Clockhands: Rename-free Instruction Set Architecture for Out-of-order Processors

Toru Koizumi
Nagoya Institute of Technology
Aichi, Japan
koizumi@nitech.ac.jp

Ryota Shioya
The University of Tokyo
Tokyo, Japan
shioya@ci.i.u-tokyo.ac.jp

Shu Sugita
The University of Tokyo
Tokyo, Japan
sugita@mtl.t.u-tokyo.ac.jp

Taichi Amano
The University of Tokyo
Tokyo, Japan
amano@mtl.t.u-tokyo.ac.jp

Yuya Degawa
The University of Tokyo
Tokyo, Japan
degawa@mtl.t.u-tokyo.ac.jp

Junichiro Kadomoto
The University of Tokyo
Tokyo, Japan
kadomoto@mtl.t.u-tokyo.ac.jp

Hidetsugu Irie
The University of Tokyo
Tokyo, Japan
irie@mtl.t.u-tokyo.ac.jp

Shuichi Sakai
The University of Tokyo
Tokyo, Japan
sakai@mtl.t.u-tokyo.ac.jp

ABSTRACT

Out-of-order superscalar processors are currently the only architecture that speeds up irregular programs, but they suffer from poor power efficiency. To tackle this issue, we focused on how to specify register operands. Specifying operands by register names, as conventional RISC does, requires register renaming, resulting in poor power efficiency and preventing an increase in the front-end width. In contrast, a recently proposed architecture called STRAIGHT specifies operands by inter-instruction distance, thereby eliminating register renaming. However, STRAIGHT has strong constraints on instruction placement, which generally results in a large increase in the number of instructions.

We propose Clockhands, a novel instruction set architecture that has multiple register groups and specifies a value as "the value written in this register group k times before." Clockhands does not require register renaming as in STRAIGHT. In contrast, Clockhands has much looser constraints on instruction placement than STRAIGHT, allowing programs to be written with almost the same number of instructions as Conventional RISC. We implemented a cycle-accurate simulator, FPGA implementation, and first-step compiler for Clockhands and evaluated benchmarks including SPEC CPU. On a machine with an eight-fetch width, the evaluation results showed that Clockhands consumes 7.4% less energy than RISC while having performance comparable to RISC. This energy reduction increases significantly to 24.4% when simulating a futuristic up-scaled processor with a 16-fetch width, which shows that Clockhands enables a wider front-end.

MICRO '23, October 28-November 1, 2023, Toronto, ON, Canada

ACM ISBN 979-8-4007-0329-4/23/10.
https://doi.org/10.1145/3613424.3614272

CCS CONCEPTS

• **Computer systems organization** → **Superscalar architectures**; *Reduced instruction set computing*; • **Software and its engineering** → *Compilers.*

KEYWORDS

Instruction set architecture, Superscalar processor, Out-of-order execution, Register renaming, Compiler, Power efficiency, Register lifetime

ACM Reference Format:
Toru Koizumi, Ryota Shioya, Shu Sugita, Taichi Amano, Yuya Degawa, Junichiro Kadomoto, Hidetsugu Irie, and Shuichi Sakai. 2023. Clockhands: Rename-free Instruction Set Architecture for Out-of-order Processors. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28-November 1, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3613424.3614272

1 INTRODUCTION

Out-of-order superscalar processors are currently the only architecture that speeds up irregular programs [24, 25]. The execution of interpreted languages such as JavaScript and Python, optimizing compilation of high-level languages, social network analysis, and video games all involve irregular processing that is difficult to parallelize, and CPU single-threaded performance is essential to reduce latency. To speed up these processes, recent commercial processors integrate more massive out-of-order cores to improve the CPU single-thread performance [5, 11, 27, 28].

However, this approach is not power efficient, as it is no longer possible to have all cores be huge out-of-order cores [10, 27]. This is because out-of-order processors consume a lot of power for additional controls, such as register renaming, scheduling, and memory order management, in addition to the computation itself. For scheduling and memory order management, various lightweight techniques have been extensively studied, which can effectively reduce the complexity [1, 2, 15, 16, 18, 32, 34, 36, 38, 44]. For register renaming, although there are also lightweight methods at the microarchitecture level [16, 30, 35, 43], it is difficult to solve the

issue. This is because false dependencies are inevitably caused as long as register numbers are used for specifying operands as in existing instruction set architectures (ISAs).

We focus on and review two existing instruction set architectures in terms of inefficiencies in operand specification. Conventional RISC specifies data transfer points by logical register numbers. Since there are only a finite number of logical registers, this leads to *overwrite* to the logical registers, which inevitably results in false dependencies. Register renaming removes this false dependency and allows efficient out-of-order execution. However, register renaming requires complex circuits, such as multi-port memory and recovery mechanisms, which consume significant power [13, 23, 35, 37, 43].

The other type of ISA, STRAIGHT, does not cause false dependencies but does increase the number of instructions compared to RISC [13, 17]. In STRAIGHT, data producer instructions are specified using inter-instruction distances. STRAIGHT allocates the destination register from a ring buffer to each instruction in turn. Thus, no overwriting occurs, and no false dependencies are caused. As a result, STRAIGHT does not require register renaming to remove false dependencies. However, specifying operands by inter-instruction distance causes strong constraints on instruction placement, which significantly increases the number of instructions executed (e.g., by about 30%) [13].

We illustrate this increase in instruction count using assemblies compiled from the code in Fig. 1(a). Compared with the code compiled for RISC (Fig. 1(b)), the code compiled for STRAIGHT (Fig. 1(c)) has a significantly increased number of instructions. The increase in instructions is caused by the need for additional instructions to adjust the inter-instruction distance. This distance adjustment is necessary because the inter-instruction distance is coupled with instruction execution and can change with it.

We focused on the sequential register allocation from the ring buffer in STRAIGHT. This approach does not cause false dependencies but requires additional instructions, as mentioned above. In particular, this increase is significant in loops because loop constants should be held on registers, as shown in Fig. 1(c). This is because each time a loop is executed, the dynamic inter-instruction distance of references from instructions inside the loop to instructions outside the loop changes. If the *distance* did not change with each loop iteration, there would be no need to add such instructions. However, this problem cannot be solved if the inter-instruction distance is used.

We propose Clockhands, an ISA with multiple register groups. In Clockhands, each operand specifies which register group and how many times ago the value was written in that group, as shown in Fig. 1(d). Because this form of operand specification enables destination register allocation from a ring buffer as in STRAIGHT (although, unlike STRAIGHT, there are multiple ring buffers), Clockhands also does not cause false dependencies and does not require register renaming. Moreover, unlike the specification by inter-instruction distance, distance change on each register group is not coupled with instruction execution; therefore, as in RISC, operands can be referenced by invariant expressions, and few additional instructions are required.

The contributions of this study are as follows:

(a) A simple code

```
void iota( int arr[], int N ) {
  int i;
  for( i = 0; i < N; ++i) {
    arr[i] = i;
  }
}
```

(b) A RISC (RISC-V) assembly

```
iota:
      ble     a1, zero, .L1
      addi    a5, zero, 0     # i
.L3:
      sw      a5, 0(a0)
      addiw a5, a5, 1       # ++i
      addi    a0, a0, 4       # &arr[i]
      bne     a1, a5, .L3
.L1:
      ret     ra
```

(c) STRAIGHT assembly

```
iota:
      ble     [3], zero, .L1
      spaddi  -8
      addi    zero, 0     # i
      sd      [4], 0(sp) # _RetAddr
      mv      [6]         # &arr[i]
      mv      [8]         # N
      j       .L3
.L2:
      addi    [6], 4      # &arr[i]
      mv      [6]         # N relay
      nop                 # dist. adjust
.L3:
      sw      [5], 0([3])
      addiw   [6], 1      # ++i
      bne     [1], [4], .L2
      ld      0(sp)
      spaddi  8
.L1
      ret     [2]
```

(d) Clockhands assembly

```
iota:
      ble     s[2], zero, .L1
      addi    t, zero, 0    # i
      mv      t, s[1]       # &arr[i]
.L3:
      sw      t[1], 0(t[0])
      addiw t, t[1], 1      # ++i
      addi    t, t[1], 4      # &arr[i]
      bne     t[1], s[2], .L3
.L1:
      ret     s[0]
```

Figure 1: (a) Simple code written in C. (b) Assembly code compiled for RISC-V, a conventional RISC architecture. `a0`, `a1`, and `a5` are logical register names. (c) Assembly code compiled for STRAIGHT, an existing rename-free architecture. The shaded parts indicate instructions that have been added compared with the RISC-V code. In STRAIGHT, the destination register of an instruction is not specified and is implicitly assigned from a ring buffer. A source operand of an instruction is specified by an *inter-instruction* distance, such as `[1]`, `[3]`, and `[6]` (e.g., `[3]` represents a reference to the result of three previous instructions.). (d) Assembly code compiled for Clockhands, our proposed architecture. `t` and `s` represent the names of *hands* (i.e., register groups). In Clockhands, the destination register of an instruction is specified by a hand identifier. A source register of an instruction is specified by combining a hand identifier and an *inter-register* distance, denoted as `[2]` (e.g., `t[2]` represents a reference to the result of three previous registers in the hand `t`.).

- We proposed a novel ISA, Clockhands, which does not require register renaming.
- We identified the cause and amount of increased instruction count in existing STRAIGHT and confirmed that it does not occur in Clockhands.
- We presented a basic compilation algorithm for Clockhands.
- We implemented a cycle-accurate simulator and first-step compiler for Clockhands and evaluated benchmarks included in SPEC CPU 2006/2017 [40, 41].
- We evaluated the performance and energy consumption of Clockhands using simulation. On a machine with an eight-fetch width, the evaluation results showed that Clockhands

consumes 7.4% less energy than RISC while having performance comparable to RISC. This energy reduction increases significantly to 24.4% when simulating a futuristic up-scaled processor with a 16-fetch width.
- We implemented soft processors with Clockhands and RISC in a field-programmable gate array (FPGA). The results showed that the look-up table (LUT) consumption of the physical register allocation stage in Clockhands is 1/4 of that in RISC, with a configuration of an eight-fetch width.

2 EXISTING OPERAND SPECIFICATION

2.1 Conventional RISC Architecture

In conventional RISC, the data exchange point is specified by the name of registers. A producer instruction writes the computation result to a logical register specified by the instruction. A consumer instruction reads the computational input from a logical register specified by the instruction.

This format causes false dependencies owing to overwriting. There are only a finite number of logical registers, which are data-exchange points. The number of logical registers is typically 32 in a RISC, which is significantly smaller than the current standard out-of-order window size (e.g., 256) [5, 11, 27, 28]. Therefore, in-flight instruction sequences involve much reuse of logical registers and overwrites to logical registers occur frequently.

Register renaming is necessary to eliminate false dependencies and efficiently execute out-of-order. In register renaming, logical register numbers are converted to physical register numbers. The destination register is assigned an out-of-life physical register obtained from a free list. An instruction sequence whose registers are converted to physical registers by register renaming has no false dependencies; therefore, it can be executed and written to registers out-of-order.

