

Article

Design of a Configurable Five-Stage Pipeline Processor Core Based on RV32IM

Yiyang Chang, Yiming Liu, Chong Peng, Jiarui Guo and Yi Zhao * 

State Key Laboratory of Integrated Optoelectronics, College of Electronic Science and Engineering, Jilin University, Changchun 130012, China; changyy20@mails.jlu.edu.cn (Y.C.); yimingl20@mails.jlu.edu.cn (Y.L.); pengchong21@mails.jlu.edu.cn (C.P.); guojr21@mails.jlu.edu.cn (J.G.)

* Correspondence: yizhao@jlu.edu.cn; Tel.: +86-130-8914-8660

Abstract: With the rapid development of the electronics industry, the scale of the global Internet of Things (IoT) industry has shown an exponential growth trend in recent years. The huge demand for IoT equipment makes low cost an important indicator for the sustainable operation of the entire IoT system. However, IoT chips also require a certain amount of performance to perform complex tasks. Aiming at the above contradiction between performance and cost, this paper proposes a configurable five-stage pipeline processor core based on RV32IM. The proposed processor core has multiple configurable modules to suit different application scenarios. In low-power mode, the proposed architecture implements only an RV32I subset, while in high-performance mode, integer division and multiplication extensions are added. Meanwhile, the processor core will also support super and user privilege levels and is equipped with CSR (Control and Status Registers). The module-level and system-level simulations of the proposed architecture are completed using a fully open-source workflow based on verilator and gtkwave. In addition, the design was prototyped and verified with FPGA. The proposed processor outperforms the performance of the classic MCU-CortexM3.

Keywords: RISC-V ISA; processor core; RV32IM chip; IoT



Citation: Chang, Y.; Liu, Y.; Peng, C.; Guo, J.; Zhao, Y. Design of a Configurable Five-Stage Pipeline Processor Core Based on RV32IM. *Electronics* **2024**, *13*, 120. <https://doi.org/10.3390/electronics13010120>

Academic Editor: Alexander Barkalov

Received: 4 December 2023

Revised: 20 December 2023

Accepted: 21 December 2023

Published: 28 December 2023



Copyright: © 2023 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

The swift advancement of the electronics industry has brought about a remarkable transformation in our daily lives, as the widespread implementation of informatization and intelligence has significantly augmented the quality of our existence. In particular, the exponential growth of the Internet of Things (IoT) has led to a remarkable surge in the deployment of connected smart devices. Ranging from smart homes to industrial automation systems, IoT is predicted to achieve the staggering milestone of connecting a mind-boggling 500 billion devices to the internet by 2030 [1–3]. Such a large number of devices makes low cost an important indicator for the sustainable operation of the entire IoT system. Therefore, as the main source of the cost of smart devices, the price of the processor often determines the stability of the entire IoT system. However, blindly reducing the cost of the processor will also cause many problems. The IoT represents a significant departure from the conventional single-scenario-oriented smart devices of the past. On the one hand, low-cost processors are the first choice for IoT devices that have a large number base but only perform a single task. On the other hand, when it comes to certain IoT devices that necessitate human–computer interactions, the requirement for a simplistic operating system is often indispensable. As a result, the processors of IoT devices will have to be equipped with logic units, such as an MMU (Memory management unit), at some point, which will lead to an increase in cost.

A configurable processor core is a potential solution to balance cost and performance.

The processor core can choose different configurations for different application scenarios. For basic application scenarios, which only perform a single repeat operation, the complex logic modules and high-performance memory will be eliminated from the

processor core to optimize energy efficiency and cost-effectiveness. Contrary to the above, when facing complex application scenarios, high-performance modules will be reserved to handle complex tasks. Through this design method, the processor core can adapt to various applications in the IoT without large-scale modifications of the project.

For such a design, which contains different modes for various application scenarios, the RISC-V (Reduced Instruction Set Computer-V) ISA [4] is one of the best potential candidates due to its customizability, scalability, and open-source feature. Benefiting from the modular design, the RISC-V ISA makes the processor suitable for use in a variety of devices. For uncomplicated devices such as microcontrollers, utilizing solely the fundamental RISC-V instruction set can lead to remarkably frugal power consumption and economical costs [5–8]. In high-performance domains such as supercomputing, the RISC-V ISA has a series of scalable subsets and can be customized with specialized instructions for specific tasks, which enables processors based on RISC-V to exhibit excellence in high-performance fields [9–11].

Hence, in this paper, we designed a configurable five-stage pipeline processor core based on RV32IM, which aimed at obtaining a processor that balances cost and performance. The design incorporates the “I” (base integer implementation) and “M” (the integer multiplication and division extension) of the RISC-V ISA. The processor core has multiple configurable modules to adapt to different application scenarios. In order to adapt to some simple micro-control applications in the Internet of Things, the processor does not need complex logical operation units and extremely high-speed and large-capacity storage architecture. For this application scenario, the most practical indicator is the low power consumption characteristics of the processor. In some more complex IoT application scenarios, such as running a simple operating system, a high-speed and large-capacity storage architecture and a logical operation unit that can handle some complex problems are indispensable options for the processor. Therefore, in order to achieve a balance of versatility in the above two application modes, the processor has two modes: low power consumption and high performance. In low power mode, a non-standard extension is added to the base integer instruction set, while there is no multiplication or division unit added to the core. In high-performance mode, the integer multiplication and division extension is added. Meanwhile, the processor core will also support the super and user privilege levels and is equipped with CSRs (Control and Status Registers). After determining the task scenario, the processor can be configured according to the specified parameters. It is worth noting that the low-power and high-performance modes here are not static. Whether it is the integer multiplication and division operation unit or the cache of different performances, they can be configured independently.

The purpose of this project is mainly to propose a general solution that can adapt to complex application scenarios of the Internet of Things, targeting low power consumption, scenarios that do not require complex calculations, and scenarios that require complex computing scenarios to adapt to different chip architectures to achieve the purpose of adapting to the diversity of the IoT market.

2. RISC-V ISA Features

The RISC-V is an open-source instruction set architecture (ISA) that is designed to be simple, modular, and extensible. It is based on the Reduced Instruction Set Computer (RISC) principles, which prioritize a smaller set of instructions that can be executed quickly and efficiently. The RISC-V ISA was developed in the Computer Science Department at the University of California, Berkeley in 2010 and has gained significant attention and adoption in recent years. This section highlights the salient features of the RISC-V ISA that render it exceptionally suitable for the design of flexible and configurable general-purpose processors.

