboxing, applicable to resource-constrained microcontrollers	Performance at 40% of native code
Jämes Ménétrey et al., 2021 [16]	TWINE trusted runtime environment	Execution of real applications like SQLite within SGX enclaves, supports real applications	Up to $4.1\times$ performance improvement with some library modifications
Robbert G. et al., 2019 [17]	WARDuino Virtual Machine	Real-time code updates, remote debugging, modularity	About $5\times$ faster performance than traditional interpreted approaches

One such study, ref. [10], introduced a compiler designed to enhance WASM performance specifically for embedded systems. This compiler uses techniques like delayed machine code emission, instruction merging, and peephole optimization to create a streaming-like single-pass compiler. It accelerates WASM execution, reducing the performance slowdown from an initial 87% to just 21% on embedded systems, making it suitable for both embedded and desktop environments.

Another approach, detailed in [15], is the introduction of eWASM, a software sandbox designed for legacy code on resource-constrained microcontrollers. This sandbox provides temporal and spatial isolation, achieving performance levels up to 40% of native C code. The study also offers recommendations for enhancing sandboxing capabilities on devices with limited resources, improving both performance and security.

Ref. [16] presented TWINE, a trusted runtime environment for securely executing language-independent applications using WASM. TWINE was tested with real-world applications like SQLite running within an SGX enclave, showing significant improvements in performance and security [19]. Modifying some libraries allowed performance gains of up to 4.1 times.

The study [17] proposed WARDuino VM, a WebAssembly-based virtual machine aimed at overcoming challenges in microcontroller programming. WARDuino VM enhances performance and development convenience by supporting real-time code updates, remote debugging, and modularization. Experimental results indicate that WARDuino VM performs about five times faster than traditional interpreted approaches.

These efforts, along with others documented in [18,35], demonstrate ongoing research and innovation aimed at overcoming the performance challenges of WASM in embedded systems. Each study contributes to enhancing the utility of WASM in resource-constrained environments, paving the way for its broader adoption and efficiency in embedded applications.

2.3.2. Hardware Acceleration Efforts for Other Problems

However, software-based acceleration methods have not completely eliminated the workload overhead associated with WASM execution. Therefore, we aimed to explore a novel approach by accelerating WASM using hardware, which had not been attempted before. Utilizing hardware for acceleration is expected to virtually eliminate compilation time and significantly reduce runtime [36]. To achieve this, we investigated various hardware accelerators to inform the design and development of our WASM accelerator.

In [37], a coprocessor was designed to address the low performance of posits in embedded systems, which were introduced to improve the accuracy of floating points. The posit

coprocessor added registers internally to reduce the data path and implemented pipelining in the posit operations, accelerating division and square root calculations performed with posits. Compared to state-of-the-art research, it achieved operations within fewer clock cycles at the same frequency, resulting in a 60.09 times performance improvement in exponential function calculations.

The paper [11] designed an accelerator to address the increasing load of 2D graphic data processing in embedded systems. The accelerator uses line-drawing operations to speed up the processing of 2D graphics. Additionally, it is connected to the system bus and directly to the frame buffer, allowing it to operate independently of the processor. This reduces the workload needed for the processor to access the frame buffer, thus improving overall system performance.

Ref. [9] proposed a compression accelerator optimized for scale-out servers in the LZ4 data center. Using a low-performance processor, the LZ4 hardware accelerator is designed to reduce the computational overhead of compression tasks in the processor. The accelerator enhances compression throughput, allowing large amounts of data to be compressed and transmitted more efficiently, which improves both the transmission bandwidth and storage space. This shows a performance 1.558 times faster than software-based LZ4 compression.

These studies confirm that designing customized hardware architectures for computation, parallelizing processes, and pipelining computational stages significantly accelerate computational speed.

3. System Architecture

To improve WASM performance in embedded systems, we designed a hardware-based WASM accelerator. Figure 4 shows the overview of the proposed system, which interprets and executes WASM bytecode directly within embedded environments. The accelerator is designed to handle WASM code obtained either from the internet or from local applications running on the embedded system, processing the code and returning the results without the need for traditional runtime execution.

By bypassing conventional WASM runtimes and directly performing computations, this accelerator is expected to significantly improve performance. As illustrated in Figure 3, the proposed system eliminates the need for the interpretation, compilation, and profiling typically required to convert bytecode into native code. This streamlining is anticipated to lead to fundamental performance gains in embedded system environments.

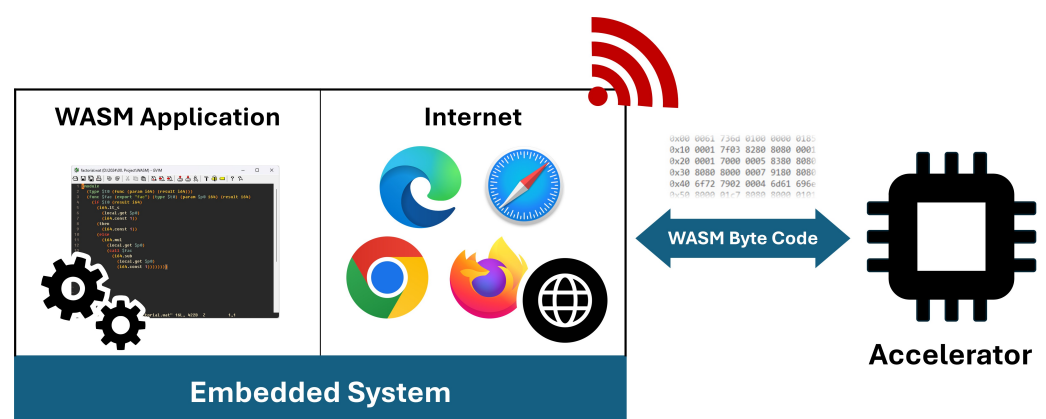


Figure 4. The overall system of the proposed system.

The accelerator, illustrated in Figure 5, connects to the embedded system via the serial peripheral interface (SPI). It has two main components: the execution part and memory part. The execution part interprets bytecode with a finite state machine (FSM) to handle each instruction. The memory part employs static random access memory (SRAM) to store instructions and leverages register memory to efficiently manage data during execution.

The realized instructions employed are designed based on the specifications provided by the official WebAssembly website [38], which is based on the standard WASM instruction set. Our accelerator supports 36 WASM instructions for 32-bit integer and floating point operations, optimized for quick processing, as detailed in Figure 6. This instruction set includes basic arithmetic operations and memory access management, and is optimized for rapid execution.

