Bulletin of Networking, Computing, Systems, and Software – www.bncss.org, ISSN 2186-5140
Volume 15, Number 1, pages 19–24, January 2026

Research on instruction scheduling for extended VLIW based on RISC-V compression instruction set

Yoshiki Yamauchi
Graduate School of Information Science & Technology
Aichi Prefectural University
Aichi, Japan
im241013@cis.aichi-pu.ac.jp

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

***Abstract*—In recent years, the Internet of Things (IoT) has become widespread across a broad range of systems. Embedded processors used in such IoT devices have tight constraints on area, energy budget, and other resources. As a processor architecture for embedded systems, Very Long Instruction Word (VLIW) is one of the useful methods for implementing embedded systems. In VLIW processors, a dedicated compiler statically schedules instructions according to the design, thereby achieving high Instruction Level Parallelism (ILP) without requiring a complex internal structure. In addition, the RISC-V compressed instructions set are well suited to embedded processors. However, compressed instructions have the problem that they must be used in combination with basic instructions. To solve the problem, a translator that converts basic instructions to compressed instruction has been proposed. Furthermore, a VLIW processor focusing on compressed instructions have also been proposed, with parameterizable numbers of functional units (e.g., ALUs). However, there is no VLIW compiler for RISC-V compression instructions, so instruction schedules must still be managed manually to use the processor. Furthermore, because this VLIW architecture supports varying the number of functional units via a configuration file for automated design, the scheduler must adapt its scheduling decisions to the VLIW configuration. To address the lack of a VLIW compiler supporting the RISC-V compressed instruction and the need to perform instruction scheduling for VLIW architectures whose configurations vary according to a configuration file, we propose a VLIW instruction scheduler for the RISC-V compressed instruction that supports configuration file defined VLIW architectures, and we validate its functional correctness and evaluate its performance. As a result, we verified that the scheduler performs correctly under varying VLIW configurations and operates with low overhead relative to hand scheduling.**

***Index Terms*—RISC-V, compressed instruction set, computer architecture, VLIW, compiler, instruction scheduling.**

I. Introduction

In recent years, the Internet of Things (IoT) [1]–[5] has become widespread across a broader range of systems. The proliferation of IoT is extending to a diverse array of systems, ranging from conventional electrical devices to those that have traditionally operated without electricity. 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 [6]. Consequently, general purpose processors are typically tend to be avoided in such designs because they incur larger silicon area and higher energy consumption.

As a processor architecture for embedded systems, Very Long Instruction Word (VLIW) has been the subject of extensive research and is regarded as an effective approach [7]–[11].

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. As a result, VLIW processors can achieve high Instruction Level Parallelism (ILP) without complex internal structure. Meanwhile, RISC-V is an open-source and free to use instruction set architecture (ISA) that has attracted significant attention from both academia and industry [12]. Owing to its open specification, a wide range of RISC-V processors such as BOOM [13], Rocket Chip [14], RVCoreP [15], and PicoRV32 [16] has been proposed. The RISC-V compressed instructions, one of the standard ISA extensions, can reduce code size by supplying 16-bit encodings for common instructions. Therefore, the RISC-V compressed instructions are well suited to embedded processors. As one of the implementations of VLIW processor based on RISC-V, RVC-VOI is proposed [17]. One of the features of this processor is that it adopts only compression (16bit) instructions. In addition, to support RVC-VOI, a translator has been developed that converts RISC-V base-ISA instructions into their compressed instructions. Furthermore, this processor, which targets compressed instructions, also has the capability to parameterize the number of functional units (e.g., ALUs). Given the characteristics of VLIW execution and the compressed instructions, RVC-VOI is a promising candidate for embedded processors.

However, there is no VLIW compiler for RISC-V compression instructions. VLIW differs from superscalar, which performs instruction scheduling on hardware, in that the compiler performs instruction scheduling. Therefore, in order to use the hardware, it is currently necessary to schedule instructions by hand. Furthermore, because this VLIW architecture supports varying the number of functional units via a configuration file for automated design, the scheduler must adapt its scheduling decisions to the VLIW configuration.

