2023 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE 10 YEAR ANNIVERSARY

Verification of an AXI cache controller with a multi-thread approach based on OOP design patterns

Francesco Rua', ST Microelectronics, Catania, Italy (francesco.rua@st.com)

and

Péter Sági, Veriest Solutions, Budapest, Hungary (peters@veriests.com)

***Abstract*— AXI protocol high-performance features and cache performance requirements demand a careful strategy for verification to be decided in advance. This is very important to achieve the main targets: minimum effort for development and maintenance of the testbench; minimum dependency on DUT microarchitecture; effective debug strategy.**

A high-performance cache controller employs intense pipelining as a significant part of its microarchitecture. Such a microarchitecture may be deeply changed for either new functionality, further improvements or bug fixes, and a well-designed testbench should not be affected heavily by those changes. Pipelining inside a cache controller also implies parallel accesses to both cache and external memories from different stages of the pipeline(s), and every access may affect the next stages as well. Therefore, a design defect could be difficult to analyze if we base our debug on mere data integrity checks.

An effective testbench for a cache controller and, in general, for pipelined designs should be abstract as much as possible in order to treat DUT's microarchitecture as a black box. The testbench can still get synchronized with the DUT behavior by means of observed transactions on its interfaces. It should also use point to point scoreboards at each DUT interface, so that the very first error will show the way to its root cause. In this paper we describe our approach based on well-known OOP design patterns, point to point scoreboards and multithreading.

Keywords*— *Verification; multi-threading; design patterns; pipeline; cache controller

I. INTRODUCTION

Cache memories are used to reduce the ever-increasing speed gap between CPU and memory. Considering that fast memories are expensive in terms of cost and space per bit, a memory hierarchy is typically employed: cache memories are small fast memories located near the processor and containing the most frequently used data and instructions. Their content is from bigger and slower memories. More levels of caches may be applied. They typically have an N-Way set-associative mapping and an efficient replacement policy to increase the probability of a HIT.

AXI protocol helps in improving cache performances thanks to its high throughput features: concurrent requests on both RD and WR channels, reordering respect to request ID and large acceptance capability for both channels. Furthermore, specific cache performance features can be employed: HIT-under-MISS and MISS-under-MISS.

As a direct consequence, the cache controller needs pipeline techniques and several FIFOs inside its microarchitecture in order to increase the throughput: the whole process required to complete a request is divided into small tasks; every stage of the pipeline(s) performs a different task related to the associated request under execution. In general, every stage in the pipeline processes a different request. As a result, the number of memory accesses in a given period of time is increased noticeably: the cache controller can grant multiple requests on its slave port, and it can perform simultaneous execution of different tasks.

II. VERIFICATION OF A CACHE CONTROLLER

The main requirement for an AXI cache controller is delivering fast and correct responses for each request granted on AXI cache slave port. The main verification focus is on checking the data consistency among the AXI ports and cache memory interfaces by means of scoreboards. To verify the desired throughput of the IP, the predicted transfers need to be provided to the scoreboards with the correct timing given by the functional specification. Unfortunately, the functional specifications are not always detailed enough to model the correct transaction timings, which denies having specific timing checks.

To provide the expected transfers to the different scoreboards in the testbench, a functional reference model should be implemented. The most challenging point for creating such a model is the pipelined processing nature of the IP. There are formal verification techniques available, which can be successfully applied for pipeline verification in processors[1][2]. Such an approach requires a well-defined architectural specification about all pipeline stages and their connections, to be able to apply the technique and implement the formal properties.

In our case the functional specification only describes the required functionality of a cache controller and nothing about exact design architecture, hence we can talk about a pipelined processing functionality rather than concrete pipelines in the design. However, the more a reference model strictly reproduces the IP architecture, the more effort is needed to rework it after every little change in the IP. Therefore, the decision was made to build a functional reference model for dynamic simulation in System Verilog using UVM which can also mimic the pipelined processing functionality.

The major requirements for the model were to be modular enough to support reusability and scalability, to have minimum dependency on the DUT microarchitecture and to provide a rich set of debugging services. During the development we wanted to explore and apply some known art object-oriented programming patterns typically employed in software programming, but still valid and helpful (even if not so diffused) in dynamic verification environments. Furthermore, in general, we wanted to create a model structure which can be applied for future IP verifications.

