Formal Proof for GPU Resource Management

Jia Zhu
Advanced Micro Devices
#46, No. 1387 Zhang Dong Road
Pudong Shanghai, China 201203
Jimmy.Zhu@amd.com

Chuanqin Yan
Advanced Micro Devices
#46, No. 1387 Zhang Dong Road
Pudong Shanghai, China 201203
Fatter.yan@amd.com

Nigel Wang
Advanced Micro Devices
#46, No. 1387 Zhang Dong Road
Pudong Shanghai, China 201203
Nigel.Wang@amd.com

***Abstract* - A GPU is a processor with a lot of compute units which processes large blocks of data in parallel. A resource management block is designed to manage all the compute resources. It's a big challenge for verification engineers to ensure that every request arbitration, resource allocation and de-allocation are correctly handled because a GPU has a single-instruction-multiple-data (SIMD) architecture and different graphics/compute tasks require different resources. It is hard for simulation-based verification to ensure that all possible scenarios are covered after running thousands of test cases. Formal verification, on the other hand, exhaustively explores the mathematical representation of the design to uncover all the incorrect behaviors that would break the design. However, the state explosion issue caused by design complexity becomes the most challenging problem in formal world. Since this block manages several big resource pools, the complexity issue get more critical for convergence. This paper presents the methods to reduce the design/testbench complexity by applying different formal techniques like blackbox, symbolic random variable, and abstraction to achieve formal proof for the whole block. Another big challenge is that resource allocation result isn't deterministic, which blocks a complete end-to-end formal check. Mutation coverage is introduced to expose our verification hole for resource leakage issue. We define functional mutation coverage and structural mutation coverage as complements to our original sign-off list. Synopsys's Certitude is used for our mutation coverage methodology.**

Keywords: *formal verification, VC formal, Certitude, formal sign-off, bounded-proof, mutation coverage, abstraction, end-to-end checker, legality checker, single-instruction-multiple-data.*

I. INTRODUCTION

Modern GPUs are becoming more and more like general-purpose-processors (GPGPUs) with a lot of compute units to process large blocks of data in parallel. To maximize the computing parallelism, usually a GPU is based on single-instruction-multi-data architecture, which requires different kinds of resources to support data processing. For example, a typical GPU needs:

1) scalar/vector general-purpose-registers to store shader instruction operands
2) local data share for inter-threads data exchange
3) barrier resource for multi-threads synchronization
4) computing slot for data computing
5) scratch buffer for temporary storage

A workgroup (A set of work-items from the same dispatch) cannot be launched onto GPU computing units until all required resources are allocated properly. Thus, a resource management block is designed to manage all these different resources. It is a big challenge for verification engineers to make sure that every request arbitration, resource allocation and de-allocation are correctly handled because the design is timing/control intensive and easy to hide corner cases. It is hard for simulation-based verification to cover all possible scenarios even after running thousands of test cases because of the huge amount of concurrent working threads in the system. Moreover, even if there is no functional bugs in the design, it's still hard to know if the arbitration result can meet the design requirement. This challenge calls for a white-box verification, which is time-consuming and hard to maintain even with current advanced verification methodologies, like CRV (Constrained-Random-Verification) or UVM (Universal Verification Methodology).

Formal verification, on the other hand, exhaustively explores the mathematical representation of the design to uncover all the incorrect behaviors and corner-cases that would break the design [1]. Model-checking based formal verification is very powerful for verifying control blocks that work under concurrent circumstances. However, the state explosion issue caused by design complexity becomes the most critical problem in formal verification world. Since resource management block manages several big resource pools to record resource status, the complexity issue gets more challenges for formal verification convergence.

This paper presents the whole process to achieve formal proof for the resource management block in a GPU design, along with the plan, implementation and result.

Section II gives a brief introduction on the Design-Under-Test, the verification challenges we encountered and the reason why we turn to formal methodology for high-quality block verification. A bounded-proof depth calculation related part of the micro-architecture will be outlined.

Section III elaborates on the way to write a complete formal verification plan. The resource management block is an arbiter plus scheduler. The verification plan covers three primary aspects to verify this DUT: end-to-end check for data integrity and forward progress (a request can be served eventually), legality check for allocation/de-allocation protocol, and local check for arbitration results. Each output of the design will be checked by at least one property. The following figure describes the whole picture of our formal testbench.

