cmt2: Rule-Based Hardware Description in Rust with Temporal Semantics

Youwei Xiao
School of Integrated Circuits, Peking University
China

Zizhang Luo
School of Integrated Circuits, Peking University
China

Yun Liang
Peking University
China

ABSTRACT

cmt2 represents the second generation of Cement [8], featuring a redesigned Rust embedding and introducing new rule-based hardware description capabilities. This paper demonstrates how cmt2 seamlessly integrates Rust embedding, rule-based hardware design, and software-like procedural control logic description. Additionally, we present a novel temporal semantics extension for cmt2's rule-based features, enabling the composition of hybrid latency-sensitive and latency-insensitive hardware. This approach supports general, modular, and efficient hardware design at an appropriate level of abstraction.

1 INTRODUCTION

Hardware design methodologies encompass a wide spectrum, each providing distinct levels of abstraction and unique features tailored to specific design tasks. Selecting an appropriate methodology—one that aligns with the desired abstraction and characteristics—is critical. In this work, we focus on general, modular, and accelerator-rich design scenarios, such as modern System-on-Chip (SoC) designs, where diverse processors and accelerators must be implemented and hierarchically integrated into a cohesive system.

To address these challenges, we present cmt2, an evolution of Cement [8]. Building on Cement's strengths in Rust embedding and software-like procedural control logic description for single-accelerator design, cmt2 introduces rule-based hardware design to better support modular design practices. We have made the cmt2 version available at https://github.com/pku-liang/Cement.

However, existing rule-based hardware design methodologies often rely exclusively on latency-insensitive composition mechanisms. While versatile, this approach can introduce unnecessary overhead for modules with statically analyzable latency, which is prevalent in accelerator design. To address this limitation, we propose a novel temporal semantics extension for cmt2's rule-based features. This extension enables hybrid latency-sensitive and latency-insensitive hardware composition, minimizing hardware overhead while preserving modularity.

In this paper, we present how cmt2 integrates diverse hardware description features into a unified embedded HDL and highlight its robust backend support.

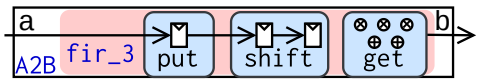
LATTE '25, March 30, 2025, Rotterdam, Netherlands


```
1  itfc_declare!{
2    param T;
3    pub struct A2B { a: input param T, b: output param T };
4    method put(a);
5    method get()->(b);
6  }
```

(a) module interface

```
1  #[module]
2  fn fir_3(t: &Type, a0: i32, a1: i32, a2: i32) -> A2B {
3    let io = io! {T: t};
4    let mut reg0 = instance!(reg(t));
5    // .. also instantiate reg1, reg2
6    let put = method!((io.a) { reg0 %= io.a; });
7    let shift = rule!(() { reg2 %= &reg1; reg1 %= &reg0; });
8    let get = method!(()->(io.b) {
9      ret!(&reg0*a0.lit(t) + &reg1*a1.lit(t) + &reg2*a2.lit(t))
10   });
11   schedule!(get, shift, put);
12 }
```

(b) module implementation



(c) illustration diagram

Figure 1: Example of FIR filter of order 3 in cmt2

2 CMT2'S DESIGN AND FEATURES

2.1 Rust Embedding

cmt2 is embedded in Rust using Rust's powerful procedural macro system, as opposed to modifying the rustc compiler with *plugins*, as done in HazardFlow [5]. The rationale behind the design choice is two-folded: Procedural macros (1) offer greater flexibility for custom syntax, and (2) provide a clear distinction between the embedded DSL and the host language. cmt2's macros which establish a well-defined boundary between hardware description regions and software code regions dedicated to parameterization and construction.

