Implementing Cache Coherence with Coroutines: A Case Study

Andrew David Alex
University of Washington
USA

Jingtao Xia
UC Santa Barbara
USA

Gus Henry Smith
Southmountain Research
USA

Rachit Nigam
MIT CSAIL
USA

Jonathan Balkind
UC Santa Barbara
USA

Gilbert Bernstein
University of Washington
USA

ABSTRACT

Hardware description languages must offer more productive abstractions without sacrificing low-level control. Coroutines are an expressive control construct that naturally describe the progression of a protocol as a sequence of steps separated by yield statements. In this report, we describe our experience modeling a hardware description language with coroutines to implement the MSI cache coherence protocol.

1 OVERVIEW

Hardware designs are full of interacting finite-state machines (FSMs), yet the languages we use to build hardware lack the abstractions to clearly and concisely express them. Many of these interactions can be characterized as a protocol, externally specified or not, between several actors that depend on specific responses from their counterparts elsewhere in the system. Typically, these FSMs are implemented as several seemingly independent blocks of processing logic under a case statement in a hardware description language (HDL). Many of these communication protocols are better expressed as a sequence of related steps rather than independent blocks. Describing protocols in this way allows more explicit state sharing between states and more clearly expresses the intent of the protocol. Coroutines naturally describe multi-step, concurrent computations in software, but have only been lightly explored for describing hardware [2]. We show how such a language is a natural choice for implementing the well-known Modified-Shared-Invalid (MSI) cache coherence protocol [1, 3].

2 DESIGN

Before illustrating our implementation, we will start with a high-level overview of the protocol and our assumptions. For the purposes of this example, we assume that each processor's private cache is connected via a shared bus. There are two FSMs involved in implementing the protocol: one handles requests from the processor and the other handles messages sent on the bus from other processors or the rest of the memory system. The processor request FSM may additionally need to request data from other caches or the rest of the memory system via its connection with the bus request FSM. Likewise, the bus request FSM needs to respond with the data

LATTE '26, March 23, 2026, Pittsburgh, PA, USA

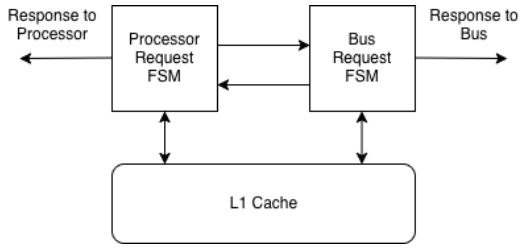

Figure 1: The FSMs involved in the protocol.

when it sees it come back on the bus. The L1 cache is assumed to have separate data and state memories, with the state being either (M)odified, (S)hared, or (I)nvalid. Both FSMs have read ports to the data and state memories in the cache, but only the processor FSM has a write port to data and only the bus FSM has write port to state. The design is illustrated in figure 1.

3 IMPLEMENTATION

To illustrate the features of our hardware description language with coroutines, we will walkthrough the processor request FSM implementation and highlight the interesting features along the way. We will first start with the "boilerplate" code that accompanies every FSM shown in figure 2. Every modeled FSM is a class with

```
class ProcReqFSM:
  def step(from_proc: ProcReq, from_bus: BusReq):
    while True:
      ...
      # processing logic to be continued
```

Figure 2: Boilerplate components around an FSM in the HDL

a method called `step`. The parameters of the method (`from_proc` and `from_bus`) are the inputs to the FSM and are passed on every call to `step`, though they may be unused. A call to `step` represents a single cycle of execution, which terminates with a `yield` statement. A parent module instantiating this component is assumed to call step every cycle. The body of `step` always contains a `while True` loop to model the non-terminating behavior of hardware, with the logic of the component as the body of the loop.

Figure 3 shows part of the body of the loop omitted previously. It describes the process for handling a read request from the processor, which is determined by the `isinstance` call on `from_proc`. When compiling to hardware, this corresponds to checking the tag of a

```
... # continued from figure 2
data, state = lookup(from_proc.addr)

# read request handler
if isinstance(from_proc, ProcReadReq):
  # cache does not have the data
  if state == INVALID:
    read_addr = from_proc.addr

    # wait for read request to go through
    _, from_bus = yield {ProcEmpty(),
                         BusRd(from_proc.addr)}
    while not isinstance(from_bus, Ack):
      _, from_bus = yield {ProcEmpty(),
                           BusRd(read_addr)}

    # wait for bus to respond with the data
    while not isinstance(from_bus, Flush):
      _, from_bus = yield {ProcEmpty(),
                           BusEmpty()}
    # got the flushed data, respond to processor
    from_proc, from_bus =
      yield {ProcResp(from_bus.data), BusEmpty()}

  # state == Modified or Shared => safe to return
  else:
    # cached read of valid data
    from_proc, from_bus =
      yield {ProcResp(data), BusEmpty()}
```

