

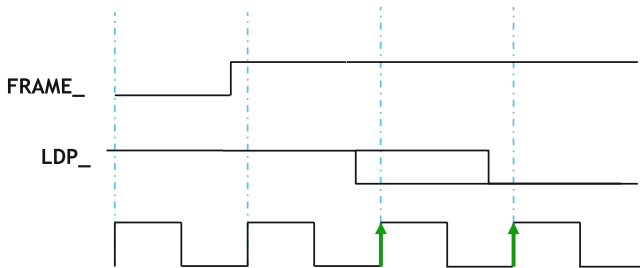
## Chapter 2

# System Verilog Assertions

### 2.1 What is an Assertion?

An assertion is simply a check against the specification of your design that you want to make sure never violates. If the specs are violated, you want to see a failure.

A simple example is given below. Whenever FRAME\_ is de-asserted (i.e. goes High), that the Last Data Phase (LDP\_) must be asserted (i.e. goes Low). Such a check is imperative to correct functioning of the given interface. SVA language is precisely designed to tackle such temporal domain scenarios. As we will see in [Sect. 2.2.1](#), modeling such a check is far easier in SVA than in Verilog. Note also



*When FRAME\_ is de-asserted, LDP\_ (last data phase) must be asserted within the next 2 clocks*

```
property ldpcheck;  
  @(posedge clk) $rose (FRAME_) |-> ##[1:2] $fell (LDP_);  
endproperty  
  
aP: assert property (ldpcheck) else $display("ldpcheck FAIL");  
cP: cover property (ldpcheck) $display("ldpcheck PASS");
```

**Fig. 2.1** A simple bus protocol design and its SVA property

that assertions work in temporal domain (and we will cover a lot more on this later); and are concurrent as well as multi-threaded. These attributes is what makes SVA language so suitable for writing temporal domain checks.

Figure 2.1 shows the assertion for this simple bus protocol. We will discuss how to read this code and how this code compares with Verilog in the immediately following [Sect. 2.2.1](#).

## 2.2 Why Assertions? What are the Advantages?

As we discussed in the introductory section, we need to increase productivity of the design/debug/simulate/cover loop. Assertions help exactly in these areas. As we will see, they are easier to write than standard Verilog or SystemVerilog (thereby increasing design productivity), easier to debug (thereby increasing debug productivity), provide functional coverage and simulate faster compared to the same assertion written in Verilog or SystemVerilog. Let us see these advantages one by one.

### 2.2.1 Assertions Shorten Time to Develop

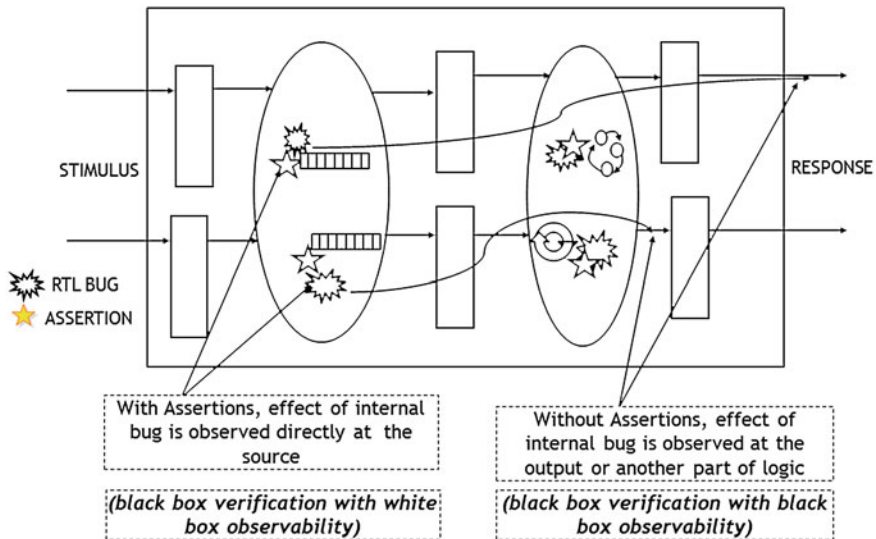
Referring to the timing diagram in Fig. 2.1, let us see how SVA shortens time to develop. The SVA code is very self-explanatory. There is the property ‘ldpcheck’ that says “at posedge clock, if FRAME\_ rises, it implies that within the next 2 clocks LDP\_ falls”. This is almost like writing the checker in English. We then ‘assert’ this property, which will check for the required condition to meet at every posedge clk. We also ‘cover’ this property to see that we have indeed exercised the required condition. But we are getting ahead of ourselves. All this will be explained in detail in coming chapters. For now, simply understand that the SV assertion is easy to write, easy to read and easy to debug.

