From the Standards to Silicon: Formally Proved Memory Controllers [⋆] .

Felipe Lisboa Malaquias (✉)[1][0000−0002−0292−3437], Mihail Asavoae[2][0000−0001−5291−8567], and Florian Brandner[1][0000−0002−2493−7864]

[1] LTCI, Télécom Paris, Institut Polytechnique de Paris, France
{flisboa,florian.brandner}@telecom-paris.fr
[2] Université Paris Saclay, CEA List, France
mihail.asavoae@cea.fr

Abstract. Recent research in both academia and industry has successfully used deductive verification to design hardware and prove its correctness. While tools and languages to write formally proved hardware have been proposed, applications and use cases are often overlooked. In this work, we focus on *Dynamic Random Access Memories* (DRAM) controllers and the DRAM itself – which has its expected temporal and functional behaviours described in the standards written by the *Joint Electron Device Engineering Council* (JEDEC). Concretely, we associate an existing *Coq DRAM controller framework* – which can be used to write DRAM scheduling algorithms that comply with a variety of correctness criteria – to a back-end system that generates proved logically equivalent hardware. This makes it possible to simultaneously enjoy the trustworthiness provided by the Coq framework and use the generated synthesizable hardware in real systems. We validate the approach by using the generated code as a plug-in replacement in an existing DDR4 controller implementation, which includes a host interface (AXI), a physical layer (PHY) from Xilinx, and a model of a memory part *Micron MT40A1G8WE-075E:D*. We simulate and synthesise the full system.

Keywords: Coq · DRAM · Hardware Design · Code Generation

1 Introduction

The limitations of approaches such as model checking and satisfiability solving – widely adopted in industrial *hardware* (HW) verification – are well-known [6]: 1) Verification effort is focused on (relatively) small components of full systems; 2) Relatively weak properties are proved, with considerable abstraction gaps from the natural correctness criteria described in the specifications; and 3) The *state-space-explosion* problem. Conversely, proof assistants (i.e., deductive verifiers, or

[⋆] This research was supported by Labex DigiCosme (project ANR11LABEX0045DIGICOSME) operated by ANR as part of the program "Investissement d'Avenir" Idex ParisSaclay (ANR11IDEX000302) theorem provers) rely on the functional paradigm and on rich expressive high-order logic specification languages to describe systems at a high abstraction level – allowing programmers to implement systems and state correctness theorems naturally, without scalability constraints. The most significant caveats are arguably a higher entry bar, given the complexity of specification languages and proof scripts, and the lack of automation. Proof assistants, like Coq, also allow users to *extract* proved programs, which can then be compiled and executed.

In this work, we use Coq to propose a trustworthy design workflow for *Dynamic Random Access Memory* (DRAM) controllers – whose timing and functional correctness is essential for a variety of computing systems (e.g., critical real-time systems, our main focus, but also parallel and distributed systems). The correct behaviour of DRAM modules, and thus that of DRAM controllers, is described in standards [10] written by the *Joint Electron Device Engineering Council* (JEDEC). Furthermore, given that the standards use textual natural language (English) to describe correctness criteria along with timing diagrams, choosing Coq to model such systems is highly convenient – given the expressivity provided by its functional high-order logic specification language.

Precisely, we connect an existing Coq framework used to develop correct DRAM scheduling algorithms [13] – which in this work will be referred to as *CoqDRAM* for brevity – to a *back-end* that generates logically equivalent *Register Transfer Level* (RTL) representations in SystemVerilog. The back-end, which will be referred to as *CavaDRAM*, is developed in Cava,[1] a *Domain Specific Language* (DSL) written in Coq for designing and proving properties about circuits.

The connection between CoqDRAM and CavaDRAM plays a vital role in the design workflow proposed in this work – presented below as a list of steps:

1. Describe a DRAM scheduling algorithm in CoqDRAM and prove its correctness against the JEDEC standards;
2. Describe the controller circuit in Cava;
3. Prove bisimilarity between the two representations;
4. Extract a SystemVerilog circuit from the Cava controller (automatically), which can then be used as a plug-in replacement in existing hardware designs.

On the one hand, CoqDRAM – written in plain Coq – has been conceived to design, explore, model, and finally prove the correctness of DRAM arbitration **algorithms**, abstracting from actual HW implementations. It has little to no size constraints, a fact that allows users to use powerful abstractions to prove strong properties. On the other hand, CavaDRAM derives real memory controller HW implementations. This means that HW limitations become relevant, e.g., CoqDRAM uses queues that can grow to arbitrary sizes to store incoming requests, which is evidently not possible in a HW model. Therefore, a logical equivalence proof will require the queue to be limited in size. This duality is formalised through a series of assumptions – which are presented further – that allow us to limit the scope of CoqDRAM algorithms.

[1] https://github.com/project-oak/silveroak

In summary, this work proposes the following **contributions**:

– A design workflow to design correct-by-construction DRAM controllers, going from correctness criteria described in the JEDEC standards to RTL code;
– A proved proof-of-concept controller implementation, corresponding to one of the controllers originally proposed in CoqDRAM;
– A methodology to validate the generated RTL code in an existing design, and consequently, validate both the CoqDRAM and CavaDRAM models;
– An extension of the CoqDRAM to include REFRESH commands and its underlying correctness criteria.

The remainder of the paper is organised as follows: Section 2 gives a concise background on the two building blocks of our work: CoqDRAM and Cava; besides presenting some fundamental concepts about DRAM systems; Section 3 introduces our novel contributions with an architectural overview of the system and presents how memory controllers (and transition systems, more generally) that are logically equivalent to CoqDRAM implementations can be written in Cava; Section 4 elaborates on our use of the term "logical equivalence", which is de facto a *bisimilarity* relationship. Moreover, it presents the theorem and key insights of the proof procedure; Section 5 details the RTL generation phase, explains how the generated code is plugged into the existing DDR4 controller implementation (written in SystemVerilog), and presents the setup and the results regarding both simulation and synthesis; Section 6 reviews and compares the state-of-the-art with our work; and Section 7 concludes by revisiting our contributions and giving pointers for future research directions.

