Phalanx: A Processor Simulator Based on the Entity Component System Architecture

Toshiki Maekawa
maekawa@matlab.nitech.ac.jp
Nagoya Institute of Technology
Aichi, Japan

Akihiko Odaki
odaki@rsg.ci.i.u-tokyo.ac.jp
The University of Tokyo
Tokyo, Japan

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

Tomoaki Tsumura
tsumura@acm.org
Nagoya Institute of Technology
Aichi, Japan

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

Abstract

In microarchitecture research, the standard approach involves implementing and evaluating new designs within processor simulators. However, leading simulators each take a different approach to implementing crucial hardware components, such as the instruction pipeline, and most were not designed with code modifiability in mind. Consequently, implementing a new microarchitecture forces researchers to first master the complex internal design of a simulator and then manually modify its code. This process often results in a codebase that is difficult to comprehend and maintain. In this paper, we demonstrate that the Entity Component System (ECS), a software architecture pattern common in video game development, is a highly effective model for structuring the timing simulation of multi-stage pipeline architectures. We also present *Phalanx*, a new, cycle-accurate processor simulator built from the ground up on the ECS software architecture pattern.

Keywords

Processor Simulator, Microarchitecture, Entity Component System

1 Introduction

In microarchitecture research, new techniques are typically evaluated by implementing them in a processor simulator. Existing cycle-accurate simulators include gem5 [7], Sniper [3], ChampSim [4], and an object-oriented simulator Onikiri2 [1].

To handle the complexity of modern instruction pipelines, these simulators use a divide-and-conquer approach. This is often done either by encapsulating the procedures for different pipeline stages and instructions or by using object-oriented programming to define them as distinct objects. However, the unit of division varies among simulators, making them difficult to understand. Additionally, simulators have two main issues that complicate modification. First, to change one functionality, we have to alter code that is spread out in multiple places. Second, it is hard for us to define a proper interface in advance.

To address these issues, we adopt a data-oriented software architecture pattern known as the *Entity Component System (ECS)*, which has become widely used in video game development [11, 12]. ECS has two main features: (1) composition over inheritance[1] and (2) separation of data from its update procedures.

We found that these two features contribute to the ease of implementation in processor pipeline timing simulators. The first feature, related to composition, enables researchers to implement new methods in the simulator easily. For example, it simplifies extending the structure that represents instructions to add new information. The second feature, related to separation, helps make it easier to understand the implementation of functionality that spans multiple stages (e.g., pipeline flushes). This is because, rather than implementing data update procedures separately for each stage to achieve a functionality, we can consolidate them into a single update procedure.

We present *Phalanx*, a cycle-accurate processor simulator designed based on ECS. In Phalanx, the data structures corresponding to the processor pipeline and the logic for updating the pipeline are designed based on ECS. We also discuss the issues that arise when adopting ECS into pipeline implementation and how to address them.

Our contributions are as follows:

- We demonstrated the advantages of using ECS for pipeline timing simulation.
- We presented a design and a systematic approach for adopting ECS to implement pipeline timing simulations.
- We demonstrated that a processor simulator implemented using our approach operates at a speed comparable to that of other simulators.

2 Problems in Existing Simulators

2.1 Scattered Implementations of Functionality

Existing processor simulators are designed so that code for a single functionality is scattered across multiple locations, which makes it difficult to identify which parts need to be modified. For example, in gem5, the *instruction wake-up*[2] involves multiple classes across source files. As another example, in ChampSim, the same process is contained in a single class but implemented through various

This work was supported by JSPS KAKENHI Grant Numbers JP20H04153, JP22K19774 and JP24K20753.

[1]The idea that code reuse should be achieved through composition rather than inheritance in cases of a *has-a* relationship rather than an *is-a* relationship [2].

[2]In an out-of-order processor, *instruction wake-up* means a process of notifying that an instruction has become ready to execute. In the simplest design, a notification is sent to its dependent instructions when a producer instruction completes.

method calls, which makes it difficult to understand which part of the code implements a single functionality.

This issue is significant when implementing and evaluating multiple techniques. In research, the full scope of modifications is rarely clear from the beginning, and useful techniques are often discovered through trial and error. As a result, if understanding scattered code is necessary each time a modification is made, the research iteration slows down.

In addition, the mutability of states further complicates code comprehension. This occurs when the code that implements a single functionality is scattered, allowing the state to be updated from multiple source locations. As a result, the behavior cannot be correctly understood unless we trace where, by whom, and in what order updates occur in the code.

As an example, consider modeling the behavior of a pipeline stall. When a stall occurs at some stage, upstream stages must also be stalled. A circuit that implements this would be as shown in Figure 1(a). If this is implemented in a design that treats pipeline stages as objects in an object-oriented style, the code would take the form shown in Figure 1(b) using the observer pattern. However, it is not immediately apparent, just by looking at this code, what specific events affect the `stall` variable.

2.2 Difficulty of Defining Interfaces

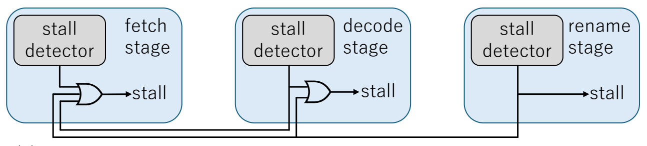
Even when we know precisely which parts of the code to modify, it is difficult to make those changes without breaking the simulator. The reason is that the predefined interfaces in processor simulators are often insufficient for microarchitecture research, which, by its nature, aims to find new techniques to improve processor performance under specific constraints. Such techniques usually require leveraging information from other hardware components, which in turn necessitates extending existing interfaces.

For example, the RUNLTS branch predictor [6] uses register values produced in the execution stage. It is essential to modify the code to implement this method because processor simulators rarely provide an interface for other parts of the system to read the execution stage outputs.

Similarly, consider a cache replacement policy interface that requires the set index and way number for each operation. While this is sufficient for common algorithms, it is difficult to implement one that deviates from this design, such as an algorithm that uses the instruction address of a load instruction[3]. Consequently, we are often forced to add interfaces that diverge from the initial design to implement new techniques.