The register renaming is generally implemented with a register map table (RMT) and dependency check logic (DCL) [20, 48]. The RMT is a table that records mappings from logical registers to physical registers. The DCL detects references to and updates of the same logical register to ensure correct renaming when multiple instructions are renamed simultaneously.

These mechanisms rapidly increase circuit area and power consumption when the rename width increases [13, 23, 35, 37, 43]. In general, the RMT consists of a multi-port RAM with a number of ports proportional to its rename width. A multiport RAM generally increases the circuit area in proportion to the square of the number of its ports [47]. In addition, the register rename state can be >500 bits, which must be checkpointed several times in case of exceptions such as branch mispredictions [23, 26, 29]. The DCL consists of matching comparators, which compare logical register numbers. The number of the necessary comparators is also proportional to the square of the rename width [20, 48]. These prevent the expansion of front-end widths and instruction windows and are significant obstacles to extracting instruction-level parallelism (ILP).

2.2 STRAIGHT Architecture

STRAIGHT is an instruction set that does not require renaming by specifying register operands at an inter-instruction distance.

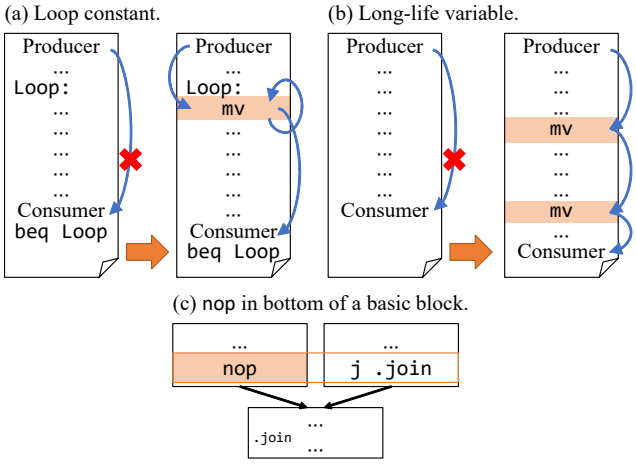


Figure 2: Three causes of the STRAIGHT instruction increase.

Because renaming is not required, it reduces power consumption by eliminating the need for ultra-multiport memory. Moreover, eliminating the rename stage from the front end facilitates front-end width expansion and speeds up recovery. Furthermore, checkpoint capacity is significantly reduced, facilitating instruction window width expansion. Consequently, STRAIGHT can efficiently fabricate processors capable of extracting more ILP.

In contrast, STRAIGHT has several problems, such as the increase in the number of instructions, because of its unique constraints. In the following, we first describe the instruction representation of STRAIGHT and then explain the problem of increased instructions.

2.2.1 Instruction Representation. In STRAIGHT instruction words, the source operand is specified as "use the result of how many instructions ago." The producer-consumer relationship is specified directly in terms of inter-instruction distance rather than indirectly via register numbers. One destination register is implicitly allocated from a ring buffer for each instruction and is used in a write-once manner. Because of allocation constraints from the ring buffer and instruction length constraints, a maximum distance (denoted M) is defined in an ISA that can be used to specify the source operand.

In a STRAIGHT processor, the physical register file is a ring buffer, which is why false dependencies do not occur. Due to the constraint of maximum reference distance, the results of older instructions become sequentially unreferenced. This allows destination registers to be allocated sequentially from the ring buffer, ensuring that overwrites occur only on registers whose lives have expired. This eliminates false dependencies and enables highly efficient out-of-order execution without register renaming.

In STRAIGHT, the process of converting register operands in the instruction word to physical register numbers is simple, unlike in RISC. Because the destinations in the instruction sequence are sequentially numbered with physical register numbers, the source physical register numbers can be obtained by simple subtraction.

2.2.2 Instruction Increase Overhead. The STRAIGHT instruction format causes constraints on instruction placement and requires

the execution of additional instructions to resolve the constraints. STRAIGHT has two fundamental constraints:

- The dynamic inter-instruction distance used to specify operands must be specified statically, not depending on the control flow.
- References must only be to instructions within the maximum reference distance.

To satisfy this constraint, the number of instructions is increased in STRAIGHT in the following cases[1]:

(1) Fig. 2(a): Holding loop constants. A dynamic inter-instruction distance of reference from instructions inside the loop to instructions outside the loop changes with each iteration. To solve this, we must add one relay instruction within the loop.
(2) Fig. 2(b): Holding long-life variables. References that exceed the maximum reference distance cannot be written in the instruction word. To solve this, we must add one relay instruction for each M instruction.
(3) Fig. 2(c): Adjustment just before convergence point. At least one branch instruction exists just before a convergence point because all paths to the convergence point cannot be fall-through. That is, the result generated by the instruction immediately before the convergence point may not exist. Thus, a program that runs correctly through any execution path does not contain a reference to the instruction immediately preceding the convergence point. As a result, a `nop` instruction is required at the end of the fall-through convergence path.

2.2.3 Investigation of Increased Instructions. We investigated the breakdown of these instruction increases. Fig. 3 shows the inevitable increase in the number of instructions when the RISC instruction sequence is converted *as is* to the STRAIGHT instruction sequence (without changing the program structure, i.e., no loop unrolling, and no addition of load/store instructions; just adding `mv`, `nop`, and `jump` instructions). In our experiments here, we obtained the lower bound by conservative counting from traces of RISC-V (a standard RISC instruction set) [46] programs rather than by creating a STRAIGHT compiler to investigate the inevitable increase in the number of instructions independent of the quality of a STRAIGHT compiler. We compiled the SPEC CPU 2006/2017 benchmark [40, 41] for RISC-V and obtained the lower bound of the instruction count increase in STRAIGHT from a trace with 3×10^{13} instructions executed from the beginning. The percentage varies widely from program to program, but on average, an increase in the number of instructions of approximately 35% is unavoidable. The breakdown of the increase is 14% for the holding of loop constants, 14% for the holding of long-life variables, and 6% for the adjustment near the convergence point.

In general, variables in programs that have long lifetimes follow a power-law distribution; therefore STRAIGHT requires many `mv` instructions to hold long-life values. Fig. 4 shows the distribution of lifetimes. This figure shows that the frequency at which a register is defined with a lifetime of N or more is approximately proportional to $1/N$. Hereinafter, we denote the frequency at which a register with a lifetime of *exactly* N is defined as $f(N)$. The above result can be expressed as $\sum_{x=N}^{\infty} f(x) \sim \int_N^{\infty} f(x)dx = O(1/N)$. Differentiating this gives $f(N) = O(1/N^2)$. This result is consistent with that shown in the previous STRAIGHT study [13]. This is also consistent with a study that shows that the count of register references follows a power-law distribution [31].

Although the frequency of defining registers with a long life is low, it is necessary to add the number of relay instructions proportional to the length of life. When the lifetime is k, $\lfloor \frac{k}{M} \rfloor$ relay instructions are needed, and the frequency at which they occur is $f(k)$; therefore $\sum_{k=1}^{P} O(\frac{1}{k^2}) f(k) \lfloor \frac{k}{M} \rfloor \sim O\left(\frac{1}{M} \log P\right)$ relay instructions are needed in the entire program, where P is a program size and M is the maximum reference distance. Accordingly, the impact of the increase in the number of instructions cannot be ignored, although the number of registers with long life is small.

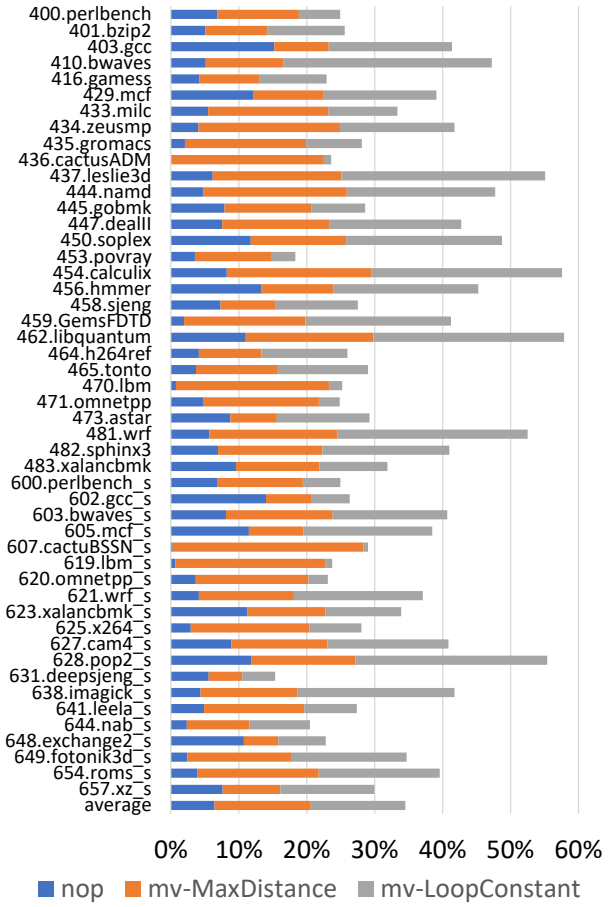


Figure 3: The number of inevitable instruction increase. The value is normalized by the number of total executed instructions. That is, 35% means that the number of instructions to be executed is increased by 1.35 times as a result of the `mv`/`nop` instructions insertion.

[1] Although there are other types of instruction increases, we do not count them because they are hard to obtain from traces. This is not a problem because the experiment is to find the *lower bounds*.

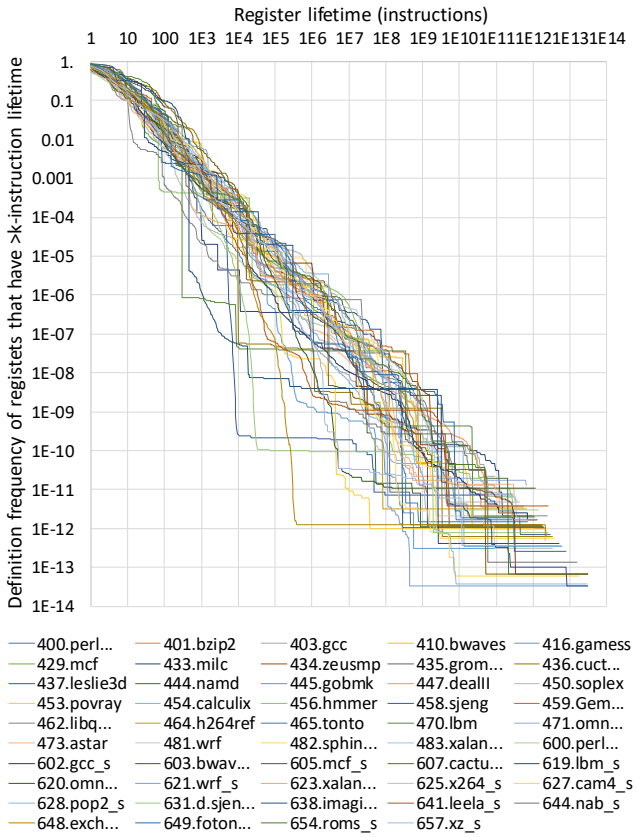


Figure 4: The frequency at which a destination register is defined with a lifetime greater than a certain number of instructions. This figure shows that the frequency at which a destination register with a lifetime of 1000 instructions or more is defined is approximately 0.001, which means that 99.9% of instruction results live in fewer than 1000 instructions.

