Prost! Coroutine-based Hardware Description

Florian Riedl
florian.riedl@tugraz.at
Graz University of Technology
Graz, Austria

Tobias Scheipel
tobias.scheipel@tugraz.at
Graz University of Technology
Graz, Austria

Marcel Baunach
baunach@tugraz.at
Graz University of Technology
Graz, Austria

Abstract

Coroutines are a suitable base abstraction for modeling synchronous circuits in hardware description languages (HDLs). They allow entire circuit designs to be written in procedural code (similar to C-like languages), while retaining most of the low-level control that is often lost with high-level synthesis.

In this paper, we discuss how hardware can be modeled using coroutines and how coroutines can be synthesized into hardware.

1 Introduction

Most HDLs use a hierarchical approach to circuit design, where the circuit is constructed from a hierarchy of modules. The innermost modules are then either primitives, operators, or other language-inherent features. One common feature is the ability to use small procedural blocks, such as the `always` blocks in Verilog and processes in VHDL. These are generally restricted to operate within one clock cycle. Some modern languages[3, 4] expand on this idea and allow the insertion of pipeline stages or cycle delays. This enables the representation of multi-cycle algorithms.

In this paper, we propose a more fundamental expansion of this concept: The entire circuit design can be made procedural by using coroutines as the fundamental abstraction. Even the top-level entry point will be fully procedural, and module instantiations will be replaced with coroutine calls. This abstraction is equivalent to the cooperative multitasking found in SystemC[1] using `SC_THREAD` and `wait()`.

2 Modeling Hardware with Coroutines

We consider *stack-less*, *first-class*, *asymmetric* coroutines[2]: By *calling* a coroutine, we transfer control to it. At any point, the coroutine may *suspend* itself, returning control back to its caller while maintaining its state. The caller is then responsible for resuming or aborting the suspended coroutine. A coroutine can also return to the caller by terminating, in which case it may provide a return value.

Coroutines are equivalent to single-clocked synchronous circuits: Local variables correspond to registers, control flow maps to next-state logic, and variable updates describe the data path. We begin execution at the entry point on reset, and at each positive clock edge, we advance the coroutine until its next suspension point.

While this equivalence of coroutines and synchronous circuits is not perfect (see Section 5), it motivates the idea to use coroutines as the fundamental abstraction of hardware. Using coroutines enables procedural programming, which has a number of advantages:

Algorithms can be expressed directly: Coroutines enable imperative programming for directly expressing multi-cycle algorithms. Instead of manually converting a multi-step process into a finite-state machine (FSM) for implementation in Verilog or VHDL, the procedural algorithm can be translated directly.

Easy to follow control flow: In traditional HDLs, complex multi-step algorithms are typically implemented using large state machines. Especially when combining multiple FSMs into a larger hierarchy, following the control and data flow can be quite difficult. Coroutines, however, read like functions with structured control flow.

Software-like execution and debugging: A cycle-accurate simulation of a design implemented using coroutines can be performed by compiling it into a software program. This can leverage existing highly optimized compiler toolchains. Coroutines also allow source-level debugging (in addition to generating waveforms), as demonstrated by Rust's[1] *async/await* syntax.

3 Prost HDL

Prost is an HDL based on coroutines and used as a demonstrative example for a coroutine-based HDL. It is semantically similar to, e.g., SystemC, but avoids supporting non-synthesizable language constructs. Syntactically, Prost is inspired by Rust's `async/await` syntax.

In Listing 1, we show a code snippet implemented in Prost HDL. A coroutine definition mostly resembles that of a function: it receives arguments when called, and may provide a value once it returns. The main extension to typical procedural languages is the `wait` statement, which suspends the task's execution until the caller resumes it. While `.in` and `.out` suffixes are used to represent I/O signals; their current value can be accessed using the `.val` projection. Output signals can be modified using the `.next` projection, which will affect all `.val` accesses after the next `wait`. Contrary to the `wait` statement, where the caller waits until the next cycle, the `.wait` expression waits for a called task to complete.

In Prost, coroutines are first-class objects. Calling a task returns a value representing its current (initial) state. This state can be passed to an `advance` function associated with the task. This function then either returns the task's return value if it ran to completion, or a new state that can be further advanced on suspension. This interpretation of coroutines greatly simplifies synthesis: The values representing a task state can be mapped directly to registers, and the `advance` function can be compiled to combinational logic.

