Transaction-Based Assertion for Transaction Level Coverage, Property and Protocol Checking

Thinh Ngo and Sakar Jain

Freescale Semiconductor, Inc.
Austin, TX, USA
www.freescale.com

ABSTRACT

We propose transaction-based assertion (TBA) that works on transactions instead of signals. Transaction-based cover assertions can be used for transaction-level coverage to detect various transactions of interest with logical (i.e. and), structural (i.e. equal) or temporal (i.e. concurrently, after) relationships. Transaction-based property assertions can be used for transaction-level protocol checking (i.e. a cache miss must be consecutively followed by a cache fetch of same cache line). Transaction-based assertions can be implemented using SystemVerilog class. Transaction class encapsulates associated signals as its properties and self-monitors the associated DUT logic for its triggering. Transaction types are uniquely defined by triggering properties (i.e a hit-valid for cache-hit transactions). Additionally, transactions can structurally associate with each other by their compare properties (i.e. transformed address). Transaction-based cover assertion class monitors triggered transactions for their required relationships structurally, logically or temporally. Transaction-based property assertion class checks temporal implication between two structurally related triggered transaction-based cover assertions.

Table of Contents

1. Introduction

SystemVerilog Assertions (SVAs) are extensively used in today's design verification. Assertion-based verification methodology is widely adopted in the mainstream verification. Specifically, cover assertions are used to determine if interesting DUT states are covered and property assertions are effectively used to check DUT logics. In addition, SVAs have been used for interface protocol checking (i.e. AXI). Syntactically, SVAs use DUT signals and logical and temporal operators to form conditions for coverage or implications for checking. These assertions can become complicated and unsuited for checking of high-level protocols (i.e. involved groups of related signals across modules and blocks); instead, procedural checking utilizing procedural constructs (i.e. if, while, queue) is more effective and appropriate. High-level protocol checking often deals with transactions (e.g. groups of related signals forming an operation on an interface) rather than individual signals. However, assertions can be made to work on transactions instead of signals. As such assertion compactness and simplicity are retained and its capability is extended. In this paper, we propose transaction-based assertion (TBA) for transaction level coverage, property and protocol checking.

2. High-Level Protocol Checking and Assertion Challenge

In today's advanced testbenches, checking is mostly performed via SystemVerilog procedural constructs (i.e. if, queue, while) except at interfaces or on structured design elements (finite state machines, fifos) where SystemVerilog assertions (SVAs) are mostly used. SVAs are preferable to procedural constructs for checking whenever possible since SVAs are easier to code and understand. Procedural checking has its advantages when:

1. Checking involves comparison of groups of related signals (e.g. transactions) and/or
2. Checking needs to use struct, array, queue, and loop constructs
3. Checking comprises multiple related checks

On the other hand, SVAs advantages are noted for:

1. Checking intent is more explicit (e.g. given by signal names and operators of the expression)
2. Checking content is more modifiable (e.g. same as above)

One can also view the two checking mechanism comparison as complexity versus simplicity. Complex coding (e.g. using procedural constructs) is required to check complex conditions but its checking intent is not explicit and its checking content is not modifiable. On the other hand, simple coding (e.g. SVAs) can use to check simple checking conditions and its checking intent and checking content are explicit and modifiable.

Software industry has adopted several methodologies (i.e. structured programming, object-oriented programming(OOP), functional programming and artificial intelligence) to reduce coding complexity and achieve complex objectives. Similarly, OOP can be used in hardware design verification to realize the advantages of both procedural checking and SVA.

3. Transaction-Based Assertion (TBA)

Transaction-Based Assertion (TBA) consolidates the capabilities and advantages of both SVA and procedural checking. Specifically, TBA retains assertion compactness and simplicity with explicit checking intent and modifiable checking content and TBA works on transactions, instead of signals, and therefore can readily handle inter-module, inter-block and high-level protocol

checking. Potential protocols which can be checked via TBAs are memory coherency, multi-core/thread memory accesses, deadlock/livelock, cache, etc.