3 CLOCKHANDS OVERVIEW

3.1 Motivation and Key Idea

The problem with conventional RISC is that it uses the operand representation, *logical register*, which causes false dependencies and thus is not suitable for out-of-order execution. Implementing the renaming mechanism in hardware, a brute-force approach, enables instructions to be executed out-of-order. These features do not contribute to essential computation and cause various inefficiency.

The problem with STRAIGHT is that the number of instructions increases because of the ISA constraints. This is because the operand specification method does not conform to the properties of general programs, which include numerous loop constants and long-life variables. Additional instructions to handle them also do not contribute to essential computation and cause various inefficiency.

We propose Clockhands, an ISA that can flexibly represent general programs and does not require register renaming even for out-of-order execution. The key idea is to provide some register groups (called *hands*) that can be referred to as "the value written k times before"[2]. This enables operands to be specified without being bound by inter-instruction distance as in RISC while retaining the rename-free characteristic in STRAIGHT. Because the microarchitecture other than the parts related to renaming is the same as RISC and STRAIGHT, Clockhands hardware can adopt the same ILP extraction mechanisms, including sophisticated speculation mechanisms.

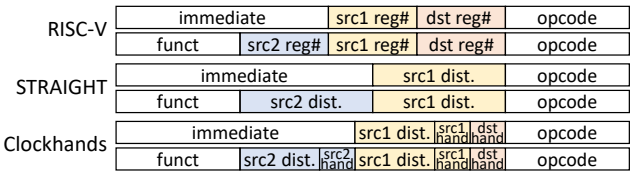


Figure 5: The instruction formats of Conventional RISC, STRAIGHT, and Clockhands. Only the fields that specify register operands are different. The "hand" field specifies a register group.

3.2 ISA Overview

Fig. 5 compares the instruction field of Clockhands with those of a conventional RISC (RISC-V) and STRAIGHT. The fields of the Clockhands instruction are the same as those of conventional RISC for `opcode` and `funct`, and they differ from RISC and STRAIGHT only in the operand specification field. The `dst-hand` field specifies which register group (hand) to write to. When an instruction with a `dst-hand` field (other than stores or non-`JAL[R]` branches) is executed, one physical register is allocated to the instruction from the register group specified in the `dst-hand` field, which becomes the destination of the instruction. The `src-hand` and `src-distance` fields are combined to specify the source operand. The value written before the `src-distance` times of the register group specified in `src-hand` becomes the source operand of the instruction.

We examined how a program is written with this instruction set. Fig. 1 shows assemblies of an `iota` function shown in (a) compiled for (b) RISC-V, (c) STRAIGHT, and (d) Clockhands. The instructions shaded in gray are the increased instructions compared to RISC. As can be observed, Clockhands has almost the same number of instructions as RISC.

Fig. 6 shows how Clockhands instructions are interpreted and how registers are rewritten during the time. The instructions executed at a given time and their source/destination registers are indicated with a background color. The registers are not updated for instructions that do not have a destination, such as the `sw` and the `bne` instructions. Regarding an instruction with a destination, such as the `addi`, the result of the instruction is written to the location indicated by the arrow on the hand specified as the destination (`t`, the hand name), and then only the specified hand is rotated. The hands that did not become a destination (for example, `v`) do not rotate; therefore, the distance remains the same, and loop constants can be referenced at the same distance in the next iteration of the loop, such as `v[0]` or `v[1]`.

[2]Derived from the hands of clocks that move at different speeds.

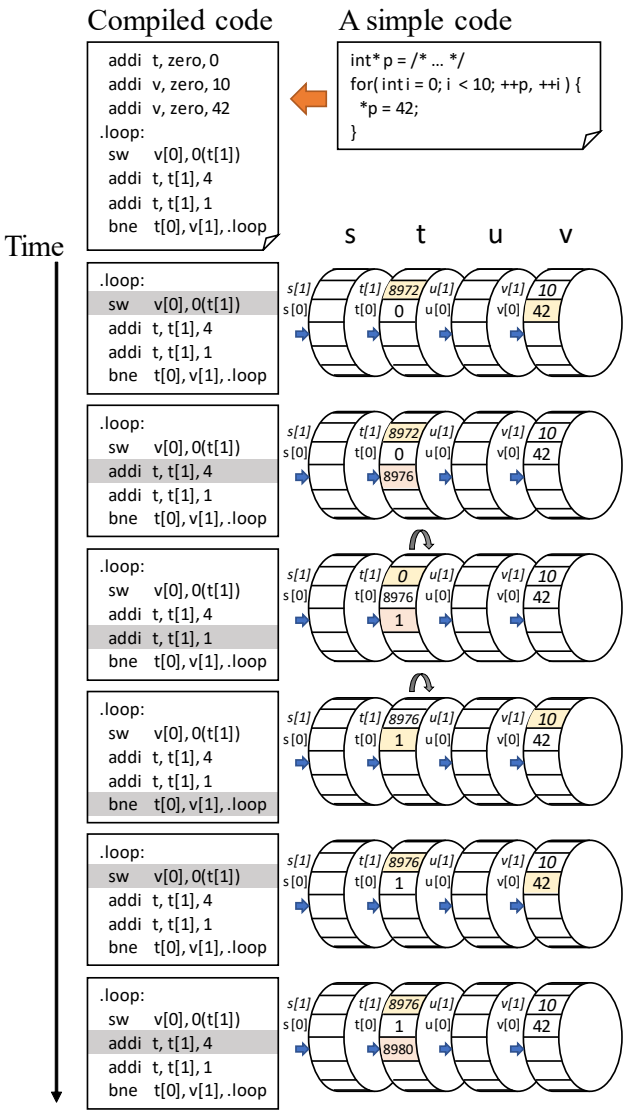


Figure 6: How Clockhands instructions are interpreted and how registers change over time. There are four hands (register groups): s, t, u, and v.

3.3 Advantages

As there are multiple hands instead of one, it reduces the increase in the number of instructions because of the three factors that occur in STRAIGHT:

(1) The copy instructions to hold loop constants are eliminated by generating code so that there is no instruction to write to the hand that records the loop constants in the loop. This is because executing an instruction in a loop does not change the distance in that hand. In STRAIGHT, executing an instruction in a loop changes the distance, but in Clockhands, this is not the case.

(2) We can reduce the copy instructions required to hold long-life values by separating a hand where temporary values are written from a hand where long-life values are written. This is because the hand that is written with long-life values advances the distance slowly. In STRAIGHT, the number of instructions until evicted from the registers is the same as the maximum reference distance. However, in Clockhands, using multiple hands makes it longer.

(3) Because the `jump` instruction is an instruction without a dst-hand and the distances of all hands do not change, there is no need to adjust near the convergence point by inserting a `nop` instruction.

4 CLOCKHANDS ISA

4.1 Appropriate Number of Hands

We must determine an appropriate number of hands because there are advantages and disadvantages to increasing the number of hands. The greater the number of hands, the more flexible code generation becomes. However, as discussed in this section, the hardware becomes more complex in proportion to the number of hands. Therefore, it is necessary to find the number of hands suitable for general programs.

To determine the appropriate number of hands, we examined the relationship between the number of hands and the number of move instructions to hold loop constants, which is the problem in STRAIGHT. We define loop constants as variables that are referenced beyond the beginning of a loop, i.e., defined outside the loop and referenced inside the loop. As described in Section 3.3, if loop constants and variables changed in a loop can be assigned to registers in different hands, no move instruction is necessary. When loops with loop constants are nested, other hands must be used. Therefore, if such loops are nested more than k times in a configuration with k hands, move instructions are needed to hold loop constants. We counted the number of such move instructions. In this experiment, the maximum reference distance of each hand is infinite. The results are shown in Fig. 7. These values were obtained from RISC-V traces of 10^{13} instruction runs of all SPEC CPU 2006/2017 benchmarks.

When there are four hands, the number of copy instructions can be reduced to 5.1% (a 94.9% reduction). When the number of hands is increased to eight, the number of instructions can only be reduced by an additional 1.3%. The gain from increasing the number of hands to eight would actually be smaller because increasing the number of hands shortens the maximum reference distance to keep the size of the instruction word constant. Therefore, considering the trade-off between increased hardware complexity and an increased number of instructions, we conclude that four is the appropriate number of hands.

From these observations, we hereinafter assume $H = 4$ where H is the number of hands. We also assume the maximum reference distance (D) is common for all hands and $D = 16$ due to the operand specification field size limits. In this case, the length of the operand specifying fields is 14 bits, which is less than that of conventional RISC ISA, 15 bits.

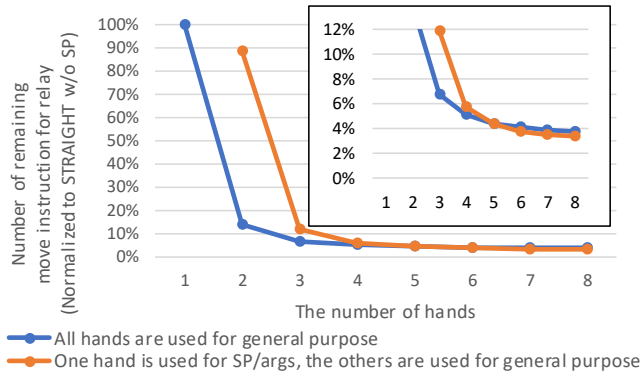


Figure 7: The remaining number of move instructions for relaying, normalized by the number in STRAIGHT. The subfigure is a zoomed-in view. When the number of hands is four and all hands are used for general purposes, 94.9% of the move instruction for relaying is eliminated. When the number of hands is increased to eight, only another 1.3% is eliminated. When one hand is fixed for SP/args holding, it increases only by 0.7%, which is acceptable.

4.2 Handling Stack Pointer

As with conventional RISC, we can place SP on a general-purpose register. In STRAIGHT, SP is a special register [13]. SP is a very long-life value, which does not go well with the distance representation in STRAIGHT. To avoid false dependencies caused by overwriting of SP, only additions and subtractions of immediate values are allowed for SP (`SPADDi` instruction). On the other hand, in Clockhands, SP is not treated as a special register but can be placed on a general-purpose register. This is possible because there are multiple hands.

It is useful to allocate one hand for SP. Fig. 7 also shows the number of copy instruction reductions when one hand is allocated for SP instead of general purpose. The increase in copy instructions by allocating one hand for SP is only 0.7% when $H = 4$. In contrast, the advantage of placing SP in a general-purpose register is considerable for the following two points.

- As discussed below, placing the SP in a general-purpose register reduces the amount of information required for recovery and also eliminates the problem of large payload RAM size.
- It enables the creation of an instruction set uniformly; that is, special instructions that use SP are unnecessary.

Therefore, we allocate one hand for SP.

4.3 Usage of Hands

Although all four hands are equal in the ISA, our compiler uses them for different purposes for simplicity. The four hands are referred to as `t`, `u`, `v`, and `s`.