The primary reason for selecting RISC-V ISA over others is that it is an open-source architecture, which not only renders it free of cost but also provides a transparent, collaborative, and constantly evolving ecosystem of innovation and development. The benefits

of the open-source architecture mentioned above are of the utmost significance, especially for individual developers and small teams who face financial constraints and have limited resources to invest in expensive proprietary technologies. In addition to its ability to sidestep the intricate and costly intellectual property issues associated with traditional commercial instruction sets such as x86 and ARM architectures [12–15], RISC-V also boasts a plethora of advantages over other open-source instruction sets such as OpenRISC, SPARC V8, etc. Compared with other ISAs, the modular and scalable design of the RISC-V ISA makes it highly adaptable to a wide range of computing applications, from embedded systems and IoT devices to high-performance computing and data centers. The inherent modularity of the RISC-V architecture empowers designers with a great level of freedom and flexibility. By adopting a modular approach, a specific subset of instruction sets can be implemented for different functions (along with base integer implementation). At the same time, unnecessary hardware can be cut off at any time to improve design efficiency.

The RISC-V ISA can be generally divided into two categories: the basic integer ISA and the optional extension of the basic ISA. In addition, the optional subset of extensions to the RISC-V ISA can be divided into two parts: standard extensions and non-standard extensions. Generally, a standard extension is a general-purpose subset that has been packaged and can be adopted at any time during the design process without worrying about conflicts with other standard extensions. In contrast, non-standard extensions are usually designed for specific tasks, often designed by developers themselves, and are highly specialized. The high degree of customization mentioned above means that non-standard extensions may conflict with other standard or non-standard extensions. In the processor development process, developers can implement any standard or non-standard extensions according to the needs of the application, so as to realize the great adaptability of the processor to different tasks. There are four standard extensions along with the base integer instructions in the RISC-V ISA. The “M” extension focuses on supporting integer multiplication and division operations. The “A” extension is the standard atomic instruction extension, which focuses on supporting atomic memory operations. The RV32A subset extends the base integer instructions of the RISC-V ISA with additional instructions that provide atomic memory operations. These instructions include atomic load (LR), atomic store-conditional (SC), and atomic memory fence (AMO) instructions. The “F” extension focuses on providing support for the single-precision floating-point arithmetic operations. The “D” extension is the double-precision floating-point extension. The architecture can be collectively referred to as “G”, while the basic integer subset is configured with all four standard extensions (IMAFD) [4,16–19].

Table 1 shows the parameter comparison of the RISC-V architecture and several other popular architectures. It can be seen that, whether comparing the classic traditional architecture or the same open-source architecture, the modular design is the core feature that makes RISC-V stand out. In addition, the RISC-V ISA not only provides extensive support for 32-, 64-, and 128-bit implementations but also boasts the capability to configure privilege levels, which makes it show obvious performance advantages compared with other simple open-source ISAs. As seen in Figure 1, the RISC-V ISA also simplifies instruction encoding and enables unconventional instruction set encoding. In the RISC-V architecture, the indexes of the general-purpose registers required by the instructions (rs1, rs2, and rd) are placed in fixed positions, so the instruction decoder can easily decode the register indexes and then access the general-purpose registers, which effectively reduces the system complexity.

The goal of this work is to develop an IoT-oriented processor core that can be configured. In addition to completing simple addition and subtraction calculation tasks, the processor core also needs the ability to handle some complex calculation tasks. Hence, the 32-bit (RV32) base integer subset (I) and the extension of integer multiplication and division (M) are implemented for this project. It is worth noting that the “M” subset is configurable and the processor core can implement RV32I alone when facing extremely

simple low-power tasks. The architecture of the proposed configurable five-stage pipeline general-purpose processor soft core based on RV32IM is presented in the next section.

Table 1. Comparison of instruction set architecture.

Features	SPARC	ARMv8	MIPs	OpenRISC	RISC-V
Free and Open Extension				✓	✓
32-bit	✓	✓	✓	✓	✓
64-bit	✓	✓	✓		✓
128-bit					✓
Privileged ISA	✓				✓
IEEE 754-2008	✓	✓			✓

31	25 24	20 19	15 14	12 11	7 6	0	
funct7	rs2	rs1	funct3	rd	opcode		R-type
imm[11:0]		rs1	funct3	rd	opcode		I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode		S-type
imm[31:12]				rd	opcode		U-type

Figure 1. RISC-V instruction encoding format [4].

3. Proposed Architecture

This section provides an overview of the design aspects and architecture of the proposed processor core. As illustrated in Figure 2, the processor is implemented with a five-stage pipelined organization, consisting of the following stages: (a) Instruction Fetch and Instruction Decode (IF and ID), (b) Instruction Issue (IS), (c) Execution (EX), (d) Memory Access (MEM), and (e) Write Back (WB). All stages of the processor pipeline are in order. The subsequent discussion will delve into the specific module design of each stage within the pipeline.

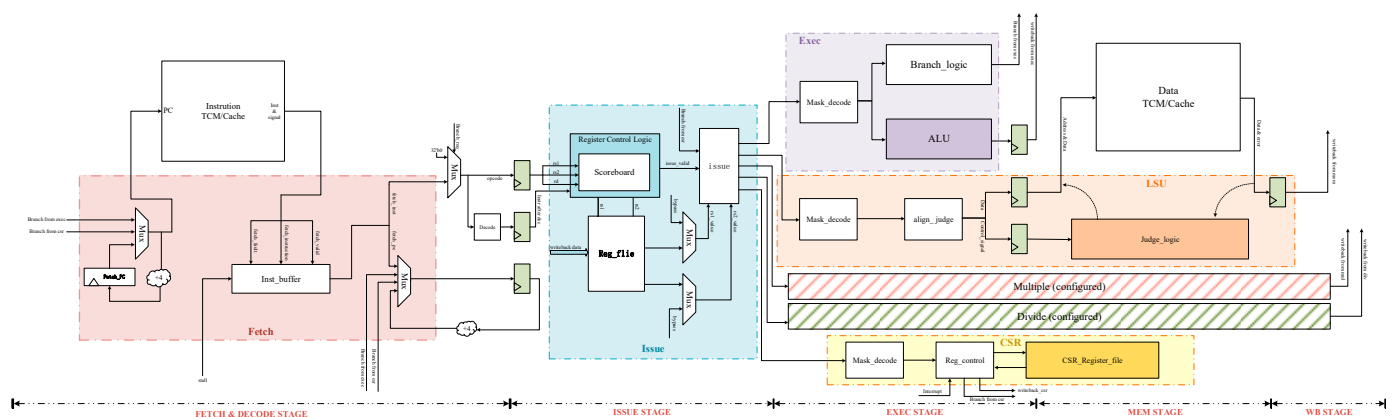


Figure 2. A high-level overview of the proposed processor's micro-architecture. The processor is implemented in a 5-stage pipelined organization.