2 Background

2.1 DRAM Basics

DRAM controllers are responsible for servicing memory requests by issuing commands to the DRAM module (among other tasks, such as translating addresses into DRAM bank groups, banks, rows, and columns; and applying some scheduling algorithm to service requests). Moreover, each bank in a DRAM module has a row-sized buffer (the row-buffer) that serves as a "cache" for the bank, storing chunks of data that can be accessed with lower latency. Although several types of commands exist, the main ones used to directly service requests are: **ACT** (Activate), used to transfer one row of a bank into the row-buffer; **PRE** (Precharge), used to re-write the content of the row-buffer back into the matrix of memory cells; and **CAS** (Column Address Strobe), used to access one of the columns from the row-buffer (a **CAS** can be either a **RD** or a **WR**). Additionally, the controller has to issue **REF** (Refresh) commands periodically to restore the charge of cell capacitors.

2.2 CoqDRAM

CoqDRAM [13] models DRAM devices as command traces. Correctness criteria coming from the JEDEC standards [10] are modelled as *proof obligations* (POs) over traces and cover both functional and timing properties. Listing 1 shows such modelling: `Trace_t` is a record (much like structures in C), with constructor `mkTrace`. The trace itself is the member `Commands` (of type `Commands_t`). It is implemented as a standard Coq list type, where elements of the list are of type `Command_t`, the type of DRAM commands (not shown here).

```
Record Trace_t := mkTrace {
 Commands : Commands_t;
 Time     : nat;
 (* PO: Ensures that the time between an ACT and a CAS commands to the
     same bank respects T_RCD *)
 Cmds_T_RCD_ok : forall a b, a \in Commands → b \in Commands →
 isACT a → isCAS b → Same_Bank a b → Before a b
 → a.(CDate) + T_RCD <= b.(CDate); }
```

Listing 1. Command trace.

The record member `Time` is the trace length, and `Cmds_T_RCD_ok` is one of many POs: it states that for any two commands `a` and `b` members of the list, if `a` is an ACT command, `b` is a CAS command, they both target the same DRAM bank, `a` is issued before `b`, then the proposition `a.(CDate) + T_RCD <= b.(CDate)` must hold, where `T_RCD` is a constraint defined in the JEDEC standards. In other words, `T_RCD` is a lower bound between the issue dates of `a` and `b`. Besides JEDEC properties, essential characteristics of real-time systems are also modelled, such as non-starvation and controller semantics (e.g., memory consistency models).

Listing 2 shows CoqDRAM's definition of a memory controller. It is made of a function (`Arbitrate`) that, for a given request arrival model (`Arrival_function_t`), produces a trace of DRAM commands (`Trace_t`) of length defined by a `nat` parameter (the number of clock cycles). The class member `Requests_handled` is a proof obligation: it states that any request that has arrived will eventually have a corresponding CAS command in the trace (i.e., requests cannot starve).[2]

```
Class Controller_t {AF : Arrival_function_t} := mkController {
 Arbitrate :  nat → Trace_t;

 Requests_handled : forall ta req, req \in (Arrival_at ta)
   → exists tc, (CAS_of_req req tc) \in (Arbitrate tc).(Commands); }.
```

Listing 2. Memory controller definition.

```
Class Implementation_t := mkImplementation {
 (* Init takes a set of incoming requests and produces a state *)
 Init : Requests_t → State_t;
 (* Next takes a set of incoming requests, a state, and produces a new
     state, a command and the request currrently being serviced *)
 Next : Requests_t → State_t → State_t * (Command_kind_t * Request_t); }.
```

Listing 3. Implementation interface for memory controllers.

[2] CAS commands tell the memory to start the data transfer – its issue date is considered to be the completion date of the corresponding request.

Concretely, memory controllers are implemented as *transition systems* (TS). The user of CoqDRAM has to implement the type class `Implementation_t` (shown in Listing 3), made of functions `Init` and `Next`. The `Next` function, for instance, takes a set of arriving requests (`Requests_t`) at an arbitrary clock cycle, an arbiter state (`State_t`), and produces a new state and an output pair made of a DRAM command (`Command_kind_t`) and the request currently being serviced (`Request_t`). These functions together define a canvas, in some sense, for implementing controllers as transition systems.

Furthermore, it is typical of high-level abstraction models of memory controllers to ignore REF commands and its impact on timing. This is however not possible in a HW implementation. Hence, we extend CoqDRAM to model REF commands, including POs that guarantee timing and functional correctness.

2.3 Cava

Cava is a DSL written in Coq designed to specify, implement and prove circuits, greatly inspired by Lava [2]. It was developed by researchers at Google as part of the *Silver Oak project*, which focuses on the verification of high assurance components of the OpenTitan[3] silicon root of trust, i.e., a set of inherently trusted functions within a platform. Cava, much like other recent Coq DSLs for hardware design (e.g., *Kami* [6] and *Kôika* [4]), follows the highly automated proof and design style proposed by Adam Chlipala in his book *Certified Programming with Dependent Types* [5]. We choose Cava over other Coq DSLs for a few reasons: 1) Cava circuit simulations generate a list of values, where each element represents the value of a wire at a given clock cycle – this emulates *time*, a key element of command traces in CoqDRAM; 2) Cava is relatively simpler and faster to get acquainted to; and 3) Cava designs resemble classic RTL design style in some sense, whereas other DSLs take an approach closer to the rule-based design of Bluespec SystemVerilog [15].

