CAMEL: A Flexible Cache Model for Cache Verification

Yue Liu
MediaTek.inc, Beijing, China
Yue.Liu@mediatek.com

Fang Liu
MediaTek.inc, Beijing, China
f.Liu@mediatek.com

Yunyang Song
MediaTek.inc, Beijing, China
Yunyang.Song@mediatek.com

***Abstract*- With the increasingly complex cache design in recent years, cache verification has been regarded more challenging. In this paper, we demonstrate a flexible cache model for cache verification. The cache model is built with a simple structure, but it can fulfill the most important verification criteria -- data correctness, even in a complex design. Extra 'location' information is added to the model for more precise checking and stimulus generation.**

I. Introduction

RISC-V becomes a hot spot in recent years, and some of the applications require high performance data access. In order to satisfy this, a lot of tricky logics is created in cache design, and cache verification has become a challenging task. In this paper, we present a flexible and extendable cache verification model which supports multi-thread and multi-core structure.

The cache model will be introduced in two perspectives: The basic structure to fulfill cache's data check, then extra 'location' and other information for more precise verification. We named the model as CAMEL (CAche ModEL) in short, as the model's characteristic is just like camel's two humps. Besides CAMEL model itself, how it is used in testbench is also introduced in this paper. Our goal is to make cache verification more general and flexible.

II. Design and Former Verification Environment Overview

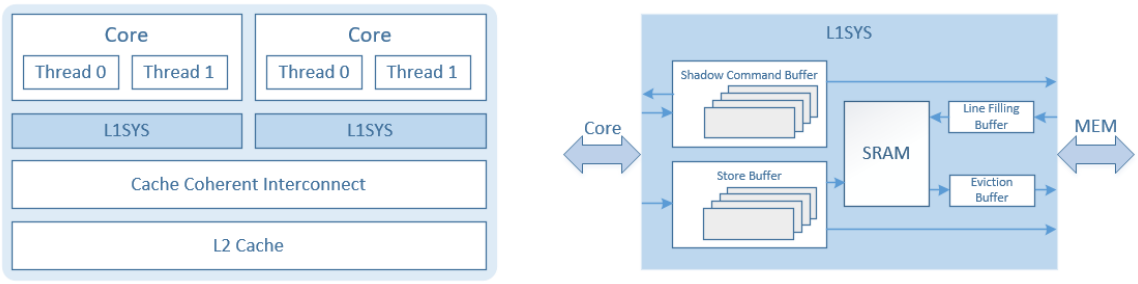
A. *Design Overview*

Our L1 Cache sub-system (L1SYS in short) is a part of multi-core and multi-thread RISC-V design, as shown in Figure 1(a). A group of internal buffers are placed in L1SYS to improve cache performance, as shown in Figure 1(b). All these buffers are working for the purpose of enhancing L1SYS's performance.

For example, to reduce the overall latency, when thread-0 encounters cache miss, that thread will be scheduled to a 'shadow command buffer' (SCB), fetching data in background, so that the design can switch to thread-1. Before the data is fetched from external memory, if thread-1 accesses the same address as the one in SCB, L1SYS will not send request to external memory again to avoid redundant bus access.

SCB is used to hide the latency of cache miss, and move the long latency data access to background. Besides SCB, the store buffer (STB) stores write data from core temporally, the line filling buffer (LFB) and the eviction buffer (EB) store data from external memory temporarily. The purpose of these three kinds of buffers is hiding the latency of accessing SRAM to improve the performance of L1SYS.

L1SYS design in multi-core scope is modified from the design in single core, and all the internal buffers already exist in single core scope. In multi-core scope, the internal buffers' structure keeps the same as single-core, but their behavior changes a lot to fulfill multi-core's ACE [1] protocol and deal with snoop requests.



