# An Efficient and Modular Approach for Formally Verifying Cache Implementations

M, Achutha KiranKumar V
Visual Parallel Computing Group
Intel Technologies Pvt. Ltd.
Bangalore, India
achutha.kirankumar.v.m@intel.com

Abhijith A Bharadwaj
Visual Parallel Computing Group
Intel Technologies Pvt. Ltd.
Bangalore, India
abhijith.a.bharadwaj@intel.com

Bindumadhava S.S.
Visual Parallel Computing Group
Intel Technologies Pvt. Ltd.
Bangalore, India
bindumadhava.ss@intel.com

*Abstract*— **Formal verification (FV) has been a proven methodology to expose deep corner cases, guarantee high design confidence and generate significant Return on Investments (RoI) [2]. In any control path oriented design with data interpretation, cache implementations are pervasive and are challenging to exhaustively validate through traditional verification methods. Control path blocks like caches, though apt, are fairly complex for FV and the crux of the problem is the high complexity in writing an exhaustive FV environment. This paper elucidates a methodical approach for FV application on a cache block, which involves formally wiggling the design, defining and proving the right set of micro-architectural properties and deploying deep bug hunting strategies on the complete functionality. A plug and play formally optimized cache property library was developed and was deployed on several cache designs. We explain the results of the modular FV approach deployed on a complex Level-3 cache subsystem which unearthed more than 25 high quality bugs during the design process. The FV aided RTL development guaranteed a robust RTL turned into the repository, limiting RTL churns and saving hours of debug time.**

***Keywords— Formal Property Verification; Cache; Control path verification;***

## I. INTRODUCTION

Formal property Verification (FPV) exhaustively verifies a given design against a specification coded as properties. The FPV environment uses mathematical techniques to verify the assertions, given a set of constraints (assumptions) [1]. A passing proof indicates that all possible behaviors were exhaustively verified and the design adheres to the given specifications.

In processor design, cache implementations are prone to exhibit corner case bugs late in the design cycle due to the sizable state space they represent. This paper presents a generic FV framework for cache designs, which has been proven to weed out several corner case issues on multiple implementations. Section II of the paper elaborates on the cache design under consideration, the verification environment and a basic expectation of properties to be proven through formal methods. Section III provides an elaborate overview of the generic formal cache library created to assist the activity. Section IV will explicate the success of the activity, by articulating the ROI in terms of quality of bugs found, resources spent and reduction of the design churn post turn-in.

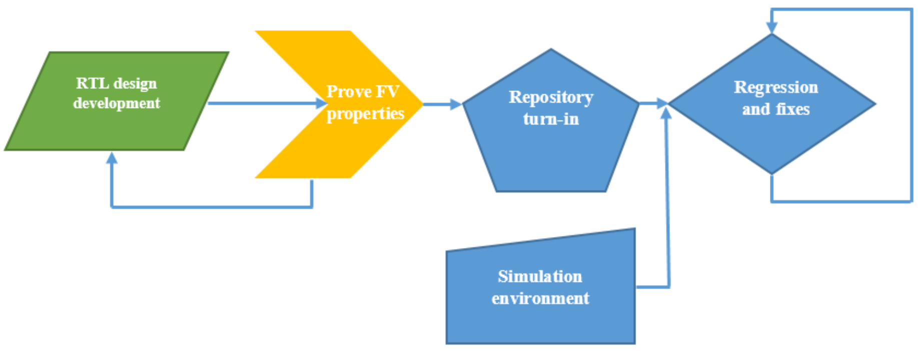

*Fig.1, Adopted FV Flow*

## II. DESIGN AND FV ENVIRONMENT

The design under consideration is a Level-3 cache (MESI protocol) used for all requests to the global memory. The design consists of two sections - a FrontEnd (FE - cache manager) and the BackEnd (BE - cache memory).

### *A. Cache FE*

The FE of the cache *(Fig.2)* coordinates various sequences of micro-operations initiated due to different cache accesses, while allowing for full out-of-order access support wherever possible. The core structure of the FE consists of a set of cycle processors, individual to a BE. A processor's lifecycle begins with allocation to receive an incoming cycle towards a BE. After allocation, it performs all necessary operations such as transferring data/state from the backing physical cache, any accesses to the Last Level Cache (LLC) /global memory, data returns to the client, and a variety of other internal operations. The requests have three different major access modes: non-coherent (traditional standalone), coherent (with LLC), and compressible (data stored locally uncompressed, but compressed to main memory). Each modes support several specialized cycle types that the design must be able to process in addition to traditional read/write cycles.