When extending an interface beyond its original design, we end up modifying internal information that was previously hidden. This means that we must carefully design interfaces while considering the invariants that the internal state is supposed to maintain. However, existing simulators often do not clearly state what these invariants for the internal state are. This forces us to investigate them ourselves, creating an additional burden.

[3]Hawkeye cache replacement policy [5] is such an example.



(a) Stall detection circuit for instruction pipeline stages.

```
class PipelineStage {
    ...
    bool stall;
    PipelineStage* previous_stage;
public:
    virtual bool decide_stall() = 0;
    void update() {
        stall |= decide_stall();
        if( stall ) { previous_stage->request_stall(); }
    }
    void request_stall() { stall = true; }
};
```

(b) Example written in an object-oriented style.

```
bool rn_stl = decide_rename_stall();
bool dc_stl = decide_decode_stall();
bool f_stl  = decide_fetch_stall();

bool rename_stall = rn_stl;
bool decode_stall = dc_stl | rn_stl;
bool fetch_stall  = f_stl | dc_stl | rn_stl;
```

(c) Example that mirrors the circuit implementation.

Figure 1: A circuit that implements pipeline stalls and example simulation code. (a) The circuit computes the stall condition by aggregating signals from multiple stages. (b) Example code that implements this behavior using the observer pattern. (c) Example code that mirrors the circuit implementation.

3 Entity Component System (ECS)

3.1 Overview

ECS is a data-oriented software architecture pattern that has gained popularity in the domain of video game development. We explain its key concepts using examples from that domain.

In ECS, every distinct object in a video game is assigned an *entity*. The term *objects in a video game* refers, for example, to characters and items on the field in an action game, or to bullets in a scrolling shooter game.

Each entity has an arbitrary combination of *components*. A component is a data unit representing attribute data associated with an entity. For example, the entity of a character or an item would have a component for its current position. If its movement is governed by physics, it would have a component storing attributes such as mass and velocity. Similarly, if it is intended to disappear after a fixed time, it would have a component that stores its remaining lifetime. Figure 2(a) shows an example design of entities and components for an action game. Because components are not organized in an inheritance hierarchy, the design remains flexible for the later addition of functionality.

A *system* is a function that modifies component data, and may optionally include other data it requires. In ECS, behaviors are decoupled from components, which makes behaviors involving multiple components straightforward to express. These features help maintain ECS-based software without unnecessary dependencies, both among components and between systems and components, thereby improving the overall clarity of the code.

3.2 Advantages

3.2.1 Centralized Functionality. We can group all the code for a single functionality together because ECS decouples functionality from specific components. This enables such functionality to be statically defined as a single procedure in one place. ECS also clearly separates which components are part of a single functionality and which are not. This design makes the code implementing any specific feature more straightforward to understand.

3.2.2 Streamlined Interface Design. By adopting ECS, we can provide a common design guideline that simplifies the definition of interfaces. This guideline is established by clarifying what should be assigned to each of the three core ECS elements: *entities*, *components*, and *systems*. Furthermore, we provide a design principle for how systems reference components, which makes the resulting interfaces straightforward and self-evident.

4 Phalanx Simulator

4.1 Entity-Component Design

Phalanx has two groups of component arrays: one for the instruction pipeline and another for the memory subsystem. These groups are updated every simulation cycle. Figure 2(b) and (c) illustrate the design of entities and components in Phalanx.

For the instruction pipeline, each entity corresponds to a single instruction, while components are defined at the granularity of the data produced by each pipeline stage. The logic of every stage is implemented as a system that computes the stage results. The system also includes certain data that is not tied to any specific instruction (e.g., branch target buffer).

For the memory subsystem, each entity corresponds to a series of memory-level accesses to a specific address, while components are defined at the granularity of the results produced as data requests traverse the memory hierarchy. For intuition, we can think of an entity as a single round-trip dataflow originating either at the execution stage in the instruction pipeline or at any level of the memory hierarchy. As with the instruction pipeline stages, the behavior of every memory level is implemented as a system. Thus, ECS can also model hardware other than the instruction pipeline.

Through the development of Phalanx, we found that treating *bundled information in dataflow* as an entity improves design clarity. Instructions and memory accesses are precise examples of bundled information in dataflow. Each time bundled information flows, either new data is added to it or some is removed from it. By exploiting this characteristic to design components based on what is added or removed, a complex concept can be naturally represented in ECS.

4.2 Design Principles

4.2.1 Maintaining a Single Source of Data. Processors determine their behavior by leveraging information from multiple sources. If this information is duplicated across sources, the copies may become inconsistent. Although existing simulators prevent such inconsistencies through encapsulation, it is incompatible with our design, which decouples data from behavior. Therefore, our design requires a principle to prevent them.

To avoid this problem, we decided not to memoize data that can be calculated from other data. Although memoizing calculation results often speeds up the simulator, this principle takes precedence. Despite this principle being in place, ECS can leverage memory access locality, easily achieving sufficient simulation speed. Although Phalanx includes several computationally intensive yet straightforward parts of code (e.g., linear searching for every simulated cycle) to follow the principle, its simulation speed remains comparable to that of existing simulators.

4.2.2 Single Primary Update Function per Component. ECS makes it difficult to identify which function updates a specific component. Proper encapsulation ensures that only the methods of a class can modify its fields. However, in ECS, functions are not tied to components, and thus, it is unclear where the code that modifies a component is located. Furthermore, when a component is modified in multiple source locations, understanding the overall behavior requires knowledge of the update order and all its actions.