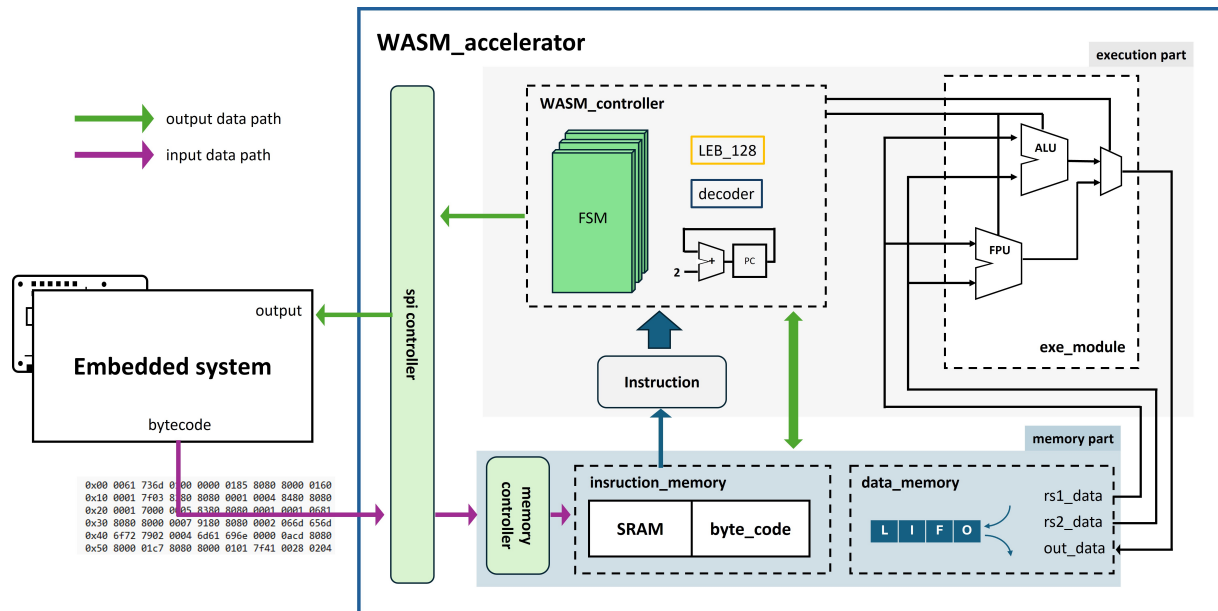


Figure 5. Hardware architecture diagram of accelerator.

byte code	instr	byte code	instr	byte code	instr
0x41	i32.const	0x67	i32.clz	0x5B	f32.eq
0x43	f32.const	0x6A	i32.add	0x5C	f32.ne
0x46	i32.eq	0x6B	i32.sub	0x5D	f32.lt
0x47	i32.ne	0x6C	i32.mul	0x5E	f32.gt
0x48	i32.lt_s	0x6D	i32.div_s	0x5F	f32.le
0x49	i32.lt_u	0x6F	i32.rem_s	0x60	f32.ge
0x4A	i32.gt_s	0x71	i32.and	0x92	f32.add
0x4B	i32.gt_u	0x72	i32.or	0x93	f32.sub
0x4C	i32.le_s	0x73	i32.xor	0x94	f32.mul
0x4D	i32.le_u	0x74	i32.shl	0x96	f32.min
0x4E	i32.ge_s	0x75	i32.shr_s	0x97	f32.max
0x4F	i32.ge_u	0x76	i32.shr_u	0x98	f32.copysign

Figure 6. Supported instructions set for the WASM accelerator.

3.1. Execution Part

The execution part of the system is divided into the *WASM_controller* and the *exe_module*. In the *WASM_controller*, the program counter (PC) decodes incoming instructions with a decoder, subsequently managing the FSM. The *exe_module* includes both an arithmetic logic unit (ALU) and a floating point unit (FPU) to perform actual computations [39]. The numbers in the instructions operate based on the LEB_128 encoding.

3.1.1. Decoder

The decoder is a critical component that interprets WASM bytecode and converts it into executable instructions. Figure 7 visually explains how the bytecode is transmitted and interpreted. Every WASM module begins with a header located at the start of the

file, which includes a magic number and version information. This header verifies the module's validity and ensures it is correctly parsed. Following the header, various sections are listed, each of which must follow a specific order and appears only once. Each section starts with its type and length, and the decoder reads these sections sequentially, accurately identifying the start and end of each.

The content of these sections includes the function definitions, code blocks, and data storage structures. The decoder uses this information to prepare the instructions for execution. Specifically, the function and code sections define the metadata and execution code for each function, and contain various instructions.

Additionally, the decoder utilizes the type system of WASM to validate the data types used by the instructions. This system ensures that the correct data types are provided for each instruction during execution, preventing runtime errors and ensuring the stability of the code. WASM operates as a stack-based virtual machine. The decoder processes instructions by pulling values from the stack, performing operations, and then pushing the results back onto the stack [40].



Figure 7. Decoding WebAssembly bytecode diagram.

3.1.2. Finite State Machine

As shown in Figure 8, the WASM accelerator utilizes several Finite State Machines (FSMs) to manage the execution flow of WASM bytecode efficiently. These FSMs are critical for managing the intricate structure of WASM and coordinating its execution within embedded systems efficiently. The primary FSMs include the *wasm_state* FSM, the *section_state* FSM, and specific FSMs for each WASM section.

The *wasm_state* FSM controls the overall operation of the accelerator. It starts by verifying the WASM bytecode's magic number and version to ensure compatibility. Following these checks, the FSM processes the bytecode through several states: In the *SEC_ID* state, it identifies the section type and determines the expected number of bytecode entries. The *SEC_SIZE* state specifies the total byte count for the current section. The *SEC_CONT* state monitors the bytecode being processed, ensuring it matches the expected count. Upon section completion, indicated by the *sec_cplt* signal, the FSM loops back to process the next section, continuing until all sections are processed. When the final section completes, signaled by *wasm_cplt*, the FSM resets to the initial state.

The *section_state* FSM identifies the type of section currently being processed. Initially, it is in the *IDLE* state. It transitions to the appropriate section state based on the bytecode provided during the *SEC_ID* state. Once a section completes, it returns to *IDLE*. The *section_state* FSM sequentially processes sections from *TYPE* to *CUSTOM* but can skip non-present sections. The *sec_done* signal, a one-hot vector, tracks which sections have been executed, preventing reprocessing and maintaining the correct order. Each section of the WASM bytecode has its own unique FSM due to their distinct structures. For example, the function section includes metadata about inputs and outputs, while the code section contains execution instructions. These specific FSMs allow the precise decoding and handling of bytecode tailored to each section's requirements.