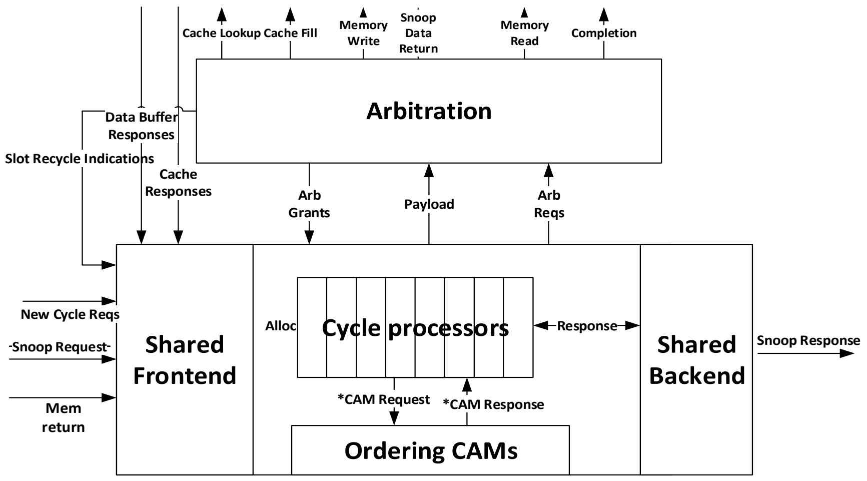

*Fig. 2, Cache FE*

### *B. Cache BE (Memory)*

The cache BE *(Fig.3)* is a generic reusable implementation of the physical cache memory. The block contains discrete memory elements to store the Cache Lines (CL) with corresponding TAG and a handler for memory management. The cache can store a maximum DEPTH number of unique entries.

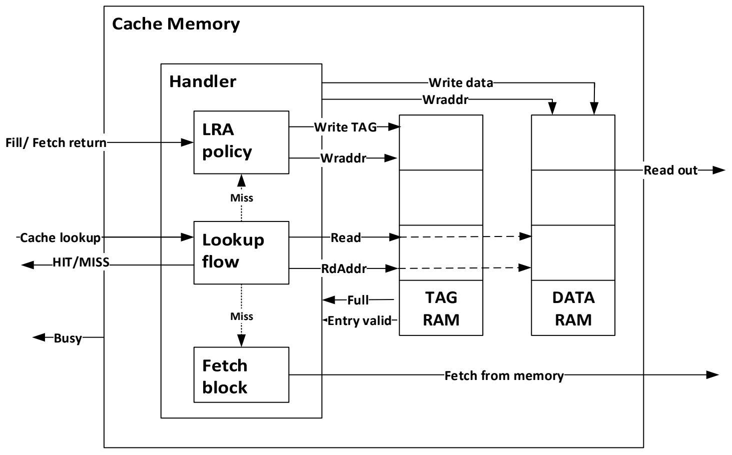

*Fig. 3, Cache BE*

The handler schedules the different CL access operations such as CL reads, lookup response, CL writes/evicts and main memory fetches. Cache reads will result either in a HIT (CL exists) or a MISS (Fetch needed), and the indications are LIVE (in the same cycle as the lookup). The lookups are based on the address of the CL. The TAG RAM stores the MISSed CL address and the DATA RAM stores the corresponding CL. The cache uses a counter based LRA policy which the Fetch-returns use to evict a CL from the cache, thereby writing the incoming data to the cache.

### C. *Environment constraints and requirements*

The controller and the cache assume certain behaviors to be respected at the interface. Non-compliance with such behaviors would cause unnecessary design hangs while subduing actual RTL issues. This section provides a summary of the mandated requirements

1. Interactions with external agents: Every interaction that either the controller or the cache initiates with any external agent must be acknowledged individually. Else, the design will wait indefinitely, resulting in Live-locks.
2. External memory (Main): The main memory is expected to be slow in responding and the responses can be out of order. In addition, each memory request is accompanied by a 'Tag', which needs to be returned with the associated data. The Formal environment must be able to provide out of order and appropriate acknowledgements, failure of which will result in data corruption or hangs.
3. Credit management: The design implements credit based FIFO at the input, which should not be overrun in the absence of credits, else such transactions will be dropped.