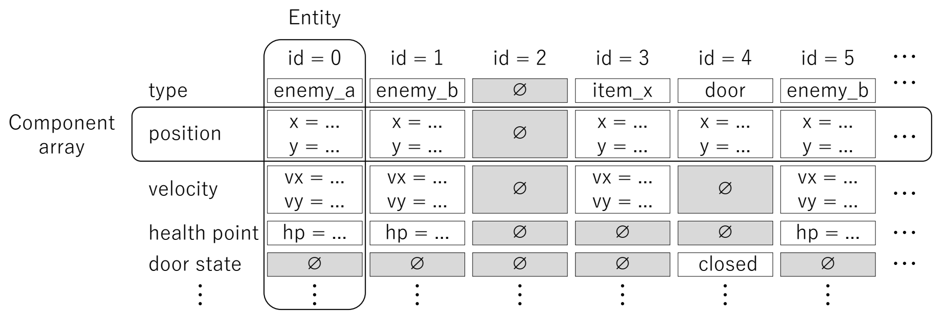
To avoid this problem, a component should be generally modified by one function. However, exceptions include functions that update data with widespread effects, such as pipeline flushes. Based on our experience developing Phalanx, this principle does not pose a significant obstacle.

4.2.3 Prioritizing Modifiability. Unlike existing simulators, Phalanx is designed to be modified by directly rewriting its code. Many existing simulators are designed as if they were libraries. In other words, they are often implemented to accept callback functions or to allow overrides through polymorphism, so that existing code does not need to be modified. However, as mentioned earlier, when implementing a new method that cannot be realized within the existing interfaces, it becomes necessary to modify the code directly.

Code modifications on the *library* can significantly burden researchers. This is because modifying the code may violate invariants that are implicitly assumed within the *library*. Consequently, researchers making the modifications must investigate the invariants that were originally guaranteed. Worse still, such invariants are often undocumented, which further increases the burden on the researchers.

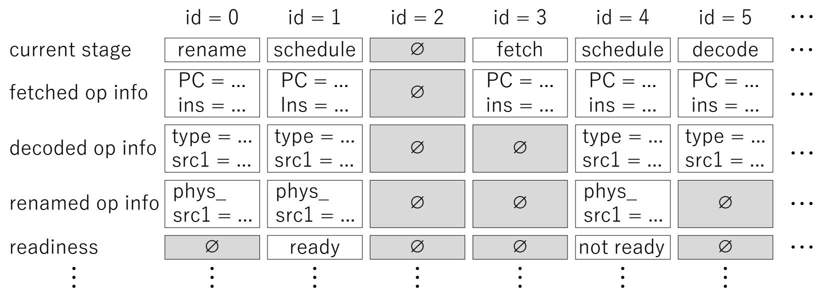
We thus concluded that it is preferable to write the code from the outset with the assumption that modifications will be made by directly rewriting it. In other words, our simulator is designed so that its modifiers (i.e., researchers) do not need to investigate data invariants. This is achieved by ensuring that our simulator does not conceal component data but instead makes all of it accessible, and by maintaining a single source of data. Because our simulator guarantees a single source of data, the researchers can freely make changes without unnecessary investigation. Although this requires the simulator provider to design it with greater care, our priority is to reduce the burden on the researchers.

It should be noted that these constraints are not necessarily imposed on the researchers. Since researchers are fully aware of their own modifications, they can ensure that the invariants remain unchanged. Thus, if necessary, modifications that violate these constraints can also be made. This approach ensures the ease of modification.


Entity
Component array

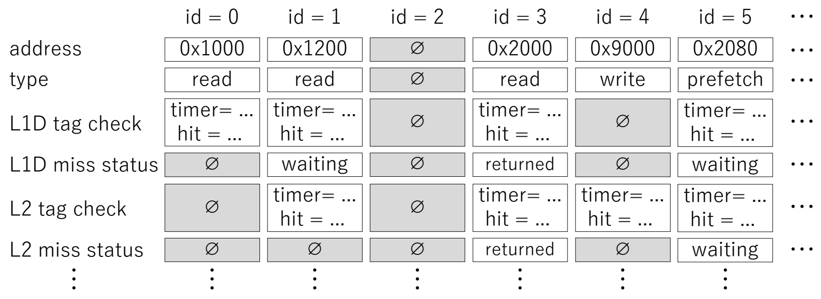
	id = 0	id = 1	id = 2	id = 3	id = 4	id = 5	⋯
type	enemy_a	enemy_b	∅	item_x	door	enemy_b	⋯
position	x = … y = …	x = … y = …	∅	x = … y = …	x = … y = …	x = … y = …	⋯
velocity	vx = … vy = …	vx = … vy = …	∅	vx = … vy = …	∅	vx = … vy = …	⋯
health point	hp = …	hp = …	∅	∅	∅	hp = …	⋯
door state	∅	∅	∅	∅	closed	∅	⋯
⋮	⋮	⋮	⋮	⋮	⋮	⋮	

(a) Example design of entities and components for video games.



	id = 0	id = 1	id = 2	id = 3	id = 4	id = 5	⋯
current stage	rename	schedule	∅	fetch	schedule	decode	⋯
fetched op info	PC = … ins = …	PC = … Ins = …	∅	PC = … ins = …	PC = … ins = …	PC = … ins = …	⋯
decoded op info	type = … src1 = …	type = … src1 = …	∅	∅	type = … src1 = …	type = … src1 = …	⋯
renamed op info	phys_ src1 = …	phys_ src1 = …	∅	∅	phys_ src1 = …	∅	⋯
readiness	∅	ready	∅	∅	not ready	∅	⋯
⋮	⋮	⋮	⋮	⋮	⋮	⋮	

(b) Example design of entities and components for instruction pipelines.



	id = 0	id = 1	id = 2	id = 3	id = 4	id = 5	⋯
address	0x1000	0x1200	∅	0x2000	0x9000	0x2080	⋯
type	read	read	∅	read	write	prefetch	⋯
L1D tag check	timer= … hit = …	timer= … hit = …	∅	timer= … hit = …	∅	timer= … hit = …	⋯
L1D miss status	∅	waiting	∅	returned	∅	waiting	⋯
L2 tag check	∅	timer= … hit = …	∅	timer= … hit = …	timer= … hit = …	timer= … hit = …	⋯
L2 miss status	∅	∅	∅	returned	∅	waiting	⋯
⋮	⋮	⋮	⋮	⋮	⋮	⋮	

(c) Example design of entities and components for memory subsystems.