By integrating these FSMs, the WASM accelerator effectively decodes and executes WASM bytecode, managing both the overall flow and the specific needs of individual sections.

This design maximizes the WASM capabilities in embedded systems, providing a robust solution for efficiently handling WASM bytecode in resource-constrained environments.

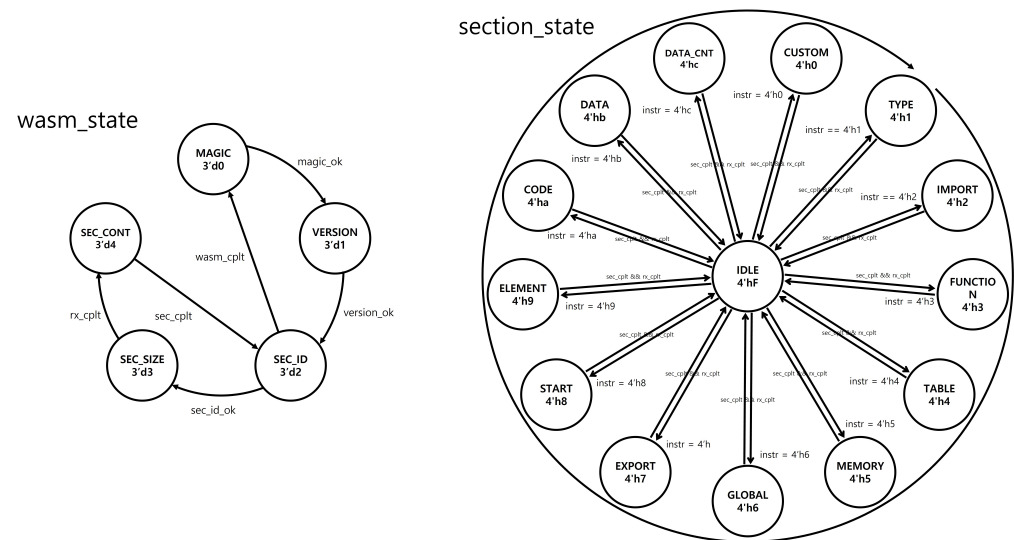


Figure 8. FSM diagram of the WASM accelerator.

3.1.3. Little Endian Base 128

Little Endian Base 128 (LEB128) is a variable-length encoding method that efficiently encodes integers and floating-point numbers. In LEB128, the most significant bit (MSB) of each byte indicates whether more bytes follow, while the remaining 7 bits represent the data value. This allows small numbers to be stored in fewer bytes and larger numbers in more bytes, optimizing file size and improving transmission and storage efficiency. This method is crucial in WASM, enhancing performance across various platforms and speeding up data transmission.

Figure 9 illustrates how LEB128 encoding works: the decimal number 122 is encoded into one byte, while 963,412 requires three bytes. The encoding process involves splitting the number into 7-bit chunks, removing leading padding bits, setting the MSB to 1, setting subsequent padding bits to 0, and arranging the data in Little Endian format. This method is particularly advantageous in environments with limited network or storage resources, such as embedded systems, as it significantly boosts the performance of WASM by reducing the size of encoded data.

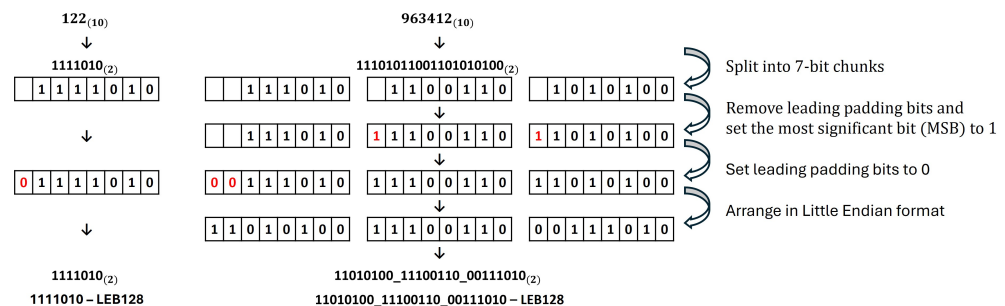


Figure 9. LEB128 encoding process diagram.

Figure 10 illustrates our LEB128 encoder and decoder. The LEB128 decoder interprets the incoming WASM bytecode instructions. As previously mentioned, the first bit is used to indicate whether more bytes follow, serving as a finish signal for the LEB instruction. The decoder employs a counter and demux to process the LEB instruction, incrementing the counter based on the number of instruction bytes and determining where to store the

data in the registers. Once stored, when the *leb_finish* signal goes high, the data are saved as either a 32-bit integer or a floating-point type.

In the encoder, LEB128 is used to output data for 32-bit integers and floating-point types. As shown in the figure, the 32-bit data are split into 7-bit segments and stored in registers. The most significant block's first bit is set to 1, and the data are transmitted starting with the least significant bits. This method is especially useful in environments where network bandwidth is limited or storage space is critical, making it ideal for embedded systems. It significantly enhances the efficiency and performance of WASM.

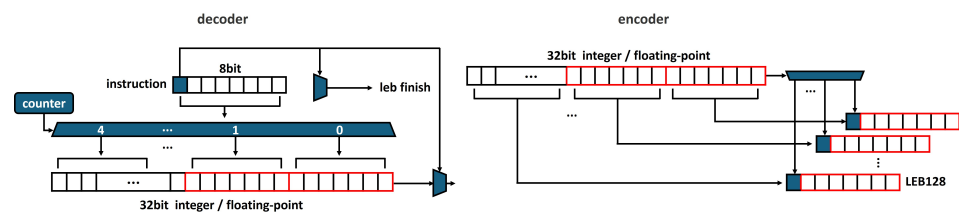


Figure 10. LEB128 decoder and encoder hardware architecture.

3.2. Memory Part

The WASM accelerator handles two primary data types: instruction data and input/output data. To efficiently process these, we adopted the Harvard Architecture as shown in Figure 11, which separates instruction memory from data memory. This architecture allows the independent handling of instructions and data, reducing bottlenecks and achieving faster processing speeds compared to the Von Neumann architecture.