Now examine the Verilog code for the same check (Fig. 2.2). There are many ways to code this. One of the ways at behavioral level is shown. Here you ‘fork’ out two procedural blocks; one that monitors LDP and another that waits for 2 clocks. You then disable the entire block (‘ldpcheck’) when either of the two procedural blocks complete. As you can see that not only is the checker very hard to read/interpret but also very prone to errors. You may end up spending more time debugging your checker than the logic under verification.

Verilog Code
<pre> always @(posedge FRAME_) begin : ldpcheck   @(posedge clk);   fork     begin       @(negedge LDP_) disable ldpcheck;     end     begin       repeat (2) @(posedge clk); \$display("ldpcheck FAIL");       disable ldpcheck;     end   join end </pre>

**Fig. 2.2** Verilog code for the simple bus protocol

### 2.2.2 Assertions Improve Observability



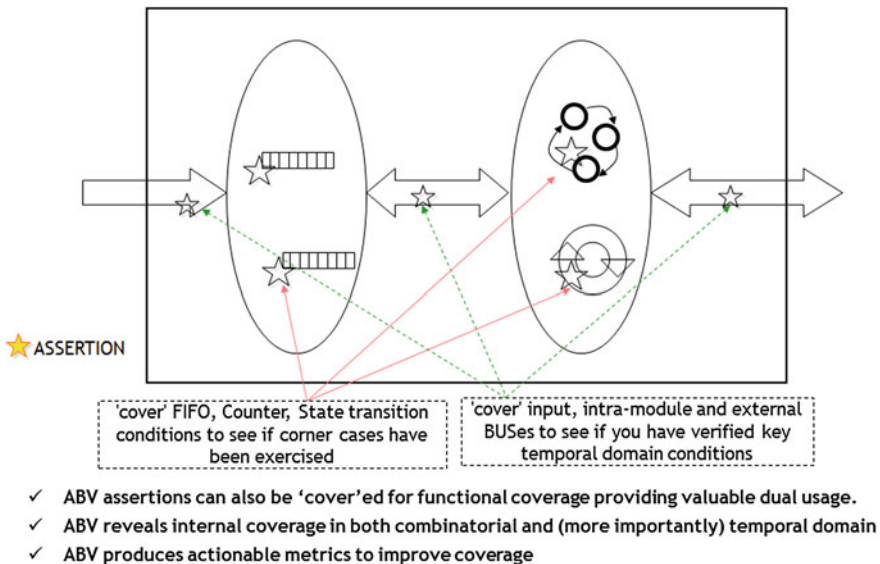
**Fig. 2.3** Assertions improve observability

One of the most important advantage of assertions is that they fire at the source of the problem. As we will see in the coming chapters, assertions are located local to temporal conditions in your design. In other words, you don't have to back trace a bug all the way from primary output to somewhere internal to the design where the bug originates. Assertions are written such that they are close to logic (e.g. @

(posedge clk) state0 l-> Read); Such an assertion is sitting close to the state machine and if the assertion fails, we know that when the state machine was in state0 that Read did not take place. Some of the most useful places to place assertions are FIFOs, Counters, block to block interface, block to IO interface, State Machines, etc. These constructs in RTL logic is where many of the bugs originate. Placing an assertion that check for local condition will fire when that local condition fails, thereby directly pointing to the source of the bug (Fig. 2.3).