Figure 2: Example design of video game entities and components, followed by processor simulator examples: an instruction pipeline and a memory subsystem. Components that can be added to a single entity are arranged *vertically*, and components of the same kind are arranged *horizontally*. (a) In the video game example, the entity with "id=3" is of type item_x and has position and velocity components, but does not have a health point component. The entity with "id=2" is unused. In ECS, each entity can have an arbitrary combination of components, making it straightforward to extend functionality through composition.

4.3 Processor-Specific Logic

4.3.1 Intra-Cycle Sequential Logic. Existing processor simulators model the simultaneous updating of clocked sequential logic in different ways. For example, ChampSim, gem5, and Sniper read and update data in an appropriate order. Onikiri2, on the other hand, creates a closure containing the read data and updates the data later using that closure.

In contrast, Phalanx double-buffers component arrays, akin to `sc_signal` in SystemC [9]. This yields deterministic results regardless of the order in which update functions are evaluated. Moreover, because the current and next component arrays are explicitly present in memory, it becomes easier to implement intra-cycle sequential behavior.

4.3.2 Multi-Cycle Pipeline. Existing simulators use the following strategies to implement stages that take multiple cycles.

- Calculate the absolute cycle to enter the next stage (ChampSim, Sniper, gem5)
- Use a queue that can be inserted at any position (Onikiri2)

The strategy of *calculating the absolute cycle to enter the next stage* is easy to understand, but its correctness is not necessarily clear. This is because if events such as pipeline stalls occur in the future, the cycle to enter the next stage will become incorrect. gem5 guarantees that every instruction can enter the next stage at the calculated cycle by providing skid buffers. However, it is difficult to model a configuration that does not use skid buffers.

The strategy of *using a queue that can be inserted at any position* is an advanced version of the previous strategy. The problem mentioned above is solved by not retrieving from the queue during cycles in which pipeline stalls occur. However, it remains difficult to account for the effects of dynamic events other than pipeline stalls. This problem is significant when implementing speculative scheduling mechanisms.

In contrast, Phalanx takes a strategy of adding a component that tracks information such as *how many more cycles until completion* (`timer`). All we need to do is decrement this variable (`--timer;`) in each cycle, and it is clear what is happening. In the event of a pipeline stall, omitting `--timer;` clearly indicates that processing has not progressed. Additionally, any other dynamic events can be accommodated by updating the `timer` value to an appropriate value after the event occurs, making the strategy highly flexible.

5 Evaluation

5.1 Methodology

By design, Phalanx includes sections of computationally intensive code. We demonstrate that it remains fast and that the design does not pose a practical problem.

The processor simulators used for comparison are ChampSim, Onikiri2, and gem5. Except for ChampSim, which uses x86, all the simulated processors use the RISC-V ISA. For reference, we also measured the performance of RSD [8] compiled with Verilator [10]. RSD is a soft processor described in SystemVerilog, and Verilator is a compiler from the synthesis subset of SystemVerilog to C++.

Table 1 summarizes the evaluation environment, and Table 2 lists the parameters of the simulated processor. The programs and execution traces used in the evaluation differ across simulators; therefore, the results should be regarded as indicative rather than strictly comparable.

5.2 Results

Table 3 shows the results of the simulation speed evaluation. We use *800 kHz* to denote that the simulator can simulate 800000 cycles in a wall-clock second. As a rough indicator of the complexity of the simulated processor, the number of pipeline stages is also shown.

Phalanx (a) is an implementation that does not double-buffer the component arrays; thus, it does not copy them every cycle. Phalanx (b) is a complete implementation that utilizes double-buffering. Both implementations perform the same simulation. Phalanx (a) ran at 800 kHz, faster than other processor simulators. On the other hand, Phalanx (b) ran at 710 kHz, which is comparable to other processor simulators. A simulation of 10^9 (=1G) cycles can be completed in approximately 23 minutes when the simulator runs at 710 kHz.

Figures 3–7 show the pipeline view diagrams of the simulated processors. Phalanx and Onikiri2 model an out-of-order execution engine that includes speculative scheduling. gem5 excels at full-system simulation; however, its model of an out-of-order core is comparatively coarse. ChampSim simulates the front-end pipeline and cache behavior in detail. However, the out-of-order engine in ChampSim employs many idealized mechanisms. The RSD processor is an actual hardware implementation rather than a simulator.

Table 1: Evaluation environment, i.e., the host machine on which the simulator program was run.

OS	Ubuntu 24.04 LTS
CPU	Ryzen 9 7950X
C++ Compiler	g++ 13.4.0
SystemVerilog compiler	verilator 4.108

Table 2: Configurations of the simulated processors.

	Phalanx	Onikiri2	gem5	ChampSim	RSD
Entities	96	96†	N/A	N/A	N/A
Fetch width	2	2	2	2	2
Physical registers	128	128	128	N/A (∞)	128
Load-store queue	16	16	16*	16*	16
Scheduler	36	36	36	36	36‡
Issue latency	2	2	N/A (0)	N/A (0)	2
Reorder buffer	64	64	64	64	64
Commit width	2	2	2	2	2

†: `OpArrayCapacity`.
‡: 16 for MatrixScheduler and 20 for ReplayQueue.
*: Set both LQ size and SQ size to 16.

Table 3: Simulation speed comparison.

	Speed	Inst. pipeline stages
Phalanx (a)	8.0×10^2 kHz	10
gem5	7.6×10^2 kHz	6–7
Phalanx (b)	7.1×10^2 kHz	10
Onikiri2	3.0×10^2 kHz	9
ChampSim	1.9×10^2 kHz	9
RSD	8.4×10^1 kHz	12–14

Because of hardware constraints, RSD does not always issue instructions in a strict oldest-first order and thus exhibits behaviors that conventional simulators typically idealize away.

