Experiencing Checkers for a Cache Controller Design

Ben Cohen
VhdlCohen Publishing
ben@SystemVerilog.us

Srinivasan Venkataramanan and Ajeetha Kumari
CVC Pvt.Ltd., Bangalore, India
srini@cvcblr.com
akumari@cvcblr.com

Lisa Piper
IC Verification Consultant
lisa_piper@systemverilog.us

ABSTRACT

The **checker** construct is a new feature defined in IEEE 1800-2009; it is intended to facilitate the definition and usage of libraries of assertions and to delineate verification code from RTL. In this paper, we describe our experience in using checkers for the evaluation of a cache controller design. The goal was to evaluate how easy it is to define and then utilize the checkers in a variety of configurations. Checkers were defined to include static concurrent assertions, procedural concurrent assertions, immediate and deferred immediate assertions. The checkers were then instantiated statically and procedurally in the design module. We also experimented with where the checkers are defined and how formal arguments were used. Simulation was used to confirm our results, with both pass and fail assertion results expected.

We start with a brief description of the design in evaluation and the verification environment, including a list of the checkers that were defined. We then provide an overview of some of the key aspects of the testing, including 1) how static and procedural assertions inside a checker are treated when the checker is instantiated statically or procedurally in the design; 2) use of deferred assertions to prevent transient errors; 3) argument passing; and 4) configuration checks (e.g., elaboration time checks). These are all areas where the 2009 release of the standard professes to have improved usability. We then show the checker definitions and instantiations and discuss our personal experiences in the use of checkers.

Categories and Subject Descriptors

IEEE 1800-2009 SystemVerilog checkers and assertions.

General Terms

Verification.

Keywords

IEEE 1800-2009, checker, SystemVerilog Assertions, verification. deferred immediate assertion, concurrent assertion, static assertion, procedural assertion, static checker instantiation, procedural checker instantiation.

1. INTRODUCTION

The **checker** construct provides an encapsulation of all assertion and coverage related code for use in RTL / behavioral designs and in verification environments. In this paper, we share our experience in creating checkers for the verification of a cache controller design. Section 2 defines the design and verification environment, including an overview of the checkers that were created. Section 3 provides an overview of the important concepts that were the focus of our analysis. These are the features that distinguish checkers from their predecessor module-based libraries. Section 4 shows the checkers that were created. Finally, Section 5 summarizes our impressions after creating and using checkers to evaluate our design. The presentation and the SystemVerilog code used on this project are available for download.[7]

2.0 DESIGN AND VERIFICATION ENVIRONMENT

This section defines the cache controller model and the key requirements that will be tested. The design requirements were intended to be very simplistic so as not to overly complicate the analysis. The verification strategy, including the overview of checkers to be defined, is also described.

2.1 Cache Controller Model

Many systems have a cache for storage of temporary data that is likely to be used again. The advantage of the cache is that access is typically much faster than accessing that data in main memory or external memory via a network. A cache controller is used to access its internal cache and to control what data is contained in the cache. An address/data pair and entry status (valid/invalid) is referred to as a cache line. The cache is implemented using three cache memories to emulate the cache entries; these consist of *cache_entry, cache_mem_addr,* and *cache_mem_data.* To quickly determine that an entry is valid (i.e., the "exists" function), a large 1-bit wide memory (*cache_valid*) with depth identical to the main memory is used. Figure 2.1-1 represents an overview of the cache memory model.

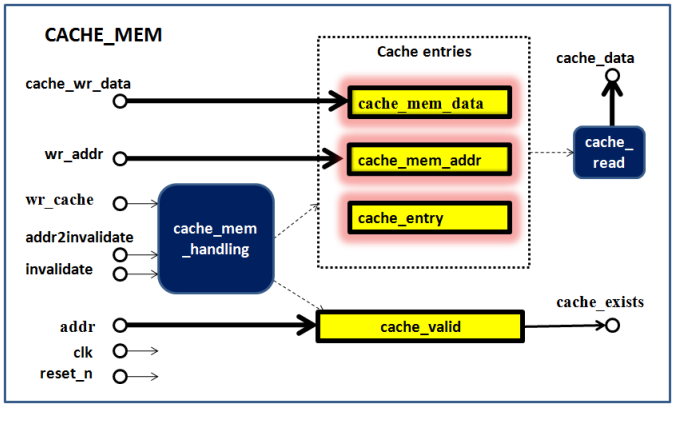

Figure 2.1-1 Cache Memory Overview

Cache reads will either result in a hit (i.e., *cache_exists*, the desired addressed data was found in the cache) or a miss (i.e., the desired addressed data is not available in the cache). A *miss* is defined in the model as the invert of *cache_exists*. When a *miss* occurs, the cache

controller will retrieve the data from main memory and then add the data to the cache.

For a read or a write transaction, if the cache is full and the cache does not contain the data, the oldest address/data in the cache is replaced. The oldest address/data algorithm for this design uses a first-in first-out (FIFO) to implement the oldest address/data replacement because it is simple. The addresses are stored in a FIFO. Address/data in the cache is replaced by popping the oldest address from the FIFO, and adding the new data and address to the cache, and the address to the FIFO.

The cache controller also controls what data is written to the cache memory. Data written by the user is always copied to the cache and to the main memory, i.e., a write-through cache. When data is written to the cache, the cache is checked to see if there already exists data in the cache at that address. If so, the data is updated. If not, a new cache line is added, i.e., the data is added to the cache and the associated address is added to the FIFO.

The cache controller has a user interface and a main memory interface, as shown in Figure 2.1-2. The cache memory is internal. The user interface consists of the following signals:

Signal	Function
data_vld	If *1'b1* then *rd_data* to user interface is valid
busy	Instructs user interface to stop sending new read / writes
rd	Read command from user interface. If *1'b1* then read
wr	Write command from user interface. If *1'b1* then write
addr	Address from user interface
rd_data	Read data to user interface from cache or main memory
wr_data	Data to write from user interface to main memory
clk	System clock
rst_n	System reset

The main memory interface consists of the following signals:

Signal	Function
mem_rd	Read command to main memory. If 1'b1 then read
mem_wr	Write command to main memory. If 1'b1 then write
mem_addr	Address to main memory
mem_rd_data	Read memory data from main memory
mem_wr_data	Write data to main memory

