International Journal of Networking and Computing – www.ijnc.org, ISSN 2185-2847
Volume 15, Number 2, pages 182–198, July 2025

Extended VLIW Processor with Overlapping RISC-V Compressed and Privileged Instructions

Haruhiro Tanaka
Graduate School of Information Science & Technology, Aichi Prefectural University
Aichi, 480-1198, Japan
im231006@cis.aichi-pu.ac.jp

Takahiro Sasaki
Graduate School of Information Science & Technology, Aichi Prefectural University
Aichi, 480-1198, Japan
sasaki@ist.aichi-pu.ac.jp

Received: February 14, 2025
Revised: May 2, 2025
Accepted: June 6, 2025
Communicated by Takashi Yokota

Abstract

RISC-V is an instruction set architecture that has attracted interest in both academic and industrial fields in recent years. RISC-V provides compressed instructions that reduce the size of each instruction. This feature contributes to the reduction of program size and is advantageous for embedded processors that have tight constraints on instruction and data memories. Therefore, implementation of compressed instructions is advantageous in terms of program size, but disadvantageous in terms of hardware for embedded processors with heavy area constraints. To solve the increase amount of hardware caused by supporting compressed instructions, we propose Converting All Integer-instructions to Compressed-instructions (CAIC) method. Additionally, we propose an extended VLIW processor called RVC-VOI (RISC-V Compressed - VLIW with Overlapping Instructions) that adapts the CAIC method. The processor implements privileged instructions which are not defined in compressed instructions, without increasing the issue slot executes privileged instructions or hardware to align instruction length, by overlapping instruction fields. This paper evaluates the code size reduction rate achieved by the CAIC method and the area of RVC-VOI. The CAIC method achieved a 7.2% reduction in the code size of QuickSort and a 25.9% reduction in the code size of Dhrystone, and RVC-VOI achieved significant reductions in energy consumption of up to 98.7% and circuit scale of up to 98.6% while maintaining execution time comparable to that of superscalar processors.

Keywords: RISC-V, Compressed instruction set, Computer architecture, VLIW

1 Introduction

In recent years, the Internet of Things (IoT) has become widespread across a broader range of systems. The global pandemic of the COVID-19 has significantly accelerated this technology, and modern IoT is now evolving into something closer to Internet of Everything [1,19]. IoT integration is advancing in many systems, from electrical devices to systems that have never used electricity before [15]. Embedded processors used in such IoT devices that have tight constraints on area and energy

budget, but the increasing complexity of IoT applications has made it difficult to meet performance, area and energy efficiency targets, increasing interest in those budgets [8]. Consequently, there is a growing trend towards designing custom processors instead of using general purpose processors, which typically have larger area and higher energy consumption.

This paper focuses on compressed instructions. Compressed instructions, such as ARM Thumb [14], have been considered useful for embedded processors. Compressed instructions are also provided as an extension function in the RISC-V instruction set architecture (ISA) [10], which has recently attracted attention in both academia and industry [4, 16]. RISC-V compressed instructions improve memory efficiency by compressing the 32-bit fixed-length instructions for basic integer instructions into 16-bit.

However, the RISC-V compressed instructions are not intended to be used independently. A typical RISC-V compiler does not have the capability to generate programs with only compressed instructions. For this reason, RISC-V compiler generates programs that combine compressed instructions and basic instructions that are not replaced by compressed instructions. The implementation of compressed instructions can reduce the size of the program code, but the processor area increases because additional hardware is required to align the lengths of fetched instructions. This is a disadvantage for embedded processors with heavy area constraints. Furthermore, running an operating system (OS) on RISC-V requires 32-bit privileged instructions. Because RISC-V compressed ISA does not support privileged instructions, it is not possible to run an OS using only compressed instructions.

To solve the above problems, we propose a method called CAIC (Converting All Integer instructions to Compressed instructions). This method converts all instructions to 16-bit compressed instructions by representing basic instructions as combinations multiple existing compressed instructions. Not implementing a custom, non-standard 16-bit instructions ensure portability to existing processors and can be used as a method to handle the increase in code size caused by the complexity of application programs. Additionally, we propose a VLIW (Very Long Instruction Word) processor called RVC-VOI (RISC-V Compressed - VLIW with Overlapping Instructions) [17] which supports programs constructed by only compressed instructions. By effectively using the VLIW structure, RVC-VOI can execute privileged instructions that are not included in compressed instructions without modifying the RISC-V compressed ISA. RVC-VOI is expected to be used as an embedded processor that requires an OS because it has a smaller area than a typical Out-of-Order processor maintaining sufficient performance.

As an evaluation, we implement our proposed RVC-VOI using Register Transfer Level (RTL) description and evaluated through simulations for the program execution time and energy consumption. Additionally, we evaluated the circuit scales by logical synthesis. For comparison, we used AnyCore [2], a tool that automatically designs superscalar processors, which are commonly used as general-purpose processors. RVC-VOI successfully reduced energy consumption to 1.3% and circuit scale to 2.6% while maintaining execution times comparable to the minimum configuration of AnyCore.

The remainder of this paper is structured as follows. Section 2 provides the RISC-V instruction set architecture and the VLIW processor architecture as background to this paper. Section 3 provides related works on processors suitable for embedded systems. In Section 4, we propose the CAIC method, which enables the standalone use of RISC-V compressed instruction set. Additionally, In Section 5, we propose the RVC-VOI processor, an extended VLIW architecture that simultaneously implements compressed and privileged instructions. Section 6 evaluates RVC-VOI in terms of execution time, energy consumption, and circuit scale. Section 7 presents the conclusions and outlines directions for future work.