Figure 1 illustrates a 2-order FIR (Finite Impulse Response) filter implemented in cmt2. The design consists of two parts: the interface declaration (Figure 1a) and the module implementation (Figure 1b). The interface, defined within the itfc_declare! macro, specifies the module's type parameters, ports, and methods. In this example, the put method processes data from the input port a, while the get method returns data to the output port b. Figure 1b demonstrates a module implementation, where the #[module] macro transforms the function fir_3 into a module adhering to the A2B interface, as illustrated in Figure 1c. On line 3, the data type t is assigned to the parameter T of A2B. Notably, a single module interface can support multiple implementations; for instance, the A2B interface can be realized by FIR filters of varying orders. For further details, refer to the full example.

```
1 let start = method!(fsm; () {
2   for_!((
3     i %= 0.lit(&at);            // init: i=0
4     0.lit(&at).lt(n.lit(&at));  // init_cond: 0<n
5     i %= &i + 1.lit(&at);       // update: i'=i+1
6     i.lt((n-1).lit(&at))        // update_cond: i'<n-1
7   ) {/* j-loop and k-loop */ }) // body can read register i
8 });
```

Figure 2: Example of GEMM loops in cmt2

2.2 Rule-Based Hardware Design

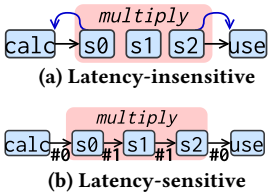
Lines 6-11 in Figure 1b describe the module logic using rules, methods, and a *schedule* specification. The methods `put` and `get` align with the interface and are analogous to methods in Bluespec SystemVerilog (BSV) and Haskell (BH). These methods execute only when invoked by a rule. The rule `shift` defines the shifting logic among registers. Similar to BSV/BH rules, `cmt2`'s rules attempt to execute proactively every clock cycle. Line 11 specifies the *schedule*, which defines the hardware concurrency strategy among rules and methods. This schedule determines the deterministic order in which the effects of rules and methods become observable within the same clock cycle, as proposed in Kôika [1].

Unlike Kôika which doesn't own a module system, `cmt2` extends the *schedule* specification to modular designs. While `cmt2`'s *schedule* specifications cannot describe the total order of all rules across multiple modules in a circuit, every pair of rules manipulating any shared states is inferred to have a deterministic order relation. Consequently, rule execution effects in `cmt2` can be serialized and deduced with cycle accuracy.

To our knowledge, `cmt2` is the first embedded HDL to support rule-based hardware design. Combining the advantages of hardware rules and rust embedding, `cmt2` becomes a good choice for hardware design tasks where different components interact closely with customization needs. Modern CPU design is one typical scenario of such characteristics. The core stages communicate intensively and mutate shared states such as the instruction queue (IQ) in an out-of-order CPU, where hardware rules robustly resolve composability challenges. Besides, CPU designs are highly parameterized with diverse instruction-set extensions implemented as hardware components to be composed, and embedding in Rust provides powerful parameterization support and allows for easy extension.

2.3 Procedural Control Logic Description

`cmt2` reimplements the event-based procedural control logic description from Cement [8] using rules. This reimplementation is straightforward, as both rules in `cmt2` and events in Cement fundamentally represent a group of hardware operations that always execute simultaneously. Figure 2 demonstrates the outermost loop description for a matrix multiplication operator in `cmt2`. For further details, refer to the full example. In `cmt2`, the `for_!` macro (line 4) is used to describe looping control logic. Additionally, `cmt2` supports primitives such as `par` and `if` for constructing procedural descriptions. The `start` method initiates the finite-state machine (FSM) synthesized from the procedural description, which is implemented as state registers and rules that update these states. With the procedural description support, `cmt2` provides great convenience for accelerator design tasks by saving efforts of designing FSMs adopted by data motion and computation phases of accelerators.



(a) Latency-insensitive

(b) Latency-sensitive

```
1  let calc = method! { (io.a, io.b) {
2    mult.s0(io.a, io.b)
3  }};
4  let use_ = method! {
5    [calc.delay_by(2)] // calc#2
6    () {let c=mult.s2(); /*use c*/}
7  };
8  let calc_use = method! {multicycle;
9   (io.a, io.b){ mult.s0(io.a, io.b);
10   let c = mult.s2(); /*use c*/ }};
```

Figure 3: Temporal semantics for rules

3 TEMPORAL SEMANTICS FOR RULES

Rule-based hardware design in `cmt2` naturally supports latency-insensitive composition, as rules are atomic and take effect only when their *guard* conditions are satisfied. This aligns directly with the principles of latency-insensitive specification. However, latency-sensitive composition often offers greater hardware efficiency but requires *temporal semantics* to statically analyze the latency of rule execution—a feature absent in prior rule-based HDLs. To address this, we extend `cmt2`'s rule abstraction with *inter-rule temporal relationships* and *multi-cycle rules*.