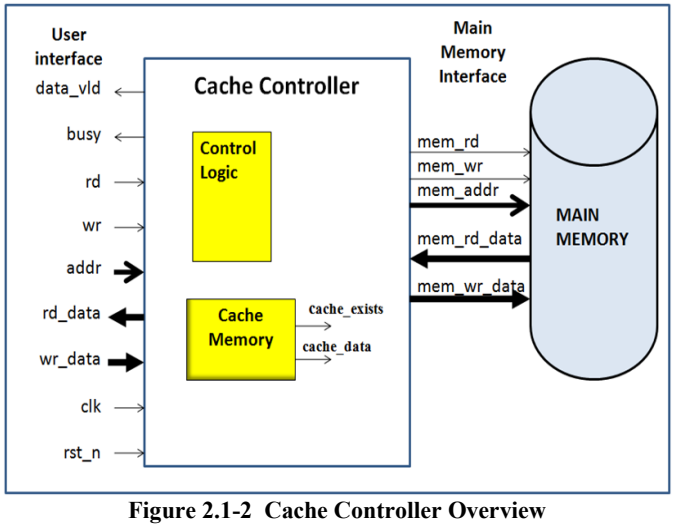

Figure 2.1-2 Cache Controller Overview

2.2 High level Requirements

This section provides a summary of the requirements. It does not necessarily define all the needed properties, nor does it define the implementation, as that could be a risk if the requirements impose an implementation.

General requirements:

1. Main memory: The main memory is slow, and requires 3 or more cycles to read or to write. The size and latency of the main memory are parameterized. The data is latched on the falling edge of the write signal and is driven on the rising edge of the read signal.

2. Cache: The cache accesses are very fast, minimally 1 clock cycle. The cache size is parameterized. This cache is a write-through cache, thus the write to memory always proceeds. Cache lines are either updated or created on writes. They are also created on cache read misses.

3. User interface write: If a write from the user interface occurs, then the write must be forwarded to main memory and to the cache. If there is a write cache hit, meaning that a cache entry already exists, then the cache line must be updated because we are doing a write. In addition, a write through to the main memory must also occur. User interface writes are always one cycle, and the cache controller is responsible for the timing to the main memory. That timing is a minimum of three rising clock edges, but could be more cycles as defined by a parameter; this latency allows time to update the main memory. See Figure 2.2-1 for the timing of a user interface write.

4. Cache full: If the cache is full and a new value must be written into the cache, then the oldest entry must be replaced.

5. User interface read, data in cache: If a read from the user interface is requested and the data is in the cache, then on the next rising edge of the clock, *data_vld* will be asserted, and the data will be supplied to the user interface on the *rd_data* bus. Figure 2.2-2 demonstrates the timing for a read request with data already in the cache.

6. User interface read, data not in cache: If a read from the user interface is requested and the data is NOT in the cache, then on the next rising edge of the clock data will be fetched from the main memory. At the last cycle of the data fetch from the main memory the main memory data is stored into the cache, and is available to the user (with *data_vld==1'b1*). See Figure 2.2-3 for the timing of a main memory read with a cache miss, assuming a main memory access time of 3 cycles.

7. User interface busy: The cache controller shall assert a *busy* signal to the user interface to instruct it to stop sending read or write commands.

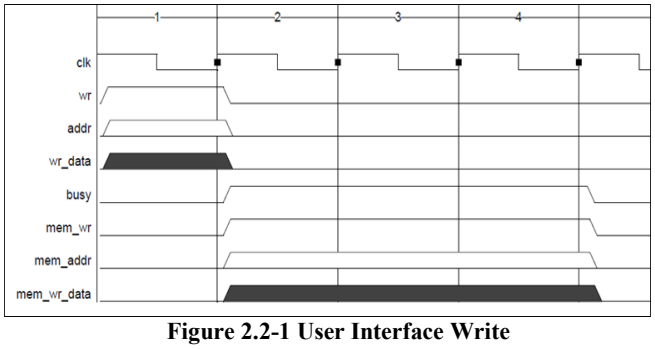

Figure 2.2-1 User Interface Write

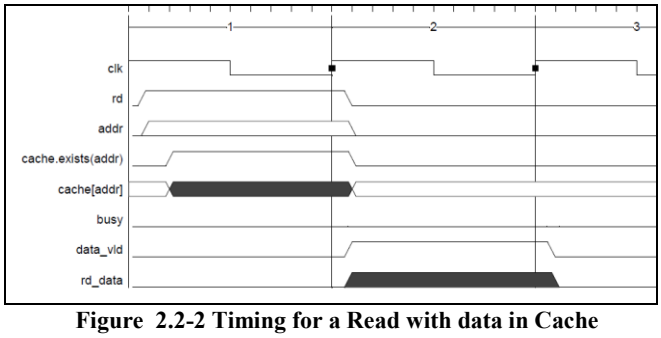

Figure 2.2-2 Timing for a Read with data in Cache

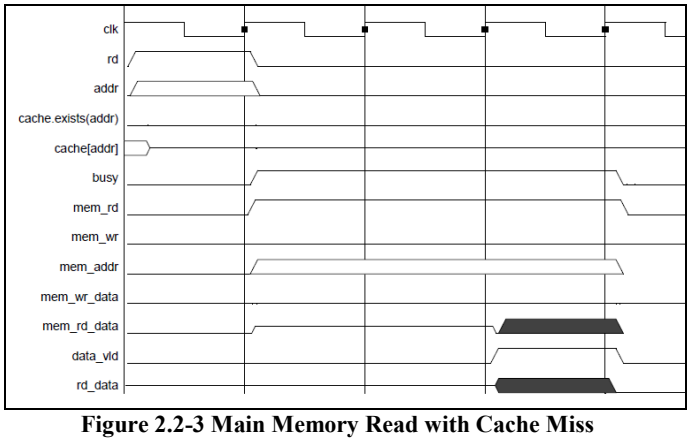

Figure 2.2-3 Main Memory Read with Cache Miss

2.3 Cache Controller Architecture

The architecture of the cache controller consists of five main elements, as shown below.

cache_mem	A cache memory instantiation
LRU_fifo	Instantiation of a FIFO emulating the least reusable algorithm
filling_cache_invalidates	Provides controls to the *LRU_fifo* and the cache memory (e.g., invalidate, push, pop)
main_mem_valid	Controls the main memory
Conditional	Select data item *rd_data*

Figure 2.3 represents a high level view of the model implementation.