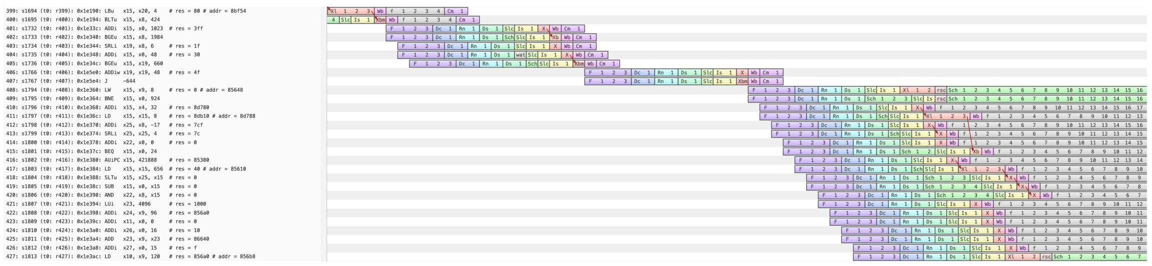
6 Conclusion

We found that the instruction pipelines of existing processor simulators are not designed to be easily modified. In other words, existing simulators each take a different approach to implementing them, and most were not designed with code modifiability in mind.

We adopted the Entity Component System (ECS) software architecture pattern into the instruction pipeline and memory subsystem design. We proposed principles on how to write code to address potential issues that may arise when implementing using ECS. Additionally, we proposed methods for implementing logic specific to processor simulators.

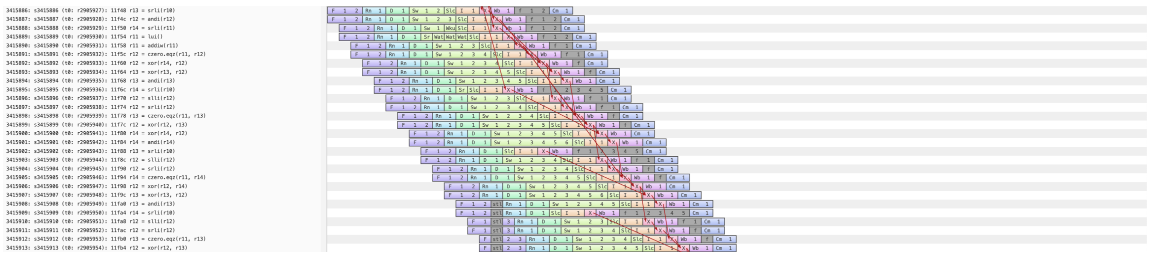
We presented a cycle-accurate processor simulator called Phalanx, designed based on ECS. We compared the execution speed of Phalanx with that of other processor simulators. Although the principles introduced in Phalanx enforce computationally intensive code implementation, Phalanx operates at about the same simulation speed as other processor simulators.

We demonstrated that ECS is useful for pipeline timing simulation. Although Phalanx is a CPU simulator, it could be extended to simulate other pipeline processors. For example, it should be possible to create highly parallelized GPU simulators. Since it adopts a double-buffering method, we expect that parallelization can be introduced without difficulty.



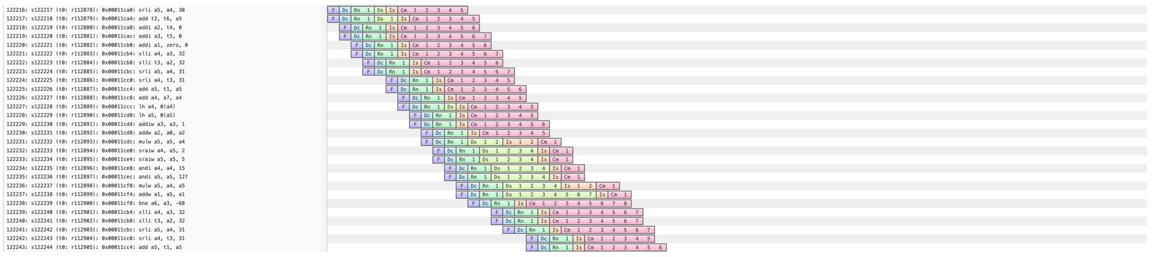
```
399: s1694 (t0: r399): 0x1e190: LBu   x15, x20, 4    # res = 80 # addr = 8bf54
400: s1695 (t0: r400): 0x1e194: BLTu  x15, x8, 424
401: s1732 (t0: r401): 0x1e33c: ADDi  x15, x0, 1023  # res = 3ff
402: s1733 (t0: r402): 0x1e340: BGEu  x15, x8, 1984
403: s1734 (t0: r403): 0x1e344: SRLi  x19, x8, 6     # res = 1f
404: s1735 (t0: r404): 0x1e348: ADDi  x15, x0, 48    # res = 30
405: s1736 (t0: r405): 0x1e34c: BGEu  x15, x19, 660
406: s1766 (t0: r406): 0x1e5e0: ADDiw x19, x19, 48   # res = 4f
407: s1767 (t0: r407): 0x1e5e4: J     -644
408: s1794 (t0: r408): 0x1e360: LW    x15, x9, 8     # res = 0 # addr = 85648
409: s1795 (t0: r409): 0x1e364: BNE   x15, x0, 924
410: s1796 (t0: r410): 0x1e368: ADDi  x15, x4, 32    # res = 8d780
411: s1797 (t0: r411): 0x1e36c: LD    x15, x15, 8    # res = 8db10 # addr = 8d788
412: s1798 (t0: r412): 0x1e370: ADDi  x25, x8, -17   # res = 7cf
413: s1799 (t0: r413): 0x1e374: SRLi  x25, x25, 4    # res = 7c
414: s1800 (t0: r414): 0x1e378: ADDi  x22, x0, 0     # res = 0
415: s1801 (t0: r415): 0x1e37c: BEQ   x15, x0, 24
416: s1802 (t0: r416): 0x1e380: AUiPC x15, 421888    # res = 85380
417: s1803 (t0: r417): 0x1e384: LD    x15, x15, 656  # res = 40 # addr = 85610
418: s1804 (t0: r418): 0x1e388: SLTu  x15, x25, x15  # res = 0
419: s1805 (t0: r419): 0x1e38c: SUB   x15, x0, x15   # res = 0
420: s1806 (t0: r420): 0x1e390: AND   x22, x8, x15   # res = 0
421: s1807 (t0: r421): 0x1e394: LUi   x23, 4096      # res = 1000
422: s1808 (t0: r422): 0x1e398: ADDi  x24, x9, 96    # res = 856a0
423: s1809 (t0: r423): 0x1e39c: ADDi  x11, x0, 0     # res = 0
424: s1810 (t0: r424): 0x1e3a0: ADDi  x26, x0, 16    # res = 10
425: s1811 (t0: r425): 0x1e3a4: ADD   x23, x9, x23   # res = 86640
426: s1812 (t0: r426): 0x1e3a8: ADDi  x27, x0, 15    # res = f
427: s1813 (t0: r427): 0x1e3ac: LD    x10, x9, 120   # res = 856a0 # addr = 856b8
```