3.1. Instruction Fetch and Decode (IF and ID)

The IF and ID stage of the microprocessor pipeline is mainly responsible for the fetching and decoding of the instructions. The processed instructions are sent to the lower-level issue module, which then distributes them to each logic unit in the execution stage. In the proposed processor core, the completion of the IF and ID stage is orchestrated by two distinguished functional modules: "FETCH" and "DECODE".

In this design, the “FETCH” module is mainly responsible for executing the operation of fetching instructions from the instruction memory. Since there are two configurable modes of low power consumption and high performance in the proposed architecture, the “FETCH” module has two different connection methods. In the low-power configuration, the ITCM will serve as the instruction memory of the proposed processor to which the “FETCH” module is directly connected. In the high-performance mode, the “FETCH” module will be connected to the MMU to support ICACHE. In fact, the difference in the above connection methods does not impact the functional realization of the “FETCH” module. Therefore, the following explanation will take the case equipped with ITCM as an example to explain the implementation of the “FETCH” module in the proposed architecture.

As illustrated in Figure 2, the “FETCH” module is mainly responsible for fetching instructions from the ITCM and transmitting them to the “DECODE” module for decoding.

In addition, the “FETCH” module also needs to be responsible for the interruption and abnormal operation of the “FETCH & DECODE STAGE”, which is embodied as the branch request from the “CSR” module and the “EXEC” module in the proposed architecture. The workflow of the proposed “FETCH” module is shown in Figure 3. In one clock cycle, if the data paths of the “FETCH” module are clear, the “FETCH” module will send a read request (referred to as Inst_1) to the instruction memory while simultaneously receiving the instruction (Inst_0) requested in the previous cycle from the instruction memory. The “FTECH” module then transfers Inst_0 to the “DECODE” module using the valid-ready handshake mechanism.

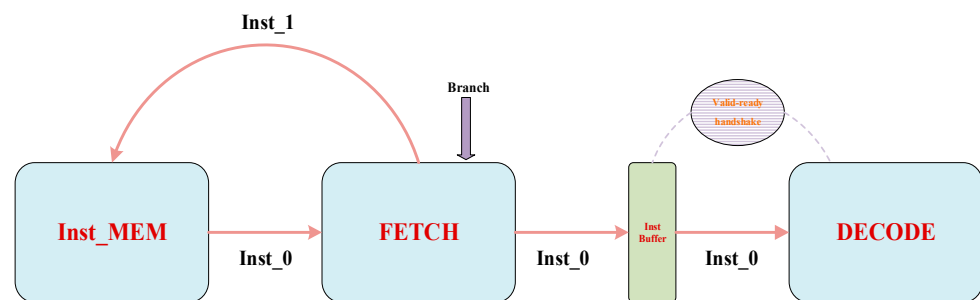


Figure 3. The workflow of the proposed FETCH module.

If the data path is not always clear, it can be blocked in the following situations: (1) the instruction memory fails to promptly return the instruction requested by the “FETCH” module in the previous cycle, as seen in Figure 4a; (2) the “DECODE” module is not ready yet, unable to handshake with the “FETCH” module, as seen in Figure 4b. For situation (1), the “FETCH” module enters the stalling state, during which no data transmission occurs along the entire data path, extending from the instruction memory to the “DECODE” module, which can be seen in cycle 1 of Figure 4a. When the instruction memory successfully returns the instruction in a certain clock cycle, the “FETCH” module restarts and continues to fetch instructions in order, which can be seen in cycles 2 and 3 of Figure 4a. As seen in Figure 4a, there is no data loss during the entire suspension of situation (1). For situation (2), the “FETCH” module also enters the stalling state. However, if the instruction memory returns the instruction in this clock cycle, there exists data transmission along the data path, which extends from the instruction memory to the “FETCH” module, as seen in cycle 1 of Figure 4b. As depicted in cycle 2 of Figure 4b, there will be data loss if the entire data path is restored in the cycle. To avert such a situation, the proposed “FETCH” module incorporates an “Inst-buffer” component, which is shown in Figure 4b. When situation (2) arises, the “Inst-buffer” stores the returned data from the instruction memory. Upon data path restoration, it is transmitted to the “DECODE” module through a handshake, ensuring that data integrity is maintained.

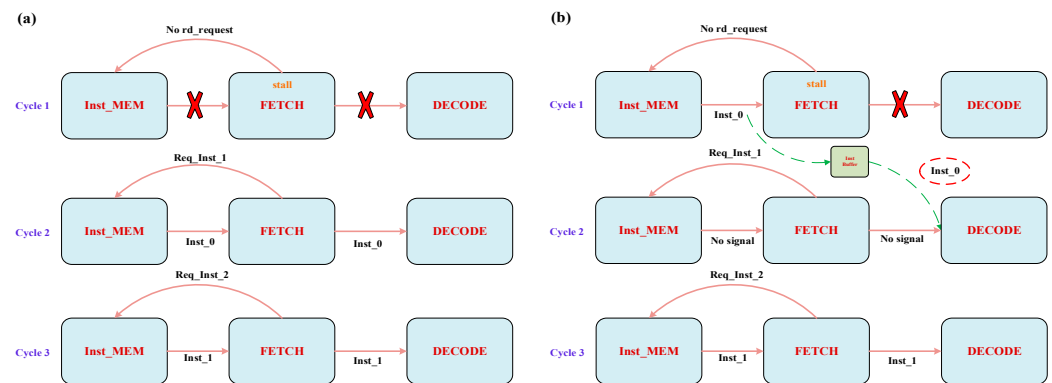


Figure 4. The “FETCH” module stall situation due to (a) an instruction memory reading delay or (b) decoding backpressure.