To solve the above problems, we develop a VLIW in-

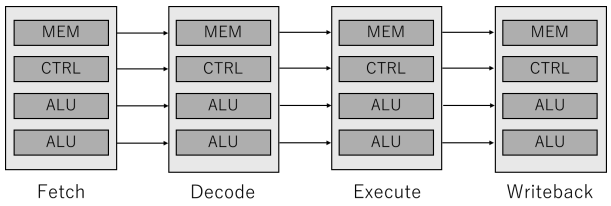

Fig. 1. 4-issue VLIW structure.

struction scheduler for RISC-V compression instruction that supports configuration file defined VLIW architectures and examine instruction scheduling methods. The scheduler receives an assembly codes composed of compressed instructions and converts the received instructions into VLIW instructions consisting of a memory access instruction, a control instruction, and two ALU instructions. In this process, the current scheduler inserts no-operation instructions to avoid register dependencies.

As part of the verification process, we actually performed the instruction conversion and confirmed the operation of the program using the converted instructions for VLIW processors with varying numbers of slots.

II. Background

RISC-V is one of the ISAs that has attracted the most attention recently [12]. While most commercial ISAs require license fees, RISC-V is an open-source ISA and free to use. By its openness, RISC-V has attracted interest not only from academia but also from industry. Consequently, many companies and universities are currently entering the development of RISC-V. Additionally, RISC-V consists of a 32-bit basic integer instruction 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 is one of the useful methods for implementing embedded systems. 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 example of 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 algorithms 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 algorithms, 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 a 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.

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 a foundation for VLIW architectures. Currently, RVC-VOI is proposed as VLIW hardware for RISC-V compression instructions. Still, because there is no dedicated compiler that accounts for VLIW configurations specified by a configuration file, the instructions must be hand-scheduled.

III. Related Works

As a VLIW based on the RISC-V ISA, Qui, Lin and Chen propose the RVCoreP-32IC, a 256-bit VLIW processor with a dynamic scheduler [18]. This processor scheduler detects the dependencies of eight instructions fetched at the same time and generates VLIW instructions within the processor. This method enables the execution of VLIW instructions without the need for a dedicated compiler. However, this processor can't fully utilize ILP, because it only detects the dependencies of eight fetched instructions. Additionally, the scheduling method increases the amount of hardware.

Fujitsu developed the FR500 as a VLIW processor for commercial products [19]. This processor can be installed in small circuits and features user-defined instructions. However, since the compiler design is proprietary and and the official download of the compiler has been discontinued, so the method of scheduling instructions is unknown.

Intel and Hewlett-Packard jointly developed IA-64 as an ISA similar to VLIW [20]. This ISA adopts an architecture called EPIC architecture, which is an improved version of VLIW. EPIC architecture has the feature of being able to change the instruction width by using identification flags. However, the design of the dedicated compiler is proprietary, so the method of scheduling instructions is unknown.

IV. Scheduling methods for compression instructions

During scheduler development, the choice of instruction scheduling method has a substantial impact on performance. Accordingly, Section IV describes several instruction scheduling strategies commonly employed in VLIW architectures.

A. List Scheduling

List scheduling is a standard instruction scheduling technique widely employed by compilers [21]. This algorithm proceeds as follows:

a) For each instruction, assign execution latency, dependency information, and a scheduling priority,
b) From the ready set, place instructions into the available VLIW slots in descending order of priority,
c) Update dependencies and priorities; if no instruction is schedulable, advance to the next cycle (padding any remaining slots with NOPs),
d) Repeat steps b) and c) until all instructions are scheduled.

The list sheduling algorithm described above schedules instructions quickly and efficiently while incurring low compile-time overhead.