TBA takes advantage of object-oriented programming (e.g. classes, information hiding and encapsulation) to alleviate coding complexity. Transaction class encapsulates groups of related signals. The Assert_Cover and Assert_Property classes emulate SystemVerilog cover assertion and property assertion functionality, respectively, to perform coverage and checking on inter-module and inter-block transactions. Transaction triggering is based on a transaction triggering signal. Assert_Cover triggering is based on individual transactions' triggering and their logical, temporal and structural relationships being met. Assert_Property triggering is based on the two constituent Assert_Covers' triggering and their temporal relationship being met. Once transactions are defined, TBAs can be formed with ease using transactions as signals in SVAs. TBAs content is specified during TBA instantiation; therefore, its checking intent is explicit and checking content is modifiable as those of SVAs.

4. TBA Functionality, Architecture and Organization

TBA consists of three primary classes: Transaction, Assert_Cover and Assert_Property.

Transaction class

Transaction class emulates DUT logic signal role in SystemVerilog assertions. At any time, a transaction is either triggered or not triggered based on the transaction event signal triggering (i.e. a cache hit valid signal is used to trigger a cache hit transaction). Transaction class has **Triggered** property to indicate its present triggering state and **Monitor** method to set or reset transaction's **Triggered** property based on the transaction event signal triggering. In addition, a transaction is more abstract and high-level by encapsulating related signals to form a more meaningful operation. An example is a peripheral access transaction composing of an address, a valid signal, an operation type, a device id and a tag.

In a SVA, individual signals are typically related to each other by the assertion intent. Assertion creator selects these related signals to create an interesting scenario to be covered or an implication to be checked. This type of relationship is functional. Transaction class naturally encapsulates functional relationship. It has **Type** property associated with transaction event signal name. Transactions of various related types can be selected to form a transaction-based assertion. For instance, a cache read transaction and a cache write transaction of the same cache line are occurring at the same time.

Moreover, transactions from related blocks often correspond via structural signals. For example, a cache invalidate cannot be immediately followed by a cache hit of the same cache line. Here, the structural relationship between the two transactions is the equality of their addresses. Transaction class is extended to encapsulate structural relationship. Transactions of different types can be structurally related via the **Comparing_Tag** property and the **Compute_Comparing_Tag** method. Specifically, transaction class can use a signal associated with the transaction event signal as its **Comparing_Tag** property (i.e. the address of a cache hit event). Often time, structurally related signals (i.e. addresses) are manipulated (i.e. part-select as tag) and needed to be transformed to the same commonality for comparison. **Compute_Comparing_Tag** method is used to compute **Comparing_Tag** from the structurally related signal.

In addition, Transaction class has **NClock** property to indicate the clock cycle of its latest triggering and **Count** property to keep track of the number of transactions triggering. In short, the Transaction class facilitates transaction based assertions via its capability to self-trigger and to relate functionally and/or structurally among transactions.

Assert_Cover class

Assert_Cover class emulates the SystemVerilog cover assertion functionality but it works on transactions instead of signals. Individual transactions and their logical, temporal and structural relationships can be put together to form a transaction based cover assertion. For instance, a cache read transaction and a cache write transaction of the same cache line are occurring at the same time. Here, logical relationship is the `and` operator between the two transactions, temporal relationship is the two transactions' occurring at the same time and structural relationship is two transactions' having the same cache line. Logical relationships include `and, or`, etc. Temporal relationships include before, after and simultaneously. Structural relationships include greater, equal and smaller.

Assert_Cover class self-triggers once individual transactions are triggered and their relationships are met. To accomplish this, Assert_cover class instantiates all possible transactions and monitor their triggering and their relationships. For instance, a transaction based cover assertion: "a cache read transaction and a cache write transaction of the same cache line are occurring at the same time" shall monitors cache write and cache read transactions triggering if they occur at the same time and if they have the same cache line. An Assert_Cover can be in one of the three assert states: not Asserted, Asserted and otherAsserted (e.g. some transaction is asserted but not those of this Assert_Cover).