As mentioned earlier, another important responsibility of the “FETCH” module is to process the branch signals from the “CSR” module and the “EXEC” module in the proposed architecture. The branch signals from the above two modules are generated at the third stage of the pipeline (EXEC STAGE). This means that the “FETCH” module should fetch the target instruction in the third cycle after receiving the branch signal to ensure the orderly execution of tasks on the pipeline.

In the proposed architecture, the “FETCH” module will deliver the retrieved instructions to the “DECODE” module to generate the corresponding control information. The “DECODE” module is tasked with the responsibility of decoding the instructions stored in memory. Its primary function is to furnish the system with the essential information necessary for the accurate execution of the code, while also identifying illegal instructions. As illustrated in Figure 1, the fields in the RISC-V ISA are always encoded in the same place inside the instruction body, which makes the decoding fairly straightforward. The “DECODE” module is entirely realized by combinational logic, wherein the instruction type is ascertained through the utilization of masks specifically tailored for different RISC-V instructions. In addition to the identification and legality judgment of specific instructions, the “DECODE” module also integrates the functionality of the instruction classification. This capability allows for the rough categorization of instructions, enabling the determination of the appropriate post-level functional module to which the instruction should be directed. To sum up, the “DECODE” module will send the generated corresponding information to the “ISSUE” module, in which the information will be used to control the transmission of instructions.

3.2. Instruction Issue (IS)

Upon completion of the IF and ID stage, the decoded instructions are fed into the Instruction Issue (IS) stage. The core focus of this stage is to achieve instruction arbitration and allocation, while also implementing data flow control across the entire pipeline. In the proposed processor core, the completion of the IS stage is orchestrated by the “ISSUE” module.

As shown in Figure 5, the “ISSUE” module consists of two main functional units: the “pipeline ctrl” module and the general register file, along with the branch request generate logic. The “pipeline ctrl” module is invoked by the “ISSUE” module in the proposed architecture. It is responsible for storing the control information emitted by the top-level “ISSUE” module, receiving the information returned by the instructions in the “EXEC”, “MEM”, and “WB” stages. It tracks the status of the instructions and issues signals to squash or stall the pipeline according to the above control signals. The “pipeline ctrl” module primarily achieves the tracking of pipeline states from two aspects: control flow and status flow. In the data flow, the “pipeline ctrl” module receives the computation results returned by instructions at different stages of the pipeline and stores them in registers, which can be seen in Figure 6a. Then, the computation result will be uniformly recorded back to the general register file and CSR register file during the WB (write back) stage. Furthermore,

in the proposed architecture, configurable bypass support has been added for the LOAD and MUL operations to enhance the efficiency of the pipeline execution. This feature is implemented in the data path of the “pipeline ctrl” through data coverage. If the bypass configuration of the processor is valid, the results of the MUL or LOAD operations will be directly forwarded to the data output path within the same cycle instead of being stored until the WB stage. The addition of the bypass avoids the situation that the results are already calculated in the pipeline and that will affect the subsequent instruction issue due to the output delay, which leads to a more efficient data flow and improves the overall throughput of the pipeline.

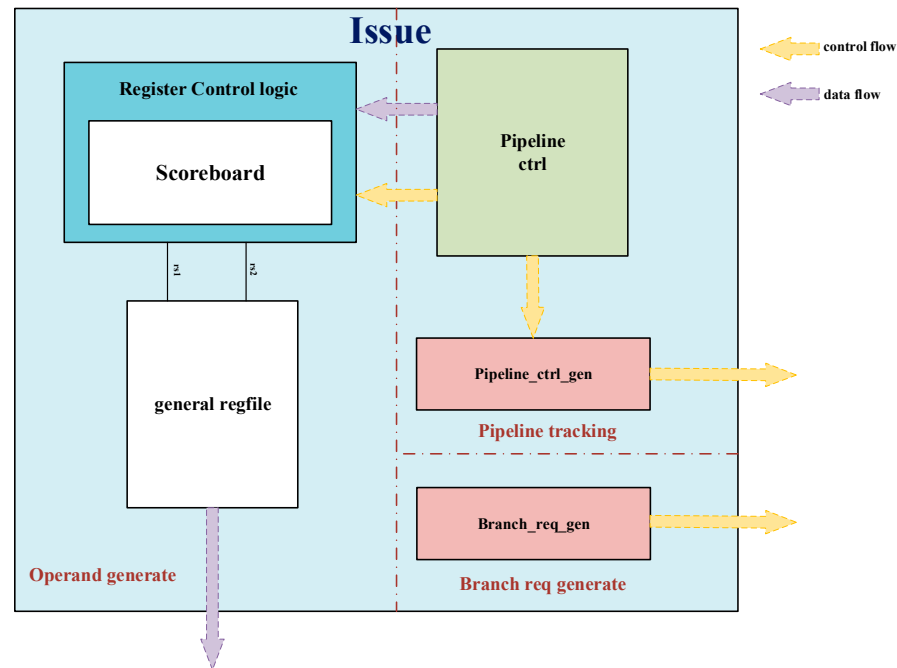


Figure 5. The internal architecture diagram of the “ISSUE” module.

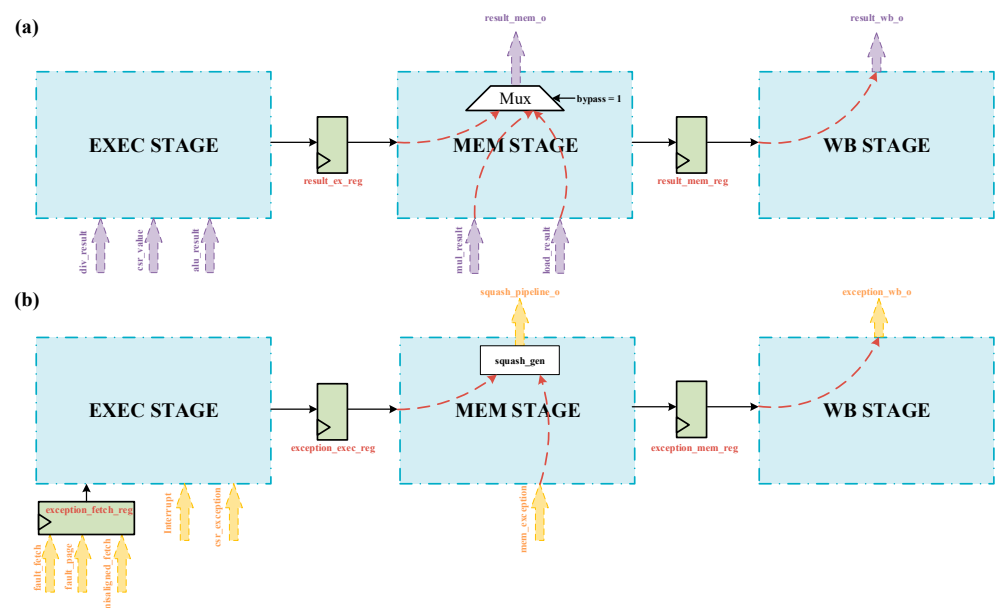


Figure 6. The flow control architecture of the “pipeline ctrl” module: (a) data flow path and (b) control flow path.