It is evident that, in case of multiple requests granted on slave port, depending on the kind of requests, on the status of the cache (HIT or MISS), on delays on AXI data ports, and so on, there may be different actions ongoing on each IP port, either associated with the same request or with different requests. A multi-threaded model is more convenient to cope with more processes happening concurrently in different stages of the IP pipeline(s). The model can be abstract enough to avoid strict match with DUT pipeline(s) stages. For sure, it needs to get synchronized with the DUT through observed transactions on its ports.

Once the minimum number of main threads has been identified, considering that pipeline stages can be thought of as states of a process, every thread is represented by a state machine that controls the evolution of a specific task in terms of predictions of transactions on IP output ports, including exchanged data.

III. Reference model structure

As already mentioned in the previous sections, some of the major requirements for the model are modularity, scalability and reusability. Therefore, we decided to divide its structure into sub-components with different dedicated responsibilities. Based on functionality we can sort these components into several kinds described in the following sections and shown in Figure 1.

A. *Process Items*

Process-items are dynamic objects shared by all the reference model components and the state machines whose states are processing it. It is a composite object including all the related prediction items, bus transaction items, state machines, coverage information and every other information needed to manage its evolution through states. Every request granted by the cache controller is associated with one (or more) process-items. Depending on the kind of process-item, there can be different state machines to handle its evolution. As an example, if a cacheable-allocate request is a MISS in cache, it will require an allocation in the cache memory at a specific cache line and, if the allocated line is already valid and dirty, it will require an eviction of the current cache content too. It means that there are up to three main actions to be executed in parallel: response handling on slave port and allocation handling as well as eviction handling on master and cache ports. Every main action requires a series of different behaviors implemented as states, depending on the kind of request.

Due to high performance features of the cache controller, there can be several actions executed in parallel that may come from the same or different process-item. In other words, due to the pipelined nature of the DUT and its internal FIFOs, there can be several process-items under execution, containing multiple state machines in different states. Furthermore, a process-item can traverse some states where synchronization is needed with other states either processed by its own state machines or from different process-items.

During the creation of a process-item every state machine inside gets its specific initial state. When the item is executed, its main thread is spawned which, in turn, spawns the execution of the state machines associated with the process-item. The first transition from the initial state to the next state is triggered by an arbiter. The sequence of state transitions for each state machine under execution depends on the kind of request and on the interaction with other monitored ports.

During a transition from one state to another one, the process-item is populated with new information coming from reference model components with monitoring functionality. Such information might be used/consumed by other components to send predictions to the scoreboards, and by states to determine the next states. When all the state machines reach their final state, the process-item will be considered done and can be destroyed. The design pattern used to manage state transitions is the state pattern, as described later in this paper.

The reference model components and the states traversed by the process-item are communicating to each other by means of notifications. The design pattern used to manage such a communication is the observer pattern (a publisher is notifying its subscribers) as described later in this paper.

B. *Process database*

Every time a new request is granted on the AXI cache slave port, or a new user command is granted on the control slave port, one or more process-items are created and pushed into specific queues inside a process database. The database acts as a central storage within the model and can be accessed by all other sub-components. It provides a wide set of queue manipulation and query methods to easily store or get a process-item during the process executions. The database also maintains special queues for history and debug purposes.