Cava combines shallow and deep embedding techniques to describe combinational and sequential circuits, respectively. In other words, sequential circuits are implemented as inductive types (which includes a wrapper to combinational circuits). Thanks to its inductive nature, circuits can be interpreted in different ways [2], a feature that allows Cava users to prove correctness, simulate, and generate a netlist from a single circuit definition. Furthermore, sequential circuits in Cava are transition systems, much like controllers in CoqDRAM.

While we lack the space to formally present Cava's syntax and semantics, we will try to give the reader an intuitive understanding of how sequential circuits – and transition systems, more generally – can be designed with the DSL. Listing 4 is a sequential circuit definition in Cava. Circuit `Foo` takes an input `i` of type `inputType` and produces an outout `o` of type `outputType`. Possible signal types in Cava are: `Void`, an empty type; `Bit`, which is interpreted in Coq as a boolean; `Vec`, which takes another Cava signal type and the vector's size; and `ExternalType`, which is non-interpreted type. Moreover, circuit `Foo` has an internal state of

[3] https://opentitan.org/

type `stateType`, a register with initial value `s_init`. The loop body `f` is a combinational function with type (`stateType`,`inputType`)→ (`stateType`,`outputType`). In other words, although the internal state `s` (a given name) is not visible from outside `Foo`, the loop body `f` takes a pair of type (`stateType` ∗ `inputType`) as input and produces another pair of type (`stateType`,`outputType`). Lastly, Cava loops can be nested in order to form circuits that have multiple internal state signals.

```
Definition Foo : Circuit
 (i : inputType) (o : outputType) :=
 LoopInit (s_init : stateType) (
  Comb (f)
  (* f : (stateType * inputType) →
  (stateType,outputType) *)
 ).
```

Listing 4. Sequential circuit in Cava.

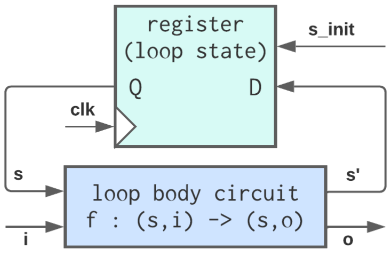

Fig. 1. Sequential circuit diagram.

3 Coupling CoqDRAM and CavaDRAM

Figure 2 illustrates the coupling between CoqDRAM and CavaDRAM. The design path starts at the JEDEC standards, modelled by `Trace_t` in CoqDRAM (c.f Listing 1). Specifically, the CoqDRAM specification covers the DDR3 and DDR4 JEDEC standards. Next, for scheduling algorithms implemented in CoqDRAM, we introduce *provably equivalent* controllers in CavaDRAM (equivalence is defined in Section 4). The RTL code produced from a controller implementation (using Cava's code extraction) can be used in existing designs. In our case, it is used as a plug-in replacement in an existing DDR4 controller implementation [20]. From a framework point of view, the additional workload introduced by the back-end coupling consists solely of writing the equivalent controllers in Cava and the equivalence proof with the representation in CoqDRAM.

[5] Figure 2 omits several components and does not represent a complete architecture, as its goals are to ease comprehension and provide an overview of the system.

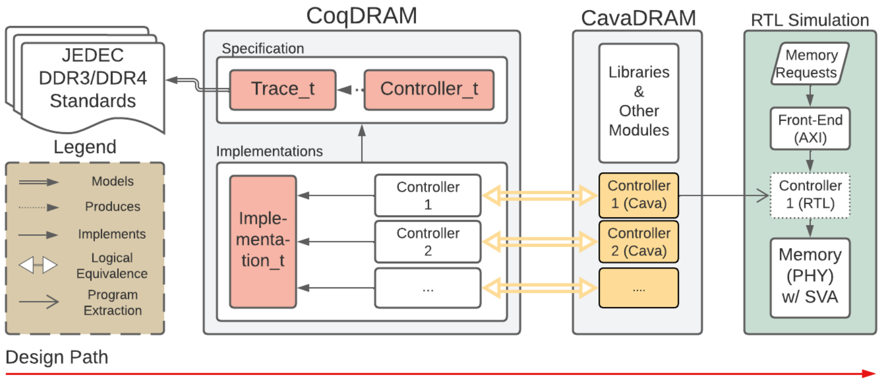

Fig. 2. System architecture.[5]
🟥 CoqDRAM classes, 🟨 Introduced workload, 🟩 DDR4 hardware simulation setup.

3.1 Controller Implementation Constraints

Hardware controller implementations impose constraints that are not captured by CoqDRAM. The setup we use for simulation and synthesis, for instance, is equipped with an AXI bus interface, which expects an interface to communicate with the memory controller. As a consequence, CavaDRAM controllers have to implement an interface containing the following input and output signals: a) the arrival of a new request is signalled by a 1-bit `pending_i` input signal; b) a *single* `request_i` is provided as a bit vector input; and c) the circuit has to produce a 1-bit `ack_o` signal as output. The `pending_i`/`ack_o` signals allow to perform a handshake (used in many bus implementations besides AXI).