Based on the description of Figure 6b, the control flow objectives of the “pipeline ctrl” module mainly involve the following two tasks:

(1) Handling Exceptions and Generating Pipeline Flush Requests (“Squash”):

The module receives and processes exceptional signals returned by instructions at different stages of the pipeline. When an exception occurs, the “pipeline ctrl” generates pipeline flush requests, also known as “Squash,” to clear or invalidate the instructions in the pipeline, preventing incorrect or corrupted results from being committed.

(2) Generating Pipeline Stall Requests (“Stall”):

The “pipeline ctrl” module generates pipeline stall requests, also referred to as “Stall”, based on the processing progress of various modules in the lower stages of the pipeline. These stall requests are used to pause the advancement of new instructions into the pipeline temporarily, ensuring that the pipeline’s stages have sufficient time to complete their current operations before accepting new instructions.

It is worth noting that the data flow and control flow of the pipeline control module described above are interleaved in some cases. This situation mainly exists in the process of writing back to the CSR. For the CSR, an exception is not just a control signal but also data information that needs to be stored, so the exception signal in the control flow needs to be interleaved into the data flow in the write-back stage and then stored in the CSR.

As illustrated in Figure 5, the pipeline control module will deliver the returned control flow and data flow results to the register control logic and pipeline control signal generation logic. Among these components, the “pipeline_ctrl_gen” logic is responsible for broadcasting flush or stall signals to the entire pipeline. On the other hand, the register control logic is tasked with determining whether the corresponding operand register is active, based on the control flow information returned by the “pipeline ctrl” module. Simultaneously, it stores the content of the data flow into the target register. In the proposed architecture, the access control of the general-purpose register file is built around a simple score-boarding mechanism, which keeps track of the status of each physical register. The score board has a total of 32 entries, one for each physical register. It keeps track of each register’s usage as well as the location of the latest data.

Alongside the “pipeline ctrl” module and the general-purpose register file, another crucial component of the “ISSUE” module is the “branch request generate logic”. This logic is implemented using pure combinational logic. Under its control, the “ISSUE” module receives branch requests from both the “EXEC” module and the “CSR” module. Simultaneously, it forwards the target PC address and target privilege level required for the branch jump.

3.3. Back-End [Execution (EXEC)/Memory Access (MEM)/Write Back (WB)]

In the “ISSUE” stage, the instructions are distributed to the functional modules of the subsequent stages in an orderly manner. Since the running time of these functional modules spans the last three stages of the entire pipeline, explaining according to the pipeline stages will lead to the separation of the functional modules. To provide a clearer understanding of the pipeline’s working mechanism in the proposed architecture, the last three stages of the five-stage pipeline (“EXEC” stage, “MEM” stage, and “WB” stage) are consolidated into the “back-end” for explanation.

In the proposed architecture, the back-end consists of five distinct functional units, which will be explained in the following sections.

3.3.1. EXEC Module

In the proposed architecture, the “EXEC” module is responsible for the following two functions:

Executing integer computational instructions in the RISC-V ISA.

Resolving all branch instructions and generating the target address and target privilege level of the branch jump.

For the first function, an Arithmetic Logic Unit (ALU) is integrated into the “EXEC” module. In order to maintain the consistency of the pipeline, the results calculated by ALU through combinational logic will be stored for one beat and then sent to the data path. For the second function, the “EXEC” module directly implements the received branch instruction with combinational logic and issues the result in the current cycle, thereby reducing the number of invalid instruction fetches and improving pipeline efficiency.

3.3.2. MUL Module

In the proposed architecture, the “MUL” module is responsible for implementing the “M” standard extension for integer multiplication of the RISC-V ISA. Without any beating processing, the “MUL” module will return the calculation result within one cycle. In order to match the pipeline, the result will be delayed by two cycles and then delivered to the data path. This module is configurable, allowing it to be removed from the design when pursuing objectives such as a small area and low power consumption. Moreover, the proposed processor offers bypass support for this module, ensuring smoother data flow and minimizing pipeline stalls.

3.3.3. DIV Module

In the proposed architecture, the “DIV” module is responsible for implementing the “M” standard extension for integer division of the RISC-V ISA. In the proposed processor, the “DIV” module is implemented using a standard shift-divider, which means that the division operation takes 2–34 cycles. Therefore, the division operation in the proposed processor is completed out of the pipeline. In other words, when a division instruction is encountered, the pipeline temporarily stalls and awaits completion of the operation by the “DIV” module.

As shown in Figure 7, in the proposed architecture, the divider employs a standard pipelined shifting method for implementation. As the divisor is shifted, and if it becomes less than or equal to the dividend during the shifting process, the shifting pointer maps its current position to the result. Simultaneously, subtraction operations are performed between “dividend-compare” and “divisor-compare” to obtain the remainder. Additionally, the proposed divider includes combinatorial logic for distinguishing between signed and unsigned operations as well as for handling both remainder and division operations.

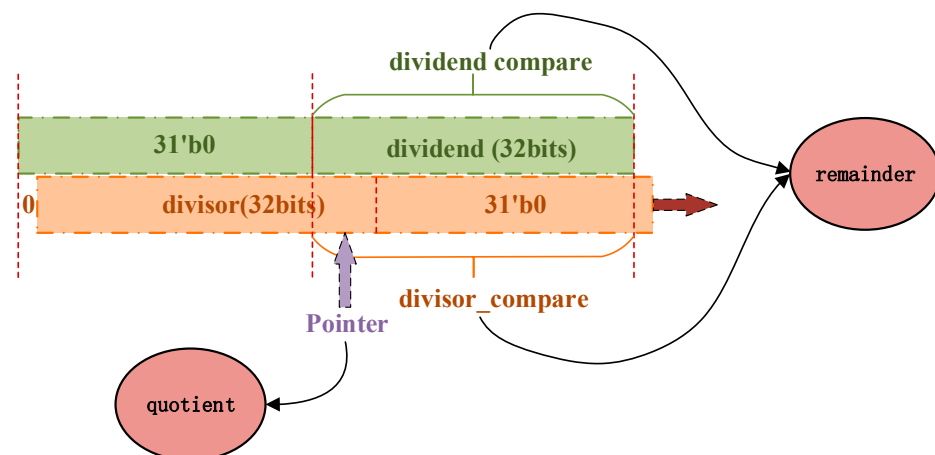


Figure 7. Divider principle diagram.