2 Background

RISC-V is one of the ISAs that has attracted the most attention recently [10]. While most commercial ISAs require license fees, RISC-V is an open ISA and free to use. By its openness, RISC-V has attracted interest not only from academia but also from industry. Consequently, many companies

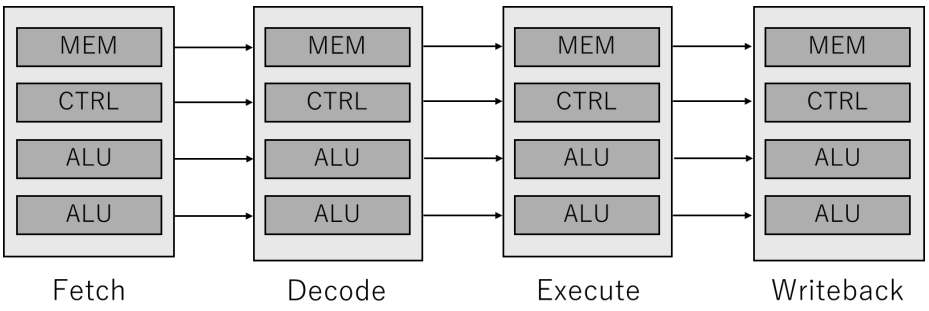

Figure 1: 4-issue VLIW structure.

and universities are currently entering the development of RISC-V. Additionally, RISC-V consists of 32-bit basic integer instructions that enables the minimal processor operation and several selectable extension instructions. These extensions can be selectively implemented during processor design, depending on the type of applications the processor is intended to run. Examples of extensions include integer multiplication and division instruction set extension, single and double precision floating-point instruction set extension, atomic instruction set extension, and compressed instruction set extension. Particularly, compressed instructions reduce certain frequently used basic instructions to 16-bit. Therefore, the implementation of this extension is expected to be used as embedded processors because it reduces code size.

As a processor architecture for embedded systems, VLIW (Very Long Instruction Word) is one of the useful methods. In a VLIW processor, the slots where specific instructions can be executed are determined during the architecture design phase, and the dedicated compiler statically reorders instructions according to the design. Figure 1 describes the architecture of a 4-issue and 4-stage pipeline VLIW processor. While superscalar processors, which are common processor architecture in recent years, depend on complex internal scheduling mechanisms to dynamically determine which instructions can be executed in parallel, VLIW processors perform this task with the compiler. This allows VLIW processors to have simpler internal mechanisms, reducing the circuit area. As a result, VLIW processors can achieve high Instruction Level Parallelism (ILP) without complex internal structure.

However, while RISC-V offers extensive support for superscalar processors, its support for VLIW processors is limited. This is due to the fact that RISC-V was not originally designed as foundation for VLIW architectures. Although the current RISC-V provides VLIW encoding through several alternative approaches, the technique for encoding instructions longer than 32-bit is not fixed. Furthermore, compressed instructions cannot be used independently and need to be combined with basic instructions. As a result, the implementation of compressed instructions has led to an increase in hardware. Detailed of the problem and our solutions to handle compressed instructions are described in Section 4.1.

3 Related Works

ρ-Vex [18] is a VLIW processor based on the VEX ISA. ρ-Vex has an advantage in integer arithmetic because it implements new instruction for generating 32-bit immediate value which is not provided in the standard VEX ISA. This new instruction overlaps 2 slots and is executed by the same ALU used for typical instructions. While this technique is similar to the approach in this work, the difference is that ρ-Vex overlaps a new instruction to reduce code size, whereas our approach overlaps existing instructions to reduce hardware complexity.

CV32E40P [7] (formerly known as RI5CY [6]) is a 4-stage in-order pipeline processor based on the RISC-V ISA. The processor provides extension instructions including multiply-divide and compressed instructions. CV32E40P is a processor with a small area, but it has low binary compatibility

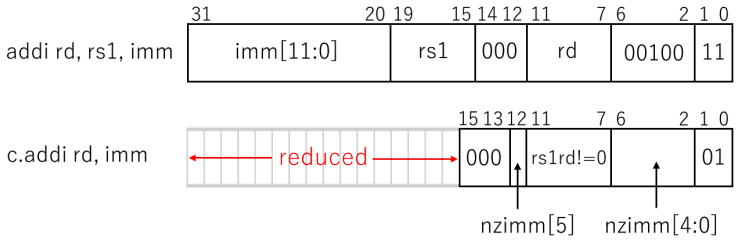

Figure 2: Comparison of basic and compressed addi instructions.

with other RISC-V processors and the hardware has also increased due to custom instruction set extensions.

RVCoreP-32IC [11] is a 5-stage pipeline processor based on the RISC-V ISA. It implements both basic and compressed instructions and achieves an efficient instruction fetch architecture by using 2 Program Counters (PC) for parallel fetching. However, this fetch architecture increases the amount of hardware required. Furthermore, unlike the processor proposed in this work, RVCoreP-32IC is a single-issue processor. Such processors have the possibility to lack sufficient performance for the complexity of embedded applications in recent years.

As a VLIW based on the RISC-V ISA, Qui, Lin and Chen propose a 256-bit VLIW processor with a dynamic scheduler [13]. This processor scheduler detects the dependencies of eight instructions fetched at the same time and generates VLIW instructions within the processor. This mechanism enables the execution of VLIW instructions without the need for a dedicated compiler. However, this processor cannot fully utilize ILP, because it only detects the dependencies of eight fetched instructions. Additionally, the scheduling mechanism increases the amount of hardware.

4 Translation Technique to Compressed Instructions

4.1 Overview of Compressed Instructions

The compressed instruction set is one of the extensions of the RISC-V ISA. Compressed instructions can reduce some of 32-bit fixed-length basic integer instructions to 16-bit. Accordingly, compressed instructions can replace equivalent basic instructions, thereby reducing the number of bits per instruction. Currently, using a general compiler such as gcc which supports compressed instructions, 50%-60% of the RISC-V instructions can be replaced with compressed instructions, which is equivalent to a 25%-30% code size reduction [10].

Figure 2 shows the formats of the basic instruction addi and the compressed instruction c.addi for performing register and immediate value addition. In Figure 2, the “rs1rd!=0” field refers to the register number field used for both the rs1 and rd registers, indicating that they cannot use a register number zero. The “nzimm” field stands for non-zero immediate, which means the immediate value cannot be zero. Additionally, the opcode of the basic addi instruction consists of 7 bits from bit 0 to bit 6, and the funct field is from bit 12 to bit 14. On the other hand, the opcode of the compression instruction consists of two bits from bit 0 to bit 1, and the funct field is from bit 13 to bit 15. In Figure 2, both instructions are described numerically. The 32-bit basic instruction addi can be replaced by the compressed instruction c.addi. However, for example, the addi instruction has a 12 bits immediate field, whereas the immediate field of the c.addi instruction is narrower and cannot handle immediate values larger than 7 bits. Instruction with such large immediate values cannot be replaced by a single compressed instruction and remains as 32-bit basic instructions during compilation. Therefore, when designing a processor that implements compressed instructions, it is necessary to include hardware to determine whether the fetched instruction is a basic or compressed instruction. It is a disadvantage for embedded processors where the hardware area is a critical concern.

Table 1: CAIC method conversion of addi instruction.