(a) Overall design block diagram (b) L1SYS interfaces and internal buffers

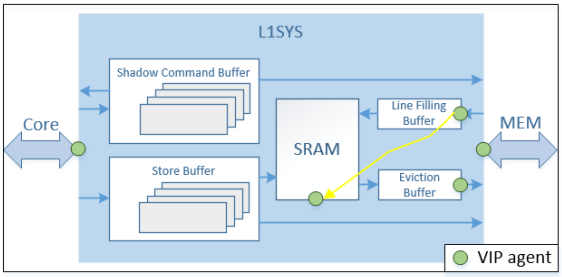
Figure 1. L1SYS related design block diagrams

B. Former Verification Environment Overview

All these buffers improve the cache's performance while they also increase the difficulty of cache verification. When a load miss command moves to background to fetch the data, the command's execution process at background would influence the data value returning to the core, which influences the data correctness. Data correctness is the most important point of cache verification, so checking the background is necessary.

In previous project, we modeled all these buffers in cycle-accurate manner in our testbench [2], which requires to hook up 5 VIP monitors on RTL external and internal interfaces, as shown in Figure 2(a). The previous testbench checking mechanism is based on MESH scoreboard. MESH scoreboard is a MediaTek in-house VIP, it is basically used in data flow checking. Each data flow checker is a stream, and many streams combine together to build a huge mesh, so the scoreboard is called MESH scoreboard. In the testbench, the data streams are separated into many sub-routines between internal buffers, and the summary is described in Figure 2(b). E1 and E2 in the Figure 2(b) mean the stages of L1SYS's pipe, and ZAC, COP, ST and RD are the acronyms of the core's instructions. The checkers are built based on the pipe's timing and instruction types, and added at the input and output ports of the data streams to ensure the data flow's correctness.

For example, in Figure 2(b) the yellow columns indicate the data stream from LFB to SRAM, which corresponds to the data flow in Figure 2(a). The testbench simulates the LFB's behavior and checks at LFB and SRAM's interfaces, so that the data correctness between the 2 points can be checked. It is because the previous testbench simulates all the buffers, the data flow can be predicted and checked in continuous data streams.



(a) The VIP agent view on testbench

from/to	Core	SCB	LFB	STB	EB	SRAM
Core	status,rda	D$ miss	ZAC hit	E3 WR hit	x	E1 RD
SCB	data back	backgroun	ZAC hit	x	x	COP hit
LFB	LFB forwar	x	x	x	x	linefilling
STB	Store it	x	x	x	EB forward	ST/ZAC
EB	x	x	x	EB forward	x	x
SRAM	E1/E2 read	COP hit	x	COP clean/	Evict	Victim

(b) The summary of data streams checker

Figure 2. Previous L1SYS verification environment overview

Hence the internal buffers are cycle-accurate modeled, the verification environment is tightly coupled with RTL design. Each time modifying buffer model's behavior costs huge efforts on both RTL designer and verification engineer. The checking mechanism can't work until the testbench is modified well along with RTL design's behavior, and the modification usually costs a lot of time.

III. CAMEL MODEL STRUCTURE

In order to reduce the effort RTL design change brings to both RTL designer and verification engineer, an enhanced checking mechanism is proposed. The main idea of the mechanism is that the cache should be 'transparent' for data access. Whether the cache exists or not, the design module between core and memory should guarantee the data correctness, regardless of how the cache is implemented. The basic instructions' behavior in L1SYS is shown in Figure 3.

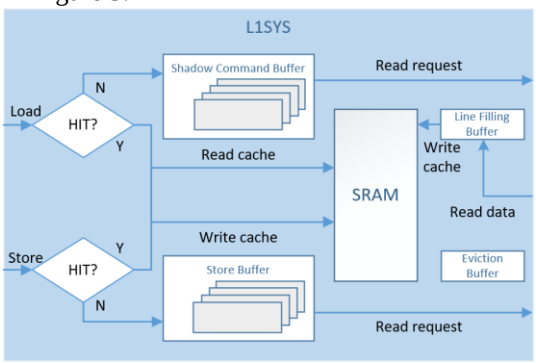


Figure 3. The basic instructions' behavior in L1SYS

CAMEL model is created to mainly focus on data correctness check, and it has low dependency on design implementation at the same time. CAMEL model is organized in byte unit in 'address vs. data+attributes' view, which covers SRAM, all internal buffers, and even external memory, as shown in Figure 4.