This interface is less *generic* than the CoqDRAM implementation. An equivalence proof thus can only succeed by introducing two additional assumptions that constrain the arrival model of CoqDRAM:

Assumption 1. The arrival function needs to be constrained to a *single incoming request per cycle*. In Coq, we model this constraint with a PO that limits the number of requests in the incoming arrival list; the PO is denoted by `HW_single` in Listing 5. For one, this models the limitation of the AXI bus mentioned in the interface above. In addition, this reflects a fundamental limitation on memories, which are typically used to implement queues in hardware: the implementation cost of memories increases drastically with the number of read/write ports. In our case (Distributed/BRAM of FPGAs), it is limited to a single read/write port each. Bear in mind that the proofs in CoqDRAM are valid for all possible arrival functions without any limitations, including `HW_Arrival_function`.

```
Class HW_Arrival_function_t {AF : Arrival_function_t} := mkHW_AF {
 HW_single : forall t, size (Arrival_at t) <= 1; (* Assumption 1 *)

 (* Assumption 2 consisting of two POs *)
 pending_i : nat → signal Bit; (* pending input for controller circuit *)
 request_i : nat → signal request_t; (* request as input *)
 HW_arrived : forall t, size (Arrival_at t) = 1 ↔ (ack_o t) ∧ (pending_i t);
 HW_request : forall t, size (Arrival_at t) = 1 →
       EqReq (Arrival_at t) (request_i t);
}.
```

Listing 5. Assumption reflecting hardware-level implementation constraints.

Assumption 2. Note that Assumption 1 does not constrain the number of outstanding requests from the requestors, it is just a constraint on the bus interfacing with requestors and memories of queues. The interface described before also comprises a handshake protocol, which allows a controller to accept (or not) newly in-coming requests. CoqDRAM only considers requests that are *accepted* by the controller, i.e., from the moment that the request is processed by the controller. Consequently, arrival functions have to be constrained to take the handshake protocol into account: a request arrives when the request is provided on the `request` input port and both the `pending` and `ack_o` signals are asserted. The two POs `HW_arrived` and `HW_request` from Listing 5 establish this relation.

3.2 From CoqDRAM to CavaDRAM Implementation

As a Proof-of-Concept, we implement a controller based on the *First-In-First-Out* (FIFO) scheduling policy, as originally proposed in CoqDRAM. The controller serves an arbitrary number of requestors. Requests are served in arrival order, without distinction between requestors. Each request is processed in a *slot* large enough to fit every necessary DRAM command while respecting all timing constraints. The controller issues DRAM commands following a *closed-page policy*, i.e., it always issues the same sequence of commands: PRE-ACT-CAS.

Representing States. Listing 6 shows CoqDRAM's FIFO state definition, an inductive type with three possible values: `IDLE`, `RUNNING`, and `REFRESHING`. Additionally, each value carries a series of arguments that extends the set of states: `Cnt_t` is a counter used to count clock cycles within a FIFO slot; `Cnt_ref_t` is another counter used to manage memory REFRESH operations, i.e., keep track of clock cycles until a REF command is needed; and `Reqs_t` is the infinite sequence of requests in the queue, i.e., waiting to be serviced. Note also that the `RUNNING` state carries an additional value of type `Req_t`, used to remember the request currently being processed by the controller. The counters are implemented in Coq as bounded integers (which carry a proof stating that the counter value is always strictly smaller than its bound) and the queue is a standard Coq list (of arbitrary size). `Reqs_t` being an unbounded list is obviously not possible in a HW model. Assumptions 1 and 2 (presented in Section 3.1) mitigate this problem, as they make it impossible for `Reqs_t` to grow arbitrarily. In connection with Listing 3, `FIFO_state_t` is a valid instance of `State_t`.

```
Inductive FIFO_state_t :=
| IDLE    : Cnt_t → Cnt_ref_t → Reqs_t → FIFO_state_t
| RUNNING : Cnt_t → Cnt_ref_t → Reqs_t → Req_t → FIFO_state_t
| REFRESHING : Cnt_ref_t → Reqs_t → FIFO_state_t.
```

Listing 6. CoqDRAM FIFO state.

Cava Implementation. A simplified version of the FIFO CavaDRAM implementation can be seen in Listing 7.[6] Figure 3 is a diagram of the resulting circuit.[7] A few points are worth emphasising:

```
Definition FIFO : Circuit (pending ∗ request) (ack ∗ request ∗ command) :=
 Loop (Loop (Loop ( (* one loop for each resigster *)
  ReadLogic >==>Queue >==>NextCR >==>CmdGen >==>Update)))
```

Listing 7. Simplified version of the FIFO implementation in CavaDRAM.

1. We use nested `Loop` constructors to manipulate multiple internal state signals, allowing us to mimic CoqDRAM states. A correspondence can be established between elements of Listing 7/Figure 3 and CoqDRAM's FIFO state

[6] In the listing, the notation >==> stands for circuit composition.
[7] Initial register values are omitted from the figure.

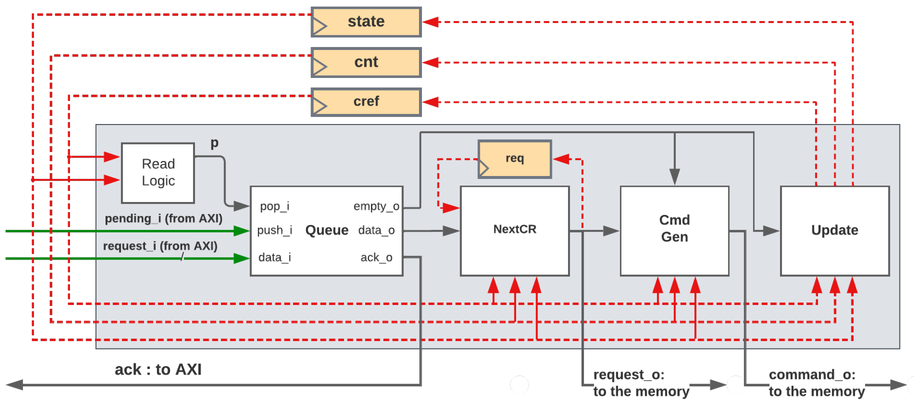

Fig. 3. FIFO circuit implementation in Cava.

definition (Listing 6): the register `state` corresponds to the state identifier (`IDLE` /`RUNNING`/`REFRESHING`); register `cnt` corresponds to the counter `Cnt_t`, and has as many bits as necessary to count up to the counter's bound; similarly, register `cref` corresponds to `Cnt_ref_t`. The request queue in CoqDRAM (`Reqs_t`) corresponds to `Queue`. Lastly, `Req_t` in `RUNNING` is a register of circuit `NextCR`.