Conditions	Basic instruction	Compressed instructions
imm $\neq$ 0 & rd == src1	addi rd, rd, imm	c.addi rd, nzimm
imm $\neq$ 0 & rd $\neq$ src1	addi rd, src1, imm	c.li rd, nzimm c.add rd, src1
imm == 0 & rd == src1	addi rd, rd, 0	c.nop
imm == 0 & rd $\neq$ src1	addi rd, src1, 0	c.mv rd, src1
$\log_2$(imm) $\geq$ 7	addi rd, rd, imm	c.li rd, imm[11:6] c.slli rd, 6 c.addi rd, imm[5:0]
rd == src1 == x2	addi x2, x2, imm	c.addi16sp imm
src1 == x2 & rd $\neq$ x2	addi rd, x2, imm	c.addi4spn rd, imm

4.2 CAIC Method

In this work, we propose a method to implement all basic instructions using only compressed instructions by mapping basic instructions to combinations of available compressed instructions. We call this method Converting All Integer-instructions to Compressed-instructions (CAIC). In this paper, we describe the CAIC method using 3 major instructions as examples. The first is the addi instruction of the simple ALU calculation, the second is the lw instruction of the data memory access, and the third is the jalr instruction for control. These instructions cover the important categories of arithmetic operations, memory access, and control flow, and comprehensively cover the basic operation of RISC-V.

Table 1 shows the CAIC method conversion of the addi instruction. The addi instruction adds the value of the src1 register and an immediate, storing the result in the rd register. The addi instruction can be converted with compressed instructions under several conditions. Conversion to standalone c.addi instruction is performed only when the destination register and the source register are the same. If the immediate value is 0 and the destination register and source register are the same, it is replaced by the c.nop instruction. If the immediate value is 0 and the destination and source register are different, it is replaced by the c.mv instruction. For addi instructions that handle the x2 register, which manages the stack pointer in RISC-V, there is a dedicated compressed instruction available. For other patterns, the processing of the addi instruction is realized using combinations of multiple compressed instructions. If the immediate value is non-zero and the destination and source register are different, the processing can be realized by splitting it into 2 compressed instructions. In the same way, if the immediate value is 7 bits or larger, the addi instruction can be converted using 3 compressed instructions.

Table 2 shows the CAIC method conversion of the lw instruction. The lw instruction adds the value of the src1 register and an immediate to compute a data memory address, then loads the data from that address into the rd register. In this paper, registers marked with a prime symbol, such as rd' and src1', refer to registers numbered from 8 to 15. Compressed instruction set includes the c.lw instruction. A general compiler conversion to the c.lw instruction occurs when the immediate value is a positive integer within 6 bits, and both the rd and src1 register numbers are between 8 and 15. When the immediate value is negative or more than 7 bits, the instruction is divided into 4 compressed instructions because the memory address needs to be generated in a register. If the rd register is outside the range of 8 to 15, one register within this range is selected for data loading. In Table 2, such registers are indicated as rx'. If no registers are available, use the c.mv or c.sw instructions to generate a free register.

Table 3 shows the CAIC method conversion of the jalr instruction. The jalr instruction jumps to the address computed by adding the src1 register and an immediate value, and stores the return address in the rd register. Compressed instruction set includes the c.jalr instruction. The jalr instruction is replaced by the c.jalr instruction when the immediate value is 0 and the rd register is

Table 2: CAIC method conversion of lw instruction.

Conditions	Basic instruction	Compressed instructions
$0 \leq \log_2(\text{imm}) < 7$ & x0 $\leq$ rd $\leq$ x15 & x0 $\leq$ src1 $\leq$ x15	lw rd', offset(src1')	c.lw rd', uimm(src1')
imm < 0 || $\log_2(\text{imm}) \geq 7$	lw rd', imm(src1')	c.li rd', imm[31:26] c.slli rd', 6 c.add rd', src1' c.lw rd', src1[25:22](rd')
rd $\leq$ x7 || x16 $\leq$ rd	lw rd, imm(src1')	c.lw rx', imm(src1') c.mv rd, rx'
rs1 == x2	lw rd, offset(x2)	c.lwsp rd, uimm(x2)

Table 3: CAIC method conversion of jalr instruction.

Conditions	Basic instruction	Compressed instructions
rd == x1 && imm == 0	jalr x1, src1, 0	c.jalr src1
imm $\neq$ 0	jalr x1, src1, offset	c.li x1, offset[11:6] c.slli x1, 6 c.addi x1, offset[5:0] c.add x1, src1 c.jalr x1
rd $\neq$ x1	jalr rd, src1, 0	c.mv rx, x1 c.jalr src1 c.mv rd, x1 (post-jump) c.mv x1, rx (post-jump)

x1. Since the c.jalr instruction cannot handle immediate values, when the jalr instruction includes an immediate value, the destination address needs to be generated in the x1 register before executing the c.jalr instruction. Because the value of the x1 register is updated after the execution of the c.jalr instruction, there is no need to save the original value of the x1 register. Additionally, the rd register for the c.jalr instruction is fixed to the x1 register. Therefore, if the rd register is not x1, the original value of x1 is temporarily stored to another register or data memory before executing the c.jalr instruction. After c.jalr execution, the value of x1 is restored. In Table 3, the process is indicated when values are stored in registers. If stored in data memory, data transfer is executed using the c.sw and c.lw instructions.

In this manner, basic instructions that are not replaced with compressed instructions by general compilers can be converted into combinations of multiple compressed instructions. This conversion was verified for all 49 basic integer instructions, and we confirmed that integer instructions can be implemented using only compressed instructions.

5 Extended VLIW Processor Architecture

Only compressed ISA cannot run an OS that requires privileged instructions, because compressed instructions do not include them. To solve this issue, this section describes an extended VLIW processor architecture capable of executing both compressed instructions and 32-bit privileged instructions. The processor is called RVC-VOI (RISC-V Compressed - VLIW with Overlapping Instructions).

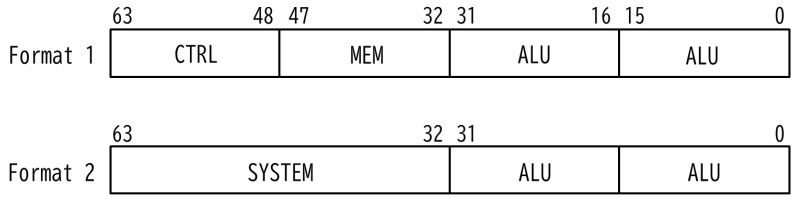

Figure 3: 4-slot RVC-VOI instruction layout.