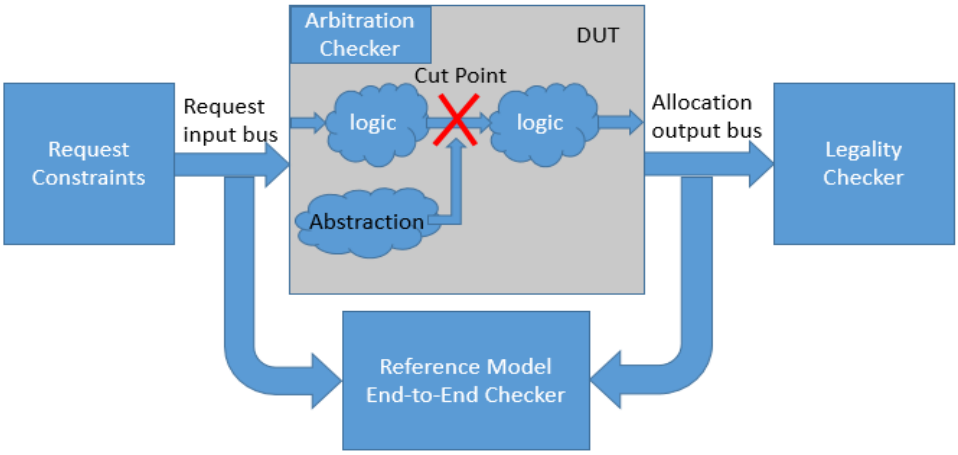

Figure 1 Formal Testbench for Resource Management

Section IV presents the complexity analysis based on VC formal (a Synopsys tool), and details the complexity issues we encountered as well as the methods to reduce the complexity for both design and formal testbench/model. This covers blackbox, symbolic random variables, reset abstraction, design scale-down, etc. After applying all these manual interventions, we can successfully achieve proof or bounded proof for the properties. Bounded proof depth is discussed based on micro-architecture and design implementation.

Section V discusses the resource leakage issue as well as mutation coverage as shown in Figure 2. Resource leakage issue was hidden by our abstraction strategies. We dig into the root cause and present our mutation coverage methodology to expose this verification hole. Functional mutation coverage and structural mutation coverage are defined to qualify our formal verification environment. Synopsys Certitude is used to facilitate this flow. Then we will list the mutation coverage guidelines for best return-on-investment.

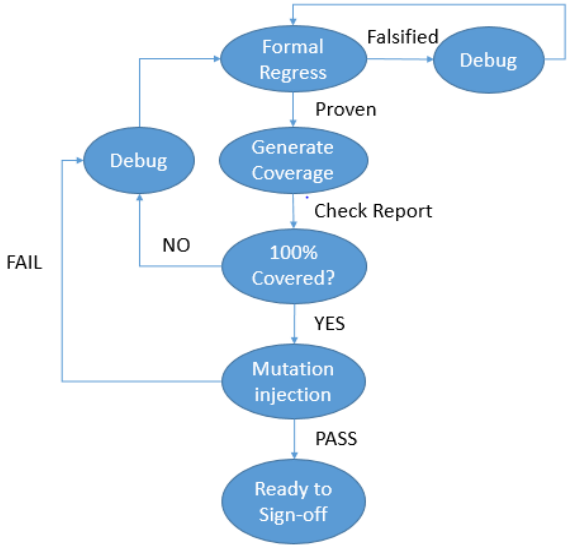

Figure 2 Formal Sign-off Flow

In the last section, we present what we learned from this case study, and give guidelines to performing mutation coverage in formal verification.

II. Design Overview and Verification Challenges

A. *Design overview*

The resource management block is responsible for accepting wavefront requests from multiple shader stages, analyzing the resource requirements of the individual requests, determining which requests can be satisfied given the current status of available resources, and deciding on which wavefront, if more than one can be provided with resources, is to be given access to the compute processor. We have 6 different types of resources managed by this block which fall into two categories: per CU (computing unit) and per SIMD (single-instruction-multi-data). The management for each type of resources are independent. Every CU contains multiple SIMDs. For the sake of confidentiality, real resource type names are represented by A/B/C/D/E/F and the accurate number of each resource is not listed.

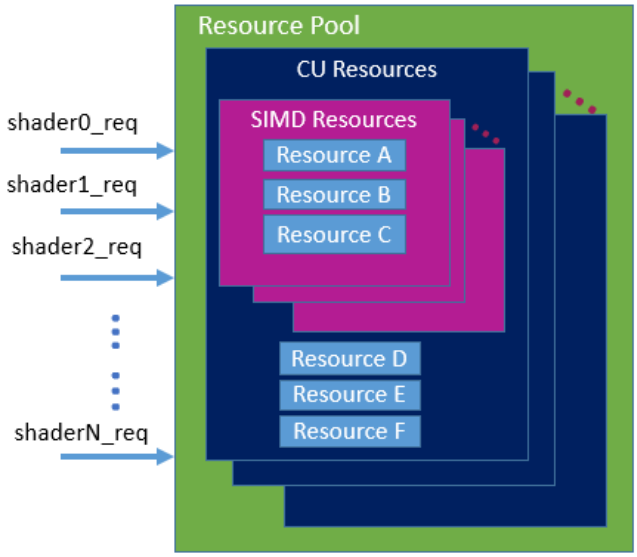

Figure 3 Resource Hierarchy Diagram

As shown in Figure 4, a bit mask (scoreboard) is used to keep track of which resources are available for allocation. Allocation requests are deemed to fit or not fit based on the maximum group of available resources. If the allocation request is found to fit and it is subsequently granted, then a group of bits in the mask are reset to indicate the resources are in use, and no longer available for allocation. When doing de-allocation, on the other side, mask is updated to set the status of the no longer needed resources back to available. When all wavefronts have completed, the bit mask will return to indicate all resources are available.