The instruction memory is implemented with SRAM, which stores the WASM bytecode received during execution. Conversely, the data memory, designed with register memory, manages the internal data required during computations. It is designed as Last-In-First-Out (LIFO), facilitating the stack-based operations typical of WASM execution. The data memory employs a read-write strategy to optimize memory usage. This means that once a data value is read, the same slot can be overwritten with new data, saving memory space. Additionally, the data memory is highly configurable, making it adaptable to various data sizes and enhancing the overall flexibility of the WASM accelerator. Figure 11 visually explains the data memory architecture of the WASM accelerator, depicted as a Last-In-First-Out (LIFO) stack structure. It illustrates how stack operations are performed. As shown, when data are stored, a single value such as *out_data* is saved. When arithmetic instructions like add, mul, or div are processed, two data values, *rs1_data* and *rs2_data*, are retrieved and sent to the ALU or FPU for execution. This architecture not only optimizes memory usage but also simplifies the implementation of stack operations, which are essential for the efficient execution of WASM bytecode.

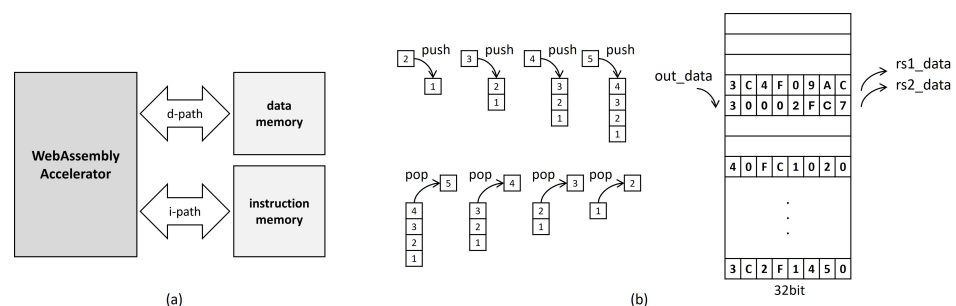


Figure 11. (a) Harvard architecture, (b) stack-based memory operation diagram.

3.3. Operational Flow

The operational flow of the WASM accelerator is demonstrated through VCS (Verilog Compiler Simulator) software simulation using Verdi Version R-2020.12-SP1 for linux64. Figure 12 shows the process where an instruction is input and its validity is checked. As illustrated, the WASM controller fetches instructions from SRAM one byte at a time via the

program counter. During this process, the decoder checks the validity of the instruction, and the check signal ensures that the accelerator continues to operate. If the check signal is not triggered, an error signal is raised, causing the accelerator to halt. In the figure, the check signal is shown to be activated after verifying the magic number and version at the beginning of the WASM bytecode.

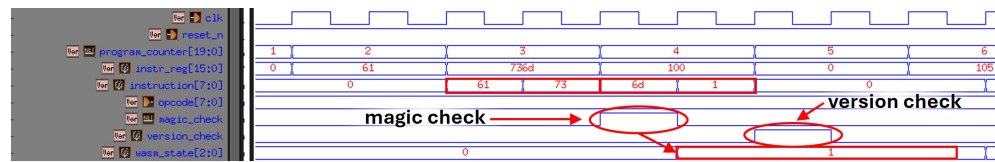


Figure 12. Checking validity of bytecode.

Figure 13 demonstrates the control of sections. After several conditions are met, the section begins execution, receiving both the section type and section number. On the left of the figure, the section type of 8'h01 and section number of 8'h05 are received. Accordingly, the FSM `wasm_state` changes. The `section_cnt` counts the instructions to ensure the section completes as described earlier in Figure 8, storing the section ID in `sec_done` to prevent repeated execution of the same section. The `section_state` FSM transitions upon each new section, and the `sec_done` register checks for any out-of-order or duplicated sections.

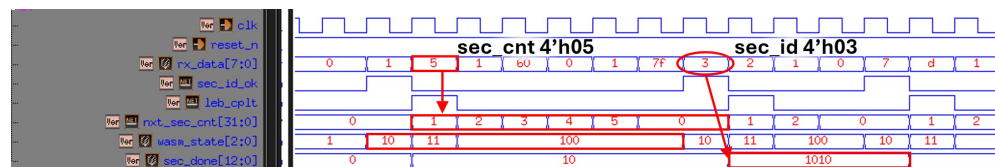


Figure 13. Counting section number and checking section ID.

Figure 14 illustrates the computational aspect. When the `i32.const` instruction (`8'h41`) is received, the following instruction is processed using the LEB128 decoder described in Figure 10, decoding the input value `24'hAF8A01` into `32'd17711`, which is stored in the data memory. Simultaneously, an `i32.add` instruction (`8'h6a`) is received, and the values from the top two addresses of the data memory stack are fetched for computation. The ALU receives the add opcode from the WASM controller, changing the *alu_mode* to execute the addition operation. The system features both an ALU and an FPU, and depending on the operation type, the appropriate unit is selected via a multiplexer (MUX). The result is immediately stored in the data memory. In the figure, the addition of `32'd28657` and `32'd17711` results in `32'd46368`, which is stored in the data memory.

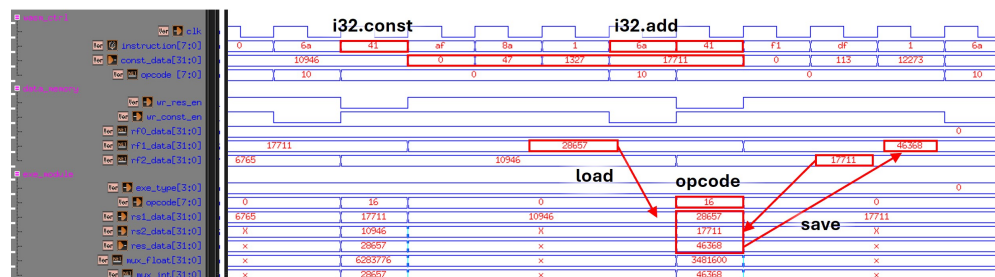


Figure 14. Actual execution in ALU and FPU.

4. Implementation and Experiments

In this section, we present the implementation of our WASM accelerator on an FPGA and describe the experiments conducted to evaluate its performance. Various algorithms are tested to compare the accelerator’s efficiency against traditional software execution methods in embedded environments.

4.1. Environment Setup

To evaluate the performance of the WASM accelerator introduced in this paper, experiments are conducted in two distinct hardware environments: an embedded system and a desktop system. The specific configurations for each environment are as follows.

