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

Francesco Rua’ (STMicroelectronics)

Péter Sági (Veriest Solutions)
Agenda

• Verification target
• Setting up strategy and exploring solutions
• Creating the model
• A typical processing example
• Q&A
Planning

Verification target, goals, possible solutions
Our verification target: AXI cache controller

• Complex behavior with throughput maximization
  • **AXI interfaces** (with response re-ordering, multiple request acceptance etc.)
  • **Pipelined operation** with buffers

• Typical cache controller operations
  • Hit/Miss check in the cache memory
  • Refill of the data from the external memory in case of a missing cache line
  • Eviction of an occupied dirty cache line back to the external memory
  • Bypass non-cacheable accesses
Verification goals

• The verification focus is mainly on checking data consistency and throughput  
  • Point2point scoreboards

• Functional reference model is needed  
  • Should be as abstract as possible  
  • Keep it modular to be able to easily follow the future functional IP CRs

• Strong debug support to speed up the verification cycles  

• Using state of the art techniques also from the OOP world
How to model it?

• Formal approach?
  • Requires to have the exact microarchitecture specification
  • We lose our goal to be as abstract as possible

• Dynamic approach?
  • Standard modelling is not enough to support the pipelined functionality
  • It can be abstract, modular and more re-usable
Multi thread model

• Multi thread model
  • Able to accept and process all concurrent input triggers

• One request, multiple threads
  • Every request on slave port spawns several threads
  • Multiple threads run in parallel
  • Emulate concurrency and pipelining inside the IP

• One thread, one main action
  • Response/allocation/eviction handling on IP ports
Thread evolution

- Every thread executes its job step by step
  - Every step requires different actions to perform
  - Steps can change dynamically
- The OOP **State pattern** can be applied here
  - **One step, one state**
  - Every state implements a different behavior
  - One **state machine** is used for each main action/thread
Using the State pattern

• State transitions are controlled dynamically
• The state needs information to
  • Set the next state
  • When to trigger the state change
• Every state consumes/produces information from/for
  • Other states
  • Model components
  • IP interfaces
Exchanging information

• Many state machines in many threads can produce high amount of information

• The OOP **Observer pattern** can be applied
  • Communication based on notifications
  • A publisher object notifies its subscribers about a context
    • A context in our case is a processing thread object
  • Subscriber objects execute some tasks upon notification
  • (Un)Subscription is dynamically controlled
Using the Observer pattern

• Every state
  • Can delegate several subscribers to perform specific tasks upon notifications
  • May trigger a publisher for notification to its subscribers
Observer pattern limitations

• The pattern implementation is not enough to cover all needs in our model

• Using multiple threads in the model requires sometimes
  • Postponed notifications
    • When the notification is produced before a subscription
  • To maintain a priority order for the multiple calls to the notify method of the same publisher
Using the Decorator pattern

- We needed a solution to modify the behavior of the notifications before being deployed to subscribers
  - The OOP **Decorator pattern** gave us the solution
- It allowed
  - To dynamically wrap publishers in a transparent way for the subscribers
  - To attach 1 or more decorations for the notifications

*Refactoring.Guru*
Let's create the model!

Put things together
Process item

• The central object type in our model is a thread object what we called: **Process item**

• Every AXI request on slave port generates one or more of these items

• A process item
  • Carries all the information related to the processing of a request
  • Information is dynamically produced and consumed all along its execution
  • It is the context item shared and exchanged among threads, states, publishers, subscribers and so on
Process item structure

- Each cache operation type has a corresponding process item type
- Executes any number of state machines
- Each state machine is running concurrent in separate threads
- All the process relevant interface events and accesses are stored in the process item
- The predicted scoreboard items are stored here
Process database

- All the process items are stored here
- Reference model components can access it
- Provides queries to get process items
- Multiple storages for different processing types
- Implements history queues to keep process items for debug
Process arbiter

• It starts the process items in expected order
• Uses the same arbitration scheme as the RTL
Interface handlers

- Connection points to the external VIP interface agents
- Creates new process items
  - In the cache controller model the AXI slave handler only
- Updates the process objects with the monitored information
- Handles predictions for expected scoreboard items
Interface schedulers

• Connection points to the external TB check components
• Sends out the predicted model output information to
  • Scoreboards
  • Registers
• Timing checks are supported
  • E.g.: for performance
Model in operation

A typical processing example
A typical processing example

• AXI Cacheable write
  • The write beats have random latency

• The accessed area is NOT in the cache memory
  • Refill operation is performed
  • The read beats have random latency

• The refilled data needs to be merged with the written data
  • Synch is needed between AXI slave and AXI master data

• After successful processing a write response is generated
Cacheable WR access

Req → Data

New REQ
START item execution
Store the new item into the DB

New BEAT
Fetch item from DB
Update SL trans in item

Create new process item.

PROCESS DB

P Item

Fetch new items from DB
Arbitrate new items

AXI_SL IF_HANDLER

State 1

PROGRESS ARBITER

Process item

Item is in INIT STATE

MAIN state machine timeline

STATE 1
Cacheable WR

Process item

MAIN State machine timeline [t]

INIT STATE

STATE_1

STATE_2

STATE_LIB

Notify: DO!

Publisher pool

#1 Refill RD

Req Resp

Resp

New RD RESP

Update M trans in item

New RD RESP

Update M trans in item

Refill REQ prediction

Send out REQ to MAR SB

PROCESS DB

P item

SUB State machine timeline [t]

➢ Next state is created by the current state
➢ State classes are stored in a library file

STATE_LIB

STATE_REFILL_REQ_PREDICT

STATE_N

AXI_M IF_HANDLER

AXI_M IF_SCHEDULER

New RD

RESP

Update M trans in item
Benefits and possible future usages

• **High modularity** allows to easily add new functionality in the future

• **State libraries are reusable** for future IP derivatives or extensions

• The model approach **allows to verify performance features** beside simple data consistency checks

• The multi thread reference model approach can be applied for pipelined designs
  • E.g.: Digital signal processors or Memory controllers etc.
Questions