Modeling and Verifying Hardware in Coq

Tz-Ching Yu
tz-ching.yu@epfl.ch

Jiawei Lin
jiawei.lin@epfl.ch

June 7, 2024

Abstract

Due to the nature of hardware design, the cost of bugs in hardware is enormous. Practically, hardware verification is done through a combination of writing down a series of tests and making sure these tests pass on the design and some formal verification techniques, such as model checking for sequential designs and symbolic execution for datapath. The former is useful for bug finding, but it by no means offers a strong correctness guarantee. On the other hand, model checking requires careful construction of the properties and formulae of the systems by the developers. For this project, we take an unconventional approach to hardware verification, focusing on the verification of a 3-stage pipelined CPU against a simple multi-cycle CPU specification. We specify both hardware implementation and specification with an existing framework, Fjfj, and prove that the implementation refines the specification, all done in Coq. The verification process involves defining a refinement mapping, proving the preservation of this mapping, and demonstrating the correctness of the implementation.

1 Introduction

A sequential circuit usually consists of two parts: the finite state machine, which represents the control of the system, and the datapath, which can be thought of as pure functions. In this project, we focus on the former part, as the datapath can usually be verified with symbolic execution without any issues. Therefore, we model the hardware systems as state transition systems, where state transitions are encoded as propositions in Coq. Specifically, there are two types of state transitions in our definition: action methods and rules. Action methods are part of the interface of hardware modules, while rules are internal state transitions that occur arbitrarily without external intervention. Besides action methods, we also define value methods, which serve as ways to get information from hardware modules without changing internal states.

In the project, we define and prove a 3-stage pipeline processor correct (perhaps with two-way superscalar). The major challenge involves the defining explicit invariants relating implementation and specification and overcoming the discrepancy of memory requests coming from the implementation and specification.

2 Correctness of Hardware Systems

In our verification framework, we start by defining two systems, implementation and specification. Then, one way to define the correctness of sequential systems would be to define it as trace inclusion which means which means that the set of all possible sequences of behaviors (traces) of the implementation are subset of the specification. This is usually proven through defining a refinement mapping which is a function that maps from a implementation state to a corresponding specification state with the same behavior. However, we follow a slightly different definition as defined in [1]. In this definition, the correctness is defined through state simulation. Given some relation Φ relating the implementation state and specification state, an implementation is correct if we can show refinement through state simulation witnessed by relation Φ and this Φ implies states are indistinguishable. Indistinguishable means that their behaviors are the same. To show state simulation over some Φ, we have to prove:

1. Initial states of implementation and specification are related, $\Phi s_i\ s_s$
2. Starting from related states

   for any rules in the implementation, if the implementation takes one rule transition and the specification takes zero or multiple rule transitions, the resulting implementation and specification states still related.

   for any action methods in the implementation, if the implementation takes one action method transition and the specification takes the same action method transition with the same arguments followed by zero or multiple rule transitions, the resulting implementation and specification states still eventually related.

As we can observe, this definition of correctness implies trace inclusion but the other direction does not necessarily hold.

2.1 Overall Proof Steps

The proof steps are quite straightforward. We start by defining the Φ relation, which sometimes can be non-trivial. Once we have done that, we can then prove a lemma stating Φ is preserved over lockstep. Lockstep defines the way the specification is simulated for each transition in the implementation. Then, refinement follows. However, it's important to note that not all Φ's are created equally. Some Φ's can be used to successfully prove refinement while some simply can't. As there's not much restriction on Φ, defining the Φ can be an iterative process where we incrementally add invariants to Φ until it's strong enough to prove the lemma.

3 Correct Micro-architectural Optimization

As mentioned in the previous section, the goal is to verify that a 3-stage pipeline CPU is a correct implementation with respect to a simple multi-cycle CPU. For the case of multi-cycle CPU, every instruction is executed one at a time, meaning there could be at most one