## III. FV ENABLING ON CACHE DESIGN

This section will elucidate the modular methodology adopted to formally verify both the FE and the cache BE.

### A. *Initial Design Exercise*

The first leg was to start early in the design process and enable FV on bifurcated design chunks of increasing complexity. The activity was started when the design was at a decent health, compiled and ready to be verified. The several Finite State Machines in the design were the initial individual targets for structural checks.

Since the FSMs were written in generic fashion, a GUI was created to automatically generate and collate generic FSM based properties when pointed to the RTL [3]. This process greatly simplified the initial wiggling as the readily generated properties required very little investment. The property failures at this stage pointed us to either an under-constrained environment or simple RTL issues. Some of the properties were:

- FSM State must never be X.
- A state must never attempt multiple arcs simultaneously.
- All valid arc transitions possible
- Invalid arc transitions never occur
- Covering that all the states are visited
- Eventually the state machine moves out of the current state

For example:

- NO_MULTIPLE_ARCS : assert property ( (FSM_now == state_X) |-> $onehot0(FSM_nxt == state_Y, FSM_nxt == state_Z) );
- NO_DEADLOCK : assert property ( (FSM_now == state_X) |=> s_eventually (FSM_now != state_X) );

### B. *FV Plan to conquer*

The next stage of the process was to divide the design and conquer. The Front End FV was chosen to be the first in the order of events, followed by the cache backend FV.

1. *Cache FE FV*

With modularity in focus, the design was bifurcated into chunks of increasing complexity and functionality was chosen as the criteria for chunk 'size'. The design was broadly divided into Non-Coherent cycles, Coherent and Compressible cycles *(Fig.4)*. The operations were broadly classified as Reads from a CL, Write to a CL and Atomics operations.

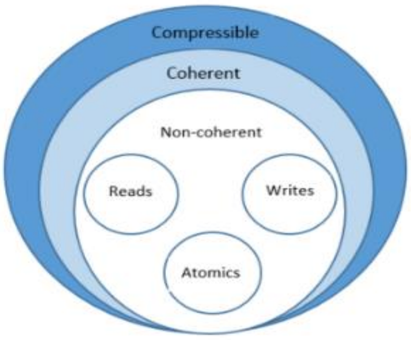

*Fig. 4, Design breakup*

The initial chunk chosen was to be relatively functionally simple yet excite a significant portion of the design. Therefore the first ‘chunk’ chosen were the Non-Coherent reads, thereby initially constraining the design to execute only NC reads.

The wiggling stage exposed an under-constrained environment, but a pattern arose – The FE would continuously interact with external agents through a series of transactions, while adhering to a ‘Request (REQ)-Acknowledge (ACK)’ protocol. One such transaction is shown in *Fig.5*.

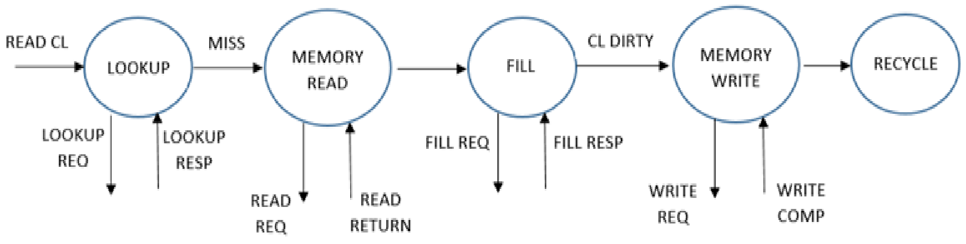

*Fig. 5, CL read flow*

These behaviors were modelled as a SystemVerilog macro which considered a REQ (Request to the macro) to create an ACK (acknowledgement), constrained by a DELAY (specified by the user). Several debug hooks were also be integrated into the macro. The implementation is as follows –

| Type | Description |
|---|---|
| Assume | A REQ is always followed by an ACK |
| Assume | ACK eventually appears only if requested (REQ) |
| Assume | No ACK for DELAY number of cycles after a REQ |
| Assume | If several different REQs request simultaneously for the same ACK, each REQ is individually ACK’d |
| Assert | No spurious REQs when a REQ is already in progress |
| Assert | If PS is the present state during a REQ, then NS should be the next state after an ACK |
| Cover | REQ and an ACK sequence appears |