Embedded System: For the embedded system tests, we utilize a Raspberry Pi 4 Model B. The Raspberry Pi 4 features a 1.5 GHz ARM Cortex-A72 microprocessor, known for its low power consumption and efficient performance, making it a popular choice in embedded systems. The WASM accelerator, designed in Verilog HDL, is implemented on an Altera Cyclone IV EP4CE115F29C7N FPGA, integrated on a DE2-115 board. This FPGA operates at a clock speed of 50 MHz. The accelerator communicates with the Raspberry Pi via an SPI interface, receiving WASM bytecode and returning the processed results. Table 3 summarizes the usage of Look-Up Tables (LUTs) and registers. The hardware implementation uses 6246 LUTs and 1563 registers, ensuring that the accelerator can be integrated into embedded systems without significant hardware overhead.

Table 3. Hardware resource utilization.

Resource	Utilization
LUTs	6246
Regs	1563

Desktop System: For the desktop system tests, we utilize an Intel i7-12700 CPU. While the base clock speed of the CPU is 2.10 GHz, we observe variations in the operating clock speed due to dynamic frequency scaling with hyper-threading enabled. To provide a more consistent representation of the CPU performance during our experiments, we report the average operating clock speed measured across multiple runs, which is approximately 3.1 GHz. The experiments are conducted on an Ubuntu 20.04.2 operating system with kernel version 5.10.16.3. This setup reflects a typical high-performance desktop workload.

In both environments, we measure the performance of various algorithms implemented in C, native code, JavaScript, and WebAssembly, and compare these to the performance of the WASM accelerator. Figure 15 shows the hardware-implemented WASM accelerator utilized in our experiments.

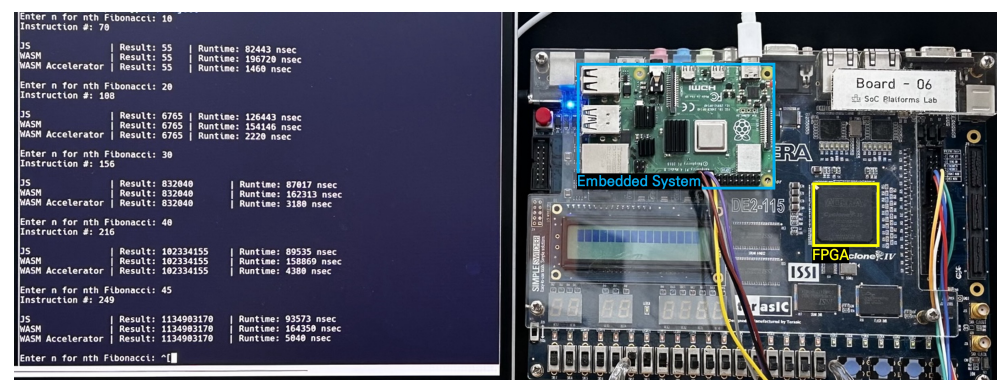


Figure 15. Hardware-implemented WASM accelerator.

4.2. Benchmark Algorithms

To assess the performance of our WASM accelerator, we perform benchmarks across various algorithms. The algorithms chosen for these experiments include Fibonacci sequence calculation, factorial calculation, binomial coefficient calculation, dice path calculation, and matmult_floating calculation. These algorithms are selected for their computational complexity and suitability for performance evaluation. Since our accelerator cannot yet interpret all WASM instructions, we focus on operations that are commonly used in WASM benchmarks and supported by our hardware. Therefore, the experiments

are specifically conducted using these five algorithms to ensure compatibility with the accelerator's current instruction set.

Each algorithm is implemented in four different forms for comparative analysis:

- **Native Code:** Native code refers to the machine code that is directly executed by the underlying hardware. It is compiled from C using GCC 9.4.0, generating platform-specific binaries. This code runs natively on the hardware architecture, such as ARM-based embedded systems (e.g., Raspberry Pi) or Intel-based desktop systems, without requiring any abstraction layers.
- **C Code:** This refers to the original platform-independent source code written in C. The C code is adaptable and can be compiled for various architectures, such as ARM or x86, using the appropriate compiler. Although the source code remains consistent across platforms, it must be compiled into native (machine) code to execute on specific hardware, making it highly reusable across different system environments.
- **JavaScript:** The algorithms are implemented in JavaScript and executed with the V8 JavaScript engine, known for its high-performance JavaScript execution and support for WASM.
- **WASM:** The WASM version is compiled and executed using the V8 engine, aiming to evaluate the performance benefits of WASM.

The experiments involve gradually increasing the input size (n) for each algorithm and measuring the computation time required. For example, in the Fibonacci sequence calculation, we start with $n = 1$ and increase the value incrementally, recording the computation time for each n . To ensure reliable results, each computation is repeated 10 times, and the average computation time is calculated. This method helps to reduce measurement errors due to fluctuating resource usage and temporary loads, enhancing the reliability of the results.

Table 4 summarizes the computation time for each algorithm in the desktop environment. As seen in the table, JavaScript and WASM generally exhibit slower performance compared to native code and C code. However, WASM shows approximately four times faster performance than JavaScript on average in the desktop environment, indicating its strong performance capabilities in this context. These results suggest that the WASM has the potential to operate efficiently in desktop.

Table 4. Performance evaluation of WebAssembly on the desktop system.

Input Dataset	Block Size	Desktop			
		Native	C	JavaScript	WASM
fibonacci	10	5.92	12.11	33,620.42	10,006.12
	20	5.63	16.34	42,200.41	11,052.30
	30	6.18	17.84	50,260.45	13,310.02
factorial	5	5.74	11.89	22,484.48	10,231.46
	10	5.89	13.11	30,221.84	11,987.33
	15	5.99	15.82	36,989.42	11,500.35
Binomial Coefficient	5	5.84	12.00	25,784.51	10,142.91
	10	5.71	13.54	35,973.13	10,147.48
	15	5.82	14.03	39,778.32	10,436.32
dyck path	3	5.67	11.68	38,536.48	10,052.13
	6	5.82	12.89	47,410.69	10,479.12
	9	5.96	13.94	76,489.12	11,978.86
matmult_float	3	5.40	13.78	95,763.18	11,331.58
	5	5.46	15.12	360,093.33	14,680.00
	7	5.72	19.42	882,115.79	14,766.67

Note: All operation results are expressed in nanoseconds [ns].