Each Assert_Cover is declared and instantiated once with the cover assertion intents (individual transactions and their relationships) being passed as instantiation arguments. For Assert_Cover class with at most two individual transactions, its new function is shown below:

```
function new(string name, string negate1 = "", CacheTxnType_enum
txn1type, string logical = "", string temporal = "", string
structural = "=", string negate2 = "", CacheTxnType_enum
txn2type = "");
```

In addition, Assert_Cover class has **Asserted** property indicating whether it is asserted at the current clock. It also has a **Comparing_Tag** property when all individual transactions have the same **Comparing_Tag**. This **Comparing_Tag** is used for structural comparison between the two Assert_Cover assertions within an Assert_Property assertion.

Assert_Property class

Assert_Property class emulates SystemVerilog property assertion but it works on transactions instead of signals. A SystemVerilog property assertion basically can be viewed as a temporal cause and effect implication between two SystemVerilog cover assertions. Similarly, Assert_Property assertion is basically formed by two Assert_Cover assertions associated with a temporal cause and effect implication.

Assert_Property self-triggers once cause and effect Assert_Covers are triggered and their temporal implication is met. SVA A |=> B will pass if A condition is matched and one clock later B condition is matched. TBA Assert_Property A |=> B will pass if A triggers and at least one clock later B triggers. At transaction level, exact timing of transaction triggering is not critical and is not considered a pass/fail criterion; however, transaction triggering order (i.e before, after) is significant and is accounted as temporal relationship pass/fail criterion. Assert_Property A |=> B will fail if A triggers and B does not trigger (i.e. C triggers where C != B). For example, a TBA "a cache miss should be consecutively followed by a cache fetch of the same cache line" shall fail if it is observed that "a cache miss is followed by a cache hit of the same cache line". Assert_Property in the form !(A |=> B) will fail if A triggers and then B triggers and pass if A does not triggers (vacuous success) or A triggers and non-B triggered.

To accomplish the above-mentioned functionality, Assert_Property instantiates both cause and effect Assert_Covers and monitors their triggering and temporal relationship. For instance, transaction-based property assertion: "a cache invalidate cannot be consecutively followed by a cache hit of the same cache line" shall instantiate Assert_Cover "a cache invalidate" and Assert_Cover "a cache hit", monitor their triggering, check if the former is triggering before the later without any other transaction in between (e.g. consecutive) and check if their cache lines are the same. If all the checks are satisfied then the Assert_Property shall trigger and fail the simulating test.

Each Assert_Property is declared and instantiated once with the property assertion intents (cause and effect Assert_Cover assertions) being passed as instantiation arguments.

```
function new(string name, string relax = "", string Pnegate =
"", string negate1 = "",  CacheTxnType_enum txn1type, string
logical1 = "", string temporal1 = "", string structural1 = "=",
string negate2 = "", CacheTxnType_enum  txn2type = "", string
Pimplicate = "|=>", string Ptemporal = "", string Pstructural =
"=", string negate3 = "", CacheTxnType_enum txn3type, string
logical2 = "", string temporal2 = "", string structural2 = "=",
string negate4 = "", CacheTxnType_enum txn4type = "");
```

5. TBA Example

The example that follows is from a real verification environment that verifies 3 different caches (primary cache, secondary cache and operation cache) of a design block named Peripheral Access Management Unit. These 3 caches store various peripheral access control data cached from a multi-core platform memory. Traditionally, coverage is used to ensure a cache is functionally active and black-box checking is used to ensure it is functionally correct. However, there were some cache defects found during silicon validation that require more thorough pre-silicon cache verification. There are choices to be made regarding how thorough the cache verification is and consequently where in the cache logics verification testbench needs to poke into. One can monitor signals inside the cache (i.e. to verify its replacement policy), at the cache interface or at the cache wrapper (where it is instantiated along with other modules). Due to the limited time availability, it is decided to verify the cache at the cache wrapper level. Specifically, information

available at this level are cache read (hit/miss), cache write, cache invalidate and cache fetch. Note that there is no victim or castout as this is a one-level cache.

