Semi-Automated Translation of a Formal ISA Specification to Hardware

Harlan Kringen, Zachary Sisco, Jonathan Balkind, Timothy Sherwood, Ben Hardekopf
{kringen,zsisco,jbalkind,tpsherwood,hardekbc}@ucsb.edu
UC Santa Barbara, USA

ABSTRACT

Software compiler engineers target ISAs so that compiled machine code can run on a common platform and is correct and as efficient as possible. On the other hand, hardware engineers target ISAs so that their processors can support certain classes of software. This separation of duties is problematic because it deepens the difficulties of verifying correctness from compiled code down to its hardware. We see an opportunity to alleviate these concerns by automating this passage from ISA to hardware. The challenge is that verification and automation tooling for ISAs and hardware description languages (HDLs) have tended to treat implementation effects, such as state, exceptions, and non-determinism, in a variety of different ways, not to mention, these tools can be spread across the imperative/functional language spectrum. By concentrating on tools which originate in the functional programming space, we observe that a simple (unoptimized), functionally correct hardware implementation for a processor may be derived in a straightforward manner from the ISA's formal specification. We propose a proof of concept for translating formal ISA specifications to valid hardware implementations, e.g. in Verilog, in a single software pipeline.

1 OVERVIEW

Our proof of concept is rooted in functional representations of ISAs and synchronous, sequential hardware and leverages existing tooling in the formal design space. Specifically, we use the Sail language, an imperative DSL, to write ISAs [2, 3]. Sail allows formally specifying an ISA with support for conventional architectural patterns such as "fetch", "decode", and "execute". From this specification we use Sail's Coq backend to extract functional representations of the ISA methods. This Coq backend represents instructions which use register and memory effects as functions running in the state monad while all other instructions without effects are pure and can be lifted into the appropriate monad.

To obtain hardware that targets this ISA, we utilize the hardware description language CλasH [4], a functional DSL written in Haskell which compiles hardware descriptions into Verilog. CλasH code is based on a top-level abstraction of a Mealy machine. A Mealy machine describes a sequential circuit element which consists of some combinational logic as well as some state information which is looped back into the combinational logic. As noted in [11, 13], Mealy machines are instances of arrows. Arrows are similar to monads but are only capable of describing data flow graphs (functions tiled in parallel, sequence, feedback), lacking the level of control flow manipulation afforded by arbitrary monads [12]. The monad, arrow, and Mealy machine types may all be seen in Figure 1. Having canvassed Sail and CλasH, we visualize our proof of concept in Figure 2.

```
1   -- the monad typeclass
2   class Monad M where
3     (>>=) :: M A → (A → M B) → M B
4     return :: M → M A
5
6   -- the Arrow typeclass, suitable for describing data flow graphs
7   class Arrow A where
8     pure :: (B → C) → A B C
9     (>>>) :: A B C → A C D → A B D
10    first :: A B C → A (B × D) (C × D)
11
12  -- Arrows can be extended with a recursion operator; the D type captures
13  -- the state that is fed back while B and C are the input/output types
14  class Arrow A ⇒ ArrowLoop A
15    loop :: A (B × D) (C × D) → A B C
16
17  -- a simplified version the of Mealy machine used in CλasH
18  Mealy :: (S → I → (S × O) -- dynamics
19          → S               -- current state
20          → Stream I        -- input stream
21          → Stream O        -- output stream
```

Figure 1: The monad, arrow, and Mealy machine types form a foundation for functional programming approaches for hardware description and design.

Building a CPU in CλasH closely follows the same format as Sail methods in the ISA. The user defines types for machine word lengths, instructions, registers, as well as functions for fetching, decoding, and executing instructions. It is this similarity we are exploring with our single software pipeline. The trick is to convert a monadic representation of RISC-style ISA methods in Sail's Coq DSL to an arrowized (Mealy machine) representation of RISC-style components in CλasH. To this end, we observe two facts about the Sail ISA code.

First, the ISA instructions typically have register and memory responsibilities for architecture-level state. Sail describes "fetch", "read-register", "write-register", etc. methods as maps of the following form , where k is the machine word length and M is the state monad (or larger monad transformer stack):

$$(\mathit{MachineWord}\ k) \rightarrow M\ (\mathit{MachineWord}\ k),$$

Second, these methods have a common form which is often termed, "Kleisli maps". In fact, Kleisli maps are themselves instances of the arrow type. This fact has been explored extensively in the Yampa and Dunai languages used in functional reactive programming [15], in the form of "monadic stream functions" (MSFs), which are shown in Figure 3. MSFs effectively parameterize Mealy machines with a monad (or monad transformer stack) to utilize side effects. MSFs consume inputs, running their computations inside the given monad, and output a monadic value as well as a continuation for future processing. This creates a type which enjoys all of the composition capabilities of arrows, the stateful, infinite stream processing of Mealy machines, and the effectful nature of the monad.

