Introducing Russet: Hardware Formal Verification with Mealy Machines

Molly MacLaren (LLNL, Carnegie Mellon University), Edwin Westbrook (LLNL), Matthew Sottile (LLNL)

Russet is a Haskell implementation of Mealy machines as a way of formally representing and reasoning about circuits. We introduce the Spud DSL to formally define circuit requirements and state transitions, and we demonstrate its effectiveness by verifying the main component of SPI.

INTRODUCTION

What is hardware verification? Why is it necessary?

- To validate hardware behavior, **simulation** with a testbench is common but not feasible to scale over every possible test case.
- **Formal verification** uses rigorous mathematical reasoning to show that a *design* meets a *specification* in every possible state.
- This approach is important for **critical systems**, where failure can have severe consequences.

What is Russet? How does it compare to other verifiers?

- Russet enables hardware engineers to define their spec as a **Mealy machine**, a type of **finite state machine** where state transitions depend on both the *current state* and *current input* and is commonly used in hardware design. [3]
- Russet applies **Constrained Horn clause (CHC)** solvers to encode circuit behavior and verification conditions to automatically compute constraint satisfaction.
- CHCs have found use across frameworks for *software verification*, but have not yet gained popularity in *hardware verification*. [2]

Research Question

Can we design a formal verification tool for practical use by hardware engineers and demonstrate that it is more effective than testing?

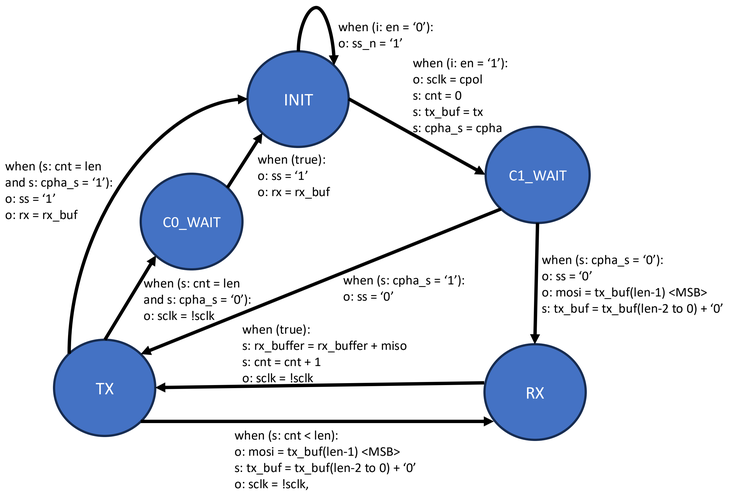

Figure 1. State machine for SPI Master with transition conditions and output. Variables are identified by (i)nput, (o)utput, and internal (s)ignals.

METHODS

Designing Russet

- Russet is written in **Haskell**, a purely functional programming language known for its strong type system, which helps to avoid mutation and runtime bugs common in imperative languages.
- Russet also defines a new language, **Spud**, to formally represent circuit specifications via finite state machines.
- Spud allows programmers to define states in a circuit, **guarded transitions** between those states, and **correctness properties** of the circuit outputs to be verified.

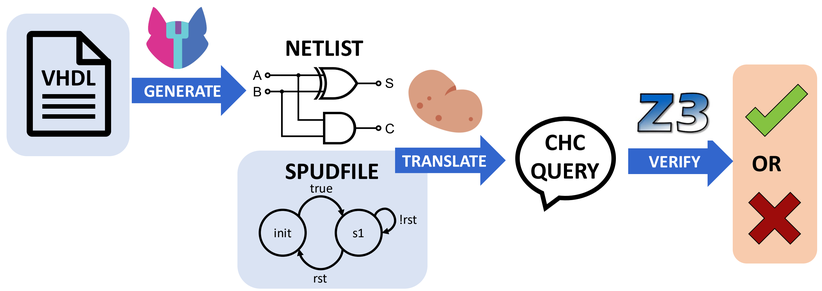

Figure 2. How Russet works with inputs in blue and outputs in orange. The hardware design is generated by Yosys, which is interpreted with the Spud specs and translated to a CHC query with Russet, and finally verified with the Z3 prover using Spacer.

Verifying Open-Source Hardware: the SPI Protocol

- **Serial Peripheral Interface (SPI)** is a design standard for synchronous serial communication between integrated circuits.
- We focus on the "master" component, which orchestrates communication with one or more peripherals, such as sensors.
- **Verification strategy:**
  1. Design a Mealy machine for the circuit based on intended behavior (see Fig 1).
  2. Incrementally construct the Spudfile, verifying the *least complex* signals first to ensure the number and order of states align with the implementation.
  3. Verify the *most complex* signals last: received and transmitted data. Start with 1-bit length and build up to longer messages.
  4. Demonstrate that Russet can also *find bugs* by manually inserting a fault into the circuit to show that Russet will generate a trace that exhibits that fault. **The bug**: if all clock input, transmitted, and received bits are 1, flip all received bits to 0.

RESULTS

Beyond Traditional Testing!

- Russet ensures SPI meets the Spud specs at every state possible.
- Russet can also generate a counterexample trace for a bug that at worst case, would require 1024 (2^{4*2+2}) test cases for 4 bit messages and 262,144 (2^{8*2+2}) test cases for 8 bits to have the same coverage!

Table 1. Time taken to compute constraint satisfaction on an M2 Mac with 32GB RAM

Message Length (bits)	1	2	3
Time to Verify	49m	3hr 8m	8hr 25min
Time to Counter (bug ver.)	1hr 23m	7hr 19m	13hr 5min

Future work

- **Optimizing** the CHC query, such as re-ordering correctness properties from most to least complex (e.g. by number of bits involved) can help speed up verification time.
- **Supporting more components** to be interpreted by Russet such as RAM and asynchronous flip-flops would increase the variety of circuits that can be verified.
- Implementing **proof modularity**, or the ability to use other proven circuits as sub-modules for larger proofs, would be useful to quickly verify multiple components of SPI together after verifying them independently.

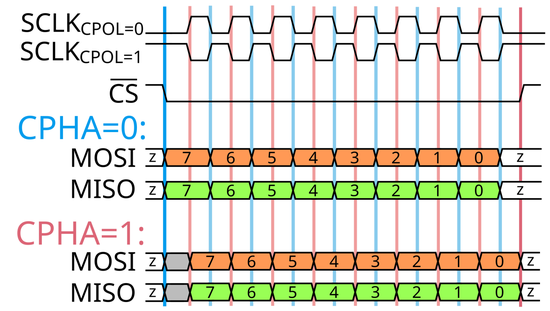

Figure 3. SPI timing diagram for both clock polarities and phases. Data bits output on blue lines if CPHA=0, or on red lines if CPHA=1, and sample on opposite-colored lines. [1]

[1] By User:Cburnett - File:SPI_timing_diagram.svg, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=134782980

[2] Gurfinkel, A. (2022, August). Program verification with constrained horn clauses. In International Conference on Computer Aided Verification (pp. 19-29). Cham: Springer Intl Publishing.

[3] Pedroni, V. A. (2013). Finite state machines in hardware: theory and design (with VHDL and SystemVerilog). MIT press.

Lawrence Livermore National Laboratory
LLNL-POST-2009382
Prepared by LLNL under Contract DE-AC52-07NA27344.
CASC
COMPUTING
National Nuclear Security Administration