3.3.4. LSU Module

The LSU (Load Storage Unit) is mainly used as a control module for memory access in the processor. In the proposed architecture, this unit is responsible for implementing the Load and Store instructions of RV32I and the CSR operations on the memory. As shown in Figure 8, the workflow of the LSU is pipelined by three stages in the proposed architecture. The following text will discuss the limit case of the pipeline in which the LSU receives

memory access instructions in three consecutive cycles. As depicted in Figure 8, there is some overlap between the three-stage pipeline of the LSU and the five-stage pipeline of the processor. In the “ISSUE” stage, the LSU receives data and control signals from the “ISSUE” module, which includes instruction types, operands, etc. The information will be registered to the next cycle and generate an access request to the memory in the “EXEC” stage. Additionally, in the EXEC stage, the LSU stores the control information corresponding to the memory access request initiated at this time into the ctrl-fifo. The information will be used in the “MEM” stage to cut and replace the bit width of the result returned by the memory. In the “MEM” stage, the LSU receives the memory access result and performs a return value judgment and exception generation.

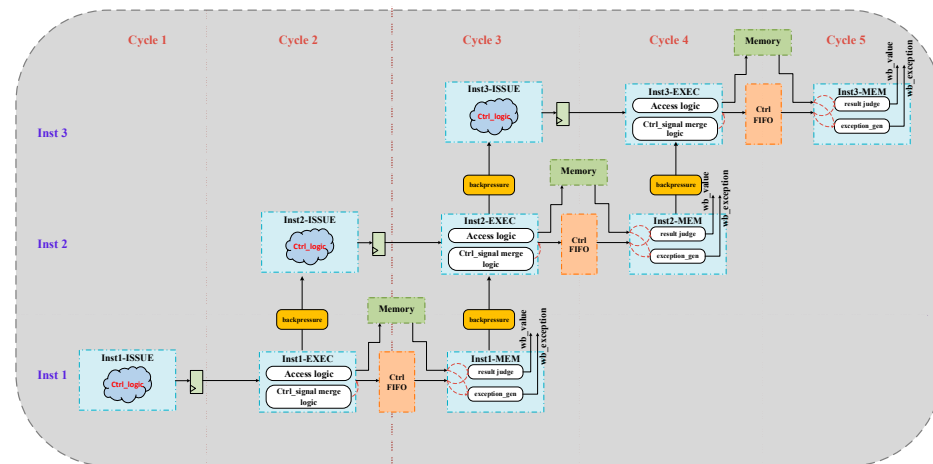


Figure 8. Schematic diagram of the workflow of the LSU.

The resulting judgment mentioned above mainly occurs when the memory reports an access error. At this point, the returned result needs to be replaced with the memory address where the error occurred. In addition, there is backpressure between each stage of the three-stage pipeline in the LSU. The LSU is designed to automatically wait for one cycle to increase redundancy when the correct memory access result is not received in the “MEM” stage.

3.3.5. CSR Module

In the proposed architecture, the “CSR” module is responsible for handling the exceptions and interrupts of the entire system. Whether it is an exception from inside the processor or an interruption from outside the processor, it will be delivered to the “CSR” module during the “ISSUE” stage or the “WB” stage of the pipeline. As illustrated in Figure 9, the internal workflow of the “CSR” module can be primarily segmented into two main parts:

1. The update of the csr-regfile: this part is completed by more complex timing logic, and the updating of the csr-register occurs under the following five conditions:
 - (1) Interrupt
 - (2) Exception-return
 - (3) Exception handled in super privilege level
 - (4) Exception handled in machine privilege level
 - (5) CSR register write
2. Interrupt signals and branch signals are generated according to the data stored in the csr-regfile in the current cycle.

It is worth noting that the write instruction for the CSR does not write data to the csr-regfile during the cycle received by the “CSR” module. These data will be returned to the “pipeline-ctrl” module for storage, and then they are written back to the csr-regfile until

the “WB” stage. Since the logic of the “CSR” module is relatively complex and is closely related to the pipeline of the entire processor, the pipeline will be stalled while the “CSR” module is running in order to avoid errors.

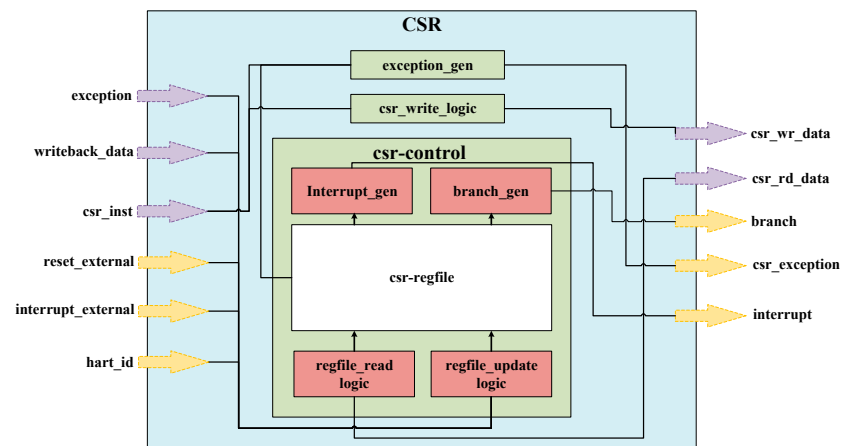


Figure 9. The internal architecture diagram of the “CSR” module.

3.4. Memory Hierarchy and Memory Interface

In the proposed architecture, configurable options are not limited to just inside the processor core. As seen in Figure 10, the memory architecture of the processor core also has two configurable modes: low power consumption and high performance. However, the focus of this article is to analyze the design inside the processor core. Therefore, only a brief description of the memory structure is given here. For the low-power mode, the proposed architecture employs a relatively simple TCM as a buffer between the processor core and the bus, while in the high-performance mode, the TCM is replaced by a cache. Both of these modes utilize a Harvard architecture, which is a storage system that separates instructions and data. In both modes mentioned above, the bus adopts the AXI-4 bus. The benchmark test results mentioned in this article are all based on the operation in the TCM mode.