5.1 Instruction Layout

One significant difference from a standard VLIW is that the slot dedicated to privileged instructions overlaps the lower 2 slots. Figure 3 shows the 2 VLIW formats in 4-slot configuration. In Format 1, slot 0 and slot 1 are allocated for integer arithmetic instructions, slot 2 is allocated for memory access instructions, and slot 3 is allocated for branch instructions. In contrast, Format 2 allows privileged instructions to utilize the slots designated for memory access and branch instructions. In other words, a dedicated privileged instruction slot overlaps with slot 2 and slot 3. Therefore, when a VLIW instruction does not contain a privileged instruction, instructions are allocated according to Format 1. When a VLIW instruction contains a privileged instruction, instructions are allocated according to Format 2, and the privileged instruction overlaps the 2 slots.

There are two alternative approaches to simply supporting privileged instructions. The first approach introduces a custom, non-standard 16-bit privileged instruction. However, this method has the issue of incompatibility with existing RISC-V processors. In contrast, our proposed method maintains compatibility with existing processors by utilizing only the RISC-V instruction set.

The second approach introduces a dedicated 32-bit slot for privileged instructions. However, since privileged instructions appear significantly less frequently than other instructions, this slot would often be filled with superfluous no-operation (nop) instructions, leading to inefficiency. Our proposed method addresses this issue by overlapping the slots for privileged and regular instructions.

Therefore, this approach allows the implementation of instructions with different lengths without adding new slots and increasing nop instructions, and also ensures compatibility with existing ISA.

5.2 Architecture

Figure 4 shows a block diagram of the proposed processor. In general, VLIW processors can maintain high ILP energy-efficiently in terms of static compiler scheduling. The extended VLIW processor proposed in this work also conforms to the static compiler of VLIW. The processor implements integer arithmetic, memory access and branch instruction slots. The number of integer arithmetic slots is parameterized and can be changed from 1 to 6. This allows the performance of the RVC-VOI processor to be adjusted according to the requirements of the embedded program during processor design. The lower 2 slots are fixed for executing memory access instructions slot and branch instructions slot. Therefore, RVC-VOI supports the design of VLIW architectures with configurations from a minimum of 3-slot to a maximum of 8-slot.

5.2.1 Pipeline Stage

RVC-VOI processor is designed with four stages: fetch, decode, execute, and writeback.

In the fetch stage, VLIW instruction is fetched from instruction memory and sent to the decode stage. The fetch width is parameterized, depending on the number of integer arithmetic slots, and the increment of the PC is also affected by the parameter.

The decode stage implements integer arithmetic instructions decoders, a memory access instructions decoders, a branch instruction decoder, and a privileged instructions decoder. Each decoder decodes the VLIW instruction that is split into individual instructions, register values are read, and these values are sent to execute stage.

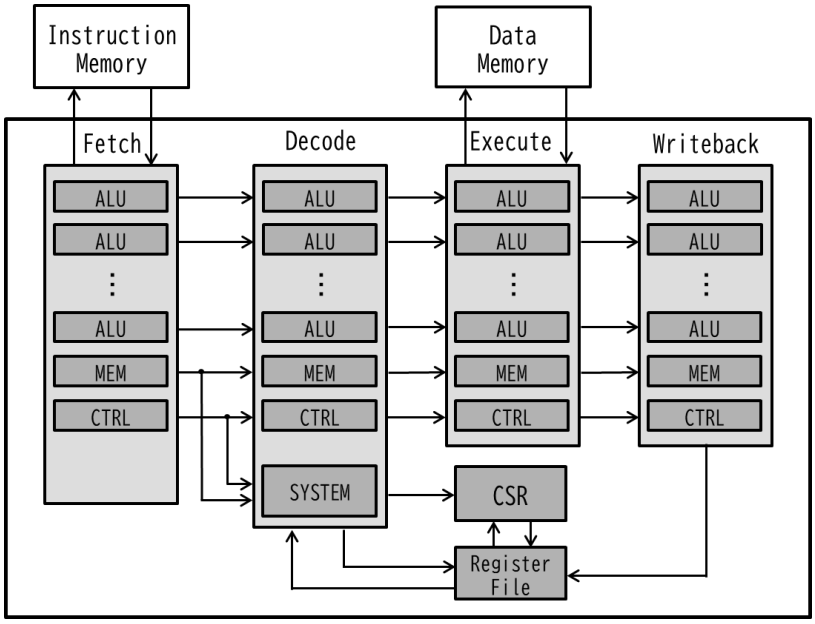

Figure 4: Block diagram of RVC-VOI.

The execute stage computes each instruction. Memory access instructions are managed by the Load Store Unit (LSU), where store instruction is completed.

In the writeback stage, register file writes and PC updates due to branch instructions. If the next instruction depends on the result, the computed value is forwarded to avoid a stall.

5.2.2 Commit Timing

RVC-VOI takes 4 cycles for most instructions, and 5 cycles for load instruction. Due to the different commit timings of these instructions, when an instruction depends on the result of a previous load instruction, it must wait for that load operation to complete. This dependency is identified at the decode stage. Therefore, when RVC-VOI detects such dependencies, it generates a 1 cycle stall in both the fetch and decode stages.

Additionally, RVC-VOI does not implement a branch predictor, a design choice aimed at minimizing circuit scale. Consequently, when a branch instruction modifies the PC, the processor must flush the instructions in the fetch and decode stages. Upon a successful branch, the execution stage sends the correct branch target PC to the fetch stage. This process results in 2 cycles delay whenever a branch instruction executes a branch.

5.2.3 Execution of Privileged Instruction

The System Decoder that handles privileged instructions decodes the lower 32 bits of a VLIW instruction and recognizes it as a valid instruction only if it is a privileged instruction. The privileged instruction slot overlaps with the memory access slot and the branch instruction slot. Therefore, the instructions provided to the Mem Decoder and Ctrl Decoder are always simultaneously provided to the System Decoder.