B. Trace Scheduling

Trace scheduling is a method of extracting the traces (instruction execution paths) of the instructions most likely to be executed and performing instruction scheduling on those traces [22]. This algorithm proceeds as follows:

a) Using profiling data or programmer-supplied annotations, determine the most probable branch outcomes and, based on this information, select the most likely execution trace.
b) Perform list scheduling of the instructions along the selected trace.
c) Generate compensation code at control-flow split and join points to reconcile discrepancies in register and memory state.
d) Repeat until no unscheduled instructions remain.

Trace scheduling has advantages that are well suited to VLIW, which is strong in static optimization. However, these techniques incur substantial compiler complexity owing to profile analysis and the need to insert compensation code.

V. Problem and Our Implementation Method

A. Problem

RVC-VOI is a VLIW architecture targeting the RISC-V compressed instruction. However, because no compiler is available to automatically schedule compressed instructions for this architecture, it cannot be utilized effectively. Moreover, because the slot configuration varies according to the configuration file, the scheduler must adapt its scheduling decisions accordingly. To solve this problem, this section describes a scheduler that reorders code consisting exclusively of RISC-V compressed instructions into instruction bundles for a VLIW architecture with a configurable number of slots. Consequently, as illustrated in Fig. 2, even when executing the same program, different VLIW slot configurations would each require a separate compiler tailored to that configuration, which is problematic. To solve this problem, this section describes a scheduler that reorders code consisting exclusively of RISC-V compressed instructions into instruction bundles for a VLIW architecture with a configurable number of slots.

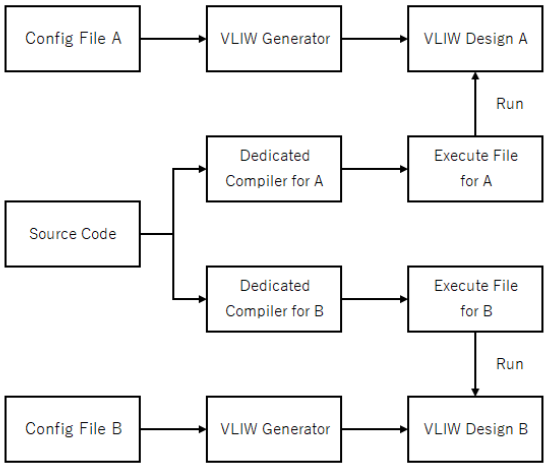

Fig. 2. Relationship Between Different Slot Configurations of RVC-VOI and the Compiler.

B. Our proposed method

As a compiler strategy for RVC-VOI, it is not practical to implement a separate compiler for each possible slot configuration. To resolve the issue illustrated in Fig.2, we adopt the approach shown in Fig.3, in which the compiler reads the RVC-VOI configuration file, obtains the slot configuration from it, and schedules instructions based on that information. Given the time constraints of this study, our current scheduler reads a configuration file, determines the VLIW slot configuration from it, and employs a simple instruction scheduling method based on that configuration. We describe the scheduler algorithm as follows:

a) Read the slot configuration from the configuration file and create the corresponding variables based on this configuration.
b) The scheduler scans the code in program order and places each instruction into the current issue bundle, while also recording its associated register-usage information with the bundle entry,
c) If the instruction cannot be placed in any slot, or if a register-dependency hazard is detected from the recorded register-usage information, the remaining slots of the bundle are padded with NOP (no operation), and scheduling proceeds to a new bundle,
d) Repeat steps b) and c) until all instructions are scheduled.

In Section V-C, we explain in detail, using pseudocode, how the scheduler identifies instructions from the input code, checks for register dependences and slot availability, and places the instructions into the appropriate VLIW slots. To illustrate the foregoing pseudocode, in Section V-D, we present an example execution of the code.