Traditional verification can be called Black Box verification with Black Box observability, meaning, you apply vectors/transactions at the primary input of the 'block' without caring for what's in the block (blackbox verification) and you observe the behavior of the block only at the primary outputs (blackbox observability). Assertions on the other hand allow you to do black box verification with white box (internal to the block) observability.

### 2.2.3 Assertions Provide Temporal Domain Functional Coverage



**Fig. 2.4** SystemVerilog assertions provide temporal domain functional coverage

Assertions not only help you find bugs but also help you determine if you have covered (i.e. exercised) design logic, mainly temporal domain conditions. They are very useful in finding temporal domain coverage of your test-bench. Here is the reason why this is so important (Fig. 2.4).

Let us say, you have been running regressions 24\*7 and have stopped finding bugs in your design. Does that mean you are done with verification? No. Not finding a bug could mean one of two things. (1) There is indeed no bug left in the design or (2) you have not exercised (or covered) all the required functions of the design. You could be continually hitting the same piece of logic in which no further bugs remain. In other words, you could be reaching a wrong conclusion that all the bugs have been found.

In brief, coverage includes three components (we will discuss this in detail in [Chap. 19](#)). (1) Code Coverage (which is structural) which needs to be 100 %; (2) Functional Coverage that need to be designed to cover *functionality* of the entire design and must be completely covered; (3) temporal domain coverage (using SVA ‘cover’ feature) which need to be carefully designed to fully cover all required temporal domain conditions of the design.

Ok, let us go back to the simple bus protocol assertion that we saw in the previous section. Let us see how the ‘cover’ statement in that SVA assertion works. The code is repeated here for readability.

```
property ldpcheck;
  @(posedge clk) $rose (FRAME_) |-> ##[1:2] $fell (LDP_);
endproperty

aP: assert property (ldpcheck) else $display("ldpcheck FAIL");
cP: cover property (ldpcheck) $display("ldpcheck PASS");
```

In this code, you see that there is a ‘cover’ statement. What it tells you is “did you exercise this condition” or “did you cover this property”. In other words and as discussed above, if the assertion never fires, that could be because of two reasons. (1) you don’t have a bug or (2) you never exercised the condition to start with! With the cover statement, if the condition gets exercised but does not fail you get that indication through the ‘pass’ action block associated with the ‘cover’ statement. Since we haven’t yet discussed the assertions in any detail, you may not completely understand this concept but *determination of temporal domain coverage of your design is an extremely important aspect of verification and must be made part of your verification plan.*

To reiterate, SVA supports the ‘cover’ construct that tells you if the assertion has been exercised (covered). Without this indication and in the absence of a failure, you have no idea if you indeed exercised the required condition. In our example, if FRAME\_ never rises, the assertion won’t fire and obviously there won’t be any bug reported. So, at the end of simulation if you do not see a bug or you do not even see the “ldpcheck PASS” display, you know that the assertion never fired. In other words, you must see the ‘cover property’ statement executed in order to know that the condition did get exercised. We will discuss this further in coming chapters. Use ‘cover’ to full extent as part of your verification methodology.

### 2.2.3.1 Assertion Based Methodology Allows for Full Random Verification

Huh! What does that mean? This example, I learnt from real life experience. In our projects, we always do full random concurrent verification (i.e. all initiators of the design fire at the same time to all targets of the design) after we are done with directed and constrained random verification. The idea behind this is to find any deadlocks (or livelock for that matter) in the design. Most of the initiator tests are well crafted (i.e. they won't clobber each other's address space) but with such massive randomness, your target model may not be able to predict response to randomly fired transactions. In all such cases, it is best to disable scoreboards in your target models (unless the scoreboards are full proof in that they can survive total randomness of transactions) BUT keep assertions alive. Now, fire concurrent random transactions, the target models will respond the best they can but assertions will pin point to a problem if it exists (such as simulation hang (deadlock) or simply keep clocking without advancing functionality (livelock)).