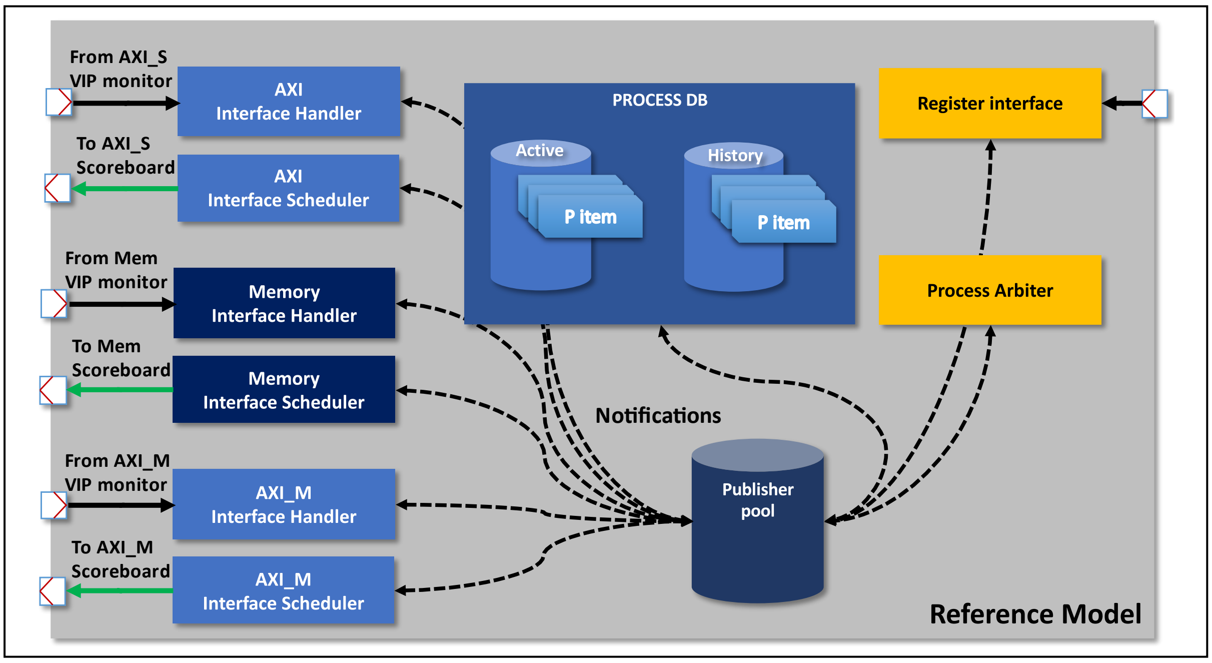

Figure 1. Reference model structure

C. *Interface handlers*

To be able to keep and maintain IP interface related actions at one place we created the interface handler components. The model contains one handler for each IP port e.g.: slave port handler, master port handler, cache port handler etc. The reference model concept is scalable: in case of any additional IP port, further handlers can be easily created and instantiated.

A handler always provides connections to external verification IPs (e.g.: through TLM ports) and relays bus events coming from the connected VIP monitors to state machines or other components within the model. At every bus event the handler is delegated to find the corresponding process-item in the database and to populate it with the monitored bus transactions. This centralized mechanism reduces the complexity of state machine code regarding interface transaction identification.

The reference model outputs are mostly realized in the form of scoreboard items. These are generated also in the interface handlers and stored as predictions within the corresponding process-items.

Sometimes a handler has extra functionality: the slave port handler also creates the process-items themselves every time a new request is granted. A special type of handler is the user interface handler, which is connected to the register model and only indirectly to the slave control port VIP.

D. *Schedulers*

The model implements for every IP interface also a scheduler component. After a scheduler receives a trigger notification, it gets the predictions from a corresponding process-item and sends them out to the related scoreboard at the proper time, based on the interface arbitration rules. This management lets the predicted items be sent to the scoreboard as close as possible to the observed ones allowing a time windowed-scoreboard approach for the verification.

E. *Process arbiter*

There is one arbiter component using the same arbitration scheme as the IP. Its purpose is to start one of the process-items under execution in the database according to arbitration rules. Starting a process-item means triggering its first state transition, from initial state to the first relevant state.

IV. Applying OOP design patterns

A. *State pattern implementation*

The state pattern allows to manage different behaviors of a context, depending on its state, by means of different state classes, one for each behavior. This means that, instead of implementing all the behaviors in the context itself in a long and complex if-else tree-structure, we can delegate a different state class for each behavior of the context depending on its current state. The context only needs a reference to its current state object which, in turn, executes the relevant tasks associated with that state. The context also provides a method to change the referenced state to a new one every time a transition to a new state is to be done. In order to be able to delegate the specific tasks to the corresponding state class, all state classes must implement the same interface. In our case such an interface provides a specific execute method. All state classes implementing the state interface provide a different body for such a method.