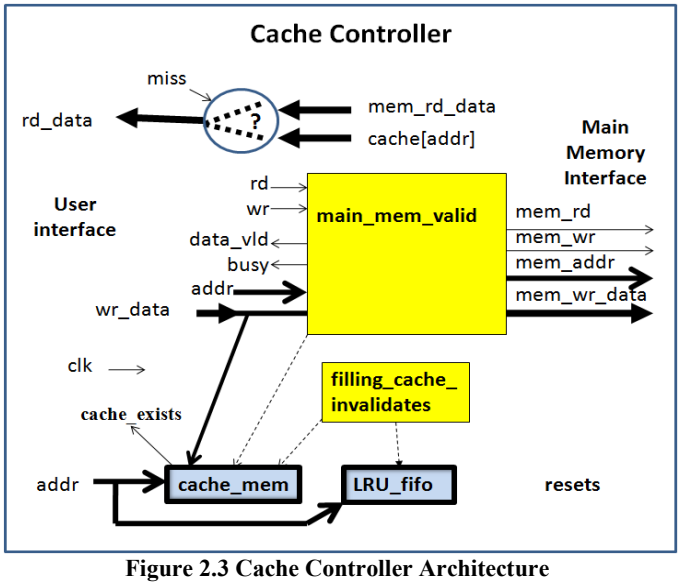

Figure 2.3 Cache Controller Architecture

2.3 Verification Environment

A simple testbench was written to provide stimulus to the design under test. Synopsys's VCS provided early access to a simulator that supports many of the new features in the 2009 release of the standard. A thorough evaluation of checkers should ideally include formal analysis as well, but the availability of tools is still limited at this time.

Since the purpose of the cache controller design is evaluating the new checker construct, we wanted to rely completely on the assertions in the checkers to verify the design. The checkers to be developed shall perform the following types of functions:

User writes: This checker checks that the main memory and the cache are updated for every user write; it also checks that the minimum access requirements are met. Since the memory is parameterized, elaboration time checks are used to confirm that the parameter value has integrity.

User reads: This checker checks that when a read miss occurs, the data is fetched from the main memory; it is then provided back to the user, and the read data is added to the cache. The *data_vld* signal is asserted *n* rising clocks after the *rd* signal is asserted, where *n* is the main memory access time. The checker also checks that when a hit (!*miss*) occurs the proper data is returned to the user and that the main memory is not accessed. The *data_vld* signal is asserted on the next rising edge of the *clk*.

Cache invalidate: When the cache is full and more data needs to be added, this checker checks that the oldest address is invalidated and then overwritten.

Data integrity: This checker checks that reads and writes to the various memories do not collide, and that control and data signals are never X or Z values.

In addition to the verification checks, cover statements will be added to the checkers.

3.0 CHECKER CONCEPTS [2]

The 2009 release of the standard professes to have improved the usability of assertion libraries with the *checker* construct. In this section, we describe what we believe are the key features that distinguish checkers from their predecessor module-based libraries. These features form the criteria for our evaluation.

Checkers group related assertions, coverage, and supporting code into individual verification units. They are designed to be able to be placed anywhere in the code; thus, they can be placed in close proximity to the code that is being evaluated. Checkers can contain static or procedural assertions, and they can be instantiated statically or procedurally. It is important to understand these classifications in order to understand how the assertions in the checker will operate.

Another new feature of the standard, not specific to checkers but allowed in checkers, are deferred immediate assertions. Deferred assertions help to prevent transient errors. Argument passing and elaboration time configuration checks make the use of libraries simpler and more reusable.

3.1 Classification of Checker Assertions

Understanding how assertions in a checker are processed requires the understanding of their classifications. Assertion statements inside a checker are classified as follows, and are shown in Figure 3.1:

1) Static concurrent assertions: Those are concurrent assertion statements that appear outside procedural code.

2) Procedural concurrent assertions. A procedural concurrent assertion is one that is in a procedural block, such an **always**, **initial**, or **final** procedure.

3) Immediate assertions. The immediate and deferred immediate assertion statements test that an expression holds. The test is performed when the statement is executed in the procedural code.

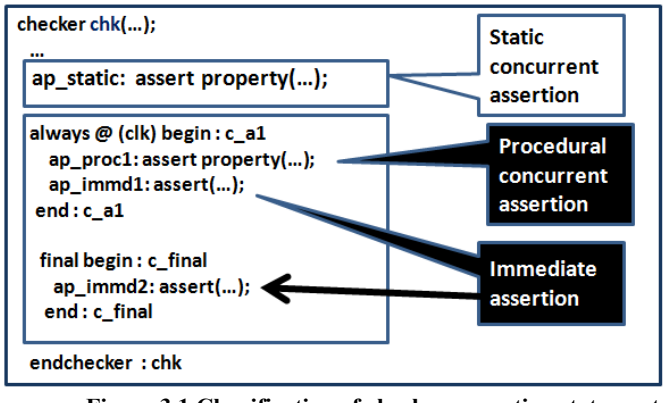

Figure 3.1 Classification of checkers assertion statements

3.2 Classification of checker Instances

A checker can then be instantiated in various units, such as modules, interfaces or programs. They can be instanced outside of procedural statements (i.e., **always**, **initial** or **final** blocks), and are called static checker instances. Checkers can also be instanced inside procedural statements, and are called procedural checker instances. As previously stated, the significance of this classification comes into play because it impacts how assertions defined in the checkers are processed by SystemVerilog. Figure 3.2 demonstrates the classification of checker instances.

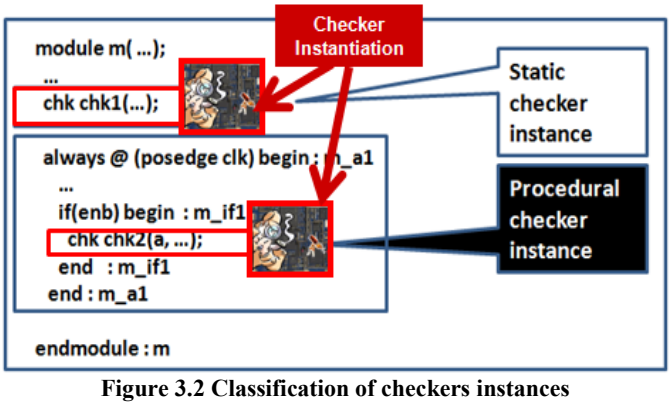

Figure 3.2 Classification of checkers instances

3.2.1 Static checker Instantiation

Checker static instances behave as instantiated code in module.

When a checker is instantiated as a static checker instance, all of its code behaves as if it were instantiated directly (or in-block) in the module after the proper argument associations are made. That code includes the static concurrent assertions and the procedural concurrent assertions. Using the checker shown in Figure 3.1, the equivalent emulation of the static checker instance in a module is shown in Figure 3.2.1. Notice that in this case, the code of the checker is directly instanced in the module, in a manner similar to an instantiation of a module.