Table 5 provides a summary of the computation time for each algorithm in the embedded environment. Due to the resource constraints of embedded systems, the overall performance is slower compared to the desktop environment. This is expected given the limited CPU performance, memory bandwidth, and power consumption of embedded platforms like the ARM-based Raspberry Pi 4 Model B. In the embedded environment, both JavaScript and WASM perform slower than C code and native code. Notably, WASM shows the slowest performance among the four forms, contrasting with the desktop results and reflecting the optimization challenges for WASM in embedded systems as discussed in Section 2. Unlike the other algorithms, which display variable computation times depending on the data size or complexity of operations, WASM exhibits consistently similar execution times regardless of workload. This uniformity can be attributed to the compact WASM bytecode format, which minimizes overhead related to data size. However, the significant delays observed in the overall WASM performance are likely a result of the embedded system's unoptimized runtime, where much of the time is spent in runtime processes such as profiling and compilation rather than the actual computation itself. Consequently, this leads to consistently high execution times, negating some of the efficiency benefits typically associated with WASM.

Table 5. Performance evaluation of WebAssembly on embedded system.

Input Dataset	Block Size	Embedded System					Accelerated
		Native	C	JavaScript	WASM	Accelerator	
fibonacci	10	101.79	111.21	65,137.12	162,588.32	1460	111.36×
	20	123.84	241.32	90,254.64	168,459.85	2220	72.88×
	30	152.05	259.14	95,227.13	165,918.36	3180	52.18×
factorial	5	103.21	111.87	47,253.12	157,486.08	1160	135.94×
	10	125.84	166.48	51,636.48	163,451.45	1500	108.97×
	15	148.15	352.16	75,122.19	167,497.34	1760	94.46×
Binomial Coefficient	5	110.18	146.64	67,843.37	175,305.13	1760	99.61×
	10	139.39	166.22	67,945.16	184,706.69	2300	80.31×
	15	203.84	537.91	74,349.51	209,022.48	2740	76.29×
dyck path	3	81.84	93.13	67,229.15	164,796.35	1160	142.07×
	6	112.98	389.09	97,853.02	167,002.48	1520	109.87×
	9	207.58	462.18	151,517.64	170,299.35	1880	90.58×
matmult_float	3	105.37	220.16	206,305.26	180,813.26	5820	31.07×
	5	119.84	426.31	758,164.32	266,517.47	30,460	8.75×
	7	135.63	731.15	2,143,623.21	400,621.11	83,580	4.79×

Note: All operation results are expressed in nanoseconds [ns].

The right side of the table displays the performance results of our designed WASM accelerator. The experiments demonstrate that the WASM accelerator processes the algorithms at remarkable speeds, achieving up to 142 times faster performance compared to traditional WASM execution in embedded system. Particularly in embedded systems, the performance improvements allow for high-performance computations in resource-constrained environments, significantly expanding the practical applications of WASM. Moreover, this acceleration surpasses even the computation speeds achieved on desktop systems, suggesting that the WASM accelerator can also be utilized in desktop environments.

5. Conclusions

This study presents the design and validation of a novel hardware accelerator aimed at enhancing the performance of WASM in resource-constrained embedded systems. Traditional software-based WASM runtimes introduce significant overhead through interpretation, compilation, and profiling processes, leading to performance bottlenecks. To address these issues, our accelerator leverages the characteristics of WASM, which shares similarities with machine code, to achieve significant performance improvements. By

directly interpreting and executing WASM bytecode, our accelerator minimizes the overhead typically associated with traditional CPU execution paths. We design the accelerator using Verilog HDL, implemented it on an FPGA, and propose an architecture that decodes WASM bytecode directly and processes it efficiently, bypassing these software-based conversion steps.

Our experimental results demonstrate that the proposed accelerator achieves up to a 142-fold performance improvement for certain algorithms compared to traditional software-based WASM runtimes. Additionally, the accelerator outperforms desktop-level WASM implementations, suggesting its potential applicability not only in constrained environments but also in high-performance systems. This performance gain indicates that WASM-based applications can be effectively used in various embedded environments, such as IoT devices, industrial control systems, and autonomous vehicles.

Moreover, the accelerator processes all received bytecode independently of the CPU and employs a hardware-based isolation mechanism to prevent WASM code from accessing system memory or interfering with other programs. This offers stronger security than conventional software sandboxes while leveraging the fast processing speed of hardware. For instance, the accelerator can perform complex security checks, such as the real-time analysis of executing code to detect malicious patterns, without causing performance degradation.

As the use of WASM continues to expand, particularly in IoT and other embedded systems, efficient interaction with the internet and faster data processing will become increasingly important. Our accelerator addresses these demands by enabling more efficient communication and rapid processing in embedded systems, expanding the potential use cases of WASM across various domains.

Currently, the accelerator supports only 36 instructions, but expanding the instruction set will allow it to handle more complex programs and algorithms. This expansion could further increase the utility of the accelerator across a wider range of applications, making it capable of performing more intricate tasks. In conclusion, this study demonstrates the effectiveness of the WASM hardware accelerator in solving performance issues in embedded systems, highlighting the potential for broader WASM adoption as the technology continues to evolve.

6. Discussion

The proposed WASM hardware accelerator offers a promising approach to improving WASM execution performance in embedded systems. However, there are several challenges that need to be addressed before the accelerator can be applied effectively in real-world IoT environments.

First, our current accelerator supports only 32-bit integer (i32) and 32-bit floating-point (f32) operations. Expanding support to 64-bit data types (i64, f64) would enable the handling of larger numerical ranges. Additionally, the current version of the accelerator supports only 36 WASM instructions as shown in Figure 6. Expanding the instruction set is crucial for broader application. To facilitate this, we have parameterized the instruction handling to simplify future updates. This modular design allows for rapid adaptation to new features or improvements in WASM.

Second, while WASM is increasingly used in combination with JavaScript, our accelerator is currently designed to process only algorithms written entirely in WASM and does not support interaction with JavaScript. To ensure seamless WASM performance in web browsers, it is essential to develop a Software Development Kit (SDK) that enables direct interaction between WASM bytecode and the browser. Integrating the hardware accelerator with software-based WASM runtimes could foster a more dynamic WASM ecosystem, potentially establishing WASM as a key technology in embedded systems.

Furthermore, our accelerator is implemented on an FPGA running at 50 MHz. As shown in Table 5 with the `matmult_float` benchmark, systems with long instruction lengths tend to see fewer performance gains at this frequency compared to those with shorter instructions. Thus, the primary target for our accelerator is smaller, resource-constrained

environments. However, with further development and the ability to operate at higher frequencies, the accelerator could offer benefits even in larger and more complex systems.

In terms of compatibility with other emerging web technologies, such as WebGPU and WebRTC, the proposed accelerator currently focuses solely on accelerating WASM bytecode execution and does not natively support these technologies. WebGPU and WebRTC have distinct requirements, including high-performance graphics processing and real-time communication, respectively, which fall outside the current design scope. However, there is potential for future extensions that could integrate support for these technologies by developing additional hardware modules or software interfaces. Such advancements would require substantial further research and development but could enhance the versatility and applicability of the accelerator in more diverse embedded system contexts.