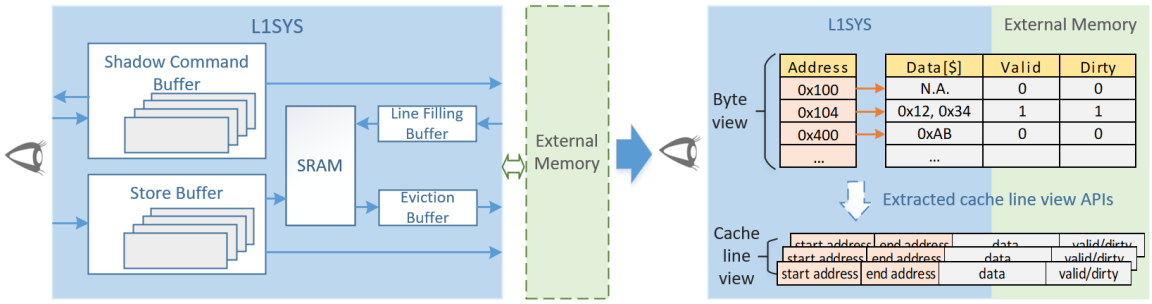


Figure 4. The Changes between Previous Buffer Models and CAMEL Model

In a 32-bit address size memory system, the address range is from 32'h0 to 32'hffff_ffff, 4G address space. The cache only contains very few part of addresses compared to the whole address space, which is 32kB in current project. The same as cache, CAMEL model only stores the valid or useful address and data into the model. When the bytes are evicted from cache, CAMEL will delete them at proper time.

Cache line is the smallest storage unit in the cache design, but it can be configured to different size in different CPU projects. To extremely reduce the effort that design change might bring, CAMEL model chooses byte as the smallest storage unit of the model. Each byte corresponds to a unique address, and CAMEL also provides a set of configurable APIs for virtual 'cache line' views for convenient usage. The two views of the data in cache are similar to the two humps of a camel. Whichever view is used to query data, the data, just like the water in camel, is the same.

Due to L1SYS's multi-thread structure and tricky design, the 'address-data' mapping relationship is not always accurate. A typical example is shown in Figure 5. If thread-0 is executing a load instruction and cache miss, the instruction will be moved to SCB and switch to thread-1. CAMEL will register the address while valid is 0 before the data is read back from external memory. Then thread-1 might store data at time A or time B to the same address before core's thread-0 gets the load data at time C. The stored data will be recorded in CAMEL and marks the dirty bit to 1. After the load data is got back from external memory, the corresponding bytes' valid are written to 1. When thread-0 gets the load data, the load data can either be the old data from external memory, or the new data overwritten by thread-1. Both data are reasonable from software view in multi-thread design, while the RTL result is different based on the store data's timing.