C. Structure

This section uses pseudocode for identifying and scheduling the RISC-V addi instruction to explain in detail how the

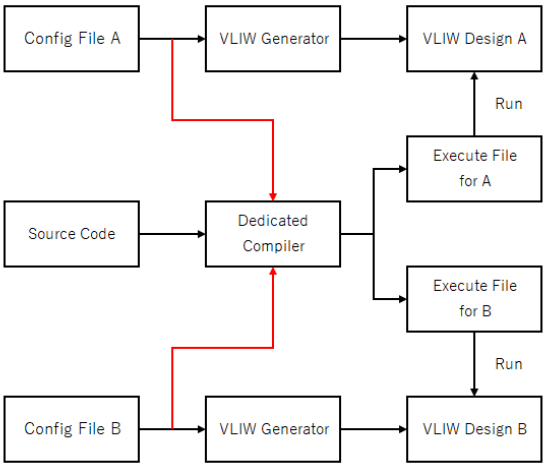

Fig. 3. A Compiler Approach for RVC-VOI with Variable Slot Configurations.

scheduler recognizes the instruction, determines whether to flush the current bundle, and inserts the instruction into an appropriate slot. Listing 1 presents the scheduler pseudocode that schedules the RISC-V addi instruction under the slot configuration shown in Fig. 1, namely a four-slot VLIW issue bundle with one memory-access slot, one control-flow slot, and two ALU slots. In this pseudo-code, inst, reg_name, and imm store the parsed instruction, register operand, and immediate value, respectively. slot1-slot4 store the instruction placed in each VLIW slot, while reg1-reg4 record the register-usage information associated with those slots. First, verify that the instruction is addi, then parse its register operand(s) and immediate value. Next, check that the target issue-bundle slot is available and that the registers used by the operation do not introduce any data-dependence hazards.. At this point, if the target slot is unavailable or a register/data-dependence hazard is detected, the scheduler writes the slot ' s contents to the output file and flushes the slot. After that, we store the instruction together with its register-usage information and use a continue statement to return to the loop and read the next line. The actual scheduler code comprises similarly structured handlers for other instructions, organized by operand formats.

Listing 1. Pseudo-code of scheduler routine that schedules the addi instruction

```
# Parse one line like: "<inst> <reg>, <imm>"
parsed = split(str, pattern = "<inst>␣<reg>,␣<
  imm>")
# returns (inst, reg_name, imm) or None

if (parsed is not None){
    inst, reg_name, imm = parsed;

    # Handle only the 'c.addi' instruction
    if (inst == "c.addi"){

        # Helper function: Fill the slot blanks
        # with c.nop and flush the slot and
        # state after output
        def flush_slots(){
            for (i = 1; i < 5; i++){
                if (sloti == ""){
                   output(c.nop\n);
                }else{
                  output(sloti);
                }
                # blank line between bundles
                output("\n");
            }

            # Frush all slots
            slot1 = slot2 = slot3 = slot4 = ""

            # Frush all tracked destination
            # registers
            reg1 = reg2 = reg3 = reg4 = ""
        }

        # Condition to flush:
        # 1. The target slot is not empty
        # OR
        # 2. Register hazard (source register
        #    matches tracked destination
        #    register)
        if ((slot1 != "" and slot2 != "") or
            (reg_name == reg1-4){
            flush_slots();
        }

        # Place the instruction into whichever
        # of the two slots is currently free
        if (slot1 == ""){
            # store the original text of the
              instruction
            slot1 = str;

            # remember tracked destination
              register
            reg1 = reg_name;
        }else{
            slot2 = str;
            reg2 = reg_name;
        }

        # Go process the next input line
        continue;
    }
```

D. Example Trace

In this section, we illustrate the scheduler's operation using the instruction sequence shown in Listing 2. First, because the first three instructions have no data dependences, we place them into the available slots together with their register-usage information. Upon reading the fourth instruction, the scheduler detects that the ALU-designated slot is unavailable. Consequently, before placing the instruction, it emits the current bundle and then inserts the instruction into the next bundle. Because the fifth instruction has no data dependences, we place it into an available slot. In this instruction sequence, there is a RAW (read-after-write) dependence on r1 between the fifth and sixth instructions. Consequently, before placing the sixth instruction, the scheduler emits the current bundle and then inserts the instruction into the next bundle.