2. `Queue` is implemented as a dual-ported memory (one port for reads and one for writes, recall Assumption 1), with additional combinational logic to determine whether the queue is full or empty using internal write and read pointers.

3. The signals `pending_i` and `request_i` (highlighted in in green on the left hand side of Figure 3) are the circuit's inputs: both signals are fed directly to the queue. The `pop_i` signal, however, comes from the `Read Logic` module, which is based on the register values and on the state of the queue itself, will produce a read enable signal. The queue produces three outputs: `empty_o`, used in the following modules; `data_o`, the request at the head of the queue; and `ack_o`, a signal used to complete the handshake with the bus arbiter and possibly stall the arrival of incoming requests (recall Assumption 2).

4. Besides `ack_o`, the circuit produces two output signals: a) `request_o` is the request currently being serviced by the controller and is eventually fed to the DRAM address bus, it contains the target bank, row, and column; and b) `command_o`, generated by `CmdGen`. These output signals are similar to those produced by CoqDRAM algorithms (see Listing 3). They are generated by circuits `NextCR` and `CmdGen`, respectively, and take as input the register values, i.e., the current state, as well as the signals `empty_o` and `data_o` from `Queue`.

5. `Update` implements the transition function: it is made of three separate combinational functions that calculate the next value of each register and thus determine the next state.

4 The Equivalence Proof

We used the word *equivalence* between transition systems throughout the paper. Here, we define equivalence, which is de facto a *bisimilarity* relation. Bisimilarity was introduced (formulated by Park [16], refining ideas from Milner [14]) as the notion of behavioural equality for processes [17].

Definition 1 (Bisimulation and Bisimilarity [18]). Given an LTS $(S, \Lambda, \rightarrow)$, where S is a set of states, Λ is a set of labels, and $\rightarrow \subseteq (S \times \Lambda \times S)$ is a transition relation, written $P \overset{\mu}{\rightarrow} Q$ for $\langle P, \mu, Q\rangle \in \rightarrow$. A binary relation $\mathcal{R}$ on the states of the LTS is a bisimulation if whenever $P \mathcal{R} Q$:

1. for all P', with $P \overset{\mu}{\rightarrow} P'$, there is Q' such that $Q \rightarrow Q'$ and $P' \mathcal{R} Q'$;
2. the converse, on the transitions emanating from Q: for all Q', with $Q \rightarrow Q'$, there is P' such that $P \rightarrow P'$ and $P' \mathcal{R} Q'$;

Bisimilarity, written $\sim$, is the union of all bisimulations; thus $P \sim Q$ holds if there is a bisimulation $\mathcal{R}$ with $P \mathcal{R} Q$.

Remark [18]. Note that although bisimulation and bisimilarity are defined on a single LTS, it is also a valid definition for distinct LTS with the same alphabet of actions; as the union of two LTSs is again an LTS.

Intuitively, two bisimilar systems match each other's moves, i.e., if we assume that two agents were playing a game according to some rules, the agents could not be distinguished from the other by an observer.

In Coq, we start by giving meaning to the $\mathcal{R}$ binary relation of Definition 1. Listing 8 shows the definition of `State_Eq`, a predicate that defines the equality between CoqDRAM and CavaDRAM states for the FIFO controller. In the listing, `fs` is the state coming from CoqDRAM (defined as shown in Listing 6), and `cs` is the Cava FIFO state (of type `State_t`). The identifiers with prefix `cs_` come from `get_` functions applied to `cs` (as in Line 2). These functions retrieve individual signals/registers from `cs`, which contains every other internal signal.

```
1 Definition State_Eq (fs : FIFO_state_t) (cs : State_t) : bool :=
2 let cs_state := get_state cs in ... (* every cs_ variable comes from cs *)
3 match fs with
4 | IDLE cnt cref P ⇒ (cs_state =? STATE_IDLE_VEC) && (cs_cnt =? cnt2Bv cnt)
5    && (cs_cref =? cref2Bv cref) && (EqMem P cs_mem) && (EqQueue P wra rda)
6 | RUNNING cnt cref P r ⇒ ... (* similar *)
7 | REFRESHING cref P ⇒ ... (* similar *)
8 end.
```

Listing 8. Predicate for state equality.

Note the pattern matching on `fs` in Line 3. If, for instance, the `fs` state is an `IDLE` state, has a `Cnt_t` denoted by `cnt`, `Cref_t` denoted by `cref`, and `Reqs_t` denoted by `P`, then, the predicate should evaluate to `true` only if a series of conjunctions are satisfied: `cs_state` – the register carrying the information if the circuit is either `IDLE`, `RUNNING`, or `REFRESHING` – has to evaluate to `STATE_IDLE_VEC`, a literal representing the idle state; `cs_cnt` has to be numerically equivalent to `cnt`, `cs_cref` has to be numerically equivalent to `cref`, et cetera.