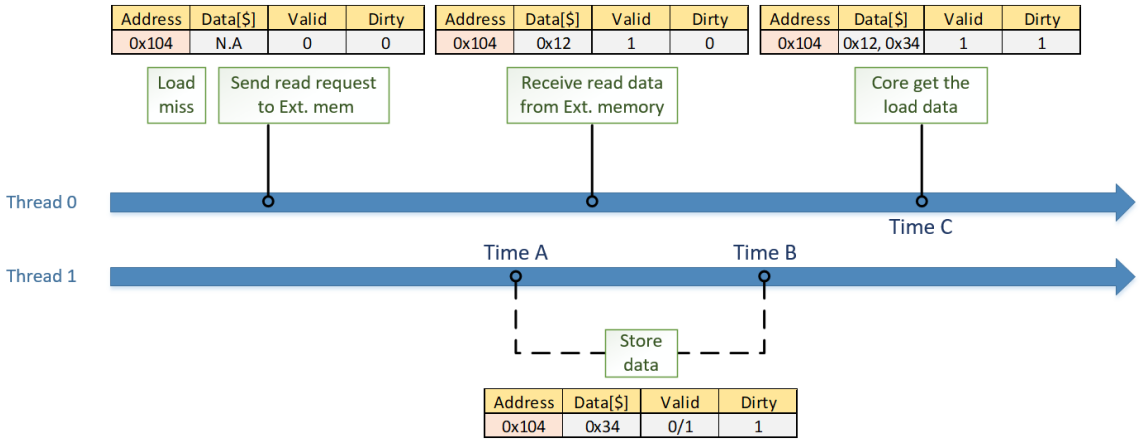


Figure 5. The Flow Chart of a Multi-thread Data Access Example

It takes a lot of effort to handle such kind of cases or even more complex cases between different threads, and the result might change depending on RTL implementation on different projects. CAMEL model takes an alternative way, it does not monitor the buffers' detail behavior in cycle-accurate mode like RTL does, but stores all possible values of the same address under such case. Either value will be considered correct during verification.

In the real L1SYS design, each load instruction will get only one piece of load data. But the checking mechanism based on CAMEL usually gives a group of data for reference at most times. Although the checking accuracy is declined by using CAMEL, the verification environment based on CAMEL can be quickly built up and easily modified for early phase checking. Using this method, we can provide a reasonable checking accuracy, without necessarily providing the cycle-accurate models to handle corner cases, and de-coupled with RTL design.

IV. Location and Extra Info For Precise Check

CAMEL model's structure is introduced at previous section, it can check data correctness in a simple method with acceptable precision. Besides data check, other critical checkers like hit/miss check and the rationality of read/write command to external memory are also hard to check based on the basic CAMEL model structure.

To solve the above problems, besides address and data, each byte unit will store an arbitrary number of extra attributes. One of the most important attributes is 'location', as the word implies, it represents where the data is locating. To solve the case between threads described in previous section, an extra attribute 'SCB data' can also be added to the model, the amount is decided by thread number. After adding 'location' and other extra information, the byte view of CAMEL model changes are shown in Figure 6.

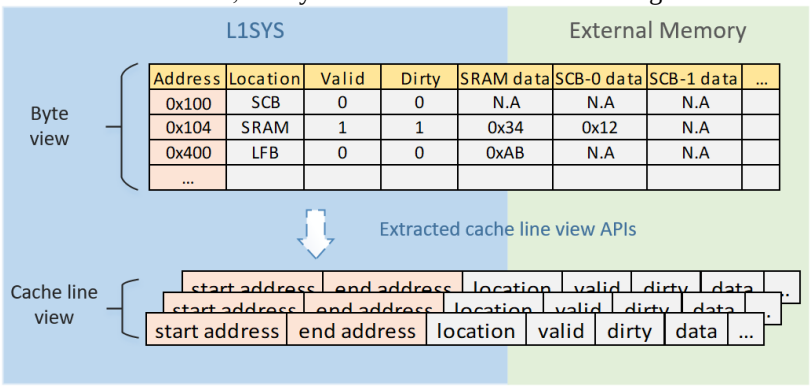


Figure 6. CAMEL model after adding 'location' and extra information

The 'location' info can be 'NA', 'SCB', 'STB', 'Ext. Mem', 'LFB' and 'SRAM' based on current L1SYS design. The transfer process of the locations is shown in Figure 7.