Through continued research and development, we aim to enhance the performance, functionality, and practicality of the proposed WASM hardware accelerator. These ongoing efforts are expected to expand the application of WASM in embedded systems and contribute to the advancement of the broader industry ecosystem.

Author Contributions: Conceptualization, J.K.; methodology, J.K. and R.K.; software, R.K. and J.O.; hardware, J.K., R.K. and J.O.; validation, J.K. and R.K.; formal analysis, J.O.; investigation, J.K.; data curation, J.O.; writing—original draft preparation, J.K.; writing—review and editing, J.K., R.K. and J.O.; visualization, R.K.; supervision, S.E.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2024-RS-2022-00156295) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation).

Data Availability Statement: The data presented in this study are available in this article.

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

Abbreviations

The following abbreviations are used in this manuscript:

WASM	WebAssembly
IoT	Internet of Things
IoV	Internet of Vehicles
DOM	document object model
AJAX	asynchronous JavaScript and XML
JIT	Just-in-Time
FPGA	field programmable gate array
WAT	WASM text format
AOT	ahead-of-time
SPI	serial peripheral interface
FSM	finite state machine
SRAM	static random access memory
ALU	arithmetic logic unit
FPU	floating point unit
LEB128	Little Endian Base 128
MSB	most significant bit
LIFO	Last-In-First-Out
SDK	software development kit
LZ4	Lempel-Ziv4

References

1. Crockford, D. *How JavaScript Works*; Virgule-Solidus: San Jose, CA, USA, 2018.
2. Southern, G.; Renau, J. Overhead of deoptimization checks in the V8 javascript engine. In Proceedings of the 2016 IEEE International Symposium on Workload Characterization (IISWC), Providence, RI, USA, 25–27 September 2016; pp. 1–10. [\[CrossRef\]](#)
3. Haas, A.; Rossberg, A.; Schuff, D.L.; Titzer, B.L.; Holman, M.; Gohman, D.; Wagner, L.; Zakai, A.; Bastien, J. Bringing the web up to speed with WebAssembly. In Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, Barcelona Spain, 18–23 June 2017; Volume 52. [\[CrossRef\]](#)
4. De Macedo, J.; Abreu, R.; Pereira, R.; Saraiva, J. On the Runtime and Energy Performance of WebAssembly: Is WebAssembly superior to JavaScript yet? In Proceedings of the 2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW), Melbourne, VI, Australia, 15–19 November 2021; pp. 255–262. [\[CrossRef\]](#)
5. Mohan, B.R. Comparative Analysis of JavaScript and WebAssembly in the Browser Environment. In Proceedings of the 2022 IEEE 10th Region 10 Humanitarian Technology Conference (R10-HTC), Hyderabad, India, 16–18 September 2022; pp. 232–237. [\[CrossRef\]](#)
6. Romano, A.; Wang, W. WasmView: Visual Testing for WebAssembly Applications. In Proceedings of the 2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), Seoul, Republic of Korea, 27 June–19 July 2020; pp. 13–16.
7. De Macedo, J.; Abreu, R.; Pereira, R.; Saraiva, J. WebAssembly versus JavaScript: Energy and Runtime Performance. In Proceedings of the 2022 International Conference on ICT for Sustainability (ICT4S), Plovdiv, Bulgaria, 13–17 June 2022; pp. 24–34. [\[CrossRef\]](#)
8. Pillay, P.; Viriri, S. Foresight: Real Time Facial Detection and Recognition Using WebAssembly and Localized Deep Neural Networks. In Proceedings of the 2019 Conference on Information Communications Technology and Society (ICTAS), Durban, South Africa, 6–8 March 2019; pp. 1–6. [\[CrossRef\]](#)
9. Choi, D.Y.; Oh, J.H.; Kim, J.K.; Lee, S.E. Energy Efficient and Low-Cost Server Architecture for Hadoop Storage Appliance. *KSII Trans. Internet Inf. Syst.* **2020**, *14*, 4648–4663. [\[CrossRef\]](#)
10. Scheidl, F. Valent-Blocks: Scalable High-Performance Compilation of WebAssembly Bytecode For Embedded Systems. In Proceedings of the 2020 International Conference on Computing, Electronics & Communications Engineering (iCCECE), Online, 17–18 August 2020; pp. 119–124. [\[CrossRef\]](#)
11. Oh, H.W.; Kim, J.K.; Hwang, G.B.; Lee, S.E. The Design of a 2D Graphics Accelerator for Embedded Systems. *Electronics* **2021**, *10*, 469. [\[CrossRef\]](#)
12. Mendki, P. Evaluating Webassembly Enabled Serverless Approach for Edge Computing. In Proceedings of the 2020 IEEE Cloud Summit, Harrisburg, PA, USA, 21–22 October 2020; pp. 161–166. [\[CrossRef\]](#)
13. Wallentowitz, S.; Kersting, B.; Dumitriu, D.M. Potential of WebAssembly for Embedded Systems. In Proceedings of the 2022 11th Mediterranean Conference on Embedded Computing (MECO), Budva, Montenegro, 7–11 June 2022; pp. 1–4. [\[CrossRef\]](#)
14. Wasmer. Available online: <https://wasmer.io/> (accessed on 23 June 2024).
15. Peach, G.; Pan, R.; Wu, Z.; Parmer, G.; Haster, C.; Cherkasova, L. eWASM: Practical Software Fault Isolation for Reliable Embedded Devices. *IEEE Trans.-Comput.-Aided Des. Integr. Circuits Syst.* **2020**, *39*, 3492–3505. [\[CrossRef\]](#)
16. Ménétrey, J.; Pasin, M.; Felber, P.; Schiavoni, V. Twine: An Embedded Trusted Runtime for WebAssembly. In Proceedings of the 2021 IEEE 37th International Conference on Data Engineering (ICDE), Chania, Greece, 19–22 April 2021; pp. 205–216. [\[CrossRef\]](#)
17. Gurdeep Singh, R.; Scholliers, C. WARDuino: A dynamic WebAssembly virtual machine for programming microcontrollers. In Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, Athens, Greece, 21–22 October 2019; MPLR 2019, pp. 27–36. [\[CrossRef\]](#)
18. Moron, K.; Wallentowitz, S. Support for Just-in-Time Compilation of WebAssembly for Embedded Systems. In Proceedings of the 2023 12th Mediterranean Conference on Embedded Computing (MECO), Budva, Montenegro, 6–10 June 2023; pp. 1–4. [\[CrossRef\]](#)
19. Kim, M.; Jang, H.; Shin, Y. Avengers, Assemble! Survey of WebAssembly Security Solutions. In Proceedings of the 2022 IEEE 15th International Conference on Cloud Computing (CLOUD), Barcelona, Spain, 10–16 July 2022; pp. 543–553. [\[CrossRef\]](#)
20. Stiévenart, Q.; Roover, C.D. Compositional Information Flow Analysis for WebAssembly Programs. In Proceedings of the 2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM), Online, 27–28 September 2020; pp. 13–24. [\[CrossRef\]](#)
21. Zakai, A. Emscripten: An LLVM-to-JavaScript compiler. In Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, Chicago, IL, USA, 17–22 October 2011; OOPSLA '11, pp. 301–312. [\[CrossRef\]](#)
22. Spies, B.; Mock, M. An Evaluation of WebAssembly in Non-Web Environments. In Proceedings of the 2021 XLVII Latin American Computing Conference (CLEI), Cartago, Costa Rica, 25–29 October 2021; pp. 1–10. [\[CrossRef\]](#)
23. Gackstatter, P.; Frangoudis, P.A.; Dustdar, S. Pushing Serverless to the Edge with WebAssembly Runtimes. In Proceedings of the 2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid), Taormina, Italy, 16–19 May 2022; pp. 140–149. [\[CrossRef\]](#)