Figure 3 demonstrates `cmt2`'s temporal extension. Figure 3a illustrates latency-insensitive composition, where blue arrows represent auxiliary signals, such as `s0`'s *readiness* for `calc` and `s2`'s *validness* for `use`. In contrast, latency-sensitive composition in Figure 3b eliminates the need for auxiliary signals through static analysis. At line 5, the `delay_by` construct specifies a temporal-relationship-based guard condition for `use`, ensuring it can only execute two cycles after `calc` executes. The compiler verifies the legality of these temporal relationships and reports any analyzable violations.

Lines 8-10 presents a *multi-cycle rule* that achieves the same functionality as lines 1-7. The compiler synthesizes each multi-cycle rule into a set of rules, prioritizing latency-sensitive composition whenever possible. Our synthesis technique inherits basic problem formulation with data dependency constraints and timing constraints from high-level synthesis (HLS) scheduling [2]. However, instead of scheduling operations to completely static timing steps as HLS does, our synthesis schedules operations into single-cycle rules, and also inserts necessary buffers and control logic for *hybrid latency-sensitive/-insensitive composition*.

4 BACKENDS AND SIMULATION SUPPORT

We implement a set of backends for `cmt2`'s compiler, targeting FIRRTL [4], SystemVerilog [3], and simulation tools such as Verilator [7] and Khronos [9]. This rich backend support enables `cmt2` to be applicable for a wide range of scenarios and audiences. For instance, the FIRRTL backend allows `cmt2` to serve as a drop-in HDL choice for Chipyard-based SoC development, where generators like Rocket Chip [6] can be reused with `cmt2` at the FIRRTL level. For simulation, `cmt2` supports embedded-in-rust rule-based testbench specification, natively enabling high-level and parallel testbench construction with flexible parameterization.

5 FUTURE WORK

We plan to stabilize `cmt2`'s temporal semantics features and make them publicly available in the near future. Our long-term goal is to develop a `cmt2`-based SoC framework to construct heterogeneous modular hardware systems.

REFERENCES

[1] Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. 2020. The essence of Bluespec: a core language for rule-based hardware design. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 243–257. https://doi.org/10.1145/3385412.3385965

[2] J. Cong and Zhiru Zhang. 2006. An efficient and versatile scheduling algorithm based on SDC formulation. In *2006 43rd ACM/IEEE Design Automation Conference*. 433–438. https://doi.org/10.1145/1146909.1147025

[3] Design Automation Standards Committee. 2024. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2023 (Revision of IEEE Std 1800-2017)* (Feb. 2024), 1–1354. https://doi.org/10.1109/IEEESTD.2024.10458102 Conference Name: IEEE Std 1800-2023 (Revision of IEEE Std 1800-2017).

[4] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, and Jonathan Bachrach. 2017. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 209–216. https://doi.org/10.1109/ICCAD.2017.8203780 ISSN: 1558-2434.

[5] Minseong Jang, Jungin Rhee, Woojin Lee, Shuangshuang Zhao, and Jeehoon Kang. 2024. Modular Hardware Design of Pipelined Circuits with Hazards. *Proceedings of the ACM on Programming Languages* 8, PLDI (June 2024), 28–51. https://doi.org/10.1145/3656378

[6] Chipyard Team. 2025. Rocket Chip Generator. https://chipyard.readthedocs.io/en/latest/Generators/Rocket-Chip.html

[7] Veripool. 2024. Verilator. https://www.veripool.org/verilator/

[8] Youwei Xiao, Zizhang Luo, Kexing Zhou, and Yun Liang. 2024. Cement: Streamlining FPGA Hardware Design with Cycle-Deterministic eHDL and Synthesis. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '24)*. Association for Computing Machinery, New York, NY, USA, 211–222. https://doi.org/10.1145/3626202.3637561

[9] Kexing Zhou, Yun Liang, Yibo Lin, Runsheng Wang, and Ru Huang. 2023. Khronos: Fusing Memory Access for Improved Hardware RTL Simulation. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 180–193. https://doi.org/10.1145/3613424.3614301