As an example, Figure 5 shows the bitmaps of the csrrs instruction, a type of privileged instruction, and the c.addi instruction, a type of compressed instruction. In RISC-V processors, the type of instruction is identified by examining the opcode (OP) and function code (funct) within the instruction. Specifically, the distinction between 32-bit and 16-bit instructions is made based on the opcode of the lower 2 bits of the instructions. In the case of compressed instructions, the OP falls into one of the values 00, 01, or 10. In contrast, the lower 2 bits of the OP for privileged instructions

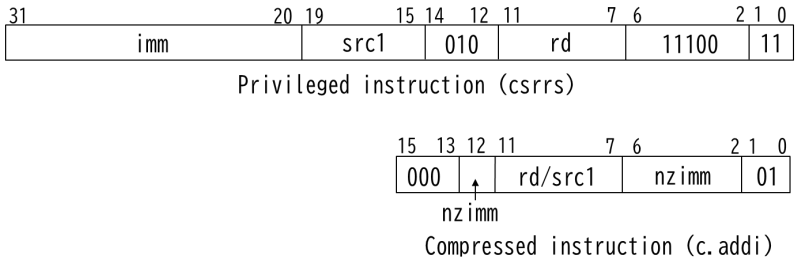

Figure 5: The bitmaps of the csrrs and c.addi instructions.

are fixed at 11. This difference in the OP allows the decoders to distinguish between compressed and privileged instructions. Therefore, when the System Decoder receives memory access and branch instructions provided in Format 1, it examines the lower 2 bits that are OP. If these bits do not match 11, the instruction is identified as a non-privileged instruction and invalidated. Conversely, if a privileged instruction is received, the same bit examination confirms its privileged instruction.

On the other hand, the process is different for the Mem Decoder and the Ctrl Decoder. The Ctrl Decoder can access the lower 2 bits of privileged instructions, allowing it to distinguish between privileged and compressed instructions. In contrast, the Mem Decoder only has access to the upper 16 bits of privileged instructions. As shown in Figure 5 with the csrrs instruction, the 16th and 17th bits of privileged instructions are not fixed but instead often represent register numbers or immediate values. Therefore, depending on the values of the register numbers or immediate fields, the Mem Decoder may incorrectly interpret the upper 16 bits of a privileged instruction as a valid compressed instruction.

To solve this problem, when the System Decoder detects a valid privileged instruction, it invalidates the instructions present in both the Mem Decoder and the Ctrl Decoder. This process prevents incorrect interpretation of instructions caused by the immediate values and register numbers in privileged instructions. Furthermore, RISC-V privileged instructions that change the internal state of the processor, such as switching machine modes or enabling interrupts, must be executed atomically. This is solved by flushing the instructions existing in the fetch stage when the System Decoder detects a valid privileged instruction. Therefore, in RVC-VOI, a privileged instruction is executed only after the completion of any integer arithmetic instructions within the same VLIW instruction. This allows safe access to the Control and Status Register (CSR) unit, which manages the state of the RISC-V machine.

As specific examples, an example of the operation of Format 1 is shown in Figure 6, and an example of the operation if Format 2 is shown in Figure 7. In Format 1, the Mem slot and Ctrl slot instructions are input into the System Decoder. The System Decoder can invalidate these compressed instructions by checking their OP. In Format 2, the System Decoder receives a privileged instruction, while the Mem Decoder and Ctrl Decoder receive the upper 16 bits and lower 16 bits of the privileged instruction, respectively. When the System Decoder detects a privileged instruction, it invalidates the instructions in the Mem Decoder and Ctrl Decoder to prevent misinterpretation of instructions. Additionally, it purges all instructions in the fetch stage to manage the processor status.

5.3 Implementation Method

Processors used in embedded systems are expected to reduce unnecessary increases in chip area by designing performance to suit specific applications. However, processor design generally requires more effort and time, designing processors for each system is challenging. To solve this issue, as described in Section 5.2, the VLIW processor proposed in this paper can be designed with performance suited to the system by adjusting the number of slots based on the supported application. In superscalar processor field, AnyCore [2] is proposed as an automated processor design tool. It is designed to generate a wide range of processors, from high-performance to low-power models by

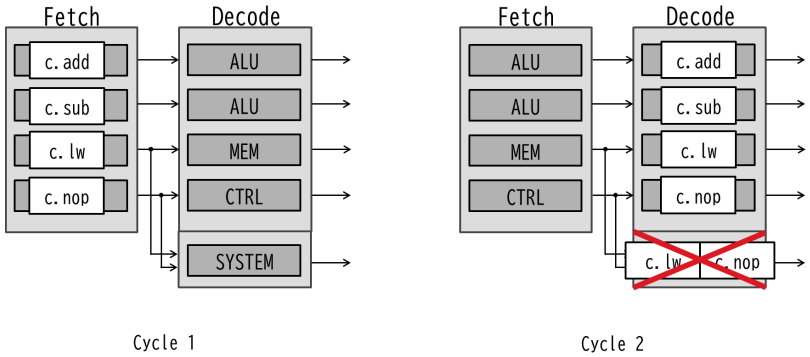

Figure 6: The operation of Format 1.

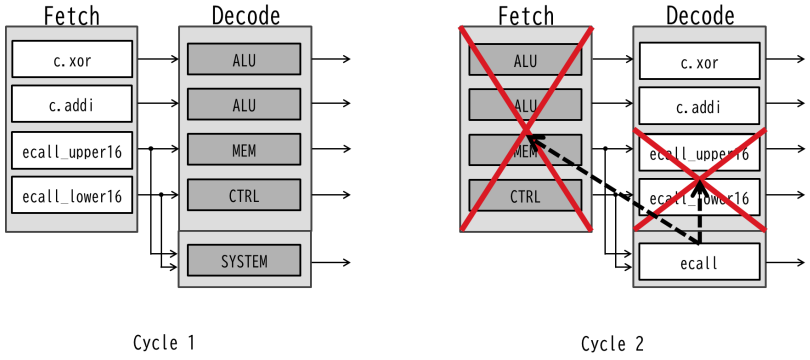

Figure 7: The operation of Format 2.

allowing parameterized configurations. The standard AnyCore cannot design VLIW processor, but RVC-VOI is designed by modifying AnyCore to simplify the design. Since AnyCore does not support compressed instructions, we have implemented decoder and execution units for them. Additionally, because VLIW is scheduled by the compiler, the internal scheduling mechanism of AnyCore has been removed to simplify the architecture. Since AnyCore manages parameters such as fetch width and issue width, RVC-VOI inherits this functionality, allowing the number of issue slots to be adjusted as parameters.

6 Evaluation

In this section, we evaluate the code size reduction rate achieved by the CAIC method. Additionally, we evaluate execution time, energy consumption, and circuit area of the RVC-VOI processor. To compare processor areas, Out-of-Order processors were designed using AnyCore. Table 4 shows the parameters of the designed processors. Since RVC-VOI is a VLIW architecture, the parameters are fundamentally identical by design. AnyCore1 has the minimal configuration of AnyCore, and AnyCore2 has the same number of fetch, dispatch, issue, and commit widths as RVC-VOI. Since AnyCore is an adaptive superscalar processor capable of dynamically adjusting structural sizes [12], superscalar widths [3], and so on, this paper focuses on generating static cores by omitting those dynamic features. This simplified configuration ensures a fairer comparison with RVC-VOI, emphasizing static design efficiency.