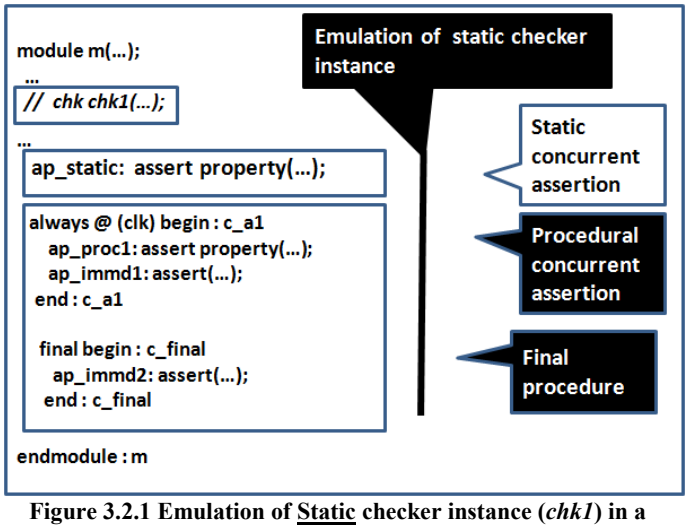

Figure 3.2.1 Emulation of Static checker instance (*chk1*) in a module

3.2.2 Procedural checker Instantiation

If a checker is instantiated in a module procedurally, then the checker's behavior depends on the type of concurrent assertions the checker includes: procedural concurrent assertions or static concurrent assertions. The rules are as follows:

Checker's static concurrent assertions are impacted by the procedural code in a module

If a module's procedural code instantiates a checker that includes static concurrent assertions, then those static concurrent assertions behave as if they were inserted inline at the point of insertion of the checker. Thus, the checker's static concurrent assertions are affected by the conditions imposed within the module's **always/initial/final** statements. For example, the static concurrent assertions within the checker will be impacted by an **if** or **case** conditions of the **always** procedure in the module that instantiates the checker.

Checker procedural concurrent and immediate assertions ignore procedural code in a module

[1] Procedural concurrent assertion statements in a checker shall be treated just like other procedural assertion statements. Thus, a procedural concurrent assertion in a checker behaves as a standalone procedural concurrent assertion, and does not inherit aspects of its enclosing checker's instantiation. The checker's procedural concurrent assertions behave as if they were directly inserted within the module, as procedure statements, and are not sensitive to the procedural code in the module in which the checker is instantiated. However, the checker's procedural assertions will be impacted by their procedural definitions. For example:

```
checker chk_test(..);
  always @ (posedge clk1)
    ap1: assert property(
          @ (posedge clk2) req |=> @ (posedge clk3) ack);
endchecker : chk_test
```

In the above procedural concurrent assertion, regardless of how the checker is instantiated, *ap1* must first wait for the clocking event (posedge clk1) before putting the assertion into the procedural assertion queue. The assertion is insensitive to any condition (**if** or **case**) under which that checker is instantiated (e.g., in a module, **always** @ (**posedge** clk) **if**(condition) chk_test *chk_test_1*(...);)

Figure 3.2.2-1 highlights a checker declaration that includes a static concurrent assertion and several procedural concurrent assertions. That checker is instantiated as a procedural checker instance within a module.

The equivalent emulation of the procedural checker instance in a module is shown in Figure 3.2.2-2. Notice that in this case, the checker's procedural concurrent assertions are directly instanced (i.e., in-block) in the module in a manner similar to module instantiation. However, the checker's static concurrent assertion is located at the point of insertion of the checker's instance (i.e., inline).

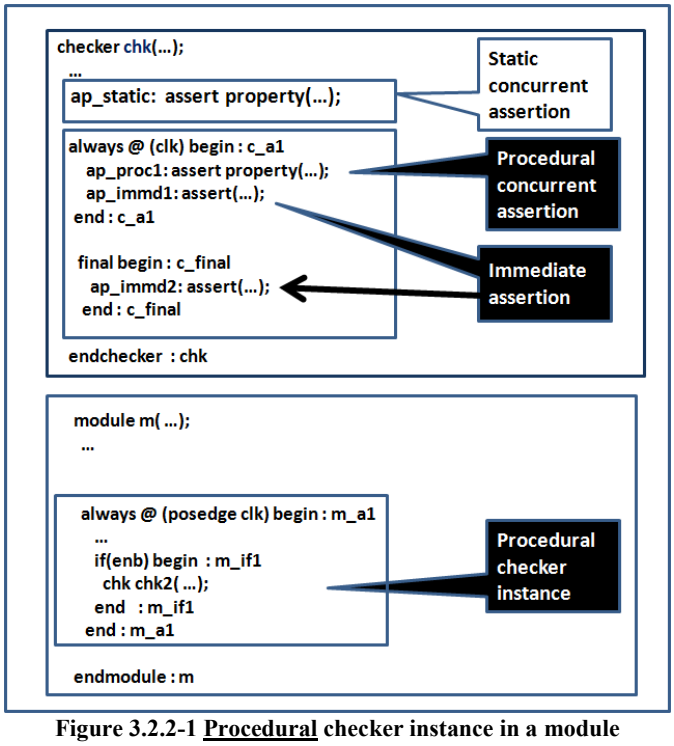

Figure 3.2.2-1 Procedural checker instance in a module

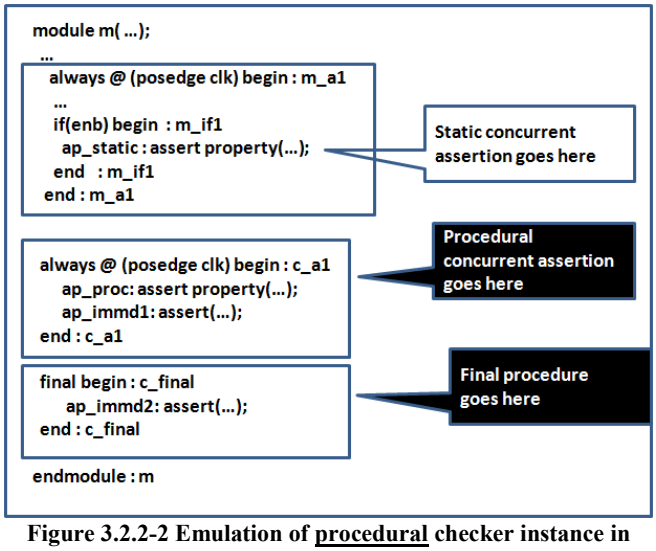

Figure 3.2.2-2 Emulation of procedural checker instance in module

3.3 Deferred Assertions

Deferred assertions are not unique to checkers; however, because they are new to the standard and they enable the creation of combinatorial library elements that were previously not possible. Deferred assertions delay the reporting of failures to later in the time step after transitioning signals have settled, and all but the last failure are filtered out. This avoids unnecessary noise in simulation reports.