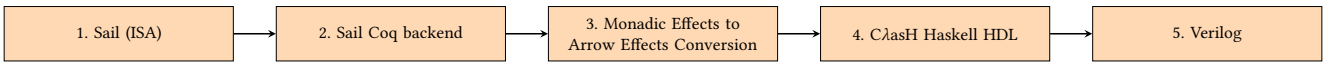

Figure 2: The software pipeline begins with an ISA written in Sail. We extract Sail's Coq representation, functional code embedded in the state monad. We convert the monadic version to an arrowized version, essentially a Mealy machine. This is then translated into the CλasH HDL, which is then automatically compiled to Verilog.

```
-- Monadic Stream Functions generalize arrows by allowing effects;
-- bear a close type-level resemblance to Mealy machines' dynamics
data MSF M A B ⇒ Monad M = MSF (A → M (B × MSF M A B))

-- instantiating types, we can describe an arrowized version
-- of Sail's 8-bit ``read register'' function
readReg      :: MachineWord 8 → State (MachineWord 8)
readRegArrow :: MSF State (MachineWord 8) (MachineWord 8)

-- a fragment of the CλasH code required to synthesize hardware designs;
-- our compiler pass takes the derived MSFs to the CλasH Mealy type, in
-- this case via the helper function, MSF2Mealy
topEntity
  :: Clock System
  -> Reset System
  -> Enable System
  -> Signal System (Signed 9, Signed 9)
  -> Signal System (Signed 9)
topEntity = exposeClockResetEnable (MSF2MealyMachine readRegArrow)
```

Figure 3: We can pass from monadic effects to monadic stream functions and directly target the CλasH DSL.

Based on this information, we can perform a compiler pass over the Sail state monad which takes it to its corresponding monadic stream function `State (Stream (Word N)) (Stream (Word M))`. Because MSFs are Arrows, and moreover, they are ArrowLoops, they can be translated into the Mealy machine type used in CλasH. The main "execute" method emitted in Sail is converted to CλasH's "topEntity" while all helper functions are translated into composable Mealy machines. We make several additions to incorporate CλasH's system of clocks, enable signals, and custom bitvector types, however, these are largely unproblematic. The translation results in CλasH code that looks nearly identical to what a single-cycle CPU would look like written directly in CλasH and is sketched in Figure 3.

This translation works straightforwardly for single-cycle, synchronous hardware. Many microarchitecures, however, can be multicycle and asynchronous and can have various optimizations that are invisible at the ISA level. For instance, the number of read and write ports, as well as their latencies on register files and memories, can be customized for hardware. This is, however, never represented at the ISA level. These are design choices we would like to explore unifying. An instruction at the ISA level *can* imply a hardware implementation consisting of registers, memory, caches, etc., and so it is important to determine what information can be helpfully included in the ISA which could form a tighter connection to an actual implementation.

2 PRELIMINARY WORK

The main contributions of this work are:

(1) The compiler pass that converts Sail's Coq backend into an arrowized Mealy machine;

```
(* Instruction encoding and decoding *)
mapping clause encdec = RTYPE(rs2, rs1, rd, RISCV_ADD)
  <-> 0b0000000 @ rs2 @ rs1 @ 0b000 @ rd @ 0b0110011
mapping clause encdec = RTYPE(rs2, rs1, rd, RISCV_XOR)
  <-> 0b0000000 @ rs2 @ rs1 @ 0b100 @ rd @ 0b0110011

(* Specifies execute behavior for ADD and XOR *)
function execute (RTYPE(rs2, rs1, rd, op)) = {
  let rs1_val = X(rs1); let rs2_val = X(rs2);
  let result : xlenbits = match op {
    RISCV_ADD => rs1_val + rs2_val,
    RISCV_XOR => rs1_val ^ rs2_val,
  };
  X(rd) = result
}
```

Figure 4: A subset of R-type instructions from the RISC-V ISA specified in Sail [3]. Architectural registers are accessed/updated from `X()`.

(2) The compiler pass that takes the arrowized Mealy machine into CλasH's particular DSL;
(3) Establishing formal criteria for correctness of the translations.