Procedural checking is naturally considered since there are more than 20 signals across several modules (i.e. fetch module, invalidate module and cache module (read/write)) and there are various address transformations and comparisons. SVAs can also be used but they can be quite involved and complex. The verification objective is to verify the 5 basic cache events namely cache miss, cache hit, cache write, cache invalidate and cache fetch – a higher-abstract level of cache verification. Specifically, a cache miss is caused by a cache read of a cache line that does not reside in the cache. A cache write is caused by a cache fetch and causes the cache line to be stored in the cache. A cache invalidate is randomly initiated by processors' snooping and causes a cache line resided in the cache to be removed from the cache. A cache fetch is caused by a cache miss and causes a cache write of the cache line. Instead of poking inside the cache to determine if a cache line resides in the cache or not to verify cache functionalities, one can use these events. This verification strategy cannot thoroughly verify a cache (i.e. checking replacement policy, set and way) but it does check the basic cache functionality. Cache functionalities based on these events to be verified are exhaustively listed below:

1. A cache miss must not be consecutively followed by a cache miss of the same cache line.
2. A cache miss must not be consecutively followed by a cache hit of the same cache line.
3. A cache miss must not be consecutively followed by a cache write of the same cache line.
4. A cache hit must not be consecutively followed by another cache miss of the same cache line.
5. A cache hit must not be consecutively followed by another cache write of the same cache line.
6. A cache hit must not be consecutively followed by another cache fetch of the same cache line.
7. A cache write must not be consecutively followed by a cache miss of the same cache line.
8. A cache write must not be consecutively followed by a cache write of the same cache line.
9. A cache write must not be consecutively followed by a cache fetch of the same cache line.
10. A cache fetch must not be consecutively followed by a cache miss of the same cache line.
11. A cache fetch must not be consecutively followed by a cache hit of the same cache line.
12. A cache fetch must not be consecutively followed by a cache fetch of the same cache line.
13. A cache invalidate must not be consecutively followed by a cache hit of the same cache line.
14. A cache invalidate must not be consecutively followed by a cache write of the same cache line.
15. A cache invalidate must not be consecutively followed by a cache fetch of the same cache line.

Note that these properties are in the form: !(A |=> B) which fail if both A and B trigger one after the other. A property like "a cache miss must be consecutively followed by a cache fetch" (A |=> B) will fail since cache invalidates are randomly injected; consequently, this assertion has to be split into 3 assertions of the form !(A |=> B) (e.g. see 1-3 above).

There have been many false fails due to transaction proximity and cache implementation simplification. For instance, a property "a cache miss must not be consecutively followed by a cache miss of the same cache line" can be violated by two back to back accesses of the same cache line. A cache design can be simplified to just let both of them as cache misses and can do the fetch later. This is a common challenge in verifying caches whose functionality deviations –

causing performance impacts - can become implementation features. The approach taken to resolve these false fails is two-fold. First, a property shall not fail if there are adjacent (i.e. <50clks) transactions of the same cache line prior to this property. Second, the constituent causal Assert_Cover and effectual Assert Cover need to be at least 5 clocks apart to give RTL time to react. These numbers of cycles are based on trial and error and are implementation-specific.

6. TBA Example Implementation

TBA declarations:

```
Assert_Property PCACHE_NOT_MISS_then_same_MISS;
Assert_Property PCACHE_NOT_HIT_then_same_MISS;
Assert_Property PCACHE_INVALIDATE_then_same_MISS;
…
```

TBA instantiations:

```
//A cache miss must not be consecutively followed by a cache miss of the same cache line.
PCACHE_NOT_MISS_then_same_MISS  =
  new("PCACHE_NOT_MISS_then_same_MISS","NOT",,, MISS,,,,,,
       "|->",,"=",, FETCH);//see the new function prototype of
Assert_Property class in Section 4
PCACHE_NOT_HIT_then_same_MISS  =
  new("PCACHE_NOT_HIT_then_same_MISS","NOT",,, HIT,,,,,,
       "|->",,"=",, HIT);
PCACHE_INVALIDATE_then_same_MISS  =
  new("PCACHE_INVALIDATE_then_same_MISS",,,, INVALIDATE,,,,,,
       "|->",,"=",, MISS);
…
```