Figure 3: Read request processing logic in loop body

tagged union input. The protocol proceeds based on the state in the cache after lookup. If the state is invalid, then this FSM must request the the bus request handling FSM to request the data from another cache or the next level of the memory system on its behalf. Here, we encounter the first `yield` statement.

Yield Statements. Each `yield` takes as arguments the outputs for the module for the corresponding cycle. For this FSM, the the first output is the response back to the processor and the second is a request to forward along to the bus. The outputs with suffixes "Empty" correspond to "invalid" or "no data." Each `yield` not only outputs values, but is also a demarcation of time in the design. A `yield` marks the end of the current cycle and the code between each `yield` models combinational logic that executes in a single cycle. That is, the order of statements between `yield` statements has no effect on the semantics of the design. A valid design contains no control-flow loops that do not have a `yield` along the path. The left-hand side are the input values for the next cycle or execution (which are punned with the parameter names for the `step` method).

Maintaining State Across Yields. Many designs, including this one, require state beyond just tracking the control-flow of the FSM. In this example, the assignment to *read_addr* is first used after a yield in several subsequent `yield` statements waiting for the bus request FSM to acknowledge the request. Variables that remain live across yields therefore correspond to state elements in the language. The variable *read_addr* must be store in a register for later use. Note, a use in that cycle would actually be the value of *read_addr* from the previous cycle which is incorrect behavior. The first `yield` after the assignment outputting *from_proc.addr* is a result of this fact.

After the bus request FSM acknowledges the request, the protocol proceeds with the processor request FSM waiting for the flushed data from its counterpart. Finally, the received data is yielded back to the processor. The other side of the branch is obviously much simpler because it just returns the cached data immediately.

We elide the rest of the state handling as well as the bus request FSM code since it follows a similar structure. The final piece to show is the top module connecting the FSMs together in figure 4. It follows the familiar pattern except for one detail: the result of the call to `step` in each FSM is passed as an argument to the other FSM. This dependence on each other illustrates that this language is indeed expressing hardware and is not just a software model of the protocol as this would not be valid in a language like Python, for example.

```
class TopModule:
  def step(req_from_proc, req_from_bus):
    # instantiate sub-modules
    procFSM = ProcReqFSM()
    busFSM = BusReqFSM()
    while True:
      # step the busFSM
      bus_to_proc_fsm, resp_to_bus =
        busFSM.step(req_from_bus, proc_to_bus_fsm)

      # concurrently step the procFSM
      resp_to_proc, proc_to_bus_fsm =
        procFSM.step(req_from_proc, bus_to_proc_fsm)

      req_from_proc, req_from_bus =
        yield {resp_to_proc, resp_to_bus}
```

Figure 4: The top module instantiating the two FSMs

4 FUTURE DIRECTIONS

This case study only handles the case where the processor has at most one memory request in flight at a time. We can extend this example to handle multiple in-flight requests by wrapping the single request handling FSM in a separate scheduling component that picks the request handler to step next, while reusing the same underlying hardware. This higher-order modularity is a key benefit to an HDL with coroutines as similar functionality is much more complicated to express in traditional HDLs. Additionally, we plan to build a type system to verify the composition of several FSMs is safe and that there are no states where one FSM is expecting a message the other will not be sending, for example. This safety is obviously not guaranteed in traditional HDLs and would be a productivity boost for hardware designers.

REFERENCES

[1] James R. Goodman. 1983. Using cache memory to reduce processor-memory traffic. *SIGARCH Comput. Archit. News* 11, 3 (June 1983), 124–131. https://doi.org/10.1145/1067651.801647

[2] Marek Materzok. 2022. Generating circuits with generators. *Proc. ACM Program. Lang.* 6, ICFP, Article 92 (Aug. 2022), 28 pages. https://doi.org/10.1145/3549821

[3] Mark S. Papamarcos and Janak H. Patel. 1984. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture (ISCA '84)*. Association for Computing Machinery, New York, NY, USA, 348–354. https://doi.org/10.1145/800015.808204