In terms of the first two contributions, we have converted Sail specifications to monadic code, extracted the Coq code into sensible Haskell code, annotated the extracted code into CλasH's DSL, and finally, confirmed that the CλasH code produces synthesizable Verilog. We have prototyped this for single instructions (e.g., the Sail code in Figure 4) and are working on full processor designs which we expect to be done shortly. We are also investigating expanding the processor designs to include multicycle and asynchronous behavior. This we expect to do through the use of deeper monad transformer stacks in our MSFs as well as augmenting our functions with timing information, similar to [8].

As for the third contribution, we have begun working on expressing correctness criteria using a predicate transformer semantics implementing pre- and post-conditions. We are roughly following past work such as [1, 14] and are using a refinement-like DSL in Coq, although we are also looking into [10] as a similar tool which focuses on correctness proofs of Bluespec-like designs. Our use of a refinement calculus allows us to prove that our translation from monadic to arrowized effects preserves the original, intended behavior. We are additionally interested in using a refinement-style framework to *derive* correct hardware descriptions as opposed to proving code correctness after the code has already been written. Finally, we are looking forward to connecting our work to current research that also explores formally verified ISAs and hardware description languages through the use of interactive theorem provers [5–7, 9].

REFERENCES

[1] João Alpuim and Wouter Swierstra. 2017. Embedding the refinement calculus in Coq. *Science of Computer Programming* 164 (05 2017). https://doi.org/10.1016/j.scico.2017.04.003

[2] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Shaked Flur, Kathryn E. Gray, Prashanth Mundkur, Robert M. Norton, Christopher Pulte, Alastair Reid, Peter Sewell, Ian Stark, and Mark Wassell. 2018. Detailed Models of Instruction Set Architectures: From Pseudocode to Formal Semantics. In *Proc. Automated Reasoning Workshop*. 23–24. Two-page abstract.

[3] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. 2019. ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS. In *Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages*. https://doi.org/10.1145/3290384 Proc. ACM Program. Lang. 3, POPL, Article 71.

[4] C. Baaij. 2009. CλasH : from Haskell to hardware. http://essay.utwente.nl/59482/

[5] Thomas Bourgeat, Ian Clester, Andres Erbsen, Samuel Gruetter, Pratap Singh, Andrew Wright, and Adam Chlipala. 2022. Flexible Instruction-Set Semantics via Type Classes. arXiv:2104.00762 [cs.LO]

[6] 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* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 243–257. https://doi.org/10.1145/3385412.3385965

[7] Thomas Braibant and Adam Chlipala. 2013. Formal Verification of Hardware Synthesis. In *Computer Aided Verification*, Natasha Sharygina and Helmut Veith (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 213–228.

[8] Manuel Bärenz and Ivan Perez. 2018. Rhine: FRP with type-level clocks. *ACM SIGPLAN Notices* 53 (09 2018), 145–157. https://doi.org/10.1145/3299711.3242757

[9] Adam Chlipala. 2017. Strong Formal Verification for RISC-V: From Instruction-Set Manual to RTL. https://riscv.org/wp-content/uploads/2017/12/Wed-1454-RISCV-AdamChlipala.pdf. Accessed: 2023–04-26.

[10] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. 2017. Kami: A Platform for High-Level Parametric Hardware Specification and Its Modular Verification. *Proc. ACM Program. Lang.* 1, ICFP, Article 24 (aug 2017), 30 pages. https://doi.org/10.1145/3110268

[11] João Paulo Pizani Flor, Wouter Swierstra, and Yorick Sijsling. 2018. Pi-Ware: Hardware Description and Verification in Agda. In *21st International Conference on Types for Proofs and Programs (TYPES 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 69)*, Tarmo Uustalu (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 9:1–9:27. https://doi.org/10.4230/LIPIcs.TYPES.2015.9

[12] Sam Lindley. 2014. Algebraic Effects and Effect Handlers for Idioms and Arrows. In *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming* (Gothenburg, Sweden) *(WGP '14)*. Association for Computing Machinery, New York, NY, USA, 47–58. https://doi.org/10.1145/2633628.2633636

[13] Hai Liu, Eric Cheng, and Paul Hudak. 2009. Causal Commutative Arrows and Their Optimization. *SIGPLAN Not.* 44, 9 (aug 2009), 35–46. https://doi.org/10.1145/1631687.1596559

[14] Thomas F. Melham. 1988. *Abstraction Mechanisms for Hardware Verification*. Springer US, Boston, MA, 267–291. https://doi.org/10.1007/978-1-4613-2007-4_9

[15] Ivan Perez, Manuel Bärenz, and Henrik Nilsson. 2016. Functional Reactive Programming, Refactored. *SIGPLAN Not.* 51, 12 (sep 2016), 33–44. https://doi.org/10.1145/3241625.2976010