Listing 2. Instruction sequence used in the illustrative example

```
c.addi      r1, 1
c.lw        r2, 0(r3)
c.addi      r4, 2
c.addi      r5, 3
c.lw        r6, 0(r7)
c.addi      r8, r6, 4
```

VI. Evaluations

In this section, to verify that our scheduler operates correctly for different VLIW configurations, we perform instruction scheduling under three slot configurations:

i) A three-slot configuration with one memory-access slot, one control-flow slot, and one ALU slot,

ii) A four-slot configuration with one memory-access slot, one control-flow slot, and two ALU slots,
iii) A six-slot configuration with one memory-access slot, one control-flow slot, and four ALU slots.

We then compare the resulting schedules with manually scheduled code and evaluate code size and execution time. For the evaluation of code size and execution time, we used long-running, highly parallel program shown in Listing 3. In this paper, we refer to this program as High IPC (Instruction Per Cycle) program. This program has been used as an evaluation workload in previous studies on AnyCore (an automated processor-design tool for superscalar architectures) [23]; we adopt it here as well to ensure that our evaluation is conducted under the same conditions and is directly comparable to the previous results. Table I presents code size, whereas Table II presents IPC.

Our evaluation confirms that the scheduler produces correct executions across multiple VLIW configurations defined by the configuration file. However, relative to the hand-scheduled baseline, code size increases by up to 18.2% and IPC decreases by up to 0.64. We hypothesize that these results arise because the schedules produced by our scheduler are not as thoroughly optimized as hand-crafted schedules. Moreover, since our current scheduling method is simple, we anticipate that implementing the strategies outlined in Section IV would narrow the performance gap with the hand-scheduled baseline.

Listing 3. High IPC program

```
1  int i, j;
2  int arr[10];
3
4  for(i = 0; i < 10; i++){
5      for(j = 0; j < 10; j++){
6          arr[j] = 0;
7      }
8
9      for(j = 0; j < 4096; j++){
10         arr[0] = arr[0] + 1;
11         arr[1] = arr[1] + 1;
12         ...
13         arr[9] = arr[9] + 1;
14     }
15 }
```

TABLE I
HIGH IPC PROGRAM CODE SIZE (BYTES)

	Our scheduler	Hand-schduled
3-slot	4,425,462	4,425,402
4-slot	4,261,744	3,606,280
6-slot	4,426,008	3,934,488

TABLE II
HIGH IPC PROGRAM IPC

	Our scheduler	Hand-scheduled
3-slot	1.11	1.28
4-slot	1.54	2.18
6-slot	2.78	3.12

VII. Conclusion and Future Work

In this paper, we develop a VLIW instruction scheduler for RISC-V compression instructions, and evaluate its performance. The results show that, compared with the hand-scheduled baseline, code size increased by up to 18.2% and IPC decreased by up to 0.64. Nevertheless, the proposed method fully automates the scheduling workflow that previously required manual effort, even for VLIW processors whose configurations vary according to a configuration file. Furthermore, incorporating techniques such as trace scheduling is expected to narrow the gap to the hand-scheduled baseline. As future work, we plan to implement the candidate scheduling strategies in our scheduler and evaluate them.