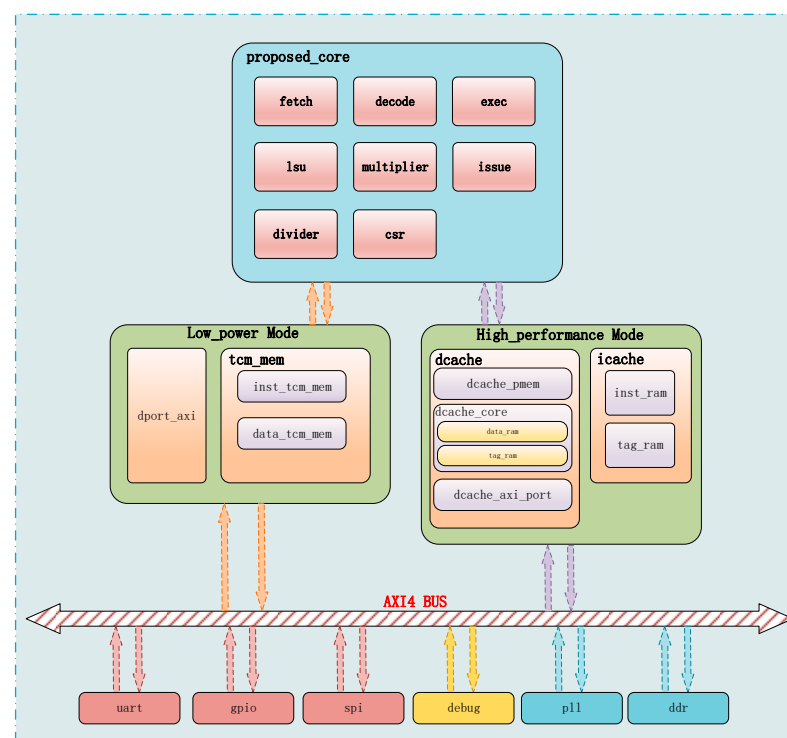


Figure 10. The Memory hierarchy and memory interface for the proposed architecture.

4. Evaluation Results

As shown in Figure 11, the proposed core design is implemented on Verilog HDL (Hardware Description Language). Following the principle of free and open-source software, we have completed both the sub-module-level and system-level verification of the proposed processor design using the verilog and gtkwave workflow. In the initial phase of verification, random instruction sequences were employed to stress each component and identify corner-case bugs. The processor virtual machine with the low power consumption configuration and high-performance configuration were realized, respectively, by System-C language at the software level. Then a virtual machine was used to carry out differential co-simulation on the proposed architecture. The performance of the proposed core was evaluated using a suite of three benchmark applications, as follows: vector-vector addition, insertion sort, and XOR cipher, corresponding to mathematical computations, data processing, and data encryption.

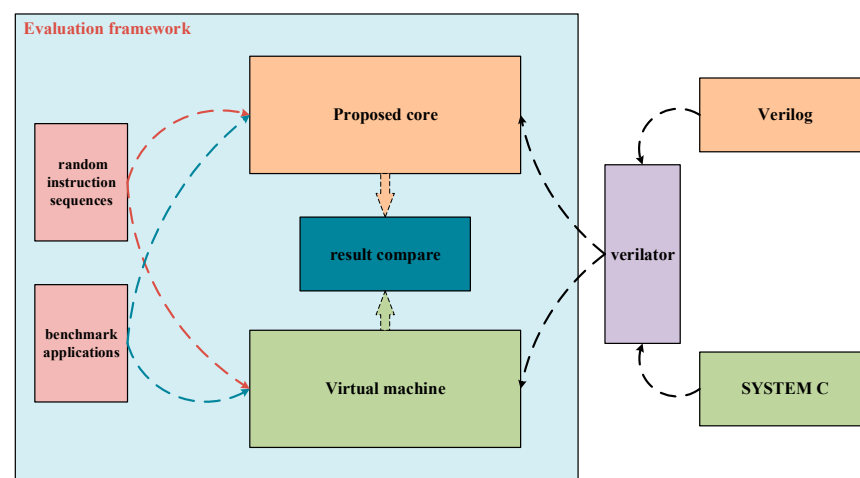


Figure 11. Evaluation framework of the proposed core.

In order to visually showcase the performance of the processor, we executed coremark and dhrystone benchmarks on the proposed architecture. As shown in Table 2, the proposed architecture outperforms classic microprocessors such as ARM’s Cortex M3 in performance.

Table 2. The performance comparison of the proposed core (low-power mode with MUL and DIV).

Features	Our Core	Cortex M0 [20]	Cortex M0+ [20]	Cortex M3 [20]
Coremark/MHz	3.51	2.33	2.46	3.53
DMIPS/MHz	1.48	0.96	0.99	1.24

For the hardware-level validation, the proposed processor is prototyped on the Xilinx Artix7 FPGA. During the FPGA prototype verification phase, the current design is capable of running at a clock rate of 200 MHz. In contrast, the Cortex-M3 operates at a clock rate of 250 MHz in 40LP and a nine-track library [20]. Therefore, there is reason to believe that the proposed architecture can have at least the same performance after being tape-out on TSMC’s 45 nm process.

In the current complex market environment, accurately estimating the price of a processor is a challenging task. However, we can still provide a preliminary estimate from a cost perspective. As shown in Table 2, the architecture proposed in this paper essentially rivals the performance of the Cortex-M3. Thanks to the adoption of a fully open-source instruction set and a comprehensive open-source design toolchain, our architecture does not incur expensive licensing fees, significantly reducing costs. This cost-effectiveness makes our architecture particularly advantageous when applied to smaller niche markets and for small-scale developers.