- Temporary values are written in the `t` (temporal) hand.
- Values with a longer lifetime are written to the `u` hand.
- Loop constants are written to the `v` hand.
- A stack pointer (SP) and function arguments are written to the `s` (SP) hand.

Hardware optimization based on this usage is also possible. For example, it would be a good idea to have more physical registers in the `t` hand than in the others. The value written to the `v` hand may be less likely to be on the critical path. Such optimization should be a topic for future work.

4.4 Calling Conventions

We designed a calling convention that stores the function arguments and return value in the `s` hand:

- The result of the `JAL[R]` instruction is written to the `s` hand, meaning that `s[0]` is the return address at the beginning of the function.
- The first argument is written to the `s` hand before the `JAL[R]` instruction, meaning that `s[1]` is the first argument at the beginning of the function.
- The second argument is written to `s` hand before the first argument is written, meaning that `s[2]` is the second argument at the beginning of the function, and so on.
- SP is restored to `s` hand, meaning that `s[0]` is the caller's SP at the point of return from a function.
- The return value is written to `s` hand before an instruction to restore SP to `s` hand, meaning that `s[1]` is the return value at the point of return from a function.
- There are eight callee-saved registers, which are written to the `v` hand, meaning that when it exits from a function, the values in `v[0]`–`v[7]` are not changed.

We defined a convention for updating SP so that a reference to `s[0]` yields SP. We ensure that the SP is in the `s[0]` in the function. To achieve this, at the beginning of the function, we execute `addi s,s[X],-(amount)` where X is the number of arguments plus one. In addition, immediately before returning from the function, `addi s,s[1],(amount)` is executed to restore the caller's SP. Here, we need to use `s[1]` to get SP value instead of `s[0]` because there is an instruction to write the return value described above.

4.5 Architectural State

The architectural state of a Clockhands processor consists of the values recorded in logical registers, a program counter (PC), and a main memory. The writing of a register by a Clockhands instruction can be logically interpreted as 1) shifting the position of all the values recorded in the destination hand by one, 2) discarding the oldest value, and 3) writing a new value. With this context, Clockhands ISA defines 64 logical registers, `t[0]`–`t[15]`, `u[0]`–`u[15]`, `v[0]`–`v[15]`, `s[0]`–`s[14]`, and `zero`.

Four RPs (described in Section 5.1) in a Clockhands processor are not included in the architectural state, similar to the RMT in RISC one. As described above, Clockhands ISA can be interpreted as having logical registers that are shifted. The RPs are just hardware-optimized implementations used to avoid shifting the actual data positions.

On a context switch, operating systems (OSs) generally save and restore only an architectural state, which is also true for Clockhands. The OS can be implemented without awareness of how the logical registers are internally implemented, even if they are implemented using the RPs or the RMT.

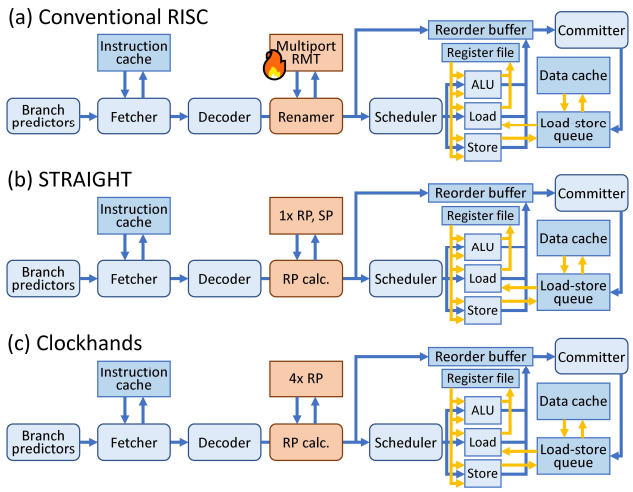


Figure 8: Out-of-order processor pipeline of conventional RISC, STRAIGHT, and Clockhands. The only difference is the physical-register-allocation stage (rename stage or RP-calculation stage).

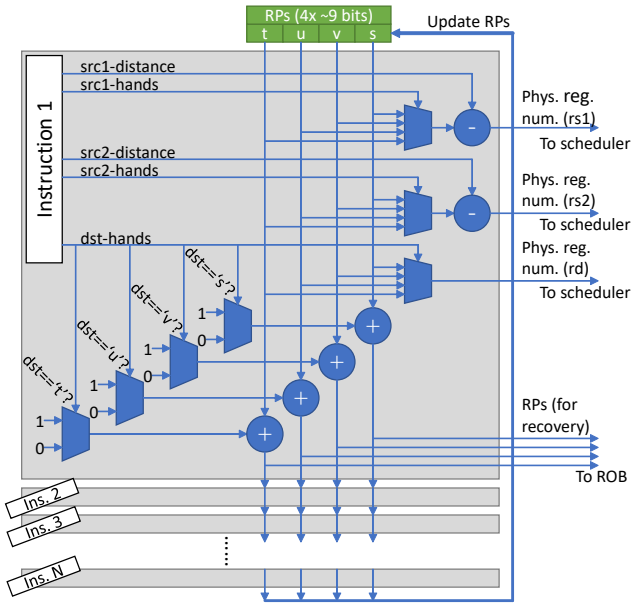


Figure 9: RP-calculation stage of the Clockhands processor. In this stage, the physical register numbers of a destination operand and source operands are determined. The implementation in this figure calculates RPs sequentially for simplicity but can be optimized for latency as described in Section 5.1.

5 MICROARCHITECTURE

A Clockhands processor has a hardware architecture similar to that of RISC and STRAIGHT. Fig. 8 shows block diagrams of (a) conventional RISC, (b) STRAIGHT, and (c) Clockhands out-of-order processors. They are identical except for the part where the register operands specified in the instruction word are converted to physical registers.

5.1 Register Pointer (RP) Calculation Stage

The primary difference in microarchitecture is the part that converts register operands to physical registers (RP-calculation stage). Fig. 9 shows the RP-calculation stage of a Clockhands processor. A Clockhands processor has four pointers, RPs, to record the range of physical registers assigned to the four hands. The physical register number of the destination of an instruction is the one pointed to by one of the four RPs, selected by the dst-hand field in the instruction word. The physical register number of the source operand is one of the four RPs selected by the src-hand field of the instruction word, subtracting the src-distance. After this calculation, the RP is incremented by one only for the hand specified by the dst-hand.

The RP-calculation stage allocates physical registers from the four ring buffers but must be stalled accordingly to avoid false dependencies. In a Clockhands processor, the physical registers have linear addresses, as in other architectures. One difference is that they are statically partitioned into four and used as four ring buffers. Each RP wraps around within each partitioned range. When a wraparound occurs, the physical registers are reused, but it is necessary to ensure that no false dependencies occur when this happens. To satisfy this constraint, a Clockhands processor must stall the RP-calculation stage when a register located within the maximum reference distance from the value of the RP of the oldest in-flight instruction is about to be allocated.

For simplicity, the diagram shown in Fig.9 calculates the RP sequentially, but this process can be optimized for latency without significantly increasing the circuit area as follows. The optimization can be done by counting P, the number of preceding instructions that write to each hand, for every instruction processed simultaneously in a group. In this calculation, the result can be shared among instructions to reduce the amount of circuit and delay, as in a tree that calculates the prefix sum in a parallel prefix adder [47]. For example, if a structure similar to the Brent-Kung tree [4, 47] is used, the delay is $O(\log W)$ and the amount of circuit is $O(W)$, assuming the parallel processing width is W. Then, the operands of each instruction can be obtained with short latency by adding P and the RP at the beginning of the group. The amount of circuit and latency is sufficiently small compared with those of the DCL used in the existing renaming logic described in Section 2.1.

5.2 Recovery Mechanism

Processors with Clockhands recover from mispredictions and exceptions by restoring the RP using the information stored in the ROB. An instruction executed in a mispredicted path may write an incorrect value to a physical register, but this does not affect the correctness of the execution as in existing processors. This recovery mechanism by restoring pointers is the same as one typically performed in ring buffers such as ROBs and LSQs, where entries are sequentially allocated.

Clockhands processor requires less information for recovery than RISC one. Table 1 summarizes the amount of information required for recovery of the physical register allocation stage for

Table 1: Recovery information size (checkpoint size) for each architecture.

Architecture	Recovery information size	
Conventional RISC	63× ~9 bits	~570 bits
STRAIGHT	~9 bits + 64 bits	~70 bits
Clockhands	4× ~9 bits	~36 bits

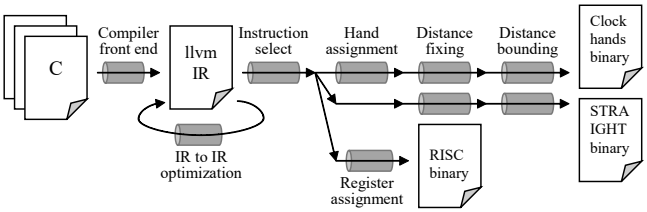


Figure 10: Compilation flow comparison.

(a) RISC-V, (b) STRAIGHT, and (c) Clockhands. The value in the rightmost column assumes that the physical register number can be specified in 9 bits. Conventional RISC requires ~570 bits of information per checkpoint to recover the correspondence between logical and physical registers. Although STRAIGHT requires only ~9 bits of information per checkpoint to recover the correspondence between logical and physical registers, 64 bits are required to recover the value of SP. Clockhands requires ~36 bits of information per checkpoint to recover the correspondence between logical and physical registers. As SP is stored in general-purpose registers, restoring RP automatically recovered its logical state, just as RISC does by restoring RMT. On the other hand, STRAIGHT processor holds SP in a special register, thus it needs to restore SP itself.

5.3 Simultaneous Multi-threading

While conventional simultaneous multi-threading (SMT) processors dynamically share physical registers among threads, Clockhands/STRAIGHT processors must partition physical registers statically because Clockhands/STRAIGHT need to allocate physical registers sequentially. This static partitioning may slightly reduce the efficiency of physical register usage, but it does not cause significant overhead in typical SMT processors for the following reason. In SMT processors, queues such as ROB and LSQ, whose entries are also allocated sequentially, are typically statically partitioned [21], which limits the number of in-flight instructions held by each thread. As a result, the number of physical registers allocated to in-flight instructions is also limited.

6 CLOCKHANDS COMPILER

6.1 Compilation Flow for Clockhands

A Clockhands compiler differs from a conventional RISC compiler only in the register allocation phase. Fig. 10 compares the compile flow of conventional RISC, STRAIGHT, and Clockhands. In a Clockhands compiler, we can implement the interpretation of high-level programming languages and instruction selection in the same way as in RISC. However, the subsequent processes differ from those of a RISC compiler. In the hand assignment phase, which is specific to

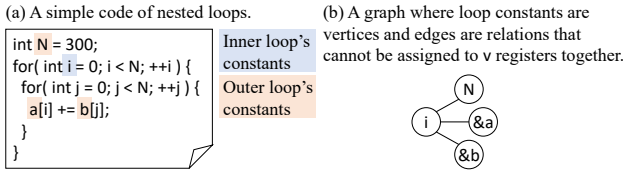


Figure 11: Independent set problem in v-hand assignment procedure.

compilers for Clockhands, a hand is first assigned to the destination operand of each instruction. Thereafter, the reference distances are determined in the same way as for STRAIGHT, and this procedure is repeated for each hand. The process of hand assignment, which is specific to Clockhands, is described below.