Figure 3: Pipeline view of the processor simulated with Phalanx.



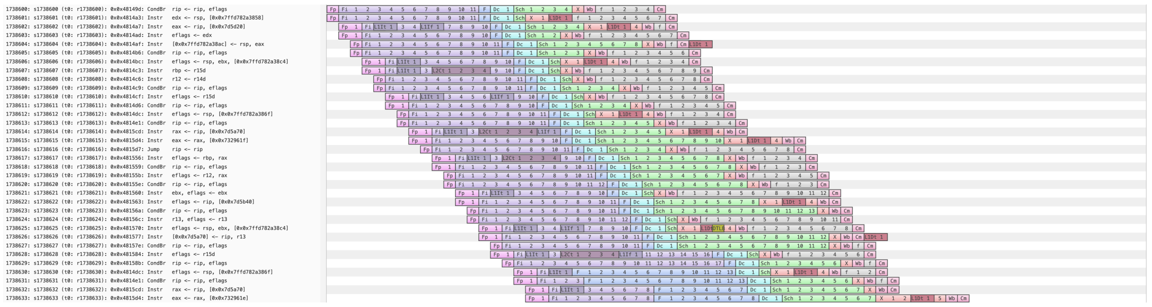
```
3415886: s3415886 (t0: r2905927): 11f48 r13 = srli(r10)
3415887: s3415887 (t0: r2905928): 11f4c r12 = andi(r12)
3415888: s3415888 (t0: r2905929): 11f50 r14 = srli(r11)
3415889: s3415889 (t0: r2905930): 11f54 r11 = lui()
3415890: s3415890 (t0: r2905931): 11f58 r11 = addiw(r11)
3415891: s3415891 (t0: r2905932): 11f5c r12 = czero.eqz(r11, r12)
3415892: s3415892 (t0: r2905933): 11f60 r12 = xor(r14, r12)
3415893: s3415893 (t0: r2905934): 11f64 r13 = xor(r13, r12)
3415894: s3415894 (t0: r2905935): 11f68 r13 = andi(r13)
3415895: s3415895 (t0: r2905936): 11f6c r14 = srli(r10)
3415896: s3415896 (t0: r2905937): 11f70 r12 = slli(r12)
3415897: s3415897 (t0: r2905938): 11f74 r12 = srli(r12)
3415898: s3415898 (t0: r2905939): 11f78 r13 = czero.eqz(r11, r13)
3415899: s3415899 (t0: r2905940): 11f7c r12 = xor(r12, r13)
3415900: s3415900 (t0: r2905941): 11f80 r14 = xor(r14, r12)
3415901: s3415901 (t0: r2905942): 11f84 r14 = andi(r14)
3415902: s3415902 (t0: r2905943): 11f88 r13 = srli(r10)
3415903: s3415903 (t0: r2905944): 11f8c r12 = slli(r12)
3415904: s3415904 (t0: r2905945): 11f90 r12 = srli(r12)
3415905: s3415905 (t0: r2905946): 11f94 r14 = czero.eqz(r11, r14)
3415906: s3415906 (t0: r2905947): 11f98 r12 = xor(r12, r14)
3415907: s3415907 (t0: r2905948): 11f9c r13 = xor(r13, r12)
3415908: s3415908 (t0: r2905949): 11fa0 r13 = andi(r13)
3415909: s3415909 (t0: r2905950): 11fa4 r14 = srli(r10)
3415910: s3415910 (t0: r2905951): 11fa8 r12 = slli(r12)
3415911: s3415911 (t0: r2905952): 11fac r12 = srli(r12)
3415912: s3415912 (t0: r2905953): 11fb0 r13 = czero.eqz(r11, r13)
3415913: s3415913 (t0: r2905954): 11fb4 r12 = xor(r12, r13)
```

Figure 4: Pipeline view of the processor simulated with Onikiri2.