For example, suppose we have an immediate assertion that checks that the *rd* and *wr* signals are never both active at the same time.

```
checker cache_integrity;
...
// read and write cannot both be active
never_rd_wr:  assert #0 ( not(rd && wr))
  $error("Simultaneous read and write are illegal");
...
endchecker : cache_integrity
```

If back to back reads and writes were legal, it is possible that the read write signals will transition from "10" -> "11" -> "01". Because the assertion reporting is delayed, the "11" value will not cause a false report. In effect, each transition flushes the previous transition so that only the stable ending value is actually evaluated.

3.4 Passing Formal Arguments

Some significant advantages exist in the way formal arguments are passed to checkers versus modules. *IEEE 1800-2009 SystemVerilog: Assertion-based Checker Libraries*[3,4,5] described this in detail. Unlike modules, the new checker construct allows for passing arguments like properties, sequences, and events. Also, defaults can be assigned to any port, including configuration parameters that are passed as part of the port list.

Inferred value system functions, *$inferred_clock* and *$inferred_disable*, exist that can be used to set the default value of a clock or a reset. The inferred value is derived from the context in which the checker is instantiated. The inferred clock is derived from procedural code first, then from a default clock specification. The inferred disable is derived from a new *default disable iff* construct if it exists; otherwise the inferred value is 1'b0.

While passing formal arguments has been simplified, it is also possible to take advantage of the checker construct without having to define any port list. Signals in a checker that are not local and not a formal parameter will resolve to the signal in the scope in which the checker is declared. So if you define the checker in the module in which it is used, it can be instantiated without formal arguments. However, as of this date, no commercials simulator seems to support nested declarations.

3.5 Configuration Checks

Configuration checks are not specific to checkers versus modules but it is new to the standard and is especially useful for library elements. A configuration check can exist anywhere that is outside of a procedural code. For example:

```
checker cache_integrity
// check that user did override the default
if (mem_size == 0)
  $error("User must configure the memory to
    a valid size.  The default value of 0  is not legal");
...
endchecker : cache_integrity
```

These types of configuration checks are identified by procedural code outside of the **always** block. They are checked at elaboration time so that configuration errors are flagged early on, before simulation even starts.

4.0 CHECKERS AND DESIGN EXPERIENCE

4.1 Checker Experience [7]

The ports and anticipated local signals of the cache controller were first declared. Because the requirements were loosely defined and needed further clarifications, the designer started to translate some of the requirements into assertions defined in a single checker. The use of checkers allowed separating assertions and supporting verification logic from the controller design. This is a very important feature of the checkers.

These assertions clarified enough details to start the controller model. This process of assertion writing and design expansion was iterated a few times until the single checker became too complex, and harder to follow. That single checker encompassed elements dealing with READS, WRITES, and CACHE accesses. It became obvious that a split of the single checker into smaller checkers, each addressing a specific functional target, is not only a better organization but also is easier to follow and maintain. Those checkers include the following:

Checker	Function
chk_immd.	**Inline assertions, *instantiated procedurally***
chk_rd_cntrl.	**Read miss/hit, *instantiated statically***
chk_rd_wr_cntlr	**RD/WR miscellaneous, *instantiated statically***
chk_wr_cntrl.	**Write through and cache, *instantiated statically***
chk_reset.	**Reset, *instantiated statically***
chk_fifo	**LRU Fifo assertions, *instantiated statically***
chk_invalidate	**Cache invalidates, instantiated procedurally**

1. The *chk_rd_cntrl* checker addresses the verification with assertions of the read transactions. Those transactions include the read from cache on a hit, the read from main memory on a miss, and the invalidation of the oldest cache entry when the cache is full. The *chk_rd_cntrl* checker also includes the validation of a cache new entry upon a miss with the verification of the timing specifications for the read from a user interface and the read of the main memory.

2. The *chk_wr_cntrl* checker addresses the verification with assertions of the write transactions. Those transactions include the write to the cache and the main memory, and the possible invalidation and rewrite of the cache entries, per the requirements. The *chk_wr_cntrl* checker also includes the verification of the timing specifications for the write from the main memory.

3. The *chk_rd_wr_cntlr* addresses the verification of items that incorporate the read and write controls, and thus does not really belong to either of the other checkers. For example, there is a need to verify that there is never a simul*taneous read and* write into t*he memory*. Another example is the verification that data written to the main memory at a specific address is correctly read later from that same address, regardless if that data was cached or not.

4. The *chk_invalidate* verifies that upon a cache *write* and the cache is *not full*, the cache line is filled. Figure 4.1-1 is the declaration of the *chk_invalidate*. Figure 4.1-2 is an example of the checker instantiated procedurally

5. The *chk_fifo* represents a move of the assertions previously written into the FIFO model into a checker.

```
import cache_ctl_pkg::*;
checker chk_invalidate(
  logic [0: 2**MAIN_MEM_ADDRESS_WIDTH-1] cache_valid,
  logic [MAIN_MEM_ADDRESS_WIDTH-1 : 0]
          [0 : (CACHE_SIZE-1)] cache_mem_addr,
  logic [CACHE_SIZE-1: 0]  cache_entry,
  logic [MAIN_MEM_ADDRESS_WIDTH-1:0] addr2invalidate,
  logic found_existing_entry,
  logic clk, rst_n);
 timeunit 1ns;   timeprecision 100ps;
 if (CACHE_SIZE > 1024) $error("cache size is too large");
 default clocking default_clk @ (posedge clk); endclocking
 default disable iff !rst_n;
function logic check_cache_entry4MT();
  automatic logic success   =0; // If==1 then it was not invalidated
  automatic int   i;
  for (i=0; i <= CACHE_SIZE -1; i++) begin : for1
    if(cache_entry[i]==1 && cache_mem_addr[i] == addr2invalidate) begin : if_1
      success  = 1; // found cache line
      break;
    end : if_1
  end : for1
  if (success) check_cache_entry4MT  =0; // was not invalidated
  else check_cache_entry4MT   =1; // was invalidated
 endfunction : check_cache_entry4MT

 ap_invalidate : assert property(  // Static assertion
     ##1 cache_valid[addr2invalidate] ==0 && check_cache_entry4MT());

 ap_nothing2invalidate: assert property (found_existing_entry)
    else $error("nothing to invalidate");
endchecker : chk_invalidate
```