6.2 Hand Assignment Algorithm

Our Clockhands compiler assigns hands to each instruction as follows: SP, function arguments, and the return value of a function are assigned to the `s` hands. Other loop constants are assigned to the `v` hand if possible. The details of this process are described below. The callee-saved registers are also assigned to the `v` hand. Otherwise, when an instruction result has a lifetime less than the maximum reference distance, it is assigned to the `t` hand. The remainder is assigned to the `u` hand.

The `v` hand assignment is performed by determining a maximal independent set. When there is a nested loop, as depicted in Fig. 11(a), and both have loop constants, we cannot assign the `v` hand to all the loop constants. For example, we cannot assign `v` hand to `N` and `i` together. This is because, in such a case, updating `i` would cause a write to `v` hand, which would change the distance to `N`. We can find out which loop constants can be assigned to `v` hand by considering a graph with loop constants as vertices (Fig. 11(b)). Each edge connects two vertices that correspond to two variables that cannot be assigned to `v` hand together. This is a relationship of two variables such that the location where one variable is defined is within the range where the other variable is a constant.

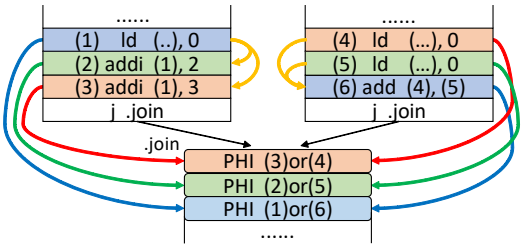
By solving the independent set problem for this graph, we can determine which loop constants can be assigned to the `v` hand. For example, {`i`} (only `i` is assigned to the `v` hand) is a solution of the problem and {`N`,`&a`,`&b`} (`N`, `&a`, and `&b` are assigned to the `v` hand) is another solution. These solutions indicate a set of loop constants that can be assigned to the `v` hand together. It is helpful to find a maximal independent set using a greedy heuristic, as shown in Algorithm 1, because assigning the inner loop constant to the `v` hand is preferred for better performance.

6.3 Advanced Register Assignment

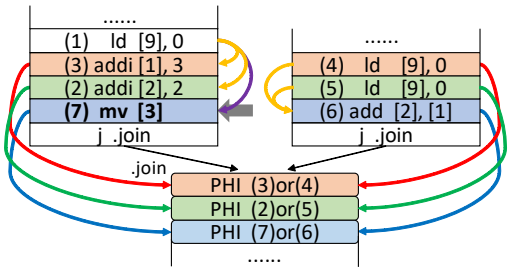
In Clockhands, in addition to those mentioned in Section 2, there are other cases where the instruction increase that occurs in STRAIGHT can be eliminated, as shown in Fig. 12. The instruction sequence depicted in Fig. 12(a) has execution order constraints that are not coherent across multiple paths. In such a case, a compiler reorders instructions to satisfy the execution order constraint in STRAIGHT. If the constraint cannot be satisfied by reordering, transfer instructions that copy live variables are inserted to keep the reference

Algorithm 1 v hand assignment algorithm

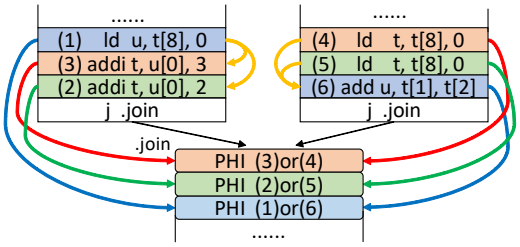
$U \leftarrow$ all loop constants
for $x \in U$ **do**
  if $\exists y \in U$ s.t. the initial value of y is defined in the loop associated with x **then**
    $U \leftarrow U \setminus \{x\}$
  end if
end for
Assign v hand to each $x \in U$.



(a) An SSA-form code.



(b) STRAIGHT code, which contains one mv instruction.



(c) Clockhands code, which contains no mv instruction.

Figure 12: Advanced register assignment.

distance constant [17]. For example, in Fig. 12(b), the mv instruction is inserted for transfer. Conversely, in Clockhands, we can generate code without the addition of mv instructions using different destination hands, as depicted in Fig. 12(c). However, how to do this with a compiler is a topic for future research.

7 EVALUATION

7.1 Methodology

We evaluated the performance, energy consumption, and resource consumption of Clockhands, STRAIGHT, and existing RISC. We also developed a Clockhands soft-core processor written in SystemVerilog and used it for hardware evaluation.

We used a cycle-accurate simulator, Onikiri2 [45], for the performance evaluation and McPAT [20] for the energy consumption evaluation. Onikiri2 is an execution-driven simulator similar to gem5 [3], but it can simulate more detailed pipeline behavior, including various speculations and replays. We implemented a Clockhands 32-bit 166-instruction RV64G-compatible ISA on Onikiri2. We also extended Onikiri2 to simulate the Clockhands pipeline behavior accurately. The parameters of the processors used in the evaluation are listed in Table 2. The parameters of the six-fetch model are derived from the parameters of Apple M1 processor [14]. In the larger models, we aggressively enlarged the ROB because it does not have complex functions such as associative search in the current mainstream architecture, while conservatively enlarged the scheduler and the load-store queue because of their complex structure and the controversial nature of their expandability.

The benchmark programs used for our evaluation were bzip2, mcf_s, lbm_s, and xz_s included in SPEC2006/2017 [40, 41] and CoreMark [8]. We use these benchmarks, which are written entirely in C, because we are currently only able to develop a C compiler, as C++/Fortran compilers are very complex and require a great deal of effort to develop. We used representative regions for each program used in a previous STRAIGHT study [17]. We modified them so that they contain >50M instructions for SPEC benchmarks.

Table 2: The parameters of the processors used in the simulation.

	4-fetch	6-fetch	8-fetch	12-fetch	16-fetch
Front-end width	4	6	8	12	16
Front-end latency	fetch(3) + decode(1) + [rename(2) +] dispatch(1) RISC-V: 7 cycles STRAIGHT, Clockhands: 5 cycles				
Issue width	8	16			
Issue latency	4 cycles (payload RAM read + register read)				
Execution units	$\lceil \frac{1}{2} \times \rightarrow \rceil$	Int×8, Float×4, Load×3, Store×2, iMul×2, iDiv×1, fDiv×1			
Reorder buffer (R)	256	640	1024	2048	4096
Register width	64 bits				
Logical registers	RISC-V: Int×31, FP×32, STRAIGHT: Unif.×127 Clockhands: s×15, t×16, u×16,v×16				
Physical registers	RISC-V: Unified×R STRAIGHT, Clockhands: Unified×$(128 + R)$ $\lceil \frac{1}{2} \times \rightarrow \rceil$ | 27-read, 14-write				
Physical register quota for each hand	s×$(32 + 2R/64)$, t×$(32 + 48R/64)$, u×$(32 + 9R/64)$, v×$(32 + 5R/64)$				
Scheduler (S)	128	192	256	384	512
Load-store queue	Load capacity: $S/2$, Store capacity: $3S/8$				
Branch predictor	8-component TAGE [33], 130-bit history, 8 KiB				
Branch target buffer	4-way, 8192 entries				
Return address stack	16 entries				
Mem. dep. predictor	Store set [7], 512 producers, 4096 store IDs				
Load lat. predictor	Optimistic (always assumes L1D cache hit)				
L1I cache	128 KiB, 8-way, 64B line, 3 cycles				
L1D cache	128 KiB, 8-way, 64B line, 3 cycles				
L2 cache	8 MiB, 16-way, 64B line, 12 cycles Stream prefetcher [39], distance 8, degree 2				
Main memory	80 cycles				

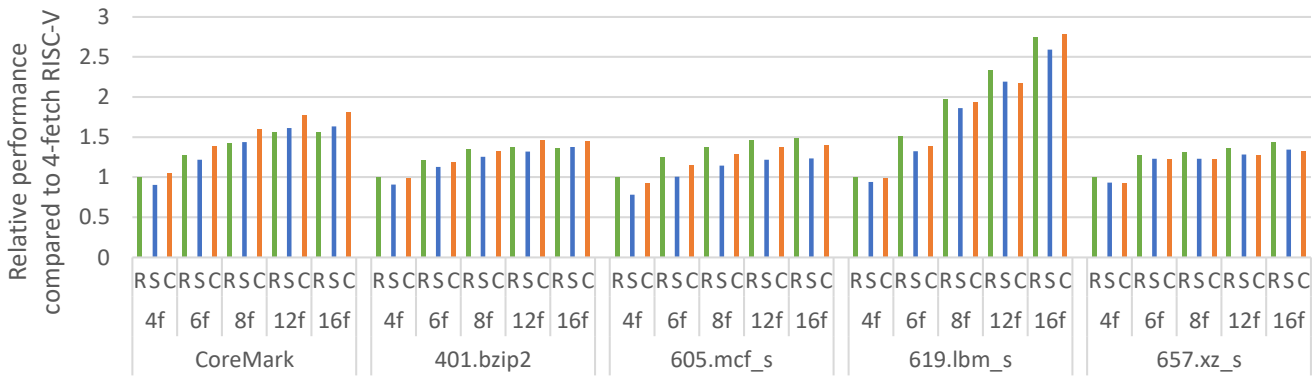


Figure 13: Performance comparison. The values are normalized to those of RISC-V's 4-fetch model. R, S, and C indicate RISC-V, STRAIGHT, and Clockhands, respectively. 4f, 6f, 8f, 12f, and 16f indicate 4-fetch, 6-fetch, 8-fetch, 12-fetch and 16-fetch, respectively.

The benchmark programs were compiled using LLVM [19]. Our compiler was built on top of LLVM version 12.0.1 and implemented the algorithms described in Section 6. The compiler for RISC-V is one with the same version of LLVM, and the compiler for STRAIGHT was obtained from the authors of the existing study [13].

7.2 Results

1) Performance: Fig. 13 shows the performance of each model. This figure shows the inverse of the cycles elapsed to run the benchmark, normalized by the value in RISC-V. This result indicates that the performance of Clockhands is almost the same as that of RISC-V while providing the advantage of no need for renaming. In the 6-fetch and above models, the performance improvement continues up to 16-fetch, even though we used a configuration of the same back-end complexity. The performance of Clockhands is 97.9%, 97.3%, 98.9%, 100.0%, and 101.6% of that of RISC-V, in 4-fetch, 6-fetch, 8-fetch, 12-fetch, and 16-fetch model, respectively. The performance of Clockhands is 9.9%, 7.6%, 6.6%, 6.5%, and 7.2% higher than that of STRAIGHT, in 4-fetch, 6-fetch, 8-fetch, 12-fetch, and 16-fetch model, respectively.