Considered that our context is the process-item that can implement one or more state machines, and in order to keep state machine functionality isolated from the context itself, a state machine object is defined and included in the process-item in a composite manner. It is more convenient keeping all the state related functionality like storing state transitions for debug purposes into a history database, in the state machine object rather than having it inside the process-item in a flat way. All state machines shall implement the same interface in order to set the next state and to propagate the delegation from the process-item to the state. The specific method provided by the state machine interface is named execute as well. The execute method in both state and state-machine interfaces will take an argument of type context to get a reference to the calling context. A specific interface class is also defined for the context itself for implementation coherency. Furthermore, considering that some test scenarios require reset assertions on the fly, all the above interfaces shall provide a flush method too.

Figure 2 shows the above description with a UML diagram.

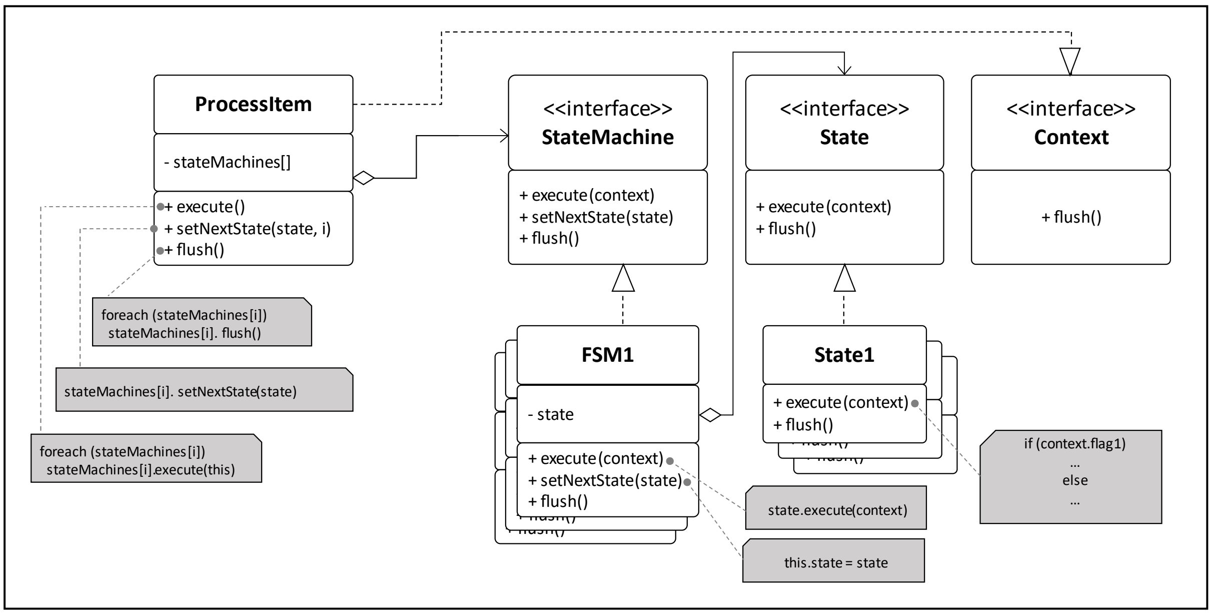

Figure 2. UML diagram of the implemented state pattern

B. *Observer pattern implementation*

The observer pattern allows to update a list of subscriber objects upon any relevant event to be notified by the publisher object they are observing. The publisher shall provide a mechanism to let subscriber objects be added or removed into/from its list of subscribers. Subscribers shall provide means to let the publisher update them. A context may be passed from the publisher to the subscribers upon notification so that subscribers can evaluate it. As there may be several subscriber classes, they must implement the same interface so that publishers can use it to update them. Similarly, as there may be several publisher classes, they shall implement the same interface so that all subscribers can use it for (un)subscription. The subscriber interface shall provide an update method taking a context as an argument. The publisher interface shall provide a notify method taking a context to be communicated as an argument, as well as methods to add and remove subscriptions. Furthermore, they shall provide a flush method for reset on the fly management.