In other words, assertions are always alive and regardless of transaction stream (random or directed), they will fire as soon as there is the detection of an incorrect condition.

- Example Problem Definition:
  - Your design has Ethernet Receive and Video as Inputs and is also a PCI target.
  - It also has internal initiators outputting transactions to PCI targets, SDRAM, Ethernet Transmit and Video outputs.
  - After you have exhausted constrained random verification, you now want to simulate a final massive random verification, blasting transactions from all input interfaces and firing transactions from internal masters (DMA, Video Engine, Embedded processors) to all the output interfaces of the design.
  - BUT there's a good chance your reference models, self-checking tests, scoreboards may not be able to predict the correct behavior of the design under such massive randomness.
- Solution:
  - Turn off all your checking (reference models, scoreboards, etc.) unless they are full proof to massive random transaction streams.
  - BUT keep Assertions alive.
  - Blast the design with massive randomness (keep address space clean for each initiator).
  - If any of the assertions fire, you have found that corner case bug.

### 2.2.3.2 Assertions Help Detect Bugs not Easily Observed at Primary Outputs

This is a classic case that we encountered in a design and luckily found before tape-out. Without the help of an assertion, we would not have found the bug and there would have to be a complex software workaround. I will let the following example explain the situation.

- *The Specification:*
  - On a store address Error, the address in Next Address Register (NAR) should be frozen the same cycle that the Error is detected.
- *The Bug:*
  - On a store address error, the state machine that controls the NAR register actually froze the address the next clock (instead of the current address the same clock when store address error occurred). In other words, an incorrect address was being stored in NAR.
- *So, why were the tests passing with this bug?*
  - The tests that were triggering this bug used the same address back to back. In other words, even though the incorrect ‘next’ address was being captured in NAR, since the ‘next’ and the ‘previous’ addresses were the same, the logic would seem to behave correct.
  - *The Assertion:* An assertion was added to check that when a store address error was asserted the state machine should not move to point to the next address in pipeline. Because of the bug, the state machine actually did move to the next stage in pipeline. The assertion fired and the bug was caught.

### 2.2.3.3 Other Major Benefits

- *SVA language supports Multi- Clock Domain Crossing (CDC) logic*
  - SVA properties can be written that cross from one clock domain to another. Great for data integrity checks while crossing clock domains.
- *Assertions are Readable: Great for documenting and communicating design intent.*
  - Great for creating executable specs.
  - Process of writing assertions to specify design requirements && conducting cross design reviews identify
    - Errors, Inconsistencies, omissions, vagueness
  - Use it for design verification (test plan) review.

- *Reusability for future designs*
  - Parameterized assertions (e.g. for a 16 bit bus interface) are easier to deploy with the future designs (with a 32 bit bus interface).
  - Assertions can be modeled outside of RTL and easily bound (using ‘bind’) to RTL keeping design and DV logic separate and easy to maintain.
- *Assertions are always ON*
  - Assertions never go to sleep (until you specifically turn them off).
  - In other words, active assertions take full advantage of every new test/stimulus configuration added by monitoring designs behavior against the new stimulus.
- *Acceleration/Emulation with Assertions*
  - Long latency and massive random tests need acceleration/emulation tools. These tools are beginning to support assertions. Assertions are of great help in quick debug of long/random tests. We will discuss this further in coming sections.
- *Global Severity Levels (\$Error, \$Fatal, etc.)*
  - Helps maintain a uniform Error reporting structure in simulation.
- *Global turning on/off of assertions (as in \$dumpon/\$dumpoff)*
  - Easier code management (no need to wrap each assertion with an on/off condition).
- *Formal Verification depends on Assertions*
  - The same ‘assert’ions used for design is also used directly by formal verification tools. Static formal applies its algorithms to make sure that the ‘assert’ion never fails.
  - ‘assume’ allows for correct design constraint important to formal.
- *One language, multiple usage*
  - ‘assert’ for design check and for formal verification
  - ‘cover’ for temporal domain coverage check
  - ‘assume’ for design constraint for formal verification.