The assertions for the FE were written with a generic intent and hence could be ported across several designs. Since the macro controls and observes all of the micro operations that a cycle processor would attempt, it was found prudent to integrate the macro with most of the checks. Some of those assertions were as follows –

- Micro operation transition checks: Checks to verify that the transitions happen according to the spec. A transition would be triggered by a response from an external agent/cache BE and therefore can be integrated into the macro.
- No spurious requests: Checks to verify that the FE does not launch spurious/multiple requests to either an external agent/cache BE which could potentially corrupt the cache data.
- BE does not recycle when active: Whenever a micro-op is active, the FE should not relinquish its possession of the CL, which would result in deadlocks.
- Lookups forwarded to the BE only once: Whenever a micro-op is active, the FE looks up the data in the BE only once, in order to avoid collisions.

One simple assertion that was found to be effective across several designs was that ‘Any cycle that locks onto a CL should eventually result in the CL being relinquished’. A liveness assertion exercising this intent would hold true for any micro-op and would help catch any Live-lock or Deadlock scenarios that might arise due to the interacting states. Such an assertion would hold even when the verification scope was augmented.

```
CL_GETS_RELEASED: assert property (CL_acquired |-> s_eventually(CL_got_releasd) );
```

Once FV on a scenario did not result in the design hanging, the next chunk was taken up. During the FV of each chunk, only one CL was allowed to be allocated. To achieve better convergence, the previously verified functionality was disabled while executing the modular assertions on the new ‘chunk’.

### 2. *Cache BE FV*

While analyzing several cache implementations in the Intel GT, it became apparent that the formal properties to be written to verify cache behaviors could be classified into two strata. The first class included safety (functional) properties which were generic enough to hold true for all cache implementations, irrespective of the replacement policies. The second set were implementation specific (performance), which would vary according to the CL replacement schemes employed.

In order to reduce the complexity of scripting the Formal properties, a Formal cache library was developed which included properties that were classified in the aforementioned manner. Each property was implemented as a SystemVerilog macro for maximum reusability. The scope of this paper will be limited to the safety properties and performance properties for a cache system using LRU replacement policy. A succinct description of the properties are given below and due to space constraint, the macro implementation is left to the user.

*Safety properties*:

- `EVERY_LOOKUP_ACKNOWLEDGED: Every lookup / allocation request from the FE warrants an acknowledgement from the cache, in the same cycle.
- `READ_RETURN_IFF_LKUP: A read return gets posted to the FE only when the FE has taken ownership of the cache and after a Lookup is posted.
- `LKUP_RETURNS_ONLY_M_CL: Only M number of CL(s) is(are) returned in a lookup cycle, where M is an argument to be passed to the macro.
- `FIRST_DEPTH_LKUPS_MISS: Right out of reset or after a cache Flush (cache invalidate), the first DEPTH number of different lookups always return a MISS. This assertion is critical in checking both the reset and flush functionality.
- `EVERY_HIT_RETURNS_DATA: Every lookup that is a HIT results in a read return being sent to the FE in M cycles (specified by the user).
- `HIT_DOES_NOT_WRITE: Any lookup that is a HIT will not result in a write_enable to the TAG_RAM
- `EVERY_MISS_GETS_WRITTEN: Every address lookup that's a MISS results in that address being written into the TAG RAM.
- `FETCH_ON_EVERY_MISS: Every MISS results in a fetch being posted after N cycles, specified by the user. This assertion will also check that a fetch will appear only on a MISS and only M number of times (also specified by the user) during a lookup.
- `MISS_AFTER_FLUSH: Any lookup after a flush returns as a MISS.