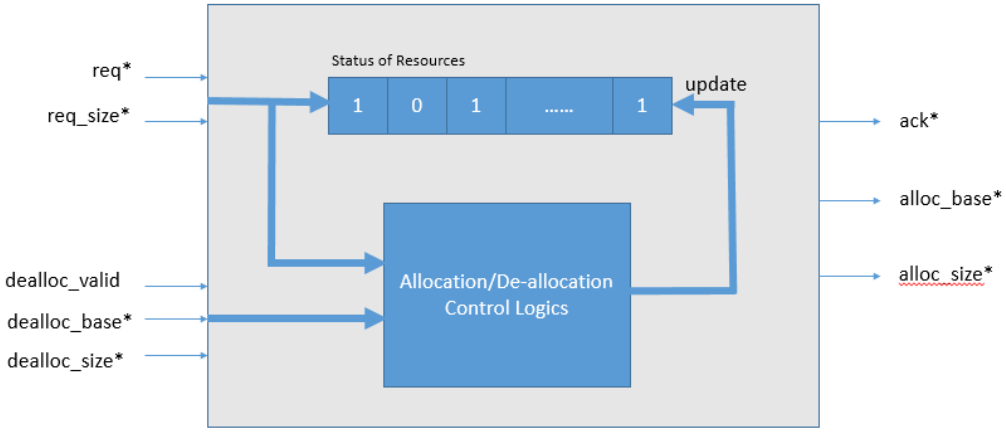

Figure 4 Resource Management Block Diagram

B. Verification challenges

For this block, there are three main verification challenges for traditional simulation-based solution:

1) It's hard to achieve verification convergence because of the extremely big resource pools (need thousands of bits to record all resources in our design). Constrained-random-verification doesn't help much.
2) The arbitration result can't be predicted accurately because it's highly timing related. Concurrent allocation/de-allocation requests from different shaders cause non-determinism. Black-box verification can't provide us with enough observability.
3) Many configuration modes should be walked through.

Formal verification is a systematic process of ensuring, through exhaustive algorithmic techniques, that a design implementation satisfies the requirements of its specification [1]. These characteristics, systematic, specification, exhaustive, perfectly solve the problems we meet in simulation. We use VC formal and Certitude from Synopsys to do formal verification for this specific DUT.

III. FORMAL VERIFICATION PLAN

With regard to the design's behavior, the most important feature we need to verify is the resource allocation and de-allocation function, but it's impossible for us to predict where the resource will be allocated accurately. The output alloc_base is non-deterministic, so the end-to-end checkers can only simply check that the output alloc_size must equal to what is requested. We can only rely on our legality checkers to ensure the alloc_base is reasonable. In this paper, we will not elaborate on end-to-end and arbitration checkers because their implementations are straightforward and have already been discussed in many formal verification papers, but mainly focus on legality checkers.

In order to define a complete set of properties, we followed the steps elaborated in Ref. [1]:

1. Define the formal specification interface.

At this step, we reviewed all the interfaces and defined which signals must be constrained or checked. Created a table to list all the signals and their functionality. So that we won't miss any interfaces.

2. Create requirement check list, which falls into five categories

1) Resource management legality checkers for every resource type
   - ✓ Resource A/B/C/D/E/F
2) End-to-End checkers.
   - ✓ Forward progress (a request can be served eventually), data integrity (e.g. alloc_size), etc.
3) Interface checkers which make sure interface protocol is legal.
   - ✓ e.g. wave_id must be consecutive in a workgroup for group requests.
4) Internal checkers which focus on arbitration mechanism.
   - ✓ e.g. LRU algorithm
5) AEP (Automatically Extracted Properties) [3] which is generated by VC formal.
   - ✓ e.g. index, counter overflow/underflow

As mentioned above, this paper will mainly focus on the legality checkers. At the step of planning, the following 3 resource allocation and de-allocation legality checkers are defined for every resource type:

1) The "non-available" resource can't be allocated.
2) The "available" resource can't be de-allocated.
3) Allocation and de-allocation for a specified resource id can't happen at the same time.