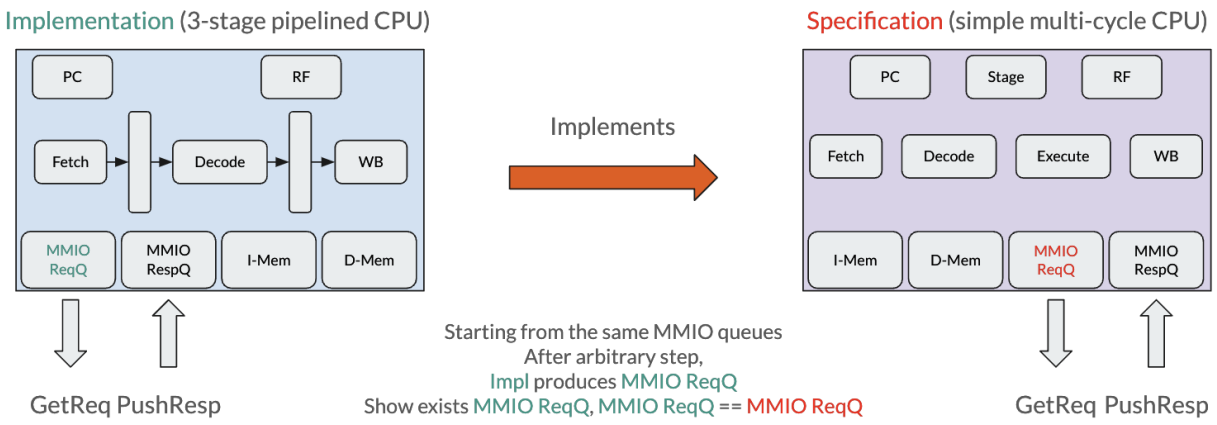

Figure 1: Implementation and specification systems

instruction in-flight, while, for the 3-stage pipeline CPU, there could potentially be at most 2 instructions in-flight simultaneously.

As shown in figure 1, these systems have one value method getting the MMIO requests from the CPU and one action method pushing the MMIO response from external devices. With our definition of correctness, it basically means the MMIO requests generated by the implementation can also be generated by the specification.

3.1 The Φ Relation

As discussed earlier, to prove refinement, we need to define a Φ relation. For this proof, we define the Φ such that the register files, the content of instruction memories, data memories, and MMIO request/response queues of the implementation and the specification are kept the same.

Besides these, we also have to relate the program counter (PC) and the stage register of the specification with the pipeline state of the implementation. There are four distinct cases.

- **Empty Pipeline:** The specification's PC matches the implementation's PC, and the stage is at Fetch.
- **One Inflight (Fetch to Decode):** The specification's PC matches the inflight instruction's PC, and the stage is at Fetch.
- **One Inflight (Decode to Writeback):** The specification's PC matches the older inflight instruction's PC, and the stage is at Writeback.
- **Two Instructions Inflight:** The specification's PC matches the older inflight instruction's PC, and the stage is at Writeback.

One important thing is that this Φ indeed implies indistinguishability between implementation and specification states, as the MMIO request queues are kept the same.

4 Conclusion

The complete definition and proof encompass approximately 800 lines of Coq code. The simplicity of the pipeline, with FIFO sizes set to one, allows for a straightforward proof. Allowing more than one entry in FIFOs would require additional components in the implementation and invariants concerning the components, such as scoreboard keeping track of the dependency of in-flight instructions and threading valid in-flight PCs. The Φ quickly becomes very hard to maintain and work with, and it motivates an implicit definition of Φ which can be derived semi-mechanically. The exploration of implicit Φ for these systems is left as future work for this project.

5 Related Work

[2] formulates the architectural design of an out-of-order processor with temporal logic which can then be verified with model checkers. [3] models processors as Term Rewriting Systems (TRS). In the work, the specification is defined as a TRS reflecting the operational semantics of the instruction set architecture, and the implementation is yet another TRS but it allows speculation and register renaming. One of the interesting insights of the work is that they relate implementation and specification when implementation is in so-called drained state. [4] is one of the works that define, implement, and verify hardware completely in Coq, where they prove the correctness of a multiprocessor and a coherent cache.

References

[1] T. Bourgeat, "Specification and verification of sequential machines in rule-based hardware languages."

[2] R. Jhala and K. L. McMillan, "Microarchitecture verification by compositional model checking," in *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*, ser. Lecture Notes in Computer Science, G. Berry, H. Comon, and A. Finkel, Eds., vol. 2102. Springer, 2001, pp. 396–410. [Online]. Available: https://doi.org/10.1007/3-540-44585-4_40

[3] Arvind and X. Shen, "Using term rewriting systems to design and verify processors," *IEEE Micro*, vol. 19, no. 3, pp. 36–46, 1999.

[4] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala, and Arvind, "Kami: a platform for high-level parametric hardware specification and its modular verification," *Proc. ACM Program. Lang.*, vol. 1, no. ICFP, aug 2017. [Online]. Available: https://doi.org/10.1145/3110268