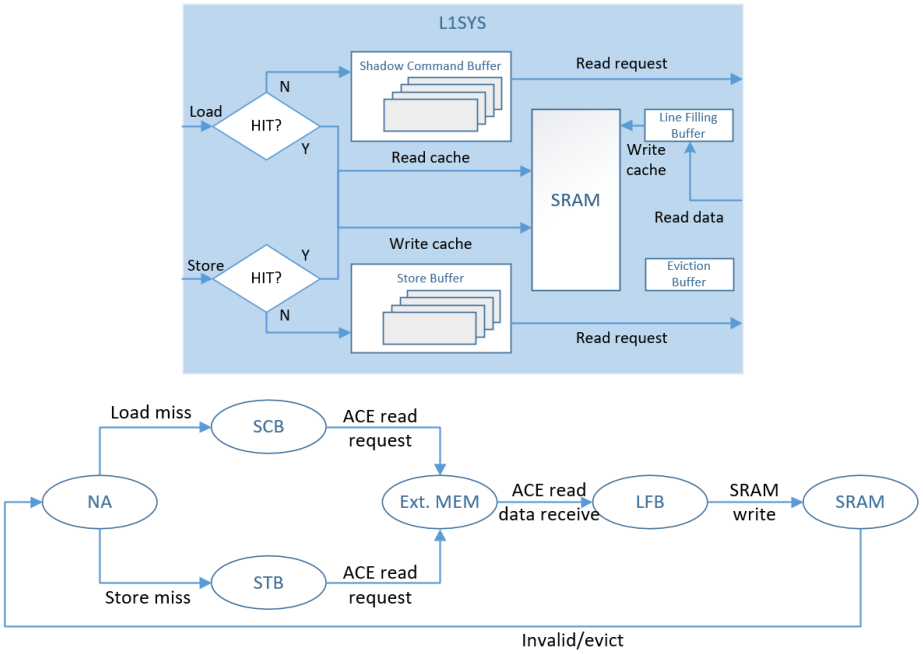


Figure 7. The transfer process of the locations in L1SYS

The transfer process of ‘location’ is changed along instruction’s executing process. The default ‘location’ of each byte is ‘NA’, and it will change to ‘SCB’ or ‘STB’ depending on the instruction type sent from core. When the ACE read request corresponding to the byte’s address appears at L1SYS’s interface to external memory, ‘location’ of the byte will change to ‘Ext. MEM’. When the read request’s data is sent back, ‘location’ will change to ‘LFB’ and ‘SRAM’ along the allocation process.

With more information added into CAMEL model, the model has 3 advantages compared to the basic one: (1) data check becomes more precise; (2) the rationality of command sent to external memory can be checked; (3) instruction from core’s hit/miss can be checked. The example in previous section changes to Figure 8.

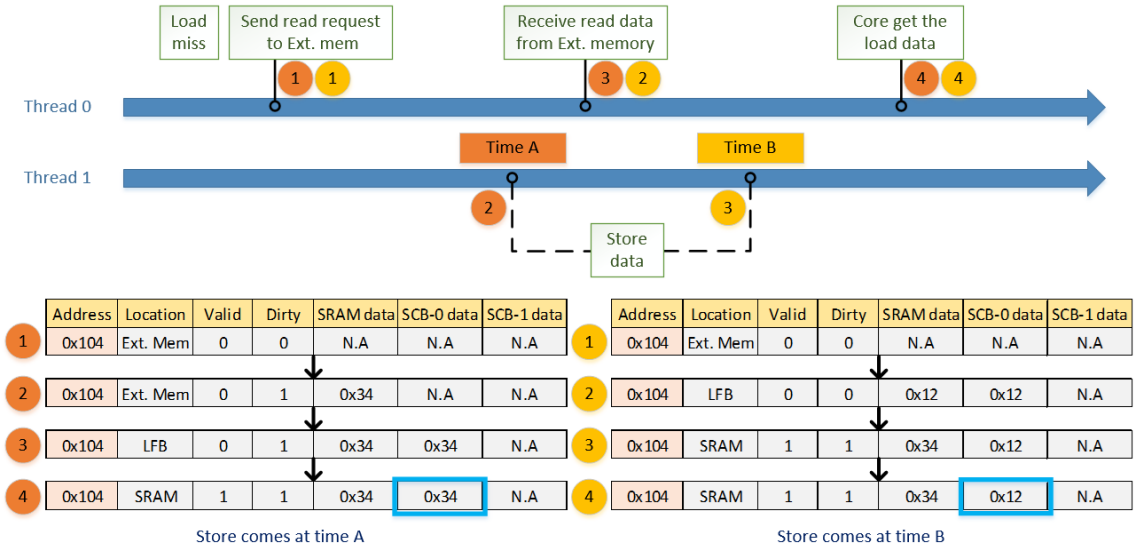


Figure 8. The Flow Chart of a Multi-thread Data Access Example Using CAMEL with Location