The predicates `EqMem` and `EqQueue` are recursive functions that establish a connection between both representations of the request queue. A simplified definition of `EqMem` is shown in Listing 9. The predicate states that each element in `CoqDRAM_Q` has a logical correspondence in `CavaDRAM_Q`. The correspondence is defined by the predicate `EqReq`, which takes as arguments a `CoqDRAM` request and a `CavaDRAM` request and outputs `true` if they are equal. Note that `x`, the element at the head of `CoqDRAM_Q`, maps to `nth rda CavaDRAM_Q`, the element of `CavaDRAM_Q` at index `rda`, the read address, where `nth` is a function used to access indexed list elements. The following element maps to index `rda` + 1, and so on. We omit the definition of `EqQueue` and `nth` for brevity.

```
Fixpoint EqMem_ {W} CoqDRAM_Q CavaDRAM_Q (rda : Bvector W) :=
 match CoqDRAM_Q with
 | [::] ⇒ true
 | x :: x0 ⇒ (EqReq x (nth rda CavaDRAM_Q)) && (EqMem_ x0 (rda + 1) CavaDRAM_Q)
 end.
```

Listing 9. Equality of memories/request queues.

Listing 10 shows the Coq version of Definition 1 applied to our problem. Consider an arbitrary Cava FIFO state (`c_state`) and an arbitrary CoqDRAM FIFO state (`f_state`) – obtained through `t` calls to `HW_Default_arbitrate`. The reason for having `t` – the number of clock cycles (i.e., the trace length) – explicit is to access the incoming request at time `t`, denoted by `R` := `Arrival_at t` and `c_req` := `request_i` for CoqDRAM and CavaDRAM respectively.

```
Theorem TS_Bisimulation : forall c_state (t : nat),
 let f_state := (HW_Default_arbitrate t).(Implementation_State) in
 let R := Arrival_at t in let c_req := request_i t in
 State_Eq f_state c_state →
 let f_nextstate := fst (Next_state R f_state) in
 exists c_nextstate,
    (c_nextstate = fst (step FIFOSM c_state (pending_i t,c_req))) ∧
    State_Eq f_nextstate c_nextstate.
Proof. ...  (* Proof script not shown *) Qed.
```

Listing 10. The bisumulation theorem.

The rest of the theorem is equivalent to clause (1) of Definition 1, it reads: if whenever `State_Eq f_state c_state`, then, for all derivative states of `f_state`, denoted by `f_nextstate`, there exists a derivative state of `c_state`, denoted by `c_nextstate`, such that `State_Eq f_nextstate c_nextstate` holds. Note that the universal quantifier of Definition 1 is encoded into `f_nextstate` itself. Furthermore, `f_nextstate` and `c_nextstate` are respectively obtained through calls to `Next_state` – an implementation of `Next` (c.f. Listing 3) – and `step`, a Cava function that produces a new circuit state given the previous state and inputs. The converse theorem, corresponding to clause (2) of Definition 1, i.e., a challenge of `c_state` (with universally quantified transitions) against `f_state`, is also true (we omit the converse lemma for brevity). We recall that from Definition 1, proving that a bisimulation exists proves that the *two transition systems are bisimilar*.

The main strategy used to drive the proof is case analysis on CoqDRAM state definitions (such as the one in Listing 6). Moreover, the `step` function in Listing 10 is unfolded and applied to every sub-circuit in a composite circuit. As a consequence, the proof of `TS_Bisimulation` is structured with lemmas stating the equivalence of individual circuits, such as in Listing 11. It states that the `step` function applied to the `CmdGen` circuit with an input containing a `cref` value equal to `CNT_REF_PREA`, a literal containing the counter value that dictates when the controller should issue a PREA command,[8] will indeed output a `PREA`.

```
Lemma CmdGen_equiv_idle_to_ref (c : circuit_state CmdGen) cnt cref:
(cref =? CNT_REF_PREA) = true → (* when cref reaches CNT_REF_PREA *)
snd (step CmdGen c (STATE_IDLE_VEC,true,cnt,cref,REQUEST_NIL)) = PREA_VEC.
Proof. ...  (* Proof script not shown *) Qed.
```

Listing 11. A lemma part of the proof tree: Equivalence of CmdGen.

We use Coq's `Ltac` language extensively in order to build automated proof procedures and thus facilitate the proofs for future implementations. The methodology for writing and proving different memory controller implementations is similar and/or follow a very specific pattern; hence, the implementation provided here, although simple, can be effectively used as a template.

Lastly, for the FIFO controller, the proof of Theorem `TS_Bisimulation` has a total of 3117 *lines of code*, and checking the proof takes an average of 22.54 minutes on a system with the following specifications: Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz, 8GB RAM, and Ubuntu 22.04. The memory usage during proof checking peaks at 5.83GB, measured with `time`. The high computational load required to check proofs about Cava circuits comes from design choices of Cava itself – Cava circuit states are large tuples that result in lengthy terms in the goals, therefore reducing the performance of reduction and rewriting tactics.

5 Simulation & Synthesis

With the goal validating our methodology in a "real world" setup, we start by (certifiably) extracting the CavaDRAM code to Haskell, using standard Coq extraction. Then, an additional Haskell script, part of the Cava tool-chain, analyses the circuit AST to generate a gate-level netlist in SystemVerilog. Next, as a host "hardware environment" for the generated code, we choose a DDR4 controller implementation for Transprecision Computing [20], which is publicly available[9] and will be referred to as *DDR4cntrl* for brevity.