A very important behavior that any cache has to adhere to is that upon repeated lookups of a particular address, the data returned has to be consistent unless there was a Flush. This intent can be verified by either painstakingly tracking data for several addresses or by cleverly leveraging the Formal environment, whose implementation is as follows –

| Type | Description |
|---|---|
| Assume | If not (reset \| flush), $stable(ADDR_FV) && $stable(CL_FV) |
| Assume | If fetch_return and fetch_addr == ADDR, fetch_data == CL_FV |
| Assume | If fetch_return and fetch_addr != ADDR, fetch_data != CL_FV |
| Assert | Upon lookup, if lookup_address == ADDR_FV, then lookup_response == CL_FV |

- Two undriven variables are declared in the RTL, one the width of the lookup address and one the width of the CL data (ADDR_FV and CL_FV) and are made to be stable unless flush or reset is seen.
- The Formal environment is still allowed to decide on the variables' values (Free variable in formal terms).
- IFF a fetch is posted for the address ADDR_FV, then only the data returned is to be CL_FV, else random.
- Now the assertion intent is simple. Every lookup for the address ADDR_FV will return only the CL_FV as the data.

Since all lookup operations are random, these properties were been found to be enough to exhaustively prove the safe functionality of the design.

Performance property for LRU policy:

A simple yet powerful concept was found enough to exhaustively verify the LRA logic. The intent was – 'If a lookup for an address returns as MISS, then until further DEPTH lookups MISS during which the same address is looked up, the status will be HIT'. The implementation was done as follows –

- To verify the LRA policy, a Formal random variable representing the lookup address was used. A constraint similar to VARS_STABLE was used to hold the value constant, yet random.
- A flag (lra_fv) is used to indicate that the window for LRA checks.

  - The flag initializes to 0 out of design reset or on a flush.
  - The flag sets to 1 when a lookup for ADDR_FV returns as a MISS.
  - The flag resets when counter miss_fv == DEPTH
- A counter (miss_fv) is used to store the number of lookups that MISS.
  - The counter initializes to 0 out of design reset or on a flush.
  - The counter increments whenever lra_fv == 1 & lookup == MISS
  - Counter resets to 0 when lra_fv resets.

The assertions now become –

- HIT_CONDITION: assert property ( lookup && lkup_addr == ADDR_FV && lra_fv |-> HIT)
- MISS_CONDITION: assert property ( lookup && lkup_addr == ADDR_FV && !lra_fv |-> MISS)

These assertions were able to find holes in replacement logic across several cache designs employing LRA policy.

### C. *Bug hunting*

Once the complete design functionality was enabled and the formal environment was allowed to decide upon operation type, the verification scope augmented greatly and convergence issues became the consequence. Therefore, Bug hunting, a semi-formal strategy, was adopted to find the deep rooted bugs.

The gist of bug hunting is to employ specialized semi-formal engines to simulatively tunnel deep into the design and break the assertions by executing formal analysis at the simulation boundary [1]. The tunneling of the design is steered through formal cover points written to reach specific sequentially deep scenarios. This method helps catch deeper bugs, which would have consumed extreme resources in mainstream FV execution.

Several modifications were made to the design environment to suit the bug hunting process. The CL control structures were driven through a FV environment controlled free variable, a structure, containing all the control fields. The complexity of the assertions used were also reduced

1. Firstly, the response (ACK) to any REQ was fixed to arrive within 7 cycles from the request (delay=3).

   REQ_EVENTUALLY_ACKD: assume property ( @ (clk) (REQ) |-> strong(##[delay:(delay+4)] ACK ) ); \

2. Secondly, all liveness assertions were made deterministic. Based on the design knowledge, a CL was expected to execute all transactions and recycle within a hundred cycles.

   CL_GETS_RELEASED: assert property (@ (clk) CL_acquired |-> strong (##[1:100] (CL_got_released)[->1] ) );

The REQ-ACK macro covers written to reach each combination of a REQ and an ACK provided cover points for the deep interesting scenarios. The framework so developed aided in unearthing several sequentially deep issues.

The design was prone to slip into deadlocks as it contained several FSMs. Liveness assertions written to check that FSMs eventually move out of a state would easily catch deadlocks. But such assertions are always the bane for convergence [1]. Therefore in this strategy, we try to push the FSMs to a state where it does not see progress for a long time.