In our case the context is the process-item. Its states need to be notified upon different events either from IP ports or from other states of different state machines inside the same process-item or other ones under execution. Furthermore, all reference model components need to be notified upon different state transitions of process-items under execution to handle and schedule predictions. Every specific task to be performed upon a specific event is delegated to a corresponding subscriber to be added to the publisher associated with that event. Every specific event from IP ports or state transitions inside process-items under execution triggers the notification of the associated publisher class. Every publisher class is delegated by states and reference model components to notify a specific event to its subscribers. For each relevant event there is a unique publisher instance shared by all the reference components and the process-items under execution.

Depending on the state of the process-item, one or more subscriptions to different publishers are performed, one for each relevant event to be notified. Every subscriber class is delegated by the state to perform a specific task associated with that state and the corresponding event to be notified. Such tasks may trigger a specific prediction followed by a new notification for other subscribers, as well as a new state transition and so on.

C. *Decorator pattern implementation*

In some cases, the behavior of the notification from some publisher needs to be modified before being deployed to subscribers. In fact, there may be cases where the same publisher is used from different contexts under execution, and we need to maintain a priority order for the multiple calls to the notify method of the same publisher. In other cases, the notification of a context from a publisher is produced before the consumer can generate its subscription to it. In such cases, the decorator pattern is used to modify the behavior of the notification in a transparent way for subscribers.

The decorator pattern allows to wrap the publisher inside an object that implements the same interface and adds new behavior either before or after passing the request to the publisher itself. Of course, this approach can be used to decorate decorators as well. Thus, the publisher's behavior can be modified several times by different cascaded decorators before performing its final notification.

D. *The benefit of using programming patterns*

With such a programming approach, all the actions associated with a specific process-item are divided into small tasks delegated to states in different state machines. Every state behaves differently: it interacts with different events and generates new ones; in turn, it performs its job in the form of smaller tasks delegated to different subscribers. This way the structure helps managing easily any new IP feature that may be added in the future, or any change in the existing ones. We only need to implement the new behavior in the form of new states, subscribers and publishers, in a modular way.

V. A TYPICAL MODEL OPERATION

In the next figures a typical reference model operation is described in case of a cacheable write miss request. In Figure 3 the slave interface handler (AXI_S IF_HANDLER) receives a new cacheable request through its TLM port from the AXI VIP and creates a new process-item, starts its execution and stores the item in the process database (PROCESS DB). The main task in the process-item spawns all the state machines threads. Every state machine is waiting in its initial state until the process arbiter notifies them through a specific publisher object. After getting the notification the first state transition is performed. The slave interface handler concurrently receives new write beats, which are automatically assigned to the corresponding process-item fetched from the process database.