3. Define coverage goals.

Coverage is one of the most important metrics for verification sign-off. Our coverage goal includes 100% code coverage, 100% functional coverage, and 100% mutation coverage. Mutation coverage is added in a late verification stage to expose our verification hole, on which will be elaborated in section V.

IV. COMPLEXITY ANALYSIS AND ABSTRACTION STRATEGIES

A. Design Complexity.

VC formal is unable to converge on this block without manual intervention. Take Resource A as an example, we need "m" bits to record Resource A's status in one SIMD, so we will need "m*n" (which is very large in our design) bits if the number of SIMDs is "n" in a configuration. And the req_size is a "w" bits input for resource A. After crossing the "bit mask" and "req_size", there are $2^{w} * 2^{m*n}$ states need to be covered. If we add other types of resources and de-allocation into consideration, the number can be much bigger.

TABLE 1
DESIGN COMPLEXITY REPORT

Parameter	Value
Line of code (macro definitions are not included)	13018
Design units	182
Instances	260
Number of standard cells	223845
Number of input ports (bit)	2295
Number of output ports (bit)	1042

B. Abstraction strategies.

1. Checker abstraction.

Writing a reference model to record status of all scoreboards is impossible. Since the "bit mask" design is a symmetric implementation, each bit of the mask has the similar RTL structure. As a result, there is no need to check the whole scoreboard. We use symbolic random variable to check if ANY single bit in the scoreboard is allocated or de-allocated legally.

In this case, three symbolic random variables are defined:

1) cu_id ($\log_2$m bits, m is the number of CUs)
2) simd_id ($\log_2$n bits, n is the number of SIMDs per CU)
3) resource_id ($\log_2$w bits, w is the number of resource ids per SIMD).

Our formal testbench implemented a one bit referenced resource scoreboard which is updated according to every allocation and de-allocation information. Any illegal allocation or de-allocation will break the checker of the referenced resource scoreboard. Take Resource A's legality checkers as an example, with this abstraction, the state space of the scoreboard reduced from 2^{m*n*w} to m*n*w. The pseudo code we defined is as follows.

```
// Symbolic variables are randomized at reset and keep stable then.
symbolic_cu_id:     assume property (@(posedge clk) (##1 $stable(watched_cu_id)));
symbolic_simd_id: assume property (@(posedge clk) (##1 $stable(watched_simd_id)));
symbolic_res_id:    assume property (@(posedge clk) (##1 $stable(watched_res_id)));

// alloc_res_hit/dealloc_res_hit is valid only when allocated/de-allocated cu_id, simd_id, and res_base/size matches.
// Note: alloc_wrap/dealloc_wrap cases are not described here.
alloc_res_hit = (watched_cu_id == alloc_cu_id) && (watched_simd_id == alloc_simd_id) &&
  (alloc_res_id >= alloc_res_base) && (alloc_base_plus_size >= alloc_res_id);

dealloc_res_hit = (watched_cu_id==dealloc_cu_id) && (watched_simd_id==dealloc_simd_id) &&
  (dealloc_res_id >= dealloc_res_base) && (dealloc_base_plus_size) >= dealloc_res_id);

// scb is the referenced resource, updated according to every allocation or de-allocation information.
alloc_legality_check_p: assert property(@(posedge clk) disable iff(rst)
                    (alloc && alloc_res_hit |-> !scb)
);
dealloc_legality_check_p: assert property(@(posedge clk) disable iff(rst)
                        (dealloc && dealloc_res_hit |-> scb)
);
exclusive_alloc_dealloc_p: assert property (@(posedge clk) disable iff(rst)
                    $onehot0({(alloc && alloc_res_hit), (dealloc && dealloc_res_hit)})
);
```

To avoid illegal input scenarios, essential de-allocation constraints are added. The guideline for constraints is the simpler the better because of the following two reasons:

1) We don't expect the constraints increase the property complexity much.
2) We should avoid over-constraining.

Therefore, we only make sure the de-allocation operation for the symbolic resource is always legal. In this way, the de-allocation constraints only add 1 flip-flop to the property complexity. The pseudo code is as follows:

```
assume_symb_dealloc_legality: assume property (@(posedge clk) disable iff (reset)
        !scb && dealloc_valid |-> !dealloc_res_hit
);
```

The complexity report of alloc_legality_check_p property generated by VC Formal is showing in Table 2.

TABLE 2
ALLOCATION PROPERTY COMPLEXITY REPORT FOR RESOURCE A

Primary inputs (bits)	Flipflops (bits)	Operators
1858	16895	78938

2. Design abstraction

Although the design complexity is reduced a lot by the automatic COI(cone of influence) technique, VC formal can't converge directly. We take advantage of two design characteristics to reduce the complexity further:

1) The resource pool managed by this block can be scaled down to a smaller configuration.
2) Different types of resources are independent from each other. The fit logic of each resource will be ANDed later. So we can blackbox other resource when checking one specific resource type.