A. Precise data check

Thread-0 sends a load instruction (0x104) and cache miss. Then a byte unit is created with address 0x104 and the ‘location’ would be default value ‘NA’. Since thread-0 misses the cache, the location of the byte unit changes to ‘SCB’. When the read request is sent to external memory, the ‘location’ of the byte unit would change to ‘Ext. MEM’, as shown in Figure 8 step-1. Then thread-1 sends a store instruction to the same address (0x104). As the address’s byte unit already exists and the coming instruction is stored, the data will be saved in the byte unit. After receiving the read data from external memory, CAMEL model will change the ‘location’ to LFB, and change the ‘location’ to SRAM when the cache line is written into the SRAM. Each thread will preserve an extra info ‘SCB data’ for SCB, which is the highlighted by blue column in Figure 8.

‘SCB data’ is the data compared with RTL’s load data, and the order of the store instruction and the read data decides which data need to be placed in ‘SCB data’. The different cases are described on step-2 and step-3 in Figure 8. If the store instruction comes before the read data, ‘SCB data’ will include the store instruction’s data. If the store instruction comes later than the read data, ‘SCB data’ will not include the store instruction’s data. With the help of location, the example in section III which may get two load data values comes to a certain value in current scope.

B. Rationality check of request to external memory

With the help of location, if any read request appears on the interface to external memory, the rationality of the request can be checked at Figure 8 step-1. If the byte unit’s location is ‘SCB’ or ‘STB’, which means there are instructions executing at background, the read request is reasonable, otherwise it means the address is not waiting to be sent and will be considered error.

A cache’s space is limited, and if a new cache line needs to update into L1SYS while there is not enough space, cache replacement or eviction will happen. In this process, the new cache line will replace an existed cache line and this should also be updated to CAMEL model. This process can be predicted when the same index address’s read data comes back from external memory. And in current L1SYS, when a cache line is replaced in cache, L1SYS will send ACE write request to inform the coherent bus. The testbench will

check between the prediction and the ACE write request to check the replacement and eviction process. CAMEL will also be updated along the process and always preserve the valid bytes align to cache.

C. hit/miss check

The 'location' and 'valid' in byte unit can be referenced to check each instruction's hit/miss. Before the cache line is invalidated or evicted, we can expect core accessing to relevant address should get cache 'hit'. Even the 'valid' in the byte unit is not 1, if the 'location' is reasonable, some store command can also be seen as cache hit. In this way, cache's hit/miss check precision can be increased from nearly zero to 80%.

There is no doubt that the more information added to CAMEL, the more complex CAMEL will be. But with the help of extra information, the checking precision and more checkers can work on the RTL design. The simple testbench and the precise checkers, you can't have it both ways. However, with the characteristic of CAMEL, this can be achieved step by step. At early time of verification, simple CAMEL structure is easy to build and can work quickly to find RTL bugs. As time goes by, more information can be added for more checkers, the process is under control.

V. CAMEL IN TESTBENCH

Previous testbench builds all the buffer models in RTL design. Relying on the cycle-accurate models, the testbench can not only check normal instructions' data correctness, but also check special instructions' process like cache operation and fence. Comparing to previous testbench, current testbench based on CAMEL mainly focuses on data correctness and address relating check. But for special instructions like cache operation and fence, they cannot be fully verified only based on CAMEL. Under such circumstances, some standalone models will be added to cover the point that CAMEL cannot cover. The current testbench's whole picture is shown in Figure 9.