NO_DEADLOCK: assert property ( (FSM != STATE_1) [*DELAY] );

Directed by the design knowledge, if the DELAY is fixed appropriately, any failure thrown will have high probability of being a deadlock. A cover is proven from this advanced stage to find a way out, failure of which indicates a deadlock: DEADLOCK_CHK: cover property (FSM != STATE_1);

## IV. RESULTS

The productiveness of this activity can be evaluated by examining three categories –

### A. *Quality of issues uncovered*

In the span of about 2 months over 30 different initial cycle types composing all the non-coherent and coherent cache accesses were fully covered, including all permutations such as FE-BE interactions, evictions on fills, data integrity and so forth. The various FV strategies used uncovered 25 different issues, ranging from minor typos to improper ordering assumptions in RTL that would not be exposed in simulation, of which three are discussed here.

1. The first issue discussed here was caught by the CL_GETS_RELEASED assertion. Non-coherent data was to be written into the L3 cache. The CL to be written was in the 'Exclusive' state (shown by ACK and Response-type collaterals during lookup). After filling the CL, the FE does not relinquish control. This is because the FE erroneously thinks that the CL was dirty and is waiting to write it to memory (shown by MEMWR pending).

   This issue was discovered by using the bug-hunting strategy. Properties were written which covered scenarios such as CL full, FE_STATE does not go into the RECYCLE_AWAIT for some cycles

   CL_FULL_cover: Cover property (!CL_FULL |-> ##[10:15] CL_FULL)

   FE_STATE_assert: Assert property ((FE_STATE != RECYCLE_AWAIT)[*50])

   As shown above, a mix of both asserts and covers were used to guide the bug-hunting engines to come up with the deadlock scenario as shown in Figure 6.

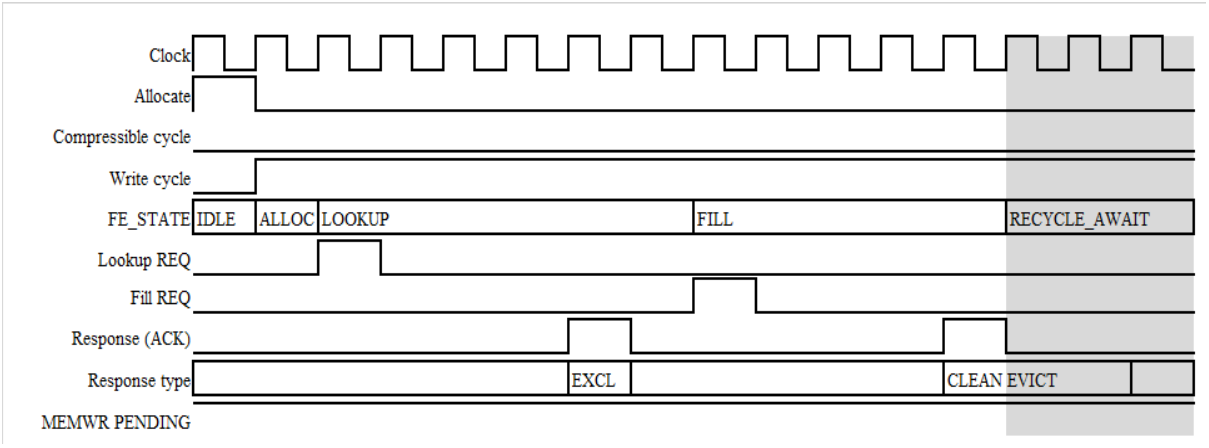

*Fig. 6, Evict issue*

2. The second issue discussed here was caught by the deadlock detection strategy. The FE_STATE state machine holding the CL status was pushed to stay in the RECYCLE await state for a long time.

   FE_STUCK: assert property (FE_STATE != RECYCLE) [*50] );

   Once the FV environment threw up a CEX, a cover to escape the scenario was tried to be proven from the functional bound of the CEX. When the cover failed to prove from the CEX depth, upon debug, it came to light that the FSM was in a deadlock. A piece of coherent data, when looked up, was found not to be present in the cache. Upon getting a MISS response, FE proceeds to do a read from the memory to fetch the data. Memory responds with an error signal which deadlocks the FE without recycling.