Clockhands shows equal to or better performance than STRAIGHT in all the benchmarks. In CoreMark, Clockhands shows higher performance than RISC-V due to faster recovery from branch mispredictions, similar to STRAIGHT. In bzip2, Clockhands shows performance equal to or better than RISC-V due to faster recovery from branch mispredictions. Although STRAIGHT has the same property, the performance degradation due to increased instruction count is larger. In mcf_s, Clockhands shows lower performance than RISC-V because it still has more instructions than RISC-V, although the number of instructions is greatly reduced than STRAIGHT, as described below. In lbm_s, as described below, unlike STRAIGHT, Clockhands succeeded in handling long-life values and was able to reduce the number of `mv` and `load` instructions, so its performance is about the same as RISC-V. In xz_s, STRAIGHT and Clockhands show performance degradation due to instruction execution order that is different from RISC-V as a result of distance adjustment. This is because xz_s is a program that uses up the integer arithmetic unit, and the instruction order greatly affects the latency.

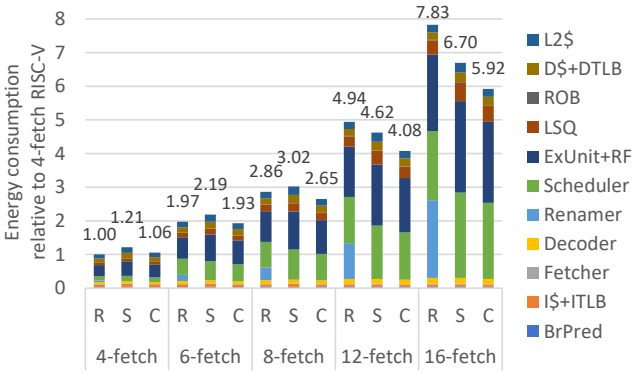


Figure 14: Energy comparison. The values are normalized to those of RISC-V's 4-fetch model. R, S, and C indicate RISC-V, STRAIGHT, and Clockhands, respectively.

2) Energy Consumption: Fig. 14 shows the energy comparison. The Clockhands processor saved 7.4% in the 8-fetch model, 17.5% in the 12-fetch model, and 24.4% in the 16-fetch model, compared to the RISC-V one owing to the elimination of the renaming process. The adoption of distance expressions has eliminated the need for renaming, and the number of instructions has hardly increased, resulting in a significant reduction in power consumption.

3) Instruction Breakdown: Fig. 15 shows a breakdown of the types of instructions executed. The number of instructions executed in Clockhands was reduced by greatly reducing the number of `mv` and `nop` instructions. In addition, the number of `load` and `store` instructions, which tended to increase in STRAIGHT, was reduced. As a result, the number of instructions executed in Clockhands was successfully reduced to the same level as RISC-V. Our compiler is still underdeveloped, and we expect to further reduce the number of instructions by further improvement.

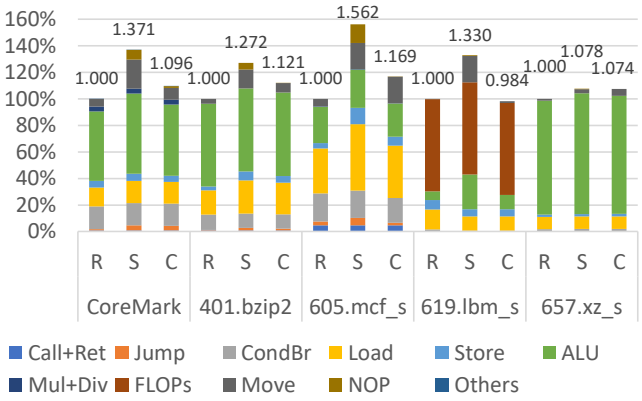


Figure 15: Executed instruction breakdown. The values are normalized to those of RISC-V. R, S, and C indicate RISC-V, STRAIGHT, and Clockhands, respectively.

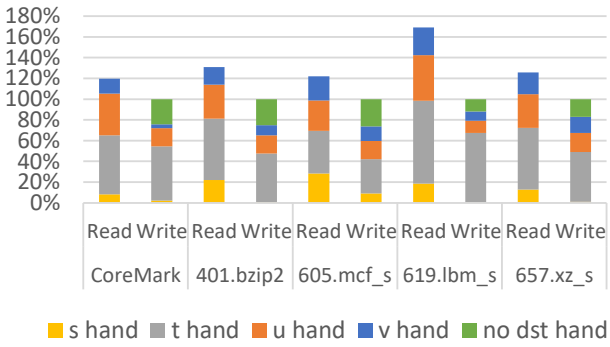


Figure 16: Breakdown of how many times each hand was read and written. The values are normalized by the number of executed instructions.

4) Hand Usage: Fig. 16 shows the distribution of which hand was written to. As mentioned in Section 4.3, the t hand, where temporary values are written, is the most commonly used. The v hand, which holds loop constants, is written less often but read more often, which is consistent with what would be expected from the nature of the loop constants. Also, the s hand is written extremely few times but read many times; this is because it holds values that are referenced many times, such as SP and arguments. In mcf_s, where there are many function calls, the s hand is often used to put in arguments, as described in Section 4.4.

5) Register Lifetime: Fig. 17 shows the register lifetime. In STRAIGHT, the distribution ends at 127, the maximum reference distance. RISC-V and Clockhands have similar distributions, which indicates that Clockhands successfully handles long-life values. Comparing RISC-V and Clockhands, Clockhands has longer vertical and horizontal lines, especially in lbm_s. This is because multiple variables co-located in one hand will have similar lifetimes.

To further clarify why Clockhands ISA was able to address long-life values, we will review the lifetime for each hand. Fig. 18 shows the register lifetime for each hand. The lifetime of registers in the

Table 3: Resource usage of soft processors.

		Phys. reg. alloc. stage		Overall	
	Architecture	Look-up tables	Flip-flops	LUTs	FFs
4-way	RISC-V	2310	998	101483	31081
	STRAIGHT	442	572	96631	28769
	Clockhands	401	560	99913	30968
8-way	RISC-V	12309	7521	190380	45708
	STRAIGHT	787	1092	188118	43928
	Clockhands	761	1086	185701	42254
16-way	RISC-V	30230	14938	350377	63338
	STRAIGHT	1641	2132	354105	57214
	Clockhands	1432	2162	349074	55220

t hand was as short as about 100 because temporary values are written in it as described in Section 4.3. The lifetime of registers in the u hand, where values with longer lifetime are written, was longer than that of the t hand. The lifetime of registers in the v hand, where loop constants are written, was more longer. The lifetime of registers in the s hand, where SP and function arguments are written, had different properties than the others. It is very short in mcf_s and very long in the others, which is very different. This is due to the frequent function calls in mcf_s. In general, SP and function arguments have a long lifetime, but this is not the case with frequent function calls. The reason Clockhands ISA can deal with long-life values is that we have used hand in this way.

6) Hardware Complexity: Clockhands architecture does not complicate hardware. The resource usage of (a) RISC-V, (b) STRAIGHT, and (c) Clockhands processors for FPGA is summarized in Table 3. For our evaluation, we used RV32IM-compatible FPGA-optimized out-of-order soft processor RSD [22] as a baseline, but with modifications for each architecture. We evaluated three front-end widths: 4, 8, and 16. We confirmed that CoreMark [8] program runs correctly and the soft processor runs on Xilinx Virtex UltraScale FPGA XCVU440. This table shows that a Clockhands processor can be built with equal or fewer resources than a RISC-V processor. Thanks to the distance representation, a lightweight physical register allocation is realized. This property is universal regardless of fetch width.

8 RELATED WORK

Some ISAs feature register windows [9, 12]. The register window switches a set of registers at the time of function calls and returns, eliminating the need to execute load/store instructions to spill values. However, when the register window is exhausted, it is necessary to save register values to memory through complex hardware mechanisms or interruptions. The register operand format proposed in this paper has similarities with register windows in that it reduces the number of instructions by storing values with significantly different lifetimes in a group of other registers. One significant difference is that the management of registers is performed purely in software and does not require hardware support.

IA-64 also has a register rotation mechanism [12]. Register rotation is a cyclical replacement of some register names, eliminating false dependencies even when logical registers with the same name are used and simplifying the description of software pipelining. The register operand format proposed in this paper has similarities

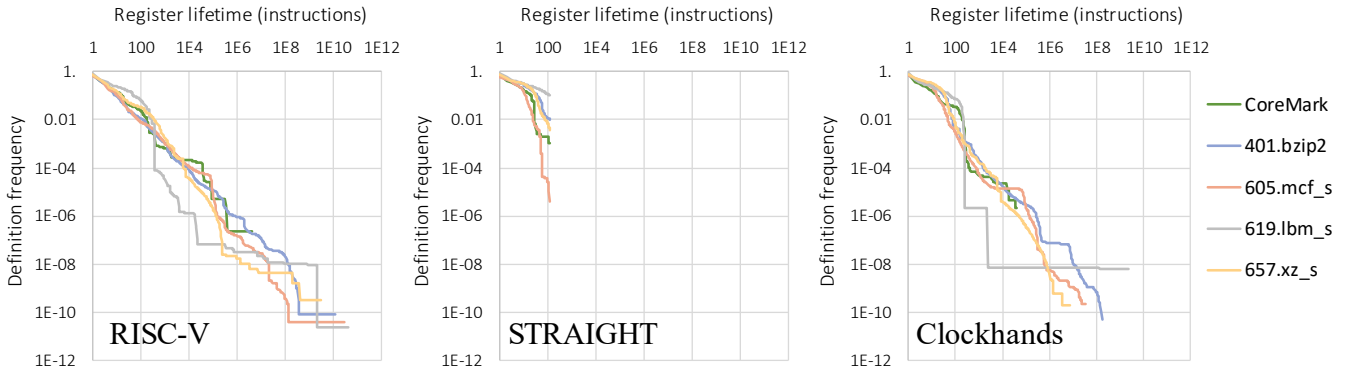


Figure 17: Frequency at which a destination register is defined with a lifetime greater than a certain number of instructions (same as Fig. 4).

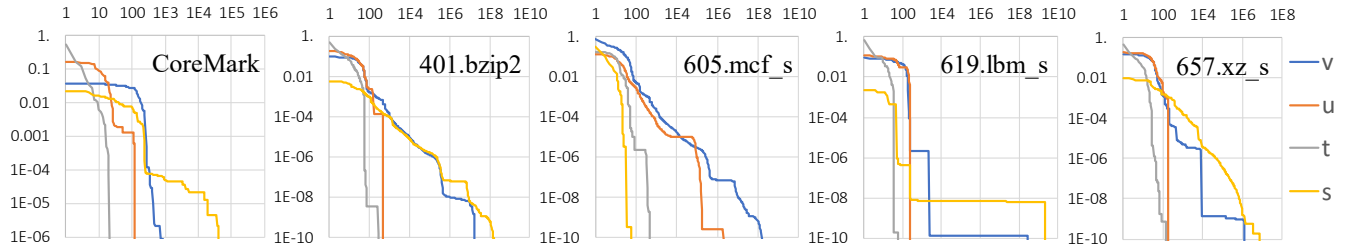


Figure 18: Frequency at which a destination register is defined with a lifetime greater than a certain number of instructions (same as Fig. 4). The vertical axes indicate definition frequency and the horizontal axes indicate register lifetime.

with register rotation in that only the names of some registers are cyclically replaced so that false dependencies do not occur. A crucial difference is that the proposed method guarantees that there are no false dependencies at the ISA level, even when out-of-order execution is performed.