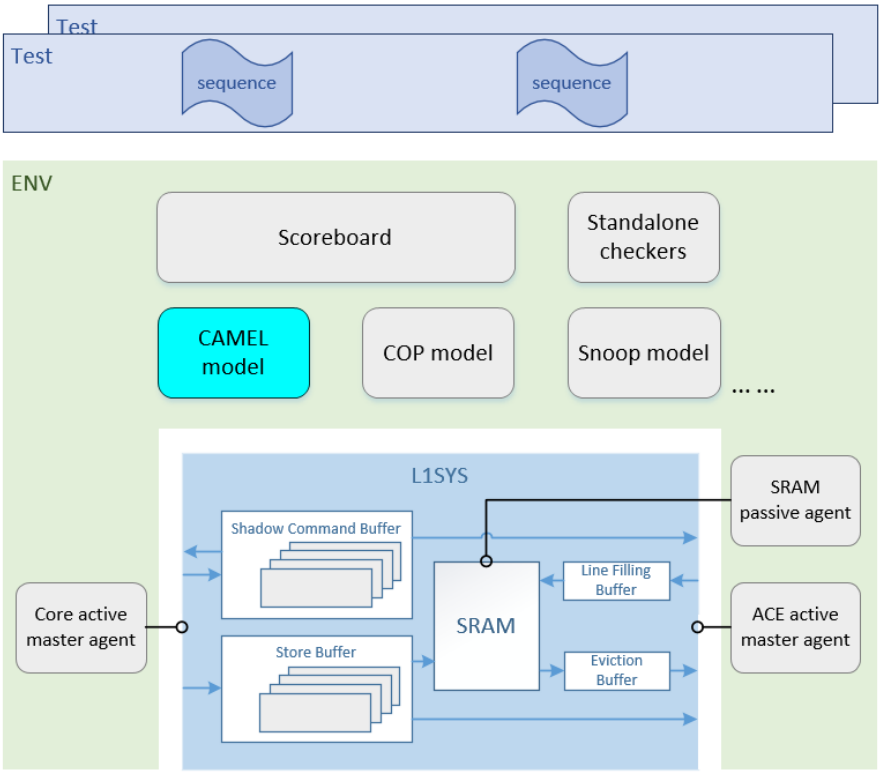


Figure 9. Overview of Current Testbench

Three VIP agents bind on L1SYS design boundaries, they are active core master agent, active ACE [1] slave agent, and a monitor on SRAM.

Besides normal data access, Cache Operation (COP) is also supported in L1SYS. This is an operation which can change cache line's state by special instruction. Normally when the core needs to change the current program, COP instruction may be sent to write all the dirty cache lines to external memory and

clear the cache. Snoop request is from coherent bus and can also change the cache line's state. These two processes change L1SYS's cache line state besides the normal data process, so two standalone models are built to monitor and check them and also update CAMEL.

A. Core active master agent

At this point, the instruction's address hit/miss will be checked based on CAMEL. If cache hit, check load data vs. CAMEL or update store data into CAMEL. If cache miss, a byte unit will be created and change 'location' to proper one, prepare for background check.

```
if (camel.hit(tr.addr)) begin                    // query CAMEL for hit/miss
  if (tr.cmd_type == LOAD) begin
    this.check_load_data(tr);                    // check load data in sub-function
  end
  if (tr.cmd_type == STORE) begin
    camel.write_data(tr.addr, tr.data);          // update data and dirty to CAMEL
    camel.write_dirty(tr.addr, 1);
  end
end
else begin
  camel.add_addr_if_not_exist(tr.addr);          // create the unit byte if not exists
  if (tr.cmd_type == LOAD) begin
    camel.update_location(tr.addr, SCB);         // update location base on instruction type
  end
  if (tr.cmd_type == STORE) begin
    camel.update_location(tr.addr, STB);         // update location base on instruction type
  end
end
```

The procedures are for normal instructions coming from the core, they focus on data check only. The hit/miss check is another standalone checker.