   FE_DEADLOCK_CHK: cover property (FE_STATE != RECYCLE) ;

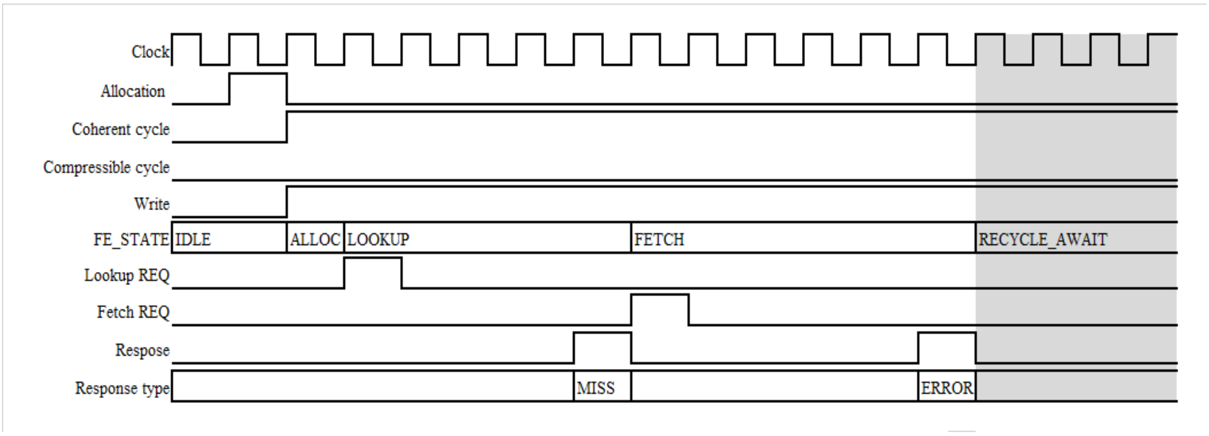

*Fig. 7, Deadlock issue*

3. The third scenario was caught by the READ_RETURN_IFF_LKUP assertion. A flush to the cache BE was being erroneously handled. A flush pulse asserted at the input would take two cycles to flush the entire cache. Here, a flush was asserted in the first cycle, followed by a lookup which returned as a MISS. The lookup, instead of being held gets partially executed. The CL does not get marked as written, but the TAG still gets written into the TAG RAM which launches a read to the DATA RAM. A fetch gets executed but the CL slot is always marked empty.

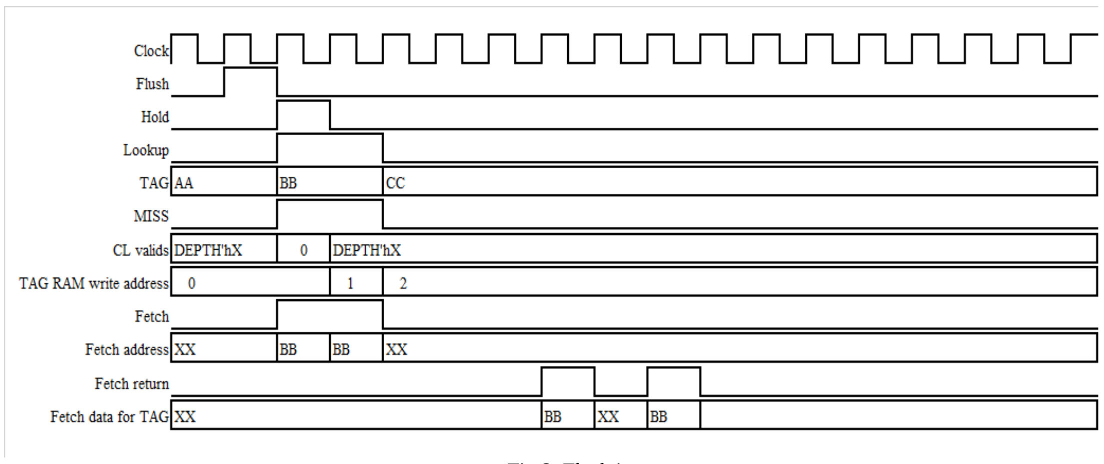

*Fig.8, Flush issue*

## B. *Return On the Investments (ROI)*

Using FV as the gating function rendered the design robust during the turn-in. When the design was integrated into the cluster, around a hundred issues were uncovered, of which only 3 were on the cache block. All three were interface issues, unrelated to the functionality verified through Formal. Without FV, the traditional simulation methodology found around 70% of the bugs after the turn-in. Coupled with the added functionality of the entire cluster, the debugs became extremely difficult. Using FV before turn-in reduced the debug complexity to a great extent as the most intricate issues spanned no greater than a hundred cycles. 80% of the issues found were before turn-in which reduced code churn to a great extent.