References

[1] J. Yan, Y. Meng, L. Lu, and L. Li, “Industrial big data in an industry 4.0 environment: Challenges, schemes, and applications for predictive maintenance,” Ieee Access, vol.5, pp.23484–23491, 2017.
[2] Y. He, J. Guo, and X. Zheng, “From surveillance to digital twin: Challenges and recent advances of signal processing for industrial internet of things,” IEEE Signal Processing Magazine, vol.35, no.5, pp.120–129, 2018.
[3] E. Bertino, M.R. Jahanshahi, A. Singla, and R.T. Wu, “Intelligent iot systems for civil infrastructure health monitoring: a research roadmap,” Discover Internet of Things, vol.1, pp.1–11, 2021.
[4] Y. Cui, F. Liu, X. Jing, and J. Mu, “Integrating sensing and communications for ubiquitous iot: Applications, trends, and challenges,” IEEE Network, vol.35, no.5, pp.158–167, 2021.
[5] P. Bamurigire and A. Vodacek, “Validating algorithms designed for fertilization control in rice farming system,” Discover Internet of Things, vol.3, no.1, p.4, 2023.
[6] M. Horowitz, “Computing’s energy problem (and what we can do about it),” 2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC), San Francisco, CA, USA, pp. 10–14, February 2014
[7] L. Benini, D. Bruni, M. Chinosi, C. Silvano, V. Zaccaria and R. Zafalon, ”A power modeling and estimation framework for VLIW-based embedded systems.”,in Proc. Int. Workshop on Power and Timing Modeling, Optimization and Simulation PATMOS, vol.1, pp.10–113, 2001.
[8] M. Sami, D. Sciuto, C. Silvano, V. Zaccaria and R. Zafalon, ”Low-power data forwarding for VLIW embedded architectures.”, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol.10, no.5, pp.614–622, October, 2002.
[9] L. Benini, D. Bruni, M. Chinosi, C. Silvano, V. Zaccaria and R. Zafalon, ”A framework for modeling and estimating the energy dissipation of VLIW-based embedded systems.”, Design Automation for Embedded Systems, vol.7, pp.183–203, 2002.
[10] C. H. Lin, Y. Xie, and W. Wolf, ”LZW-based code compression for VLIW embedded systems,”, in Proc. Design, Automation and Test in Europe Conf. and Exhibition, vol. 3, pp.76–81, 2004.
[11] Yuan Xie, W. Wolf and H. Lekatsas, ”Code compression for embedded VLIW processors using variable-to-fixed coding”, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol.14, pp.525–536, May 2006.
[12] RISC-V international, RISC-V Specifications, “Volume 1, Unprivileged Specification version 20240411” [Online] Available: https://riscv.org/technical/specifications/, Accessed: June 26, 2024.
[13] RISC-V BOOM [Online] Available: https://boom-core.org, Accessed: September 4, 2025.
[14] Rocket Chip Generator, Github, [Online] Available: https://github.com/chipsalliance/rocket-chip, Accessed: September 4, 2025.
[15] H. Miyazaki, T. Kanamori, M. A. Islam, and K. Kise, ”RVCoreP: An optimized RISC-V soft processor of five-stage pipelining,”, IEICE Transactions on Information and Systems, vol. E103-D, no.12, pp. 2494–2503, December 2020.
[16] PicoRV32 - A Size-Optimized RISC-V CPU, [Online] Avaiable: https://github.com/YosysHQ/picorv32, Accessed: September 4, 2025.

[17] Haruhiro Tanaka and Takahiro Sasaki, "Extended VLIW Processor Based on RISC-V Compressed Instruction Set", Proc. of the Twelfth International Symposium on Computing and Networking Workshops (CANDARW), pp. 360–364, 2024.

[18] N. M. Qui, C. H. Lin and P. Chen, "Design and Implementation of a 256-Bit RISC-V-Based Dynamically Scheduled Very Long Instruction Word on FPGA," in IEEE Access, vol. 8, pp. 172996–173007, September 2020.

[19] Fujitsu, "FR500 VLIW-architecture High-performance Embedded Microprocessor", [Online]. Available: https://www.fujitsu.com/global/documents/about/resources/publications/fstj/archives/vol36-1/paper06.pdf

[20] Intel, "IA-64 Application Developer's Architecture Guide", [Online]. Available: https://www.ece.lsu.edu/ee4720/ia-64.pdf, Accessed: July 19, 2025.

[21] PB Gibbons, SS Muchnick, "Efficient instruction scheduling for a pipelined architecture.", Proceedings of the 1986 SIGPLAN symposium on Compiler construction, 1986.

[22] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction,", IEEE Transactions on Computers, vol. C-30, no. 7, pp. 478–490, July 1981.

[23] R. B. R. Chowdhury, A. K. Kannepalli, S. Ku and E. Rotenberg, "AnyCore: A synthesizable RTL model for exploring and fabricating adaptive superscalar cores," 2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Uppsala, Sweden, pp. 214–224, April 2016.