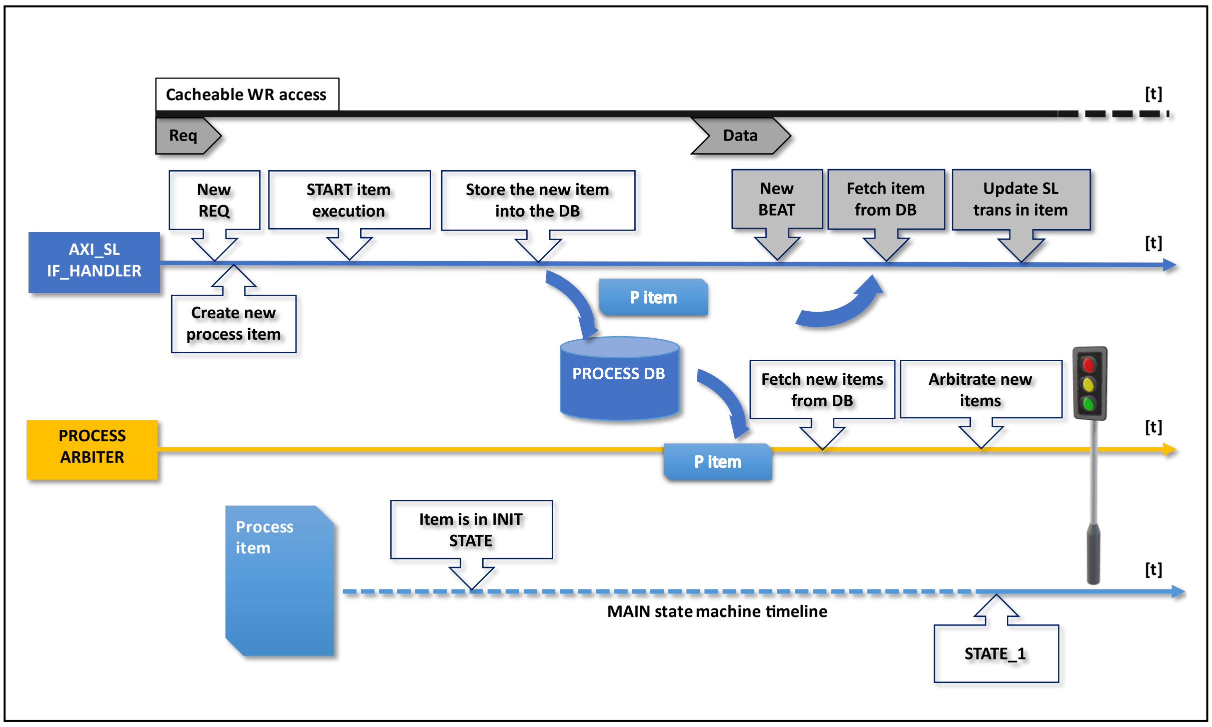

Figure 3. Model operation scheme at cacheable request reception

State classes are stored in a library file (STATE LIB). Every state implements a different behavior and sets the next state according to some conditions. In general, states can directly notify other states or reference model components through publishers or delegate subscribers for executing some tasks upon specific events. Figure 4 shows some example of state transitions and a scenario about notification from a state to a reference model component. In the example shown in the figure the process-item executes at least two state machines, where the first one handles the merging of incoming slave port write beats and data from refill, while the second one handles the refill process on the AXI master port. Similarly, to the slave interface handler in the previous figure, the master handler (AXI_M IF_HANDLER) concurrently receives new read response beats, which are automatically assigned to the corresponding process-item fetched from the process database.