The following table shows the allocation property complexity report at different configurations for Resource A with other resources blackboxed. Please note only some configurations are listed for the sake of confidentiality.

TABLE 3
ALLOCATION PROPERTY COMPLEXITY REPORT AT DIFFERENT CONFIGURATIONS FOR RESOURCE A

Configuration(CU NUM)	Primary inputs(bits)	Flipflops (bits)	Operators
16	1031	5647	21948
8	951	3463	14478
4	911	2371	10401
2	891	1825	8377
1	881	1552	7409

With these abstractions, the proven depth is moving forward. Since the alloc/de-alloc size is variable and the resource pools are very big, we expect a very deep sequential depth to traverse all states of a scoreboard and resource search logic. We also increased formal run time to 10 hours, and reached a depth of 21, but VC formal still can't converge.

TABLE 4
PROOF DEPTH AT DIFFERENT CONFIGURATION

Configuration (CU NUM)	Depth	Time	Result
16	11	1h	Inconclusive
8	12	1h	Inconclusive
4	12	1h	Inconclusive
2	14	1h	Inconclusive
1	16	1h	Inconclusive
1	21	10h	Inconclusive

3. Reset abstraction and bounded proof depth calculation

The method we used to reduce required proof depth is to apply reset abstraction for the resource scoreboard (the bit mask). If we start model checking from a reset state, it requires VC formal to reach a deep sequential depth but if the scoreboard starts with an arbitrary legal status, then doing one allocation is enough to cover all of the possible allocation scenarios. The same goes for de-allocation. It's easy to calculate the number of possible initial states for each resource.

For SIMD resources:

$$\text{Number of initial states} = 2^{\text{per SIMD bit mask * number of SIMDs * number of CUs}}$$

For CU resources:

$$\text{Number of initial states} = 2^{\text{per CU bit mask * number of CUs}}$$

Based on this reset abstraction, the required proof depth can easily be determined. The proof depth equals the cycles we need to do one time allocation and/or one time de-allocation. To calculate the required proof depth, we followed the 6 steps described in Ref. [2]:

- Latency analysis

We wrote 2 covers, one for allocation and one for de-allocation. The allocation costs 4 cycles from request to allocation output. The de-allocation cover is a little bit different because de-allocation is input to the design. The only way we can make sure a de-allocation happens is to check the internal scoreboard of DUT. To cover de-allocation property, we need 5 cycles. Therefore, the initial proof depth is 5 cycles.

- Micro-architectural analysis:

As discussed in reset abstraction, one time allocation or de-allocation is sufficient.

- Covers for interesting corner-cases:

The interesting corner-case we need to cover for the resource management design is the allocation wraps around behavior when the (alloc_base + alloc_size) exceed the range. For example, consider a resource with just 8 bits to record status. If the initial state of resource is 8'b11000011 (1 means available), and the request alloc_size is 4, the request can still be fit. The alloc_base will be 6 and we have alloc_base + alloc_size = 10 > 8.

Based on the reset abstraction strategy, this can also be covered in one time allocation. The same goes for de-allocation.

- Formal coverage:

After running all of the legality checkers, formal code coverage generated by VC Formal is used to validate the proof bound. The 100% code coverage target was achieved at 5 cycles. This also increases the confidence that the inputs are not over-constrained.

- Failures seen during formal verification:

All the resource allocation or de-allocation related counter-examples exposed are less than or equal to 5 cycles.

- Safety net:

At last, we add 1 more cycle for a safety net. The required proof depth we defined is 6 cycles.

TABLE 5
RESULT OF LEGALITY CHECKERS FOR RESOURCE A

Property	Depth	Time
alloc_legality_check_p	6	27 minutes
dealloc_legality_check_p	6	2 minutes
exclusive_alloc_dealloc_p	6	28 minutes

Other resources legality checkers have similar results. The only difference is we need more time to achieve bounded proof for larger resource pools (e.g. the biggest resource pool needs almost 6 hours to reach depth 6), and less time for smaller ones. Most of the end-to-end checkers can be easily proven in a short time, like the assertion alloc_size equals to req_size. Some are complicated, but after applying tricks, they are proven. For example, there is a counter (the task id) in the design, which resets to 0 and increases at every wave acknowledge but wraps at N. VC Formal can't converge this to prove task id must be consecutive. However, if we split the assertion into N assertions (e.g. 0->1, 1->2, … , (N-1)->N,N->0), we can fully prove them in 2 hours. Those end-to-end checkers and local checkers will not be covered in this paper even if they are non-trivial.

V. RESOURCE LEAKAGE AND MUTATION COVERAGE

A. Resource leakage issue

It looks like the verification is finished because we guarantee the correctness of forward progress, arbitration, and legal allocation/de-allocation. We also meet all the sign-off requirements.