B. ACE active slave agent

At this point, when read request command appears, the testbench will check the address's rationality. When the read data is sent back from external memory, the corresponding byte units will change 'location' to 'LFB' and update data, dirty and unique attributes. Testbench will predict if the line filling process will trigger eviction based on CAMEL and record the possible write cache line address. When the write request command appears, it will be compared to previous record.

```
if (tr.exact_type == READ) begin
  wait (tr.addr_status == ACCEPT);
  this.check_rationality_of_read_command(tr);        // check rationality base on location in CAMEL
  camel.update_location(tr.addr, EXT_MEM);

  wait (tr.data_status == ACCEPT);
  camel.update_location(tr.addr, LFB);
  camel.write_clean_data(tr.addr, tr.data);          // write data while not cover exist dirty data
end
if (tr.exact_type == WRITE) begin
  wait (tr.addr_status == ACCEPT && tr.data_status == ACCEPT);
  camel.read_data(tr.addr, ref_data);
  this.check_write_data(tr.data, ref_data);          // compare write data with data in CAMEL
  camel.write_valid(tr.addr, 0);                     // if evict from CAMEL, invalid bytes
  camel.delete_bytes(tr.addr);                       // if need delete from model, delete bytes
end
```

C. SRAM monitor

Whichever cache line is written to valid at SRAM interface will be monitored at this interface. After RTL design writes the cache line to valid, CAMEL will also write the byte units to valid. This point's monitor is not as important as the other two, even if without monitor at this point, the whole checker can also work. This point's monitor is mostly for debug convenience.

D. Standalone checker models

There are some standalone checkers which cover the point that CAMEL cannot cover. Take COP model as an example, when core sends special instruction like flush all to L1SYS, COP model will record the instruction. In this checker, the testbench references a RTL internal signal which represents the COP process is done. When the model receives the COP event, it will examine CAMEL model and predict the evict cache lines to external memory. The prediction will later compare with the write commands appearing at ACE interface and check the COP process.

In current testbench, checkers are broken down to different layers for different precision's check. The main checking point, data correctness, is guaranteed by CAMEL and others are covered by standalone models. All the models and checkers work together to verify L1SYS properly.

VI. Feedback Mechanism Based on CAMEL

Our previous testbench offers an abundance of feedback APIs [2] for effective stimulus generation based on the cycle-accurate buffer models. In current testbench, not only core instructions need to reference address from feedbacks, but also snoop requests need reference too.

L1SYS's design is complex when handling the conflict of foreground and background. For example, when a cache line miss is executed at background and fetch data from external memory, in the meanwhile, if the core accesses the same address cache line from foreground, the conflict appears. This kind of circumstances are the key point of verification, so feedback for effective stimulus is necessary.

The 'location' information in CAMEL can take the role of feedback function. By sending instruction whose address is located at SCB or other buffers, we can generate more effective stimulus on complex cases, and speed up the verification process.

```
rand  my_instruction     tr;
      int                base_addr, end_addr;

constraint reasonable_addr_range{
  tr.addr>=base_addr;tr.addr<=end_addr;          // constraint of transaction's address
}

task body();
  `uvm_create_on(tr, m_sequencer)
  get_addr_range();                              // get the address range by CAMEL model
  this.randomize();
  `uvm_send(tr)
endtask

function get_addr_range();
  int   addr_q[$];
  camel.get_addr_by_location(SCB, addr_q);      // query CAMEL model for certain location
  set_addr_range(addr_q, base_addr, end_addr);  // set address range by the result of CAMEL
endfunction
```

After those buffer models are replaced with CAMEL model, those feedback APIs are modified accordingly based on 'location' information. Such modification is transparent to external users, and the stimulus effectiveness is almost the same.

VII. SUMMARY

In this paper, we present an implementation independent cache verification model. The basic structure is rather simple and easy to maintain, 'location' information is added for more precise checking and stimulus generation. With the help of CAMEL model, the testbench can de-couple with RTL design and save coding and maintained effort dramatically, as shown in Table I.

Table I Comparison between Previous models and current models

	Previous	Current	
		CAMEL model	Other models
VIP number	5	3	
model number	6	1	2
model code lines	8513	1915	622

REFERENCES

[1] ARM Corporation. AMBA AXI and ACE Protocol Specification, 2019
[2] Chenghuan Li, et al., An Enhanced Stimulus and Checker Mechanism on Cache Verification, DVCon2019