24. Gadepalli, P.K.; Peach, G.; Cherkasova, L.; Aitken, R.; Parmer, G. Challenges and Opportunities for Efficient Serverless Computing at the Edge. In Proceedings of the 2019 38th Symposium on Reliable Distributed Systems (SRDS), Lyon, France, 1–4 October 2019; pp. 261–2615. [\[CrossRef\]](#)
25. Krylov, G.; Jelenković, P.; Thom, M.; Dueck, G.W.; Kent, K.B.; Manton, Y.; Maier, D. Ahead-of-time compilation in eclipse OMR on example of WebAssembly. In Proceedings of the 31st Annual International Conference on Computer Science and Software Engineering, Toronto, ON, Canada, 22–25 November 2021; CASCON '21, pp. 237–243.
26. Zhang, Y.; Cao, S.; Wang, H.; Chen, Z.; Luo, X.; Mu, D.; Ma, Y.; Huang, G.; Liu, X. Characterizing and Detecting WebAssembly Runtime Bugs. *ACM Trans. Softw. Eng. Methodol.* **2023**, *33*, 37. [\[CrossRef\]](#)
27. Wasm3. Available online: <https://github.com/wasm3/wasm3> (accessed on 23 June 2024).
28. V8. Available online: <https://v8.dev/> (accessed on 23 June 2024).
29. Wasmtime. Available online: <https://wasmtime.dev/> (accessed on 23 June 2024).
30. Wang, Z.; Wang, J.; Wang, Z.; Hu, Y. Characterization and Implication of Edge WebAssembly Runtimes. In Proceedings of the 2021 IEEE 23rd Int Conf on High Performance Computing & Communications; 7th Int Conf on Data Science & Systems; 19th Int Conf on Smart City; 7th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys), Haikou, China, 20–22 October 2021; pp. 71–80. [\[CrossRef\]](#)
31. Klehm, V.D.S.; Sardinha, E.D.; Junior, V.F.d.L.; Mendonca-Neto, R.; Cordovil, L. A comparative analysis between Lua interpreter variants compiled to WASM, JavaScript and native. In Proceedings of the 2024 IEEE International Conference on Consumer Electronics (ICCE), Las Vegas, Nevada, USA, 6–8 January 2024; pp. 1–5. [\[CrossRef\]](#)
32. Lee, S.; An, S.; Kim, J.; Namkung, H.; Park, J.; Kim, R.; Lee, S.E. Grid-Based DBSCAN Clustering Accelerator for LiDAR's Point Cloud. *Electronics* **2024**, *13*, 3395. [\[CrossRef\]](#)
33. Li, B.; Dong, W.; Gao, Y. WiProg: A WebAssembly-based Approach to Integrated IoT Programming. In Proceedings of the IEEE INFOCOM 2021—IEEE Conference on Computer Communications, Vancouver, BC, Canada, 10–13 May 2021; pp. 1–10. [\[CrossRef\]](#)
34. Gadepalli, P.K.; McBride, S.; Peach, G.; Cherkasova, L.; Parmer, G. Sledge: A Serverless-first, Light-weight Wasm Runtime for the Edge. In Proceedings of the 21st International Middleware Conference, Online, 7–11 December 2020; Middleware '20, pp. 265–279. [\[CrossRef\]](#)
35. Jiang, B.; Chen, Y.; Wang, D.; Ashraf, I.; Chan, W. WANA: Symbolic Execution of Wasm Bytecode for Extensible Smart Contract Vulnerability Detection. In Proceedings of the 2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS), Hainan, China, 6–10 December 2021; pp. 926–937. [\[CrossRef\]](#)
36. An, S.; Oh, J.; Lee, S.; Kim, J.; Jeong, Y.; Kim, J.; Lee, S.E. Lightweight and Error-Tolerant Stereo Matching with a Stochastic Computing Processor. *Electronics* **2024**, *13*, 2024. [\[CrossRef\]](#)
37. Oh, H.W.; An, S.; Jeong, W.S.; Lee, S.E. RF2P: A Lightweight RISC Processor Optimized for Rapid Migration from IEEE-754 to Posit. In Proceedings of the 2023 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED), Vienna, Austria, 7–8 August 2023; pp. 1–6. [\[CrossRef\]](#)
38. Rossberg, A. WebAssembly Specification. Available online: <https://webassembly.github.io/spec/> (accessed on 28 April 2024).
39. Park, J.; Shin, J.; Kim, R.; An, S.; Lee, S.; Kim, J.; Oh, J.; Jeong, Y.; Kim, S.; Jeong, Y.R.; et al. Accelerating Strawberry Ripeness Classification Using a Convolution-Based Feature Extractor along with an Edge AI Processor. *Electronics* **2024**, *13*, 344. [\[CrossRef\]](#)
40. Stiévenart, Q.; Binkley, D.W.; De Roover, C. Static stack-preserving intra-procedural slicing of webassembly binaries. In Proceedings of the 44th International Conference on Software Engineering, Pittsburgh, PA, USA, 25–27 May 2022; ICSE '22, pp. 2031–2042. [\[CrossRef\]](#)

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.