Transaction class:

Transaction class properties:

```
   typedef struct packed {
     CacheTxnType_enum Type;
     logic Triggered;
     logic [0:39] Comparing_Tag;
     logic [0:63] Address;
     int NClock; //time of triggering
     int Count; //number of triggerings
   } txn;
```

Transaction Cache Write monitor method:

```
task PCacheWriteTxn::monitor ();
@(PCacheIF.cb);

if (PCacheIF.cb.pamu_pcache_write) begin
   Txn.Address[24:57] = PCacheIF.cb.pcache_updt_req_addr[24:57];
   Txn.Triggered = 1'b1;
   Txn.Comparing_Tag = Compute_Comparing_Tag();
   ++Txn.Count;
end else begin
```

```
   Txn.Address = 64'h0;
   Txn.Triggered = 1'b0;
   Txn.Comparing_Tag = 0;
end
end else @(PCacheIF.cb);

++Txn.NClock;
endtask
```

Transaction Cache Write Compute Comparing_Tag:

```
function logic [0:39] PCacheWriteTxn::Compute_Comparing_Tag();
logic [0:11] liodn;
logic [0:63] pcache_base_addr;

pcache_base_addr[0:63] = {PCacheIF.cb.pcache_base_addr_h[0:31],
                          PCacheIF.cb.pcache_base_addr_l[0:31]};
liodn[0:11] = Txn.Address[46:57] - pcache_base_addr[46:57];
Compute_Comparing_Tag[0:39]={(pcache_base_addr[24:57]+
                             liodn[0:11]), 6'b0};
endfunction
```

Assert Cover class:

```
typedef struct {
  CacheTxnType_enum Txn1Type, Txn2Type, otherTxn1Type,
                    otherTxn2Type;
  string Relax, Negate1, Negate2;
  string Logical, Temporal, Numerical;
  logic Asserted, otherAsserted;
  logic [0:39] Comparing_Tag, otherComparing_Tag;
  int NClock;
  int Count;
} _cover;

task Assert_Cover::monitor ();
oneclock ();
...
if (Hit.Txn.Triggered) begin
   Txn.Asserted = 1'b1;
   Txn.Type = Hit.Txn.Type;
   Txn.Comparing_Tag = Hit.Txn.Comparing_Tag;
   Txn.NClock = Hit.Txn.NClock;
   Txns_queue.push_front(Txn);
   Txns_5clks_queue.push_front(Txn);
end
if (Miss.Txn.Triggered) begin ...
if (Fetch.Txn.Triggered) begin ...
if (Write.Txn.Triggered) begin ...
```

```
if (Invalidate.Txn.Triggered) begin ...

if (Txns_queue.size() > 0) begin
   Cover.Asserted = 1'b0;
   Cover.otherAsserted = 1'b0;
   while (Txns_queue.size() > 0) begin
        Txn1 = Txns_queue.pop_back();
        if (Cover.Txn1Type == Txn1.Type) begin
           Cover.Asserted = 1'b1;
           Cover.Comparing_Tag = Txn1.Comparing_Tag;
           Cover.NClock = Txn1.NClock;
           ++Cover.Count;
        end else begin
           Cover.otherAsserted = 1'b1;
           Cover.otherTxn1Type = Txn1.Type;
           Cover.otherComparing_Tag = Txn1.Comparing_Tag;
           Cover.NClock = Txn1.NClock;
        end
   end
...
```