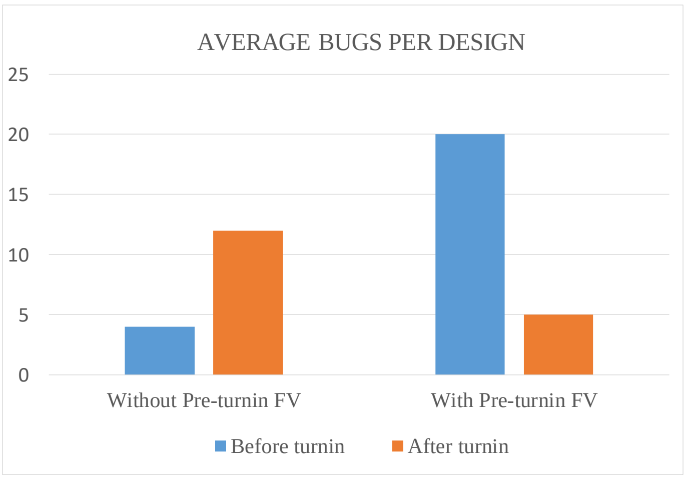

*Fig. 9, Bugs chart*

The FV regressions still continue to be run on the design post check-in, alike any simulation strategy. While it boosts the confidence on the design, it also helps in catching the bugs as soon as they surface. The relative investment on FV is considerably less compared to the simulation counterpart. The absence of bugs in the cache units that employed FV motivated the management to mandate FV for many such blocks that are a part of the feature turn-in. While FV guaranteed the design correctness, it also served as a huge motivator for deep bug hunting on many more cache units.

## V. FUTURE WORK AND SUMMARY

The primary goal of this activity was to establish the importance of FV on cache designs and thereby to propose a robust modular approach for exhaustive validation. As clearly seen from the results, FV adds real value, as an alternative to early simulation, especially when the simulation environment is unstable or unavailable. Formal methods had been extensively used in the L3 cache subsystem primarily for arbitration validation. The cache FV being the first time formal was applied to any other type of logic in L3, and the complexity being very high, the ROI was considerable. The scope for future work includes expanding the number of cache units that use FV as the pre turn-in gating activity and to make use of FV to verify more complex functionalities. The designers are now motivated to embrace the methodology, based on the results from this unit. The design managers are now open to demand FV for new units bring up and as a result several more new units have currently signed up for FV.

## References

[1] “Formal Verification: An Essential Toolkit for Modern VLSI Design” by Erik Seligman, Tom Schubert, M V Achutha Kiran Kumar, Elsevier Publications, 2016

[2] M, Achutha KiranKumar V, Erik Seligman, Aarti Gupta, Bindumadhava, Abhijith Bharadwaj, “Making Formal Verification Mainstream: A Graphics Experience”, DVCON USA, February 2017.

[3] M, Achutha KiranKumar V, Ss Bindumadhava, Abhijith Bharadwaj, “Democratizing FPV”, DVCON India 2016

[4] Kenneth McMillan, “Modular specification and verification of a cache-coherent interface” Formal Methods in Computer-Aided Design (FMCAD), October 2016, pp. 109-116

[5] Barada P. Biswal, Anurag Singh, Balwinder Singh, “Cache coherency controller verification IP using SystemVerilog Assertions (SVA) and Universal Verification Methodologies (UVM)”, International Conference on Intelligent Systems and Control (ISCO), Vol 11, pp. 21-24, January 2017.

[6] Freek Verbeek, Pooria M. Yaghini, Ashkan Eghbal, Nader Bagherzadeh, “ADVOCAT: Automated deadlock verification for on-chip cache coherence and interconnects”, Design, Automation & Test in Europe Conference & Exhibition (DATE), March 2016, pp. 1640-1645.

[7] Swadhesh Kumar, P K Singh, “An overview of modern cache memory and performance analysis of replacement policies”, IEEE International Conference on Engineering and Technology (ICETECH), 2016