1) Every output signal is covered with at least one checker.
2) We achieved bounded proof for all of the checkers with depth 6.
3) We can achieve virtually 100% code coverage and 100% functional coverage (16/16, only for legality check) in 6 cycles.

However, those above can only ensure the reachability of our formal verification isn't broken by different abstraction techniques. Is our environment able to catch all possible bugs? In other word, do we have complete observability? To measure this, the next step in our verification sign-off flow is mutation injection. For the resource management block, one of the most important functions we need to check is whether there is any resource leakage issue. Therefore, we defined four types of error of our interests to challenge the formal verification environment:

1) The DUT allocates more resources than expected.
2) The DUT allocates less resources than expected.
3) The DUT de-allocates more resources than expected.
4) The DUT de-allocates less resources than expected.

After manual error-injection into the design, our properties are still proven by VC formal, which means there must be some holes in our verification environment or sign-off methodology.

With some analysis, we found that this hole resulted from reset abstraction. The reset abstraction is based on induction. Refer to Kripke structure [5] to represent our design M = (S, I, R, L):

- S: a finite state set of design
- I: initial state set, where $I \in S$
- $R \subseteq S \times S$, the set of state transitions
- L: a function that labels each state with a set of properties that are true at that particular state

Now let I equals to S, then we can start from any state and only need one transition to reach every possible s_next

$$\forall s \in S \; \exists s\_next \in S \text{ such that } (s, s\_next) \in R$$

This abstraction technique reduces the sequential depth to reach every legal state. However, since our legality checkers only check that the output alloc/dealloc behavior is legal, it cannot tell a correct s_next from an incorrect state s_leakage, if:

$$\forall s \in S \; \exists s\_leakage \in S \text{ such that } (s, s\_leakage) \notin R$$

Which means a wrong state transition between scoreboard states can escape from our property check.

Classic formal proof requires a complete end-to-end check. Say we have an ideal formal tool with unlimited capacity to search state space, the resource leakage issue will finally violate forward progress property because a resource request cannot be acknowledged any longer. In other words, we don't have immediate observability for this issue based on outputs legality check. After applying reset abstraction, we reduce the sequential depth for reachability but hide this issue. So for our specific DUT (resource allocation behavior is not deterministic, which avoids a robust end-to-end check), the root cause is that we need to provide immediate observability on scoreboard to ensure our induction won't bring in any side effect. The solution is to add two properties to check if each scoreboard state transition works as expected. Symbolic random IDs discussed above are used to reduce complexity.

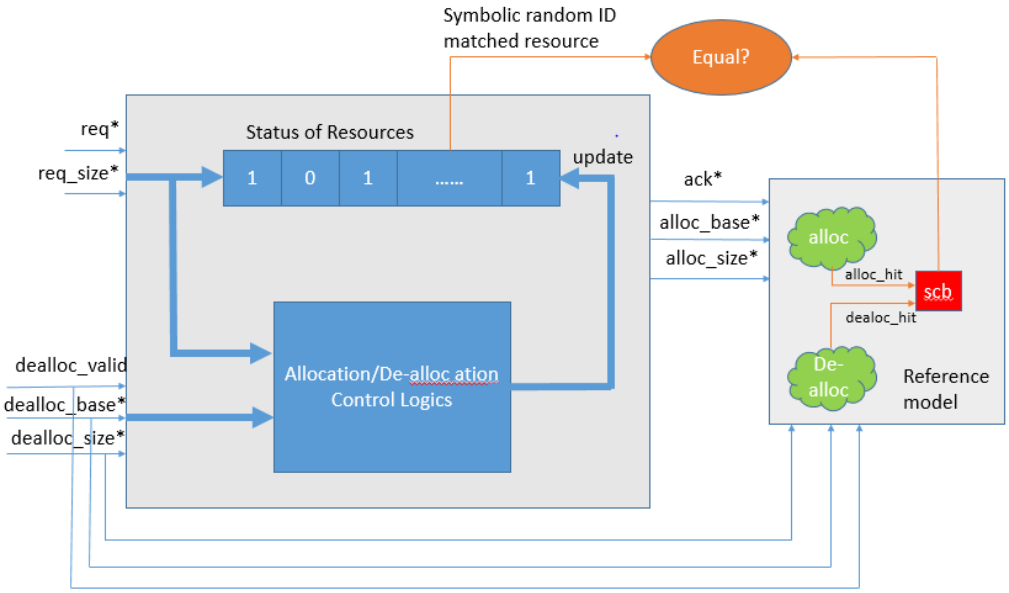

Figure 5 End-to-End Checker for Scoreboard

```
//After alloc, the scoreboard bit must be the same as reference bit, scb
alloc_legality_check_internal_p: assert property (@(posedge clk) disable iff(rst)
                        (alloc && alloc_res_hit |-> scb == internal_bit_mask[symbolic_id])
);

// After dealloc, the scoreboard bit must be the same as reference bit, scb
dealloc_legality_check_internal_p: assert property (@(posedge clk) disable iff(rst)
                        (dealloc && dealloc_res_hit |-> ##3 scb = internal_bit_mask[symbolic_id])
);
```

Now there are end-to-end checkers for resource status to make sure our induction is safe. VC Formal is able to catch all those four errors in a short time after adding these two new checkers.