Assert_Property:

```
typedef struct {
     string Relax, Pnegate, Pimplicate, Ptemporal, Pnumerical;
     _cover CoverCause, CoverEffect;
     logic Asserted, otherAsserted;
     logic [0:39] Comparing_Tag;
     int NClock;
     int Count;
} _property;

task Assert_Property::monitor ();
oneclock ();

if (Assert_Cover_Effect.Cover.Asserted |
       Assert_Cover_Effect.Cover.otherAsserted) begin
   Property.CoverEffect.Asserted =
      Assert_Cover_Effect.Cover.Asserted;
   Property.CoverEffect.otherAsserted =
      Assert_Cover_Effect.Cover.otherAsserted;
   Property.CoverEffect.Comparing_Tag =
      Assert_Cover_Effect.Cover.Comparing_Tag;
   Property.CoverEffect.NClock =
      Assert_Cover_Effect.Cover.NClock;
end else begin
   Property.CoverEffect.Asserted = 1'b0;
   Property.CoverEffect.otherAsserted = 1'b0;
```

```
   Property.CoverEffect.Comparing_Tag = 0;
   Property.CoverEffect.NClock = 0;
end

Property.Asserted =
  (Property.CoverCause.Asserted & Property.CoverEffect.Asserted)
 &(Property.CoverEffect.NClock > Property.CoverCause.NClock + 4)
 &((Property.CoverCause.Comparing_Tag ==
       Property.CoverEffect.Comparing_Tag)
 |(Property.CoverCause.Comparing_Tag == 40'h9999999900)
 |(Property.CoverEffect.Comparing_Tag == 40'h9999999900));

Property.otherAsserted =
  (Property.CoverCause.Asserted &
      Property.CoverEffect.otherAsserted)
 &(Property.CoverEffect.NClock > Property.CoverCause.NClock + 4)
 &((Property.CoverCause.Comparing_Tag ==
      Property.CoverEffect.Comparing_Tag)
 |(Property.CoverCause.Comparing_Tag == 40'h9999999900)
 |(Property.CoverEffect.Comparing_Tag == 40'h9999999900));

if (Property.Pnegate == "NOT")
  if (!Property.Asserted) begin
    if (Property.CoverCause.Asserted &
           Property.CoverEffect.otherAsserted) begin
        Property.CoverCause.Asserted = 1'b0;
        Property.CoverCause.otherAsserted = 0;
        Property.CoverCause.Comparing_Tag = 0;
        Property.CoverCause.NClock = 0;
        ++Property.Count;
    end
  end else `error("*** Negated Property Asserted ***");
else if (!Property.Asserted) begin
        if (Property.CoverCause.Asserted &
               Property.otherAsserted) begin
           `error("*** Property Failed ***");
        end
end else begin //only reset CoverCause when hit
            Property.CoverCause.Asserted = 1'b0;
            Property.CoverCause.otherAsserted = 0;
            Property.CoverCause.Comparing_Tag = 0;
            Property.CoverCause.NClock = 0;
            ++Property.Count;
end

if (Assert_Cover_Cause.Cover.Asserted) begin
   Property.CoverCause.Asserted = 1'b1;
```

```
    Property.Comparing_Tag =
      Assert_Cover_Cause.Cover.Comparing_Tag;
    Property.CoverCause.otherAsserted =
      Assert_Cover_Cause.Cover.otherAsserted;
    Property.CoverCause.Comparing_Tag =
      Assert_Cover_Cause.Cover.Comparing_Tag;
    Property.CoverCause.NClock = Assert_Cover_Cause.Cover.NClock;
end
endtask
```

7. Conclusion

In this paper, we proposed Transaction-Based Assertion (TBA) for transaction level coverage, property and protocol checking. TBAs can be formed using transactions located across modules and blocks with various logical, temporal and structural relationships. TBA is implemented using SystemVerilog classes to facilitate information hiding and encapsulation. We successfully applied and implemented TBAs in verifying three caches of a design block. TBA cannot replace procedural checking where complicated checking is required (i.e. modelling). TBA is based on transactions; therefore, it is most applicable where there are several transaction types and they are related to each other logically, temporally and structurally (e.g. high-level protocol). Furthermore, TBA can be more applicable if the SystemVerilog language extends SVA capabilities to handle transactions,

8. Reference

[1] Janick, B., Eduard, C., Alan H.and Andrew N. 2005. Verification Methodology Manual for SystemVerilog.