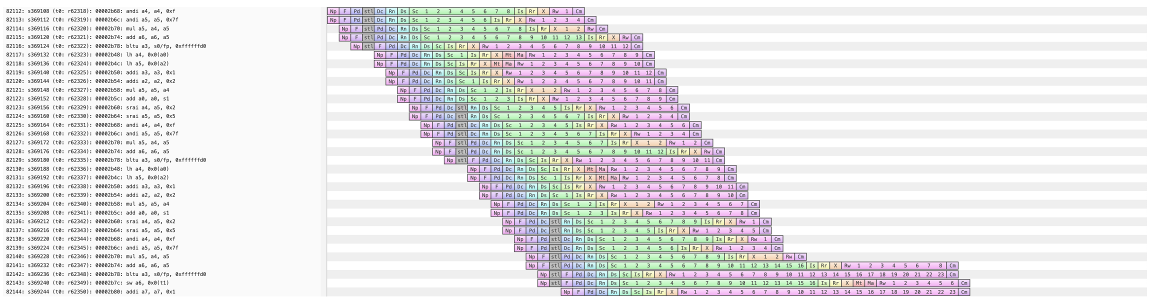
```
122216: s122217 (t0: r112878): 0x00011ca0: srli a5, a4, 30
122217: s122218 (t0: r112879): 0x00011ca4: add t2, t6, a5
122218: s122219 (t0: r112880): 0x00011ca8: addi a2, t4, 0
122219: s122220 (t0: r112881): 0x00011cac: addi a3, t5, 0
122220: s122221 (t0: r112882): 0x00011cb0: addi a1, zero, 0
122221: s122222 (t0: r112883): 0x00011cb4: slli a4, a3, 32
122222: s122223 (t0: r112884): 0x00011cb8: slli t3, a2, 32
122223: s122224 (t0: r112885): 0x00011cbc: srli a5, a4, 31
122224: s122225 (t0: r112886): 0x00011cc0: srli a4, t3, 31
122225: s122226 (t0: r112887): 0x00011cc4: add a5, t1, a5
122226: s122227 (t0: r112888): 0x00011cc8: add a4, a7, a4
122227: s122228 (t0: r112889): 0x00011ccc: lh a4, 0(a4)
122228: s122229 (t0: r112890): 0x00011cd0: lh a5, 0(a5)
122229: s122230 (t0: r112891): 0x00011cd4: addiw a3, a3, 1
122230: s122231 (t0: r112892): 0x00011cd8: addw a2, a0, a2
122231: s122232 (t0: r112893): 0x00011cdc: mulw a5, a5, a4
122232: s122233 (t0: r112894): 0x00011ce0: sraiw a4, a5, 2
122233: s122234 (t0: r112895): 0x00011ce4: sraiw a5, a5, 5
122234: s122235 (t0: r112896): 0x00011ce8: andi a4, a4, 15
122235: s122236 (t0: r112897): 0x00011cec: andi a5, a5, 127
122236: s122237 (t0: r112898): 0x00011cf0: mulw a5, a4, a5
122237: s122238 (t0: r112899): 0x00011cf4: addw a1, a5, a1
122238: s122239 (t0: r112900): 0x00011cf8: bne a6, a3, -68
122239: s122240 (t0: r112901): 0x00011cb4: slli a4, a3, 32
122240: s122241 (t0: r112902): 0x00011cb8: slli t3, a2, 32
122241: s122242 (t0: r112903): 0x00011cbc: srli a5, a4, 31
122242: s122243 (t0: r112904): 0x00011cc0: srli a4, t3, 31
122243: s122244 (t0: r112905): 0x00011cc4: add a5, t1, a5
```

Figure 5: Pipeline view of the processor simulated with gem5.



```
1738600: s1738600 (t0: r1738600): 0x0x48149d: CondBr  rip <- rip, eflags
1738601: s1738601 (t0: r1738601): 0x0x4814a3: Instr   edx <- rsp, [0x0x7ffd782a3858]
1738602: s1738602 (t0: r1738602): 0x0x4814a7: Instr   eax <- rip, [0x0x7d5d20]
1738603: s1738603 (t0: r1738603): 0x0x4814ad: Instr   eflags <- edx
1738604: s1738604 (t0: r1738604): 0x0x4814af: Instr   [0x0x7ffd782a38ac] <- rsp, eax
1738605: s1738605 (t0: r1738605): 0x0x4814b6: CondBr  rip <- rip, eflags
1738606: s1738606 (t0: r1738606): 0x0x4814bc: Instr   eflags <- rsp, ebx, [0x0x7ffd782a38c4]
1738607: s1738607 (t0: r1738607): 0x0x4814c3: Instr   rbp <- r15d
1738608: s1738608 (t0: r1738608): 0x0x4814c6: Instr   r12 <- r14d
1738609: s1738609 (t0: r1738609): 0x0x4814c9: CondBr  rip <- rip, eflags
1738610: s1738610 (t0: r1738610): 0x0x4814cf: Instr   eflags <- r15d
1738611: s1738611 (t0: r1738611): 0x0x4814d6: CondBr  rip <- rip, eflags
1738612: s1738612 (t0: r1738612): 0x0x4814dc: Instr   eflags <- rsp, [0x0x7ffd782a386f]
1738613: s1738613 (t0: r1738613): 0x0x4814e1: CondBr  rip <- rip, eflags
1738614: s1738614 (t0: r1738614): 0x0x4815cd: Instr   rax <- rip, [0x0x7d5a70]
1738615: s1738615 (t0: r1738615): 0x0x4815d4: Instr   eax <- rax, [0x0x732961f]
1738616: s1738616 (t0: r1738616): 0x0x4815d7: Jump    rip <- rip
1738617: s1738617 (t0: r1738617): 0x0x481556: Instr   eflags <- rbp, rax
1738618: s1738618 (t0: r1738618): 0x0x481559: CondBr  rip <- rip, eflags
1738619: s1738619 (t0: r1738619): 0x0x48155b: Instr   eflags <- r12, rax
1738620: s1738620 (t0: r1738620): 0x0x48155e: CondBr  rip <- rip, eflags
1738621: s1738621 (t0: r1738621): 0x0x481560: Instr   ebx, eflags <- ebx
1738622: s1738622 (t0: r1738622): 0x0x481563: Instr   eflags <- rip, [0x0x7d5b40]
1738623: s1738623 (t0: r1738623): 0x0x48156a: CondBr  rip <- rip, eflags
1738624: s1738624 (t0: r1738624): 0x0x48156c: Instr   r13, eflags <- r13
1738625: s1738625 (t0: r1738625): 0x0x481570: Instr   eflags <- rsp, ebx, [0x0x7ffd782a38c4]
1738626: s1738626 (t0: r1738626): 0x0x481577: Instr   [0x0x7d5a70] <- rip, r13
1738627: s1738627 (t0: r1738627): 0x0x48157e: CondBr  rip <- rip, eflags
1738628: s1738628 (t0: r1738628): 0x0x481584: Instr   eflags <- r15d
1738629: s1738629 (t0: r1738629): 0x0x48158b: CondBr  rip <- rip, eflags
1738630: s1738630 (t0: r1738630): 0x0x4814dc: Instr   eflags <- rsp, [0x0x7ffd782a386f]
1738631: s1738631 (t0: r1738631): 0x0x4814e1: CondBr  rip <- rip, eflags
1738632: s1738632 (t0: r1738632): 0x0x4815cd: Instr   rax <- rip, [0x0x7d5a70]
1738633: s1738633 (t0: r1738633): 0x0x4815d4: Instr   eax <- rax, [0x0x732961e]
```