TABLE 6
MANUAL MUTATION INJECTION AND RESULT

Mutation type	Failed legality checkers	Depth	Time
Allocate more	alloc_legality_check_internal_p	4	20s
Allocate less	alloc_legality_check_internal_p	4	22s
De-allocate more	dealloc_legality_check_internal_p	5	26s
De-allocate less	dealloc_legality_check_internal_p	5	29s

By definition, formal verification is exhaustive. However, when manual interventions are applied to help convergence, some subtle errors may be introduced unintentionally. For this specific case:

1) Code or functional coverage can't help because reset abstraction doesn't change the reachability of the design states.
2) Resource leakage issue requires a deep depth to expose, which formal verification isn't good at.

Therefore, mutation coverage gives us an intuitive way to check whether any property holes are hidden. Because manual error injection is trivial and error-prone, we turn to Synopsys's Certitude for our mutation coverage methodology. We expect to catch more issues in the whole scoreboard logic given that it hasn't been fully verified with end-to-end checker (e.g. resource search logic).

B. Mutation coverage by Certitude

Nowadays, code coverage and functional coverage are still the mainstream metrics for verification convergence. However, those controllability coverages can only measure if the DUT is well traversed, but they can't guarantee that each error activated would be detected by the verification environment. On the other hand, mutation coverage, as observability coverage, is able to measure the quality of verification environment itself. The disadvantages of mutation coverage are:

1) The number of possible mutations is infinite for real design.
2) To check each mutation coverage is time-consuming.

Synopsys Certitude is a powerful mutation coverage tool, which is able to:

1) Inject different kinds of errors into RTL automatically.
2) Be compatible with both simulation and formal verification.

We integrate it into our verification environment to do automatic faults injection and mutation coverage collection, checking if it can replace manual injection completely.

It costs nearly 208 hours to insert 7893 faults to find out the resource leakage issue discussed above while the manual way only requires 40 minutes to expose the hole. Please note Certitude runs with only the legality checkers. The run time cost could be even more expensive for a full FPV run. We can't expect to fully rely on automatic error injection.

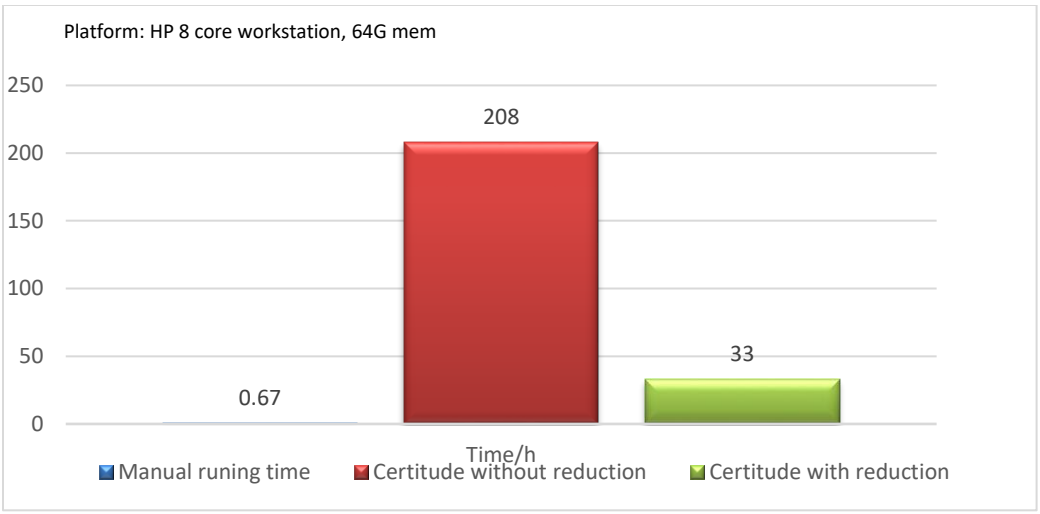

Figure 6 Manual running time, Certitude without reduction run time, Certitude with reduction run time

This reminds us of the contrast between code coverage and functional coverage. Usually in simulation world, verification engineer will take both types of coverage into consideration. Though functional coverage requires non-trivial workload for implementation, it's more meaningful and friendly to review. We define manual mutation coverage as functional mutation coverage:

- Explicit mutation coverage, based on design specification directly
- Independent from formal verification environment and abstraction techniques
- Measurement for observability of interests
- Limited number