Formal argument

Elaboration time check

Defaults

Function -- Must return a single value

Function used here

To be instantiated procedurally

Figure 4.1-1 Declaration of the *chk_invalidate*

.

```
always @ (posedge clk) begin : cache_mem_handling
  automatic logic success, found_existing_entry, found_empty_line;
  automatic int   i, j, found_index;
  success =0;  found_existing_entry =0;
  found_index =0;  found_empty_line =0;
  if (invalidate) begin : if1
   cache_valid[addr2invalidate]<= 1'b0;
   for (int j=0; j<= CACHE_SIZE-1; j=j+1) begin : for1
     if(cache_entry[j]==1 && cache_mem_addr[j] == addr2invalidate) begin : if2
     found_existing_entry= 1'b1;
     found_index=j;
     success=1;
     if (found_existing_entry) cache_entry[j] <=1'b0;  //  empty line
     break;
    end : if2
   end : for1
   chk_invalidate chk_invalidate_1(.*);   // checks for invalidates
end : if1
```

```
ap_invalidate : assert property(  // Static assertion
    ##1 cache_valid[addr2invalidate]  ==0 && check_cache_entry4MT());
ap_nothing2invalidate: assert property (found_existing_entry)
    else $error("nothing to invalidate");
endchecker : chk_invalidate
```

Figure 4.1-2 Example of a checker instantiated procedurally

This splitting of the original common one checker into multiple checkers, each targeted to a specific aspect of the design, helped in further understanding the design, and facilitated the coding of the

RTL. The designer then used several *emacs editor fr*ames to display the design and the checkers respectively.[6]. Having separate checkers, each addressing a specific aspect of the assertions, allows the designer to freely add supporting code without being concerned about interference or confusion with the RTL. Supporting code consisted of items such as signal declarations, generation of registered signals, function declarations, and the declaration of the **let** statements

As the design was being refined, more assertions were added into the checkers and additional checkers were added. The checkers were statically and procedurally instantiated into the cache controller. As the requirements were being further reviewed along with the assertions, several other issues were brought up. In particular:
- The lack of *chip select*, as most memory interfaces have such a signal.
- The real purpose of the *busy* signal,

Those comments were addressed with elaboration time checks, the **let** constructs, and the **assume** assertion statement. Specifically:
Lack of chip select: The design represents a partition of the whole subsystem. The *rd/wr* signals are inputs, and chip select (*cs*) is outside the DUT. It is the responsibility of the external logic to enable the *rd/wr* signals based on some external chip select. To clarify this point, we added the following IEEE 1800-2009 statements to the *chk_rd_wr_cntlr chk:*

```
let cs = rd || wr;  // demonstrates the equivalency of this signal
always @ (posedge clk) mp_no_simultaneous_rd_wr: assume property (
                                                    not (rd && wr));
```

Role of the *busy* signal:
The role of the *busy* signal is to put the burden of stopping read/write transactions when the main memory is being accessed. Since this stopping action.is outside the boundaries of the DUT, an **assume** property is used to clarify its intent. That assume statement was added into the *chk_rd_wr_cntlr chk* checker:

```
always @ (posedge clk) mp_busy_no_rd_wr: assume property (
                                                busy |-> !rd && !wr);
```

As key sets of assertions within the checkers were written, RTL code was then defined. The review process of the checkers with the code brought up errors in the code. For example, a reviewer questioned why the *push* was not in the RTL on a cache *miss*. That assertion was later replace in the *chk_rd_cntrl* checker with:

```
always @ (posedge clk) ap_fifo_push_on_miss: assert property(
                                                rd && miss |=> push);
```

The review process also brought up the point that an FSM with a fixed set of states was used to count the main memory access time. The FSM was replaced with separate read and a write cycle timers (*rd_cycle_timer*; *wr_cycle_timer*) to allow for the parameterization of the main memory access time for reads and writes. The *chk_imm* checker (discussed later on) was used inline in the *cache_controller* to verify that the proper conditions of the timer were correct upon a read.

```
if(rd && miss) begin : rd_from_main
   chk_immd chk_immd_rd_cycle_timer_not_zero(  // checker instantiation
         .the_what(rd_cycle_timer==0),
         .msg("wr signal when memory counter !=0"),
         .clk(clk));
…
```

As partitions of the RTL design and the checkers were completed and integrated, the design was verified using a simple testbench that generated random transactions. The simulation helped in debugging errors in both the design and the checkers. However, even errors in the checkers helped the designer in further understanding the intended operations of the targeted machine.

4.2 Debugging the Design

We experienced tool issues in that not all tools currently support the checker construct, as it is part of IEEE 1800-2009 that was just approved in late December 2009. Because of the tight schedule, one designer used QuestaSim to start the debug process; however, that tool does not yet support the checker construct. Thus, the debugging process relied on assertions added directly into the various modules, instead of using checker instantiations. These were later converted to checkers in the VCS environment and the results compared.

Some of the most helpful assertions were the immediate assertions injected directly into procedures at various locations within the conditional ***if*** statements. For example, in the cache memory module:

```
if (invalidate) begin : if1
 …
 ap_nothing2invalidate:
   assert(found_existing_entry) else $error("nothing to invalidate");

if (wr_cache) begin : wr_cache_updates
 …
 ap_no_cache_line_found_4write:
   assert(found_empty_line || found_existing_entry) else
     $error("attempt to write a cache line into non existent space %t", $time);
```

A better methodology that we adopted with the checkers is to create a checker that handles immediate assertions that are instantiated procedurally. This would allow consistency in code style, and the capability to add additional coverage or assertions if needed with little modifications to the RTL. We thus wrote the following checker that includes a static concurrent assertion. The checker can be instantiated wherever immediate assertions are needed. Note that an immediate assertion in a checker behaves as a procedural assertion because it has an implied *always_comb* associated with it. Thus, if a checker has immediate assertions, they will be instantiated in-block, regardless of how the checker is instantiated (statically or procedurally). An immediate static assertion is needed to enable inline behavior.

```
checker chk_immd(logic the_what, string msg, logic clk);
 // static concurrent assertion
 ap_test_now: assert property(@ (posedge clk) the_what) else
        $error("msg, the_what=%b at %t", the_what, $time);
endchecker : chk_immd
//-------------------------------------------------------------------------------------
// cache_controller
always @ (posedge clk) begin : main_mem_accesses
    …
 else  begin : else1_mt4wr// find an empty line
      …
      chk_immd chk_immd_no_cache_line( // checker instantiation
         .the_what(found_empty_line || found_existing_entry),
         .msg("L111 attempt to write a cache line into non-existent space"),
         .clk(clk));…

     end : else1_mt4wr
   end : wr_cache_updates
 end : cache_mem_handling
```

As design errors were corrected, the test engineer reached a point where the design appeared to work correctly. However, to ensure that data written into the memory is correctly read, the following assertion was written.

```
sequence q_wr_wr;
  int v_addr;
  @ (posedge clk) ($rose(wr), v_addr=addr) ##[1:$] wr && addr==v_addr;
 endsequence : q_wr_wr

 property p_wr_rd;
  int v_addr, v_data;
   disable iff(q_wr_wr.triggered)
 ($rose(wr), v_addr=addr, v_data=wr_data) |->
    ##[1:$] rd && addr==v_addr ##[1:3] rd_data==v_data;
 endproperty : p_wr_rd
 always @ (posedge clk)ap_wr_rd : assert property (p_wr_rd);
```