[1]https://rust-lang.org/

```
 1 task mac(rx: Bit.in, acc: Int.out) {
 2   acc.next = 0;
 3   loop {
 4     acc.next = acc.val * rx_byte(rx).wait;
 5     acc.next = acc.val + rx_byte(rx).wait;
 6   }
 7 }
 8
 9 task rx_byte(rx: Bit.in) -> [Bit; 8] {
10   let byte = [0; 8];
11   for i in 0..8 {
12     wait_n(250).wait;
13     byte(i) = rx.val;
14   }
15   byte
16 }
17
18 task wait_n(cycles: Int) {
19   while cycles > 0 {
20     cycles -= 1;
21     wait;
22   }
23 }
```

Listing 1: A code example written in the Prost HDL.

4 Synthesizing Coroutines

Listing 2 shows the next-state logic synthesized from the code in Listing 1, translated to Verilog. The compiler identified three states in the `mac` task: The `RESET` state corresponds to the initial state at the task entry. The other two states correspond to the single `wait` statement in the `wait_n` task, which is reachable via two separate calls in the `mac` task.

The `RESET` task will be entered on an external reset. On the next positive clock edge, it updates all registers according to the code between the task entry point and the first `wait` statement, before transitioning to the corresponding new state. Note that all control flow has been unrolled into nested `if-else`-blocks in Verilog: Just as software can support all control flow with conditional jumps, this means we can support any control flow construct. The `STATE_0` (Listing 2, line 19) task then continues execution from the `wait` statement (Listing 1, line 21). Depending on the current loop variables `i` and `cycles`, it either arrives at `STATE_0` or at `STATE_1`.

5 Future Work

One requirement of the Prost compiler is to reject code that cannot be synthesized. Determining statically whether a program reaches a certain state is undecidable in the general case, especially if the control flow depends on external inputs. Therefore, we envision the compiler using a heuristic instead, which must be well-defined and computationally efficient while rejecting as few valid programs as possible. Currently, the compiler requires each loop to contain at least one `wait` statement and to run for at least one iteration. Future work shall develop more precise heuristics and also formally verify their correctness.

```
 1 case (state_reg)
 2     RESET: begin
 3         acc = 0;
 4         byte = 0;
 5         i = 0;
 6         cycles = 250;
 7         cycles = cycles - 1;
 8         next_state = STATE_0;
 9     end
10     STATE_0:
11         if (cycles > 0) begin
12             cycles = cycles - 1;
13             next_state = STATE_0;
14         end
15         else begin
16             byte[i] = rx;
17             i = i + 1;
18             if i < 8 begin
19                 cycles = 250;
20                 cycles = cycles - 1;
21                 next_state = STATE_0;
22             end
23             else begin
24                 acc = acc * byte;
25                 byte = 0;
26                 i = 0;
27                 cycles = 250;
28                 cycles = cycles - 1;
29                 next_state = STATE_1;
30             end
31         end
32     STATE_1:
33         ...
34 endcase
```

Listing 2: The next-state logic in Verilog, translated from the code example in Listing 1.

Another design challenge comes from the handling of multi-clock circuits. While most designs are largely single-clocked, many real-world devices have multiple internal clocks. Existing languages like SystemC solve this by splitting the design into clock domains. In future extensions, Prost may support to explicitly `wait` on different clock signals.

Finally, by design, Prost tries to model synchronous circuits. While modeling pure combinational circuits via regular functions is a goal, combinational cycles are currently not expressible. If this capability is desired, e.g., for modeling latches, it needs to be reconsidered in the future.

6 Conclusion

Compared to the traditional hierarchical design methodology, Prost coroutines compose cleanly and reduce duplication in the FSM description. The function-like syntax of Prost lends itself well to implementing algorithms and should feel very familiar to software developers, especially if they use asynchronous programming.

References

[1] 2023. IEEE Standard for Standard SystemC® Language Reference Manual. *IEEE Std 1666-2023 (Revision of IEEE Std 1666-2011)* (2023), 1–618. doi:10.1109/IEEESTD.2023.10246125

[2] Ana Lúcia de Moura and Roberto Ierusalimschy. 2009. Revisiting coroutines. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 31, 2 (2009), 1–31.

[3] Frans Skarman and Oscar Gustafsson. 2023. Spade: An Expression-Based HDL With Pipelines. arXiv:2304.03079 [cs.AR] https://arxiv.org/abs/2304.03079

[4] sylefeb. 2025. Silice - A language for hardcoding algorithms with pipelines and parallelism into FPGA hardware. https://github.com/sylefeb/Silice Last accessed 10 January 2026.