Figure 6: Pipeline view of the processor simulated with ChampSim.



```
82112: s369108 (t0: r62318): 00002b68: andi a4, a4, 0xf
82113: s369112 (t0: r62319): 00002b6c: andi a5, a5, 0x7f
82114: s369116 (t0: r62320): 00002b70: mul a5, a4, a5
82115: s369120 (t0: r62321): 00002b74: add a6, a6, a5
82116: s369124 (t0: r62322): 00002b78: bltu a3, s0/fp, 0xffffffd0
82117: s369132 (t0: r62323): 00002b48: lh a4, 0x0(a0)
82118: s369136 (t0: r62324): 00002b4c: lh a5, 0x0(a2)
82119: s369140 (t0: r62325): 00002b50: addi a3, a3, 0x1
82120: s369144 (t0: r62326): 00002b54: addi a2, a2, 0x2
82121: s369148 (t0: r62327): 00002b58: mul a5, a5, a4
82122: s369152 (t0: r62328): 00002b5c: add a0, a0, s1
82123: s369156 (t0: r62329): 00002b60: srai a4, a5, 0x2
82124: s369160 (t0: r62330): 00002b64: srai a5, a5, 0x5
82125: s369164 (t0: r62331): 00002b68: andi a4, a4, 0xf
82126: s369168 (t0: r62332): 00002b6c: andi a5, a5, 0x7f
82127: s369172 (t0: r62333): 00002b70: mul a5, a4, a5
82128: s369176 (t0: r62334): 00002b74: add a6, a6, a5
82129: s369180 (t0: r62335): 00002b78: bltu a3, s0/fp, 0xffffffd0
82130: s369188 (t0: r62336): 00002b48: lh a4, 0x0(a0)
82131: s369192 (t0: r62337): 00002b4c: lh a5, 0x0(a2)
82132: s369196 (t0: r62338): 00002b50: addi a3, a3, 0x1
82133: s369200 (t0: r62339): 00002b54: addi a2, a2, 0x2
82134: s369204 (t0: r62340): 00002b58: mul a5, a5, a4
82135: s369208 (t0: r62341): 00002b5c: add a0, a0, s1
82136: s369212 (t0: r62342): 00002b60: srai a4, a5, 0x2
82137: s369216 (t0: r62343): 00002b64: srai a5, a5, 0x5
82138: s369220 (t0: r62344): 00002b68: andi a4, a4, 0xf
82139: s369224 (t0: r62345): 00002b6c: andi a5, a5, 0x7f
82140: s369228 (t0: r62346): 00002b70: mul a5, a4, a5
82141: s369232 (t0: r62347): 00002b74: add a6, a6, a5
82142: s369236 (t0: r62348): 00002b78: bltu a3, s0/fp, 0xffffffd0
82143: s369240 (t0: r62349): 00002b7c: sw a6, 0x0(t1)
82144: s369244 (t0: r62350): 00002b80: addi a7, a7, 0x1
```

Figure 7: Pipeline view of the RSD processor.

References

[1] 2006. Onikiri2. https://github.com/onikiri/onikiri2.

[2] Joshua Bloch. 2018. *Effective Java* (3 ed.). Addison-Wesley Professional.

[3] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. 2014. An Evaluation of High-Level Mechanistic Core Models. *ACM Trans. Archit. Code Optim.* 11, 3, Article 28 (Aug. 2014), 25 pages. https://doi.org/10.1145/2629677

[4] Nathan Gober, Gino Chacon, Lei Wang, Paul V. Gratz, Daniel A. Jimenez, Elvira Teran, Seth Pugsley, and Jinchun Kim. 2022. The Championship Simulator: Architectural Simulation for Education and Competition. arXiv:2210.14324 [cs.AR] https://arxiv.org/abs/2210.14324

[5] Akanksha Jain and Calvin Lin. 2016. Back to the future: leveraging Belady's algorithm for improved cache replacement. *SIGARCH Comput. Archit. News* 44, 3 (2016), 78–89.

[6] Toru Koizumi, Toshiki Maekawa, Mananari Mizuno, Maru Kuroki, Tomoaki Tsumura, and Ryota Shioya. 2025. RUNLTS: Register-value-aware predictor Utilizing Nested Large TableS. In *The 6th Championship Branch Prediction*.

[7] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Carlos Escuin, Marjan Fariborz, Amin Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Anthony Gutierrez, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Miquel Moreto, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, William Wang, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. 2020. The gem5 Simulator: Version 20.0+. arXiv:2007.03152 [cs.AR] https://arxiv.org/abs/2007.03152

[8] 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. https://github.com/rsd-devel/rsd. In *2019 International Conference on Field-Programmable Technology (ICFPT)*. 63–71.

[9] Preeti Ranjan Panda. 2001. SystemC: a modeling platform supporting multiple design abstractions. In *Proceedings of the 14th International Symposium on Systems Synthesis* (Montréal, P.Q., Canada) *(ISSS '01)*. Association for Computing Machinery, New York, NY, USA, 75-80. https://doi.org/10.1145/500001.500018

[10] Wilson Snyder. 2004. Verilator and systemperl. https://github.com/verilator/verilator. In *North American SystemC Users' Group, Design Automation Conference*, Vol. 79. 122–148.

[11] Unity Technologies. 2022. ECS for Unity. https://unity.com/ecs.

[12] Catherine West. 2018. Using Rust for Game Development (and What You Can Learn From It). https://kyren.github.io/2018/09/14/rustconf-talk.html.