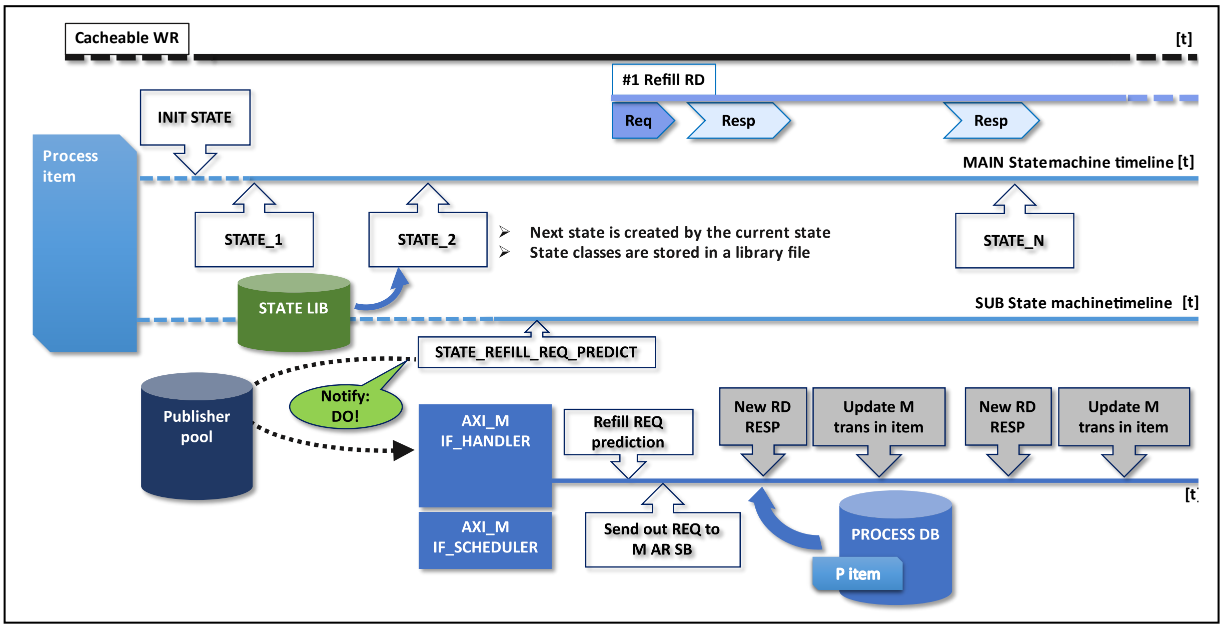

Figure 4. Model operation scheme during cache line refill

Once the processing reached the write-response prediction for the slave port, the state machines will reach their final state. The process-item is considered finished when all its state machines are done, and it can be removed from the database and destroyed. This is shown in Figure 5.

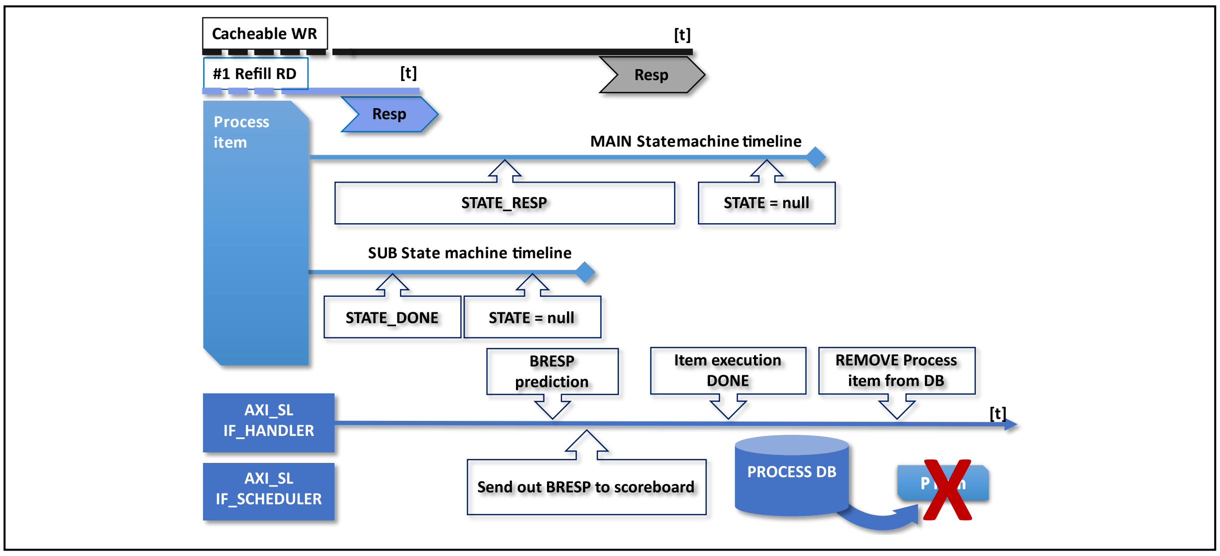

Figure 5. Model operation scheme at the final stage of processing

VI. Conclusions

With this approach we were able to successfully verify a cache controller IP. It was easy to model the pipelining nature of the hardware without any concern about its microarchitecture. We modeled the main functional stages of the IP pipelines as states, and we used multi-threading to emulate parallel processing. Using libraries of state classes allows us to easily extend or modify the IP behavior in the reference model in the future.

Dividing the model into several sub-components dedicated to monitoring and predictions and having a control layer in the form of state machines helps to extend and scale the model in case of any change requests.

Debugging facilities such as history queues for process-items, state transitions, notifications from publishers or cache line access history helps to analyze DUT failures even in the most complex cases.

The same reference model approach can be applied for any similar pipelined designs, e.g.: memory controllers or DSP (Digital Signal Processor). After identifying the main functional stages of the pipeline which are interacting with the IP ports, we can define a set of states (behaviors) and actions (predictions, notifications, timing) controlled by these states to predict the expected functionality.

References

[1] Jeremy Levitt and Kunle Olukotun, “A Scalable Formal Verification Methodology for Pipelined Microprocessors” in DAC 96 - 06/96 Las Vegas, NV, USA.

[2] Djuro Grubor, “Verification strategy for pipeline type of design” in DVCon-US 2018.