Figure 4 illustrates the *DDR4cntrl* architecture, with our modifications highlighted. In summary, we replace a module called `rank machine` with the CavaDRAM

[8] PREA commands are PRE commands sent to every bank at once.
[9] https://github.com/oprecomp/DDR4_controller

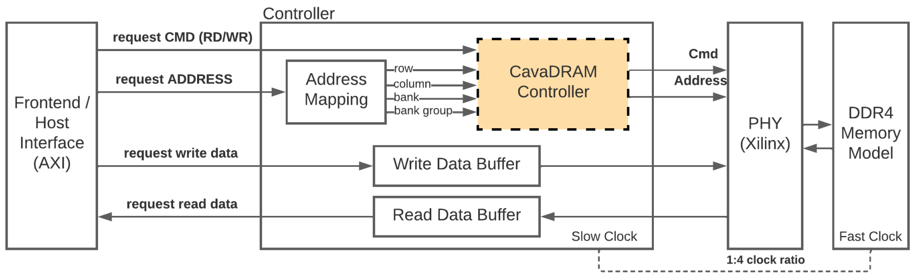

Fig. 4. An illustration of *DDR4cntrl* [20] with our modifications highlighted.

controller. Originally, `rank machine` was responsible for scheduling requests, generating and issuing DRAM commands, as well as managing REF commands; the same tasks performed by the CavaDRAM controller.

Every other functionality is kept unchanged: the AXI logic and its interface to the controller, the logic to control the read and write data buffers, the PHY, and the memory model. The PHY, a Xilinx IP in this case, generates the signal timing and sequencing required to interface to the memory device, e.g. phase alignment between DQ and DQs signals, logic for initialising the DRAM after power-up, and conversion of slow clock to fast clock.[10]

Simulation. We validate the approach with the two testbenches provided in *DDR4cntrl*. The simulation is filled with SVA, which trigger if timing constraints are not respected, invalid commands are issued, or transactions do not complete. The simulation goes through with no assertions triggered.

We emphasise that achieving good performance is not the goal of this work, but rather present the methodology and provide a proof-of-concept. Considering that, although RTL code generated from Cava performs as well as standard SystemVerilog designs,[11] both simulation time and controller bandwidth worsen, for two straightforward reasons: 1) The CavaDRAM controller was not designed to exploit the ratio between the different clock domains of the system and the memory; therefore, it only operates at 1/4 of the available bandwidth in *DDR4cntrl*. 2) The FIFO algorithm does not offer competitive bandwidth compared to state-of-the-art memory controllers.

Synthesis. Using a testbench, which is part of the *DDR4cntrl* source code, we synthesise the FIFO CavaDRAM controller. A comparison of synthesis util-

[10] Inasmuch as the PHY runs at the system clock frequency (1/4 of the DRAM clock frequency), it expects four command/address per system clock and issues them serially on consecutive DRAM clock cycles on the DRAM bus. This means that the PHY interface provides four command slots: 0,1,2, and 3, which it accepts each system clock. To cope with the different clock domains, we insert CavaDRAM commands always in the first slot. The proofs in CoqDRAM do not lose validity, as lower-bounds still hold. The only proofs that need adapting are REF related proofs, as they are upper bounds on the spacing between REF commands. We write modified version of such constraints considering the different clock domains.

[11] https://silm-seminar.gitlabpages.inria.fr/season2/episode5/singh.pdf

isation metrics between the original *DDR4cntrl* and the modified *CavaDRAM* version can be seen in Table 1. The results were obtained with a Xilinx Virtex UltraScale 095FFVB2104-2 board, considering a request queue in CavaDRAM that can store up to 256 requests.

Table 1. Key metrics from the synthesis report showing the resource utilisation on a Xilinx Virtex UltraScale 095FFVB2104-2.

		LUTs	Flip-Flops	bram_fifo_52x4
DDR4cntrl	`rank_machine`	8195	4644	16
	Full Design	9263	6129	16
CavaDRAM	`CavaDRAM`	5312	8605	0
	Full Design	6344	9832	0

Unsurprisingly, the CavaDRAM version uses fewer LUTs, since the FIFO scheduling logic is simpler than what had been originally proposed in *DDR4cntrl*. The total number of Flip-Flops (FFs) in the design augments, since the Vivado synthesizer chooses to represent Cava queues with FFs rather than native FPGA FIFOs (accessible through the *bram_fifo_52x4* macro). Note also that *DDR4cntrl* took 88,46% and 75,77% of the total amount of LUTs and FFs in the design, respectively. For *CavaDRAM*, these percentages are 83,73% and 87,52%, respectively. These results show that the generated code introduces only a negligible imbalance w.r.t resource utilisation, compared to the replaced module.

6 Related Work

DRAM & Formal Methods. The idea of applying formal methods to verify that DRAM controllers comply to the standards was first introduced by Datta et al. [7]. The authors perform a manual translation of the DDR2 standards into *SystemVerilog Assertions* (SVA). Kayed et al. [11] improves this idea by automatically deriving SVA from timing diagrams in the standards. More recently, Steiner et al. [19] go further by proposing automatic generation of SVA from *DRAMml* scripts, a DSL that models the functional and timing properties from JEDEC standards as *Petri nets*. While these approaches have incrementally succeeded at automatically capturing and formally verifying JEDEC properties, they cannot be used to verify broader aspects, such as latency bounds, controller semantics, and security properties.

Li et al. [12] use the Uppaal model checker [1] to analyse memory controller models described as *Timed Automata* (TA). However, in order to keep the state-space manageable, they assume that each requestor has at most one outstanding request, i.e., they constrain how requests arrive in the system. Moreover, there is a large abstraction gap from the standards to the TA models, as they are complex and written by hand [13]. Moreover, Hassan et al. [9] use Linear Temporal Logic (LTL) formulas to specify the correctness of DRAM controllers. However, the specification is not used to prove the actual implementation correct. Instead, counterexamples are obtained from models by bounded model checking, which are then used to build a test bench for the validation of the implementation. This latter approach can be seen as complimentary.