In dataflow architecture, a sticky bit has been proposed [42]. In dataflow, produced values vanish when they are consumed. However, such a scheme is unsuitable for frequently referenced values, such as loop constants. Therefore, a sticky bit, that is, a marker, is used to prevent the value from vanishing even if it is consumed. The register operand format proposed in this paper is similar to sticky bits in that it provides special treatment to values referenced many times, such as loop constants. However, the proposed method provides a general-purpose register use not limited to holding loop constants, such as holding SP or a value with a slightly longer lifetime.

EDGE architecture [6] improves power efficiency by introducing direct communication between closely located instructions, exploiting the fact that many computational results are short-lived and references are often resolved locally. STRAIGHT improves power efficiency by introducing a static guarantee of register lifetime, exploiting the same property. Clockhands also improves power efficiency by introducing lifetime classification, exploiting a more generalized property that many instructions refer to recently generated results *within each lifetime class*. Clockhands can efficiently handle both short-lived and long-lived values in a single framework.

9 CONCLUSION

In this paper, we proposed Clockhands, an ISA with a register operand specification scheme that has multiple register groups and specifies the value as "the value written to this register group k times before." This scheme enables operands to be specified without being bound by inter-instruction distance, as in RISC, while retaining the rename-free characteristic of STRAIGHT. A Clockhands processor is similar to a RISC processor except for the physical register allocation mechanism, and it can exploit existing sophisticated speculation mechanisms. In this study, in addition to the Clockhands instruction set architecture, its RTL design and compiler are provided, and its performance and energy consumption are evaluated using simulation. The evaluation results show that Clockhands processors do not increase complexity over RISC processors, and the performance of Clockhands processors was equivalent to that of RISC processors.

ACKNOWLEDGMENTS

This work was supported by JSPS KAKENHI Grant Numbers JP19H04077, JP20J22752, JP20H04153, JP23H03360, JP23KJ0500, and JP23K19975. This work was also supported by Premo Inc. This work was also supported through the activities of VDEC, The University of Tokyo, in collaboration with Nihon Synopsys G.K. and Siemens Electronic Design Automation Japan K.K.

REFERENCES

[1] Mehdi Alipour, Stefanos Kaxiras, David Black-Schaffer, and Rakesh Kumar. 2020. Delay and Bypass: Ready and Criticality Aware Instruction Scheduling in Out-of-Order Processors. In *2020 IEEE International Symposium on High Performance Computer Architecture* (San Diego, California) *(HPCA 2020)*. IEEE Computer Society, Los Alamitos, CA, USA, 424–434. https://doi.org/10.1109/HPCA47549.2020.00042

[2] Mehdi Alipour, Rakesh Kumar, Stefanos Kaxiras, and David Black-Schaffer. 2019. FIFOrder MicroArchitecture: Ready-Aware Instruction Scheduling for OoO Processors. In *2019 Design, Automation & Test in Europe Conference & Exhibition* (Florence, Italy) *(DATE)*. EDAA, Leuven, Belgium, 716–721. https://doi.org/10.23919/DATE.2019.8715034

[3] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[4] Richard P. Brent and H. T. Kung. 1982. A Regular Layout for Parallel Adders. *IEEE Trans. Comput.* 31, 3 (March 1982), 260–264. https://doi.org/10.1109/TC.1982.1675982

[5] Thomas Burd, Wilson Li, James Pistole, Srividhya Venkataraman, Michael McCabe, Timothy Johnson, James Vinh, Thomas Yiu, Mark Wasio, Hon-Hin Wong, Daryl Lieu, Jonathan White, Benjamin Munger, Joshua Lindner, Javin Olson, Steven Bakke, Jeshuah Sniderman, Carson Henrion, Russell Schreiber, Eric Busta, Brett Johnson, Tim Jackson, Aron Miller, Ryan Miller, Matthew Pickett, Aaron Horiuchi, Josef Dvorak, Sabeesh Balagangadharan, Sajeesh Ammikkallingal, and Pankaj Kumar. 2022. Zen3: The AMD 2nd-Generation 7nm x86-64 Microprocessor Core. In *2022 IEEE International Solid-State Circuits Conference* (Virtual Conference) *(ISSCC, Vol. 65)*. IEEE, New York, NY, USA, 1–3. https://doi.org/10.1109/ISSCC42614.2022.9731678

[6] Doug Burger, Stephen W. Keckler, Kathryn S. McKinley, Mike Dahlin, Lizy K. John, Calvin Lin, Charles R. Moore, James Burrill, Robert G. McDonald, and William Yoder. 2004. Scaling to the end of silicon with EDGE architectures. *Computer* 37, 7 (July 2004), 44–55. https://doi.org/10.1109/MC.2004.65

[7] George Z. Chrysos and Joel S. Emer. 1998. Memory dependence prediction using store sets. In *The 25th Annual International Symposium on Computer Architecture* (Barcelona, Spain) *(ISCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 142–153. https://doi.org/10.1109/ISCA.1998.694770

[8] EEMBC. 2009. CoreMark. https://www.eembc.org/coremark/

[9] Robert B. Garner, Anant Agrawal, Fayé Briggs, Emil W. Brown, David Hough, Bill Joy, Steve Kleiman, Steven Muchnick, Masood Namjoo, Dave Patterson, Joan Pendleton, and Richard Tuck. 1988. The scalable processor architecture (SPARC). In *Digest of Papers. COMPCON Spring 88 Thirty-Third IEEE Computer Society International Conference* (San Francisco, California). IEEE Computer Society, Los Alamitos, CA, USA, 278–283. https://doi.org/10.1109/CMPCON.1988.4874

[10] Peter Greenhalgh. 2011. Big.LITTLE Processing with ARM Cortex™-A15 & Cortex-A7. *ARM white paper* (Sept. 2011), 1–8.

[11] Linley Gwennap. 2019. Cortex-A77 Improves IPC. *Microprocessor Rep.* (2019), 1–4.

[12] Jerry Huck, Dale Morris, Jonathan Ross, Allan Knies, Hans Mulder, and Rumi Zahir. 2000. Introducing the IA-64 architecture. *IEEE Micro* 20, 5 (Sept.–Oct. 2000), 12–23. https://doi.org/10.1109/40.877947

[13] Hidetsugu Irie, Toru Koizumi, Akifumi Fukuda, Seiya Akaki, Satoshi Nakae, Yutaro Bessho, Ryota Shioya, Takahiro Notsu, Katsuhiro Yoda, Teruo Ishihara, and Shuichi Sakai. 2018. STRAIGHT: Hazardless Processor Architecture without Register Renaming. In *The 51st Annual IEEE/ACM International Symposium on Microarchitecture* (Fukuoka, Japan) *(MICRO-51)*. IEEE Computer Society, Los Alamitos, CA, USA, 121–133. https://doi.org/10.1109/MICRO.2018.00019

[14] Aakash Jani. 2021. Apple Ships Its First PC Processor. *Microprocessor Rep.* (2021), 1–5.

[15] Ipoom Jeong, Jiwon Lee, Myung Kuk Yoon, and Won Woo Ro. 2022. Reconstructing Out-of-Order Issue Queue. In *2022 55th Annual IEEE/ACM International Symposium on Microarchitecture* (Chicago, Illinois) *(MICRO 2022)*. IEEE Computer Society, Los Alamitos, CA, USA, 144–161. https://doi.org/10.1109/MICRO56248.2022.00023

[16] Ipoom Jeong, Seihoon Park, Changmin Lee, and Won Woo Ro. 2020. CASINO Core Microarchitecture: Generating Out-of-Order Schedules Using Cascaded In-Order Scheduling Windows. In *2020 IEEE International Symposium on High Performance Computer Architecture* (San Diego, California) *(HPCA 2020)*. IEEE Computer Society, Los Alamitos, CA, USA, 383–396. https://doi.org/10.1109/HPCA47549.2020.00039

[17] Toru Koizumi, Shu Sugita, Ryota Shioya, Junichiro Kadomoto, Hidetsugu Irie, and Shuichi Sakai. 2021. Compiling and Optimizing Real-world Programs for STRAIGHT ISA. In *2021 IEEE 39th International Conference on Computer Design* (Virtual Conference) *(ICCD 2021)*. IEEE Computer Society, Los Alamitos, CA, USA, 400–408. https://doi.org/10.1109/ICCD53106.2021.00070

[18] Rakesh Kumar, Mehdi Alipour, and David Black-Schaffer. 2019. Freeway: Maximizing MLP for Slice-Out-of-Order Execution. In *25th IEEE International Symposium on High Performance Computer Architecture* (Washington, D.C.) *(HPCA 2019)*. IEEE Computer Society, Los Alamitos, CA, USA, 558–569. https://doi.org/10.1109/HPCA.2019.00009

[19] Chris Lattner and Vikram Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization* (San Jose, California) *(CGO 2004)*. IEEE Computer Society, Los Alamitos, CA, USA, 75–86. https://doi.org/10.1109/CGO.2004.1281665

[20] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (New York, New York) *(MICRO 42)*. ACM, New York, NY, USA, 469–480. https://doi.org/10.1145/1669112.1669172

[21] Artemiy Margaritov, Siddharth Gupta, Rekai Gonzalez-Alberquilla, and Boris Grot. 2019. Stretch: Balancing QoS and Throughput for Colocated Server Workloads on SMT Cores. In *25th IEEE International Symposium on High Performance Computer Architecture* (Washington, D.C.) *(HPCA 2019)*. IEEE Computer Society, Los Alamitos, CA, USA, 15–27. https://doi.org/10.1109/HPCA.2019.00024

[22] Susumu Mashimo, Akifumi Fujita, Reoma Matsuo, Seiya Akaki, Akifumi Fukuda, Toru Koizumi, Junichiro Kadomoto, Hidetsugu Irie, Masahiro Goshima, Koji Inoue, and Ryota Shioya. 2019. An Open Source FPGA-Optimized Out-of-Order RISC-V Soft Processor. In *2019 International Conference on Field-Programmable Technology* (Tianjin, China) *(ICFPT 2019)*. IEEE Computer Society, Los Alamitos, CA, USA, 63–71. https://doi.org/10.1109/ICFPT47387.2019.00016

[23] Andreas Moshovos. 2003. Checkpointing Alternatives for High Performance, Power-Aware Processors. In *Proceedings of the 2003 International Symposium on Low Power Electronics and Design* (Seoul, South Korea) *(ISLPED '03)*. ACM, New York, NY, USA, 318–321. https://doi.org/10.1145/871506.871585

[24] Tony Nowatzki, Vinay Gangadhar, and Karthikeyan Sankaralingam. 2015. Exploring the Potential of Heterogeneous von Neumann/Dataflow Execution Models. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. ACM, New York, NY, USA, 298–310. https://doi.org/10.1145/2749469.2750380