INCORRECT ASSERTION

After a review, it was determined that the above assertion is incorrect, even though at first glance, it seems to just ignore two write transactions to the same address. It actually ignores after the first write, all writes to the same address because of the **disable iff** condition. Thus, even though the model worked with no violations, this poorly written assertion gave a false sense of security. The point: **ASSERTIONS NEED TO BE WELL EXAMINED AND BLESSED FOR ACCURACY. Having all the assertions collocated in checkers help in the review process of assertions.** The above assertion was correctly rewritten as:

Correct assertion

```
 property p_wr_rd;
   int v_addr, v_data;
    first_match(($rose(wr), v_addr=addr, v_data=wr_data) ##1
    !(wr && addr==v_addr)[*1:$] ##1 rd && addr==v_addr) |->
                    ##[1:MEM_CYCLES] rd_data==v_data;
 endproperty : p_wr_rd
 always @ (posedge clk)
   ap_wr_rd : assert property(p_wr_rd);
```

That assertion also led to the need of another related assertion to verify that if data is written to the main memory, then the data read from the main memory at that address should not change. The assertion takes into account that the memory model can be peaked.

```
 property p_wr_rd_memory;
  int v_addr, v_data;
   first_match(($rose(wr), v_addr=addr, v_data=wr_data) ##1
   !(wr && addr==v_addr)[*1:$] ##1 rd && addr==v_addr) |->
                    main_mem[addr] ==v_data;
 endproperty : p_wr_rd_memory
 always @ (posedge clk) ap_wr_rd_memory : assert property(
                                          p_wr_rd_memory);
```

Those two assertions helped in the detection of the source of errors when the *ap_wr_rd* assertion failed at different time slots. The issue was the code that selected the source of data to the user interface from either the cache or from the main memory. Below is copy of the erroneous code and the corrected code.

```
// Erroneous code:
// assign rd_data = miss? mem_rd_data :  // If miss, return memory data
//              cache_data; // If !miss, return cache data
//-----------------------------------------------------------------
```

```
// Corrected code, after assertion pointed to region of the error:
 always @ (posedge clk) begin : rd_data_output
   if (rd && miss) begin : rd_miss_data_from_mem
        data_from_mem <= 1'b1;
   end :rd_miss_data_from_mem
   else if (rd && cache_exists) begin : rd_hit
        data_from_mem <= 1'b0;
    end : rd_hit
  end : rd_data_output

 assign rd_data = data_from_mem && data_vld ? mem_rd_data :
                                          // If miss, return memory data
          cache_data; // cache[addr];   // If hit, return cache data
```

Once this correction was done, the design seemed to be fully functional. Since we worked as a team, we needed to verify the model with the checkers and the remaining assertions. The model was then transferred to a test facility where we had access to Synopsys VCS and simulation with the checkers.

5.0 CHECKER IMPRESSIONS

The use of the checker provided several benefits during the design and verification of the cache controller. The following lists the set of observations:

1. **Benefits:** The most beneficial use of the checkers is that they provide a clear demarcation between the RTL design and the verification code.
   a. The code within the RTL is reduced in size because the verification code (assertions and supporting code) is separately encapsulated. In the cache controller model, we experienced a 75% ratio of assertion lines of code (LOC) to DUT lines of code. However, when the assertions are embedded in checkers, the ratio of checker instantiation LOC to DUT LOC was less than 3%. These statistics are shown below.

Checkers	
LOC	330
Assertions	34
Cover property	6
Declarations	7
Instantiations	10
DUT	
LOC	440
LOC: checker/DUT	**~ 75%**
Checker instances/DUT	**< 3%**

   b. There is no concern about interference or confusion between the verification code and the RTL. This separation relieves concerns that verification code can be synthesized inadvertently. Of course, there are other ways to prevent the synthesis of code, such as the use of pragmas, but the use of checkers is much clearer and less error prone.

   c. The checker provides an organized and structured solution that is amenable to building small to medium verification units.

2. **Instantiation impact:** Checkers can be instantiated statically or procedurally within RTL. This allows the insertions of checkers in locations within the RTL where they make more sense. Checkers allow for the definition of static or procedural concurrent assertions. If a checker is instantiated procedurally,

the static concurrent assertions inside the checker get affected by the conditions under which the checker is procedurally instantiated. However, regardless of the instantiation method, assertions inside of a checker's procedural code are not affected by the conditions under which the checker is instantiated. This demarcation in use model maybe confusing for a first-time user; however, it has value because a checker is a conglomeration of several assertions, and there are cases where it is desired that the instantiation method does not impact the behavior of some assertions (i.e., the assertions are not affected by a *case* or *if* condition in the instantiation of the checker). That feature is a flexibility offered to the users. However, caution must be used in the construction of the assertions within a checker.

3. **Multiple checkers:** Having separate checkers with each addressing a grouping of the assertions targeted to specific functions allowed for a more organized and structured solution; this is amenable to the building of small to medium verification units instead of a large set in the RTL or in modules bound to the design. For the case of the cache controller, we wrote checkers that addressed the read, write, the read with a cache hit, the invalidate, and generic violation rules functions,

4. **Arguments with default values using inferred functions**: IEEE 1800-2009 allows the use of optional arguments. For example:

```
default disable iff rst;
default clocking default_clk @(posedge clk1); endclocking
property pReqAck(request, acknowledge,
  reset = $inferred_disable, clk1 = $inferred_clock);
  @clk disable iff (reset)
     request |=> @ (posedge clk2) acknowledge;
endproperty : pReqAck
```

```
always @ (posedge clk) begin : reqack
  // with defaulted clock and resets
  apReqAck_ck1 : assert property(pReqAck(req, ack));
  // with specific clock, and defaulted reset
  apReqAck_ck2 : assert property(pReqAck(req, ack,,clk2));
end : reqack
```

We did not make use of inferred signals as arguments. We did not have multiple clocks and we relied on the defaults for resets and clocks.

5. **Checkers in design process:** The assertions within the checkers helped in the understanding of the requirements and the definition of the RTL. During simulation, the checkers pointed to various errors, most of them attributed to design errors, but a few to errors in the assertions within the checkers. Even though checker errors may sound like an overhead because the design was correct, they were not an overhead; they demonstrated a misunderstanding in the requirements between the actual design and the verification code. Resolution of those issues helped in better solidifying the design and requirements.