Hardware & Deductive Verification. The idea of describing circuits using functional programming languages was first introduced by Bjesse et al. in Lava [2], a DSL written in Haskell. The verification part of Lava, however, was limited, since logical formulas extracted from circuit definitions were exported to automatic theorem provers (ATPs), and therefore limited to decidable properties. Cava implements the key concepts of Lava in Coq, allowing its user to prove circuit properties directly from its definitions. Cava's most notable use case is an *Advanced Encryption Standard* (AES) implementation proven correct against the AES NIST standard [8].

Other DSLs for hardware design in Coq are Kami [6] and Kôika [4]. The former has been used to verify a RISC-V implementation against simple ISA semantics [3]. In short, the latter extends Kami by providing mechanisms to enhance per-cycle performance of designs. In this work, we use Cava – which, like the other DSLs, is built to describe circuits at a low level – to bridge the gap from a high-level framework (CoqDRAM, written in plain Coq) to actual synthesizable RTL. CoqDRAM captures correctness criteria directly from the standards in their most natural form and is mainly used to prove algorithms, using general Coq abstractions and ignoring HW limitations. Conversely, proving the properties captured by CoqDRAM directly from a Cava or Kami circuit definition would be certainly more difficult. In our approach, by doing a single equivalence proof, the properties ensured by CoqDRAM are inherited by the CavaDRAM HW implementations (under some assumptions modelling HW limitations).

7 Conclusion

We developed a framework for designing correct-by-design, standard-compliant DRAM controllers. The feasibility of the methodology is demonstrated by one proof-of-concept implementation. We use the generated RTL as a plug-in replacement in an existing hardware design – the simulation and synthesis results provide confidence in the correctness of the Coq models. In the future, we plan to implement state-of-the-art bus arbiters and memory controllers using this methodology. Another research path would be to develop a DSL tailored to the needs of DRAM controller algorithms (handling of queues and DRAM requests/commands) that allows an automatic translation to Cava, while maintaining the versatility of plain Coq used in the current version of CoqDRAM.

Acknowledgement. We acknowledge and are grateful for the contributions of Sumantha Chaudhuri, who provided help setting up the simulation environment; Chirag Sudarshan, who made himself available for discussing the implementation of *DDR4cntrl*; and Lirida Naviner for insightful discussions. This work is supported by the European Union's Horizon 2020 research and innovation program under grant agreement No. 101070627 (REWIRE).

References

1. Behrmann, G., David, A., Larsen, K.G., Håkansson, J., Pettersson, P., Yi, W., Hendriks, M.: Uppaal 4.0 (2006)
2. Bjesse, P., Claessen, K., Sheeran, M., Singh, S.: Lava: hardware design in haskell. ACM SIGPLAN Notices **34**(1), 174–184 (1998)
3. Bourgeat, T., Clester, I., Erbsen, A., Gruetter, S., Wright, A., Chlipala, A.: A multipurpose formal risc-v specification. arXiv preprint arXiv:2104.00762 (2021)
4. Bourgeat, T., Pit-Claudel, C., Chlipala, A.: The essence of bluespec: a core language for rule-based hardware design. In: 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 243–257 (2020)
5. Chlipala, A.: Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant. MIT Press (2022)
6. Choi, J., Vijayaraghavan, M., Sherman, B., Chlipala, A., et al.: Kami: a platform for high-level parametric hardware specification and its modular verification (2017)
7. Datta, A., Singhal, V.: Formal verification of a public-domain DDR2 controller design. In: 21st International Conference on VLSI Design. pp. 475–480. IEEE (2008)
8. Dworkin, M.J., Barker, E.B., Nechvatal, J.R., Foti, J., Bassham, L.E., Roback, E., Dray Jr, J.F.: Advanced encryption standard (AES) (2001)
9. Hassan, M., Patel, H.: Mcxplore: Automating the validation process of dram memory controller designs. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **37**(5), 1050–1063 (2017)
10. Joint Electron Device Engineering Council: DDR4 SDRAM standard (2021)
11. Kayed, M.O., Abdelsalam, M., Guindi, R.: A novel approach for SVA generation of DDR memory protocols based on TDML. In: 2014 15th International Microprocessor Test and Verification Workshop. pp. 61–66. IEEE (2014)
12. Li, Y., Akesson, B., Lampka, K., Goossens, K.: Modeling and verification of dynamic command scheduling for real-time memory controllers. In: Real-Time and Embedded Technology and Applications Symposium (RTAS). pp. 1–12. IEEE (2016)
13. Lisboa Malaquias, F., Asavoae, M., Brandner, F.: A Coq framework for more trustworthy DRAM controllers. In: Proceedings of the 30th International Conference on Real-Time Networks and Systems. pp. 140–150. ACM (2022)
14. Milner, R.: A calculus of communicating systems. Springer (1980)
15. Nikhil, R.: Bluespec system verilog: efficient, correct RTL from high level specifications. In: Proceedings 2nd ACM and IEEE International Conference on Formal Methods and Models for Co-Design. pp. 69–70. IEEE (2004)
16. Park, D.: A new equivalence notion for communicating systems. Bulletin EATCS **14**, 78–80 (1981)
17. Pous, D., Sangiorgi, D.: Bisimulation and coinduction enhancements: A historical perspective. Formal Aspects of Computing **31**(6), 733–749 (2019)
18. Sangiorgi, D.: Introduction to bisimulation and coinduction. Cambridge University Press (2011)
19. Steiner, L., Sudarshan, C., Jung, M., Stoffel, D., Wehn, N.: A framework for formal verification of DRAM controllers. arXiv preprint arXiv:2209.14021 (2022)
20. Sudarshan, C., Lappas, J., Weis, C., Mathew, D.M., Jung, M., Wehn, N.: A lean, low power, low latency DRAM memory controller for transprecision computing. In: International Conference on Embedded Computer Systems. pp. 429–441. Springer (2019)