[25] Tony Nowatzki, Vinay Gangadhar, and Karthikeyan Sankaralingam. 2019. Heterogeneous Von Neumann/Dataflow Microprocessors. *Commun. ACM* 62, 6 (May 2019), 83–91. https://doi.org/10.1145/3323923

[26] Salvador Petit, Rafael Ubal, Julio Sahuquillo, and Pedro López. 2014. Efficient Register Renaming and Recovery for High-Performance Processors. *IEEE Trans. VLSI Syst.* 22, 7 (July 2014), 1506–1514. https://doi.org/10.1109/TVLSI.2013.2270001

[27] Efraim Rotem, Adi Yoaz, Lihu Rappoport, Stephen J. Robinson, Julius Yuli Mandelblat, Arik Gihon, Eliezer Weissmann, Rajshree Chabukswar, Vadim Basin, Russell Fenger, Monica Gupta, and Ahmad Yasin. 2022. Intel Alder Lake CPU Architectures. *IEEE Micro* 42, 3 (May 2022), 13–19. https://doi.org/10.1109/MM.2022.3164338

[28] Satish Kumar Sadasivam, Brian W. Thompto, R. Kalla, and William J. Starke. 2017. IBM POWER9 Processor Architecture. *IEEE Micro* 37, 2 (March 2017), 40–51. https://doi.org/10.1109/MM.2017.40

[29] Elham Safi, Patrick Akl, Andreas Moshovos, Andreas Veneris, and Aggeliki Arapoyianni. 2007. On the Latency, Energy and Area of Checkpointed, Superscalar Register Alias Tables. In *Proceedings of the 2007 International Symposium on Low Power Electronics and Design* (Portland, Oregon) *(ISLPED '07)*. ACM, New York, NY, USA, 379–382. https://doi.org/10.1145/1283780.1283863

[30] Rama Sangireddy. 2006. Reducing rename logic complexity for high-speed and low-power front-end architectures. *IEEE Trans. Comput.* 55, 6 (June 2006), 672–685. https://doi.org/10.1109/TC.2006.88

[31] Hiroshi Sasaki, Fang-Hsiang Su, Teruo Tanimoto, and Simha Sethumadhavan. 2017. Why do programs have heavy tails?. In *2017 IEEE International Symposium on Workload Characterization* (Seattle, Washington, USA) *(IISWC)*. IEEE Computer Society, Los Alamitos, CA, USA, 135–145. https://doi.org/10.1109/IISWC.2017.8167771

[32] Andreas Sembrant, Trevor Carlson, Erik Hagersten, David Black-Shaffer, Arthur Perais, André Seznec, and Pierre Michaud. 2015. Long Term Parking (LTP): Criticality-Aware Resource Allocation in OOO Processors. In *Proceedings of the 48th International Symposium on Microarchitecture* (Waikiki, Hawaii) *(MICRO-48)*. ACM, New York, NY, USA, 334–346. https://doi.org/10.1145/2830772.2830815

[33] André Seznec and Pierre Michaud. 2006. A case for (partially) TAgged GEometric history length branch prediction. *J. Instruction-Level Parallelism* 8 (Feb. 2006), 1–23. https://jilp.org/vol8/v8paper1.pdf

[34] Tingting Sha, Milo M. K. Martin, and Amir Roth. 2006. NoSQ: Store-Load Communication without a Store Queue. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* (Orlando, Florida) *(MICRO-39)*. IEEE Computer Society, Los Alamitos, CA, USA, 285–296. https://doi.org/10.1109/MICRO.2006.39

[35] Ryota Shioya and Hideki Ando. 2014. Energy efficiency improvement of renamed trace cache through the reduction of dependent path length. In *2014 32nd IEEE*

International Conference on Computer Design (Seoul, South Korea) *(ICCD)*. IEEE Computer Society, Los Alamitos, CA, USA, 416–423. https://doi.org/10.1109/ICCD.2014.6974714
[36] Ryota Shioya, Masahiro Goshima, and Hideki Ando. 2014. A Front-End Execution Architecture for High Energy Efficiency. In *47th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, United Kingdom) *(MICRO 2014)*. IEEE Computer Society, Los Alamitos, CA, USA, 419–431. https://doi.org/10.1109/MICRO.2014.35
[37] Ryota Shioya, Kazuo Horio, Masahiro Goshima, and Shuichi Sakai. 2010. Register Cache System Not for Latency Reduction Purpose. In *The 43rd Annual IEEE/ACM International Symposium on Microarchitecture* (Atlanta, Georgia) *(MICRO 2010)*. IEEE Computer Society, Los Alamitos, CA, USA, 301–312. https://doi.org/10.1109/MICRO.2010.43
[38] Faissal M. Sleiman and Thomas F. Wenisch. 2016. Efficiently Scaling Out-of-Order Cores for Simultaneous Multithreading. In *2016 43rd International Symposium on Computer Architecture* (Seoul, South Korea) *(ISCA 2016)*. IEEE Computer Society, Los Alamitos, CA, USA, 431–443. https://doi.org/10.1109/ISCA.2016.45
[39] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. 2007. Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture* (Scottsdale, Arizona) *(HPCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 63–74. https://doi.org/10.1109/HPCA.2007.346185
[40] Standard Performance Evaluation Corporation. 2006. Standard performance evaluation corporation CPU2006 benchmark suite. https://www.spec.org/cpu2006/
[41] Standard Performance Evaluation Corporation. 2017. Standard performance evaluation corporation CPU2017 benchmark suite. https://www.spec.org/cpu2017/
[42] A.S. Tanenbaum. 1980. The Future of Distributed Computer Architecture. *Informatie* 22, July/Augus (1980), 500–503.
[43] Sriram Vajapeyam and Tulika Mitra. 1997. Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code Sequences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture* (Denver, Colorado) *(ISCA '97)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/264107.264119
[44] Naveen Vedula, Arrvindh Shriraman, Snehasish Kumar, and William N Sumner. 2018. NACHOS: Software-Driven Hardware-Assisted Memory Disambiguation for Accelerators. In *24th IEEE International Symposium on High Performance Computer Architecture* (Vienna, Austria) *(HPCA 2018)*. IEEE Computer Society, Los Alamitos, CA, USA, 710–723. https://doi.org/10.1109/HPCA.2018.00066
[45] Kenichi Watanabe, *et al.* 2005. Processor simulator Onikiri2. https://github.com/onikiri/onikiri2
[46] Andrew Waterman and Krste Asanovíc. 2019. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*. RISC-V Foundation.
[47] Neil H. E. Weste and David Money Harris. 2010. *CMOS VLSI Design: A Circuits and Systems Perspective* (fourth ed.). Addison Wesley, Boston, MA, USA.
[48] Kenneth C. Yeager. 1996. The MIPS R10000 superscalar microprocessor. *IEEE Micro* 16, 2 (April 1996), 28–41. https://doi.org/10.1109/40.491460

A ARTIFACT APPENDIX

A.1 Abstract

We have prepared this artifact for you to reproduce the results of Fig. 3, 4, 7, 13, 15, 16, 17, and 18. Following the instructions below, you can make CoreMark binaries for RISC-V, STRAIGHT, and Clockhands and run them on Onikiri2, a processor simulator.

A.2 Artifact check-list (meta-information)

- **Program:** Onikiri2, sasm2, clockhands-assembler2, musl, and CoreMark are included in our artifact. RISC-V GNU Compiler Toolchain and Clang compiler are available in public GitHub repositories. You can use SPEC 2006/2017 if you have a license.
- **Compilation:** g++ 11.4.0.
- **Run-time environment:** Ubuntu 22.04 LTS, GNU tar 1.34, GNU Make 4.3, GNU Awk 5.1.0, GNU sed 4.8, GNU grep 3.7, perl v5.34.0; Windows 11, Excel version 2307.
- **Metrics:** Improvement in the number of cycles taken to execute a measurement section (simulated value, not wall-clock time).
- **Output:** XML file that contains simulation results. We also provide Excel files to make figures.
- **Experiments:** Download our artifact; prepare binaries of CoreMark; build Onikiri2; run Onikiri2; observe results.
- **How much disk space required (approximately)?:** 2 GiB.
- **How much time is needed to prepare workflow (approximately)?:** For building GCC, it takes about 1 hour. The others are done in five minutes.
- **How much time is needed to complete experiments (approximately)?:** Five minutes for Fig. 13, 15, 16, 17, and 18. For Fig. 3, 4, and 7, they require 3×10^{13} instructions executions, which take 60 days. You can shorten the measurement region by rewriting input parameter files.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** Creative Commons Attribution 4.0 International, except for third-party codes.
- **Archived (provide DOI)?:** 10.5281/zenodo.8218698

A.3 Description

A.3.1 How to access. Our artifact is hosted on Zenodo at https://doi.org/10.5281/zenodo.8218698.

A.3.2 Hardware dependencies. Any hardware capable of compiling Onikiri2 [45].

A.3.3 Software dependencies. We have tested our artifact with the compiler and run-time environment listed in Section A.2. It may work in other environments, such as another Linux Distribution, but we have yet to check.

A.3.4 Data sets. Our artifact includes assemblers, a simulator, and CoreMark necessary for the experiments, except for RISC-V GNU Compiler Toolchain and Clang compiler. If you have a SPEC 2006/2017 license and want to try bzip2, mcf_s, lbm_s, and xz_s, please get in touch with us. We will give you the assemblies of them.

A.4 Installation

Download our artifact from Zenodo and extract it.

```
$ wget https://zenodo.org/record/8218698/files/
Clockhands_Artifact_MICRO2023.tar?download=1
$ tar xvf Clockhands_Artifact_MICRO2023.tar
$ cd Clockhands_Artifact_MICRO2023
```

You will find two README files, `ClockhandsEvaluation/README.md` and `ClockhandsPreliminaryExperiments/README.md`. Please follow their instructions to install RISC-V GNU Compiler Toolchain, Clang compiler, and other required software.

A.5 Experiment workflow

The workflow of the experiment is outlined as follows:

(1) Make CoreMark binaries for the three ISAs. For RISC-V, make a binary from the source code in our artifact with Clang compiler and assemble it using RISC-V GNU Compiler Toolchain and musl. For STRAIGHT and Clockhands, make a binary from the assemblies of CoreMark in our artifact. To do this, you can use pre-built binaries of sasm2 (a STRAIGHT assembler) and clockhands-assembler2 included in our artifact.
(2) Build Onikiri2.

(3) Copy the three CoreMark binaries and the Onikiri2 binary to the evaluation directory.
(4) Run Onikiri2.

For detailed instructions, refer to the README files.

A.6 Evaluation and expected results

All results of the experiment are output to XML files in directories ending with "result." You will find instructions on extracting the results in the Excel files we provide. Extract and compare the results with the expected ones in the Excel files. Finally, you will get the same charts as in Fig. 13, 15, 16, 17, and 18. You will also get similar charts as Fig. 3, 4, and 7.

A.7 Experiment customization

If you have enough time, you can obtain the same charts as Fig. 3, 4, and 7 by running 3×10^{13} instructions. It will take 60 days.

A.8 Methodology

Submission, reviewing, and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html