6. **Checker restrictions:** In the definition of the checkers, the use of static concurrent assertions felt more natural than the use of procedural concurrent assertions, perhaps from previous experiences with assertions. Procedural concurrent assertions were only used when it was important that the assertions are not impacted by where the checkers were to be instantiated. However, there are restrictions in the use of checkers. Those illegal constructs include the inclusion of:
   a) Parameter, localparam and specparam
   b) Module, interface, program, class
   c) Task, void functions, blocking assignments,
   d) Functions with side effects
   e) **if, for, while, case** statements (in **always,** and **initial** procedures)
   f) All hierarchical referencing, into or out of a checker, is disallowed

Even though those illegal constructs sound like an impactful deficiency in the use of checkers, we did not experience those limitations as road blockers. For example, instead of applying the **if, case**, or loop statements within the **always** procedures, functions can be used in a procedural concurrent assertions to produce the computational results of an expression. Other options include the use of those conditional statements from within the assertions. Specifically, a property statement is specified as follows:

property_statement ::=
  property_expr ;
  | ***case*** *(expression_or_dist)* *property_case_item*
   { property_case_item } **endcase**
  | ***if*** *(expression_or_dist)* *property_expr*
    [***else*** *property_expr]*

It is important though to understand where these limitations exist. Final blocks in a checker are no different than final blocks in RTL code, and the same goes for assertion action blocks; however, an action block may not call a task, as tasks are disallowed in checkers. However, an action block may assign a value to a signal. The limitations apply only to the checker body. For example, the following is legal:

```
ap1: assert property(@ (posedge clk) a |=> b) else
  begin
    c<= 1'b1;
    $info ("expected b==1")
  end ) ;
```

Another restriction in the use of the checkers is where they can be instantiated. A checker may be instantiated wherever a concurrent assertion may appear. However, it is illegal to instantiate checkers in **fork**...**join**, **fork**...**join_any**, or **fork**...**join_none** blocks. This was not an impactful restriction in the use of the checkers.

Figure 5.1 is an example of legal and illegal code extracted from the chk_*rdhit* checker (not used in the model).

```
checker chk_rdhit(
    logic data_vld, busy, rd, wr, cache_exists, clk, reset_n);
  always @ (posedge clk) begin : a1

    if(rd) begin : rd1 // Illegal in a checker
      assert(wr  == 1'b0);
      if ($past(cache_exists))  // if is illegal in a checker
        assert (busy==1'b0);
    end : rd1
  end: a1

  function logic rd_cache(logic rd, wr, cache_exists, busy);
    if(rd && !wr && cache_exists & !busy)
      rd_cache  = 1'b1;
    else
      rd_cache  = 1'b0;
  endfunction : rd_cache

 ap_rd_hit: assert property(@ (posedge clk)
              rd_noWr(rd, wr, cache_exists, busy));
 ap_rd_hit2: assert property(@ (posedge clk)
        rd && cache_exists |-> !wr ##1 !busy);
endchecker : chk_rdhit
```

If, for, case are not allowed here

if, for, while, case allowed in functions

Figure 5.1 Legal and illegal code in a checker declaration

7. **Elaboration time check**: Those checks are part of IEEE 1800-2009; they were used within the checkers to ensure that the parameters did not exceed intended limits. For example,
```
if (CACHE_SIZE > 1024) $error("cache size is too large");
```
Those elaboration time checks serve as good documentation, and as verification prior to simulation.

8. **Instantiation of undefined checkers during RTL design:** Checkers are intended to be totally passive, and do not in any way impact the environment in which they are used. For example, checkers can be instantiated during the design definition of the RTL when one really does not yet know what is in the checker, but feel that there should be something. Thus, the marking of such a checker helps in the initiation of verification code to be done at a later time. For example,
```
module dut(..);
  ...
  always @ (posedge clk) begin
   if (..) begin
     some_code;
     chk_rd_mode_fast chk_rd_mode_fast_1 (.*);
  ...
```
In the above example, one may not know what is in *chk_rd_mode,* but the designer feels that some checks are needed. In addition, that checker may in time evolve, and be changed, as the requirements may change.

6.0 SUMMARY

Our overall impression is that the checker construct does provide many advantages over the module-based library elements or simple assertions inserted directly into the design module. Checkers enable the grouping of related code and assertions into entities that can be instantiated inline with RTL. They also reduce the size of the RTL code because the verification code is external in checkers. That allows the separate detailed reviews of the assertions within the checkers. They can also be easily linked to the usage locations within the RTL code; we took advantage of the notation for checker named prefixed with "*chk_*" and used the *Unix grep* command to locate the files that used checkers (i.e., *grep chk_ *.sv > file.txt*).

Writing basic checkers of assertions is fairly straightforward because checkers bear similarities to modules. However, checkers are not identical to modules, and do have rules that must be understood.

We like the capability to group related assertions and supporting code in individual checkers, and we appreciated the capability to instantiate checkers anywhere inline with the RTL code.

Elaboration time configuration checks are very useful in catching usage errors up front, before time is spent with the simulator.

We recommend the use of checkers. The main hurdle, as of today, is tool support. However, we believe that the checker construct will be widely supported in the future.

7.0 REFERENCES

[1] *IEEE Standard for SystemVerilog Unified Hardware Design, Specification, and Verification Language*, IEEE 1800-2009.

[2] Ben Cohen, Srinivasan Venkataramanan, Ajeetha Kumari, and Lisa Piper c. 2010 *SystemVerilog Assertions Handbook, 2nd edition for Dynamic and Formal Verification,* ISBN 878-0-9705394-8-7 http://SystemVerilog.us/

[3] E. Cerny, S. Dudani, and D. Korchemny, *IEEE 1800-2009 SystemVerilog: Assertion-based Checker Libraries*, Design Verification Conference (DVCon) 2010.

[4] E. Cerny, D. Korchemny, L. Piper, E. Seligman, S. Dudani, *Verification case studies: evolution from SVA 2005 to SVA 2009*, Proc. Design Verification Conference (DVCon) 2009.

[5] *Accellera Standard Open Verification Library* (OVL), Version V2.4, Accellera, 2009.

[6] SystemVerilog Snippets for Emacs http://tinyurl.com/ygtg8cw

[7] Slides and code can be downloaded from http://SystemVerilog.us/DvCon2010/

8. 0. ACKNOWLEDGMENTS

We thank *Synopsys* for providing us access to their *VCS* platform that supports many of the SystemVerilog IEEE 1800-2009 features, including the checker construct.

We thank *Mentor Graphics* for granting us licenses of *QuestaSim.* Since the authors are located in different places (California, Florida, and Bangalore, India), having access to this simulator helped us in the early compilations and debugging of the code with inline assertions (instead of checkers).