We define automatic mutation coverage as structural mutation coverage:

- Implicit mutation coverage, based on RTL structural only
- Independent from formal verification environment and abstraction techniques
- Generated by EDA tool
- Can be numerous

This discrimination helps us to make some trade-off between the two kinds of mutation coverage. Our mutation coverage steps are listed below:

1. Listed possible errors of interests in the formal verification plan
   - ✓ Based on specification, just like functional coverage plan
   - ✓ Pay attention to abstraction techniques which may introduce verification hole
   - ✓ Focus on high priority errors and keep the runtime cost in mind
   - ✓ Alloc more/less, de-alloc more/less, fits logic in this case
2. Use SNPS Certitude to inject those functional mutation coverage
   - ✓ Identify arithmetic errors and the target design files for this case.
3. Use SNPS Certitude to auto-inject structural mutation error into design
   - ✓ Reset, connectivity, arithmetic, etc.
   - ✓ Restrict the error number to avoid extremely long running time
   - ✓ Low priority compared to functional mutation error.
4. Add mutation coverage into our formal sign-off list.

After this optimization, resource leakage issue can be exposed by Certitude easily as before and it takes 33 hours to check 698 related errors injected by Certitude as Figure 6 shows. Now the running time is more acceptable. Further reduction is feasible, but this isn't the focus of this paper.

TABLE 7
COMPARISON BETWEEN MANUAL AND AUTOMATIC MUTATION INJECTION

	Manual Injection	Automatic Injection
Engineering work	Big	Small
Running time	Short	Long
Design knowledge	Yes	No

Final mutation coverage result is showing below.

- 33% non-detected faults.
    - 9 of the non-detected errors are the resource leakage related issue.
    - 19 of the non-detected errors are related to logic redundancy.
    - 37 of the non-detected errors are covered by other types of checkers, like the "reserve" and "cu_locking" logics.
    - 172 of the non-detected errors are related to resource search logic as we expect, which is similar to the issue in resource status logic. When we do reset abstraction to the scoreboard, resource search logic is cut off from the design. So we need to add a new checker to guarantee that the correction of resource search logic.
- 67% detected faults.

Based on our experience in Certitude, we think that mutation coverage is a big plus for formal verification:

1. Compared to simulation, formal verification usually takes much less running time.
2. Formal verification is exhaustive, which avoids non-activated error (untouched by stimulus) seen in constrained-random-verification.
3. Abstraction techniques can be very tricky to introduce subtle errors, mutation coverage can expose holes in formal properties.
4. A planned functional mutation coverage can give us high confidence in formal verification environment.
5. A well-thought-out trade-off between functional mutation coverage and structural mutation coverage can achieve best return-on-investment.
6. With some guidance, Certitude can generate more intentional mutation coverage even there are still many redundancies which can be optimized further.

VI. CONCLUSION

Formal property verification has been proven to be a reliable method for many kinds of designs' verification sign-off. However, it's unrealistic to achieve fully proof without any manual abstraction strategies as designs are becoming more and more complex. The more manual interventions, the bigger risk to have verification holes in the testbench. Mutation coverage can be a big help for formal sign-off flow to expose subtle verification holes. Some guidelines are listed as following:

1. Plan your mutation coverage according to the formal verification plan (functional mutation coverage).
2. Use EDA tool (e.g. SNPS Certitude) to automatically insert different kinds of errors into your design (structural mutation coverage).
3. Makes trade-off between two mutation coverages for best return-on-investment because mutation coverage can be infinite and expensive.
4. Pay more attention onto the functionality which you can't do end-to-end check (non-determinism).
5. Pay more attention onto the functionality which you apply abstraction techniques.
6. Separate it from your constraints, checkers or abstraction technique for the sake of independency.

ACKNOWLEDGEMENT

The authors would like to thank Vigyan Singhal and Chirag Agarwal from Oski, Xiaolin Chen, Jimmy Sun, Sward Xie and Huang Feng from Synopsys, Matthew Znoj and Mark Anderson from AMD Shader Processor Input team, Christeen Gray from AMD GFXIP methodology team for their great support during our development of formal verification for this resource management block. Without their help, we could not have finished this very first try on our GPU design.

REFERENCES

[1] D. Perry, H. Foster, "Applied formal verification: for digital circuit design", McGraw Hill, 2005.
[2] N. Kim, J. Park, H. Singh, and V. Singhal. "Sign-off with Bounded Formal Verification Proofs", DVCon 2014.
[3] Synopsys. (2015) VC Formal Verification User Guide, version K-2015.09-1.
[4] Synopsys. (2015) Certitude User Manual, version K-2015.09.
[5] H. Foster, A. Krolink and D. Lacey. (2004) Assertion-Based Design, Second Edition.