5. Conclusions and Future Work

In this paper, we present the design of a configurable five-stage pipeline processor core based on the RV32IM architecture. The primary objective is to create a processor that strikes a balance between cost and performance. Our design incorporates the “I” (base integer implementation) and “M” (integer multiplication and division extension) components of the RISC-V ISA. Notably, our processor core features a range of configurable modules, enabling a seamless adaptation to diverse application scenarios. The processor core operates in two distinct application modes: a low-power mode and a high-performance mode. In the low-power mode, the core adheres to the base integer instruction set without incorporating any standard or non-standard extensions. On the other hand, the high-performance mode introduces integer multiplication and division extension. Moreover, the processor core extends its support to the super and user privilege levels, complemented by a comprehensive array of Control and Status Registers (CSRs). We completed the module-level and system-level verifications of the processor using verilator + gtkwave’s fully open-source workflow, and we completed the functional verification using random instruction generation sequences, followed by performance evaluation using a representative benchmark program. After evaluation, the proposed processor exhibits higher performance than a classic commercial MCU-Cortex M3.

Future work can address further customization and optimization of the architecture for low power. The multicore configuration snap-in will be added to the design. The final design will be synthesized with a commercial 45 nm CMOS process technology node and then complete the back-end process.

Author Contributions: Concept and structure of this paper, Y.C.; Resources and Supervision, Y.Z.; Review and editing, Y.L., C.P. and J.G. All authors have read and agreed to the published version of the manuscript.

Funding: This work was financially supported by Yi Zhao’s National Natural Science Foundation of China (NSFC) grant number 61675089.

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

Conflicts of Interest: The authors announce that they have no conflicts of interest concerning article publication.

References

1. De Donno, M.; Tange, K.; Dragoni, N. Foundations and evolution of modern computing paradigms: Cloud, iot, edge, and fog. *IEEE Access* **2019**, *7*, 150936–150948. [[CrossRef](#)]
2. Song, S.; Li, S.; Gao, H.; Sun, J.; Wang, Z.; Yan, Y. Research on multi-parameter data monitoring system of distribution station based on edge computing. In Proceedings of the 2021 3rd Asia Energy and Electrical Engineering Symposium (AEEES), Chengdu, China, 26–29 March 2021; pp. 621–625.
3. Mahbub, M.; Gazi, M.S.A.; Provat, S.A.A.; Islam, M.S. Multi-access edge computing-aware internet of things: MEC-IoT. In Proceedings of the 2020 Emerging Technology in Computing, Communication and Electronics (ETCCE), Dhaka, Bangladesh, 21–22 December 2020; pp. 1–6.
4. Waterman, A.; Lee, Y.; Avizienis, R.; Patterson, D.A.; Asanovic, K. *The Risc-V Instruction Set Manual Volume 2: Privileged Architecture Version 1.7*; University of California: Berkeley, CA, USA, 2015.
5. Pinyotrakool, K.; Supmonchai, B. Design of a low power processor for embedded system applications. In Proceedings of the 2020 8th International Electrical Engineering Congress (iEECON), Chiang Mai, Thailand, 4–6 March 2020; pp. 1–4.
6. Budi, S.; Gupta, P.; Varghese, K.; Bharadwaj, A. A risc-v isa compatible processor ip for soc. In Proceedings of the 2018 International Symposium on Devices, Circuits and Systems (ISDCS), Howrah, India, 29–31 March 2018; pp. 1–5.
7. Schiavone, P.D.; Conti, F.; Rossi, D.; Gautschi, M.; Pullini, A.; Flamand, E.; Benini, L. Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications. In Proceedings of the 2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS), Thessaloniki, Greece, 25–27 September 2017; pp. 1–8.
8. Ramos, A.; Maestro, J.A.; Reviriego, P. Characterizing a RISC-V SRAM-based FPGA implementation against Single Event Upsets using fault injection. *Microelectron. Reliab.* **2017**, *78*, 205–211. [[CrossRef](#)]
9. Ficarella, F.; Bartolini, A.; Parisi, E.; Beneventi, F.; Barchi, F.; Gregori, D.; Magugliani, F.; Cicala, M.; Gianfreda, C.; Cesarini, D. Meet Monte Cimone: Exploring RISC-V high performance compute clusters. In Proceedings of the Proceedings of the 19th ACM International Conference on Computing Frontiers, Turin, Italy, 17–22 May 2022; pp. 207–208.

10. Marena, T. RISC-V: High performance embedded SweRV™ core microarchitecture, performance and CHIPS Alliance. *West. Digit. Corp.* **2019**, *1*, 1–21.
11. Wu, N.; Jiang, T.; Zhang, L.; Zhou, F.; Ge, F. A reconfigurable convolutional neural network-accelerated coprocessor based on RISC-V instruction set. *Electronics* **2020**, *9*, 1005. [[CrossRef](#)]
12. Domas, C. Breaking the x86 ISA. *Black Hat* **2017**, *1*, 1–6.
13. Sankaralingam, K.; Menon, J.; Blem, E. *A Detailed Analysis of Contemporary Arm and x86 Architectures*; University of Wisconsin: Madison, WI, USA, 2013.
14. Liu, Y.; Ye, K.; Xu, C.-Z. Performance Evaluation of Various RISC Processor Systems: A Case Study on ARM, MIPS and RISC-V. In Proceedings of the Cloud Computing–CLOUD 2021: 14th International Conference, Held as Part of the Services Conference Federation, SCF 2021, Virtual Event, 10–14 December 2021; Springer: Cham, Switzerland, 2022; pp. 61–74.
15. El Kady, S.; Khater, M.; Alhafnawi, M. MIPS, ARM and SPARC—an architecture comparison. In Proceedings of the Proceedings of the World Congress on Engineering, London, UK, 2–4 July 2014.
16. Waterman, A.; Lee, Y.; Patterson, D. The RISC-V instruction set manual. In *Volume I: User-Level ISA, Version 2.0*; EECS Department, University of California: Berkeley, CA, USA, 2014.
17. Höller, R.; Haselberger, D.; Ballek, D.; Rössler, P.; Krapfenbauer, M.; Linauer, M. Open-source risc-v processor ip cores for fpgas—Overview and evaluation. In Proceedings of the 2019 8th Mediterranean Conference on Embedded Computing (MECO), Budva, Montenegro, 10–14 June 2019; pp. 1–6.
18. Waterman, A.S. *Design of the RISC-V Instruction Set Architecture*; University of California: Berkeley, CA, USA, 2016.
19. Patterson, D.; Waterman, A. *The RISC-V Reader: An Open Architecture Atlas*; Strawberry Canyon: Berkeley, CA, USA, 2017.
20. Martin, T. *The Designer's Guide to the Cortex-M Processor Family*; Newnes: Boston, MA, USA, 2022.

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.