In this paper, we also present the verification and evaluation of RVC-VOI using EDA/CAD. The EDA/CAD tools and libraries used are listed in Table 5.

Table 4: Parameters of the evaluated processors.

	RVC-VOI	AnyCore1	AnyCore2
Fetch width	4	1	4
Dispatch width	4	1	4
Issue width	4	3	4
Commit width	4	1	4

Table 5: EDA/CAD environment for evaluation

Functional verification	Cadence NC-Verilog 15.20-s020
Logical synthesis	Synopsys PrimeTime (Ver. U-2022.12-SP5-4)
Power estimation	Synopsys Design Compiler (Ver. S-2021.06-SP2)
Layout	Synopsys IC Compiler II (Ver. R-2020.09-SP5)
Standardcell library	Rohm CMOS 0.18μm

6.1 Code Size

In the code size evaluation, we used a simple QuickSort program that was self-written in C programming language and Dhrystone benchmark provided by RISC-V international [9]. These programs were compiled into the programs with only 32-bit basic instructions using the riscv-gnu-toolchain [5], and then manually converted to create a program consisting of only compressed instructions using the CAIC method. The code size is evaluated by comparing the object file sizes of programs with only 32-bit basic instructions, with only 16-bit compressed instructions and with a combination of these instructions. Table 6 shows the results of code size. Since the code size of VLIW depends on the number of slots, these results are based on the non-VLIW code size with CAIC applied.

By adapting the CAIC method, the program size of the QuickSort reduced 7.2% from using only 32-bit basic instructions, and indicating a 2.1% increasing rate from the combination program to the compressed program. Additionally, the program size of the Dhrystone reduced 25.9% from using only 32-bit basic instructions Therefore, the code size increase due to the CAIC method is considered sufficiently small.

The primary cause in the code size increase for the QuickSort program when adapting the CAIC method is considered the load/store instructions with negative immediate values. These instructions are frequently used in accessing the stack. As shown in Table 2, load/store instructions with negative immediate values need to be modified to create the address in a register because compressed load/store instructions cannot have negative immediate values. As a result, these instructions are broken down into at least 3 compressed instructions, and if the negative value becomes larger or if registers that cannot be used with the c.lw instruction are involved, the number of instructions increase further.

In the Dhrystone benchmark, 8-bit and 16-bit store instructions are the main cause of code size increase. Since compressed instructions do not provide 8-bit and 16-bit load/store instructions, these instructions are implemented by loading the destination data, modifying specific bits, and storing it back. However, compressed instructions are not suited for generating large values, and it takes time to generate masks to modify specific bits. Consequently, a single sb (store byte) instruction has the possibility to be converted into more than 10 compressed instructions in the worst case.

Subsequently, we decomposed the basic instructions addi, lw, and jalr into multiple compressed instructions and analyzed the occurrence frequency of each. The results are presented in Table 7. The Reduction rate in Table 7 indicates the code size reduction rate from assembly programs composed of basic instructions. This value is positive when the code size decreases and negative when the code size increases. Instructions that do not have the number of splits corresponding to Table 1

Table 6: Benchmark program code size.

	QuickSort	Dhrystone
Integer (bytes)	2,872	4,728
Compressed by CAIC method (bytes)	2,664	3,504
Integer and Compressed (bytes)	2,608	3,104

Table 7: Breakdown of converted instructions.

		1 inst.	2 inst.	3 inst.	4 inst.	Total	Reduction rate
QuickSort	addi	31	0	0	0	31	50%
	lw	20	8	10	7	45	-4%
	jalr	0	2	1	3	6	-58%
Dhrystone	addi	34	1	1	0	36	46%
	lw	30	0	4	6	40	18%
	jalr	0	0	0	3	3	-100%

through Table 3 are due to an increase or decrease in the number of instructions by the generation of immediate values. For example, if a single lw instruction is divided into 3 compressed instructions, the operation is decomposed into the following steps: storing the data of src1 into an available register rx, adding an immediate value to rx, and executing c.lw.

These results indicate that the addi instruction contributes to a decrease in the number of instructions, while the jalr instruction tends to increase the number of instructions. On the other hand, the reduction rate of the lw instruction was observed to vary depending on the program. Although a comprehensive investigation using various benchmark programs is currently difficult due to the absence of a compiler, we plan a more detailed analysis after the compiler under development is completed.

6.2 Execution Time

For execution time evaluation, we used a QuickSort program and a highly parallel and long-running program, as shown in Listing 1. In this paper, we call this program a high-IPC program. This program has been used for evaluation in the research of AnyCore [2], and it is also employed in this paper to ensure evaluation under the same conditions. The high-IPC is written to have high ILP and is suitable for execution on superscalar processors. Additionally, the programs for the RVC-VOI were generated by rearranging the assembly programs converted by the CAIC method into 4-issue VLIW format. Since a compiler for RVC-VOI is still under development, this process was conducted manually.

Listing 1: high-IPC program

```
1  int i,j,arr[10];
2  for(i=0; i<10; i++){
3      for(j=0; j<10; j++){
4          arr[j] = 0;
5      }
6      for(j=0; j<4096; j++){
7              arr[0] = arr[0] + 1;
8              arr[1] = arr[1] + 1;
9              ...
10             arr[9] = arr[9] + 1;
11     }
12 }
```

Table 8: Execution time.

		RVC-VOI	AnyCore1	AnyCore2
QuickSort	Number of execution instructions	3,942	1973	1,969
	Execution time (Cycle)	4,080	15,017	17,319
	IPC	0.97	0.13	0.11
High-IPC program	Number of execution instructions	983,384	573,644	573,665
	Execution time (Cycle)	573,705	577,620	244,985
	IPC	1.71	0.99	2.34

Table 9: Power consumption of each processor at 100 MHz and energy consumption when executing the high-IPC program.

	RVC-VOI	AnyCore1	AnyCore2
Power consumption (W)	0.0030	0.2383	0.4176
Execution time (Cycle)	573,705	577,620	244,985
Runtime energy consumption (J)	1,751	137,646	102,305

Table 8 shows the execution results, including the number of executed instructions, execution time, and IPC for each program. In the QuickSort program, the RVC-VOI processor executes 3,942 instructions, while the AnyCore processors executes 1,970 instructions, indicating a difference of about 2 times in the number of executed instructions. Similarly, in the high-IPC program, the RVC-VOI processor executes 983,384 instructions compared to about 573,600 instructions on the AnyCore processors, resulting in a difference of about 1.7 times. The reason for this is that the CAIC method divides operations that typically be executed with a single basic instruction into multiple compressed instructions.

In the QuickSort program, the RVC-VOI processor achieved the fastest execution time. This result can be attributed to the low parallelism of the QuickSort algorithm. The processors generated by AnyCore had significantly slower execution times due to penalties associated with load instructions. Superscalar processors execute instructions as soon as they are ready by scheduling them internally. Consequently, AnyCore2, which fetches more instructions simultaneously than AnyCore1, retains a larger number of in-flight instructions within the processor. This leads to delayed execution timing of load instructions, which become a bottleneck. As a result, AnyCore2 exhibited the slowest execution time, while RVC-VOI, which incurred minimal penalties from load instructions, achieved the fastest execution time.

In the high-IPC program, AnyCore2 achieved the fastest execution time. The bottleneck for RVC-VOI in this program lies in branch instructions. AnyCore is equipped with a branch predictor, and the penalty of branch mispredictions occurs only at the end branch instruction of the for-loop shown in Listing 1. On the other hand, RVC-VOI does not have a branch predictor, causing a 2-cycle penalty each time the for-loop branches. Given that the high-IPC program executes about 40,000 loop iterations, this penalty becomes significant. Nevertheless, even though the CAIC method increased the number of instructions to be executed by 1.7 times that of a superscalar processor, RVC-VOI demonstrated faster execution than AnyCore1, indicating that it possesses sufficient processing capability.

6.3 Energy Consumption

This section estimates power consumption using Synopsys PrimeTime (Ver. U-2022.12-SP5-4), as shown in Table 5, and evaluates the total energy consumption required during the high-IPC program execution. Table 9 shows the estimated power consumption of the RVC-VOI and the comparison processors at a frequency of 100MHz. To shorten the design period, we designed the VLSI using the ROHM 0.18μm CMOS process in this study. As a result, the clock frequency was relatively low.

Table 10: Area evaluation.

	RVC-VOI	AnyCore1	AnyCore2
Number of gates*	70,210	2,698,259	4,920,978

* In terms of equivalent NAND gates.

Table 11: Area breakdown of each processor.

	RVC-VOI	AnyCore1	AnyCore2
Fetch	2,944	2,128,994	4,119,850
Decode	59,252	146,939	289,907
Rename	0	10,236	28,285
Dispatch	0	67	399
Issue	0	55,372	81,107
Execute	12,678	261,434	278,747
Writeback	1,532	95,444	123,296

Since RVC-VOI is currently under development, the processors are evaluated at the same frequency in this paper, and a more realistic evaluation is a future work. along with the estimated energy consumption when executing the high-IPC program. The total energy consumption was derived based on the number of execution cycles obtained for the high-IPC program in Section 6.2.

As a result, the total energy consumption of RVC-VOI was successfully reduced to 1.3%, representing a 98.7% reduction compared to AnyCore1, and to 1.7%, representing a 98.3% reduction compared to AnyCore2. Therefore, RVC-VOI achieved a significant reduction in energy consumption compared to superscalar processors.

6.4 Circuit Scales

RVC-VOI and the processors were designed by using AnyCore to evaluate the area by logical synthesis using Synopsys Design Compiler Version S-2021.06-SP2. Table 10 shows the circuit evaluation results in terms of the number of equivalent NAND gates.

Compared to the number of equivalent NAND gates of AnyCore2 which has the same issue widths as RVC-VOI, the RVC-VOI area was reduced by 98.6%, achieving only 1.4% of the original footprint. Similarly, compared to the number of equivalent NAND gates of AnyCore1 which is the smallest configuration of AnyCore, the RVC-VOI area was reduced by 97.4%, resulting in 2.6%. This reduction is attributed to RVC-VOI not having the complex configurations of Out-of-Order processors, as well as its simpler architecture and not having a branch predictor.

A comparison of the circuit scales of the fetch stage shows that the reduction of a branch predictor contributes to a significant reduction in circuit scales.

Table 11 shows the area breakdown for each processor. Due to the complexity of the architecture, superscalar processors consist of many pipeline stages. In contrast, VLIW processors have a simpler design, allowing certain stages to be eliminated. Additionally, branch predictor of AnyCore is configured in the fetch stage.

Figure 8 shows the circuit scales based on changes in the number of ALU slots in RVC-VOI. It can be observed that the circuit scales increase proportionally with the number of slots. However, even at the maximum configuration of 8-slot, the scales remain smaller than that the superscalar processors shown in Table 10, indicating that the RVC-VOI maintains a compact design.

Additionally, the layout design of the RVC-VOI processor configured with 4-slot was performed using Synopsys IC Compiler II (Ver. R-2020.09-SP5). The layout diagram is shown in Figure 9. As a result, the chip size was measured to be $1{,}089\mu m \times 1{,}088\mu m$. Since the size of the AnyCore chip within Reference [2] is $25mm^2$, the RVC-VOI chip is significantly smaller in comparison.

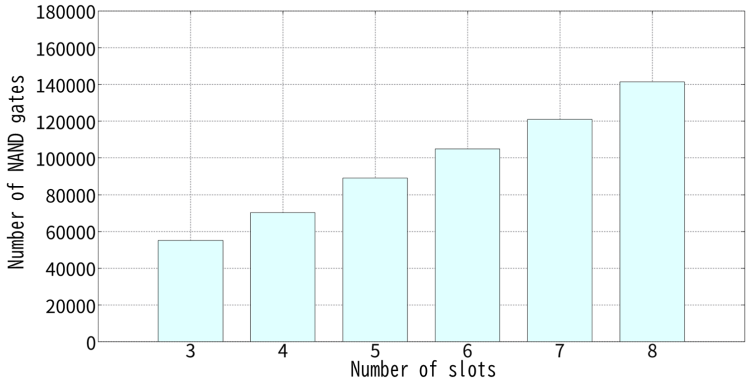

Figure 8: Circuit scales of RVC-VOI at each parameter.

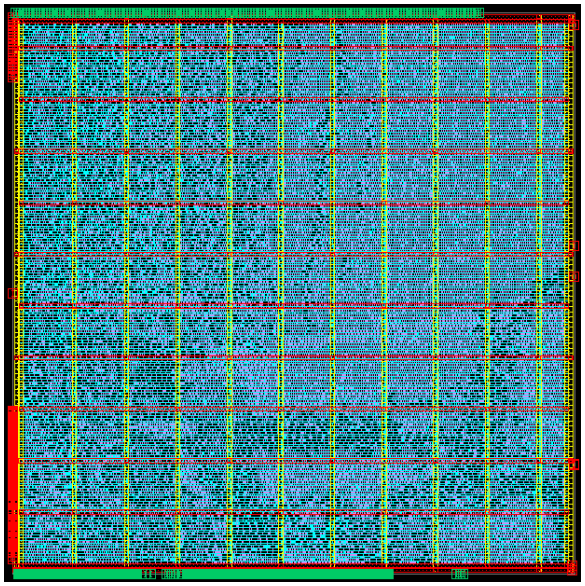

Figure 9: Layout diagram of a 4-slot RVC-VOI.

7 Conclusion and Future Work

In this paper, we proposed a method called CAIC, which converts all integer instructions to compressed instructions, and an extended VLIW processor called RVC-VOI that adapts CAIC method and implements privileged instructions. Through the CAIC method, compressed instructions, which traditionally required simultaneous use with basic instructions in RISC-V, can be utilized independently. Additionally, the RVC-VOI architecture enables the implementation of compressed instructions along with privileged instructions of different instruction lengths, without increasing the number of slots and nop instructions.

The CAIC method achieved code size reductions of 7.2% for the QuickSort program and 25.9% for the Dhrystone program compared to programs consisting only of basic instructions. The evaluation of RVC-VOI covered execution time, energy consumption, and circuit scale. For execution time, RVC-VOI outperformed the superscalar processor in the QuickSort program and indicated sufficient execution times in the high-IPC program. In terms of energy consumption, it achieved a reduction of up to 1.3%. Additionally, the circuit scale was successfully reduced to 2.8% compared to the minimal

configuration of AnyCore. Therefore, RVC-VOI achieved performance comparable to superscalar processors while significantly reducing both energy consumption and circuit scale.

In future work, we plan to improve the CAIC method for more efficient conversion by supporting the Zc extension that provides byte and half word size memory access compressed instructions, and developing a compiler that supports the RVC-VOI architecture to evaluate more realistic conditions. We also plan to design the VLSI of RVC-VOI using more advanced process technology.

Acknowledgment

This work was supported VLSI Design and Education Center (VDEC) of the University of Tokyo in collaboration with NIHON SYNOPSYS G.K., Cadence Design Systems and Rohm Corporation. The authors are grateful for them for their support. We are also grateful to Eric Rotenberg and his students for research collaboration on FabScalar and AnyCore at North Carolina State University.

References

[1] Keyan Cao, Yefan Liu, Gongjie Meng, and Qimeng Sun. An overview on edge computing research. *IEEE Access*, 8:85714–85728, 2020.

[2] Rangeen Basu Roy Chowdhury, Anil K. Kannepalli, Sungkwan Ku, and Eric Rotenberg. Anycore: A synthesizable rtl model for exploring and fabricating adaptive superscalar cores. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 214–224, 2016.

[3] Eric Chun, Zeshan Chishti, and T. N. Vijaykumar. Shapeshifter: Dynamically changing pipeline width and speed to address process variations. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 411–422, 2008.

[4] Islam Elsadek and Eslam Yahya Tawfik. Risc-v resource-constrained cores: A survey and energy comparison. In *2021 19th IEEE International New Circuits and Systems Conference (NEWCAS)*, pages 1–5, 2021.

[5] RISC-V Foundation. riscv-gnu-toolchain. `https://github.com/riscv-collab/riscv-gnu-toolchain`, Accessed on July 31, 2024.

[6] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K. Gürkaynak, and Luca Benini. Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(10):2700–2713, 2017.

[7] OpenHW Group. Cv32e40p. `https://github.com/openhwgroup/cv32e40p`, Accessed on July 31, 2024.

[8] Mark Horowitz. Computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14, 2014.

[9] RISC-V international. riscv-tests. `https://github.com/riscv-software-src/riscv-tests`, Accessed on August 10, 2024.

[10] RISC-V international. Volume 1, unprivileged specification version 20240411, 2024. `https://riscv.org/technical/specifications`, Accessed on June 26, 2024.

[11] Takuto Kanamori and Kenji Kise. Rvcorep-32ic: An optimized risc- v soft processor supporting the compressed instructions. In *2021 IEEE 14th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, pages 38–45, 2021.

[12] Vasileios Kontorinis, Amirali Shayan, Dean M. Tullsen, and Rakesh Kumar. Reducing peak power with a table-driven adaptive processor core. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 189–200, 2009.

[13] Nguyen My Qui, Chang Hong Lin, and Poki Chen. Design and implementation of a 256-bit risc-v-based dynamically scheduled very long instruction word on fpga. *IEEE Access*, 8:172996–173007, 2020.

[14] S. Segars, K. Clarke, and L. Goudge. Embedded control problems, thumb, and the arm7tdmi. *IEEE Micro*, 15(5):22–30, 1995.

[15] Kinza Shafique, Bilal A. Khawaja, Farah Sabir, Sameer Qazi, and Muhammad Mustaqim. Internet of things (iot) for next-generation smart systems: A review of current challenges, future trends and prospects for emerging 5g-iot scenarios. *IEEE Access*, 8:23022–23040, 2020.

[16] Manoj Sharma, Ekansh Bhatnagar, Kartik Puri, Amitav Mitra, and Jatin Agarwal. A survey of risc-v cpu for iot applications. In *Proceedings of the International Conference on Innovative Computing & Communication (ICICC)*, 2022.

[17] Haruhiro Tanaka and Takahiro Sasaki. Extended vliw processor based on risc-v compressed instruction set. In *2024 Twelfth International Symposium on Computing and Networking Workshops (CANDARW)*, pages 360–364, 2024.

[18] Stephan Wong, Thijs van As, and Geoffrey Brown. ρ-vex: A reconfigurable and extensible softcore vliw processor. In *2008 International Conference on Field-Programmable Technology*, pages 369–372, 2008.

[19] Andrea Zanella, Nicola Bui, Angelo Castellani, Lorenzo Vangelista, and Michele Zorzi. Internet of things for smart cities. *IEEE Internet of Things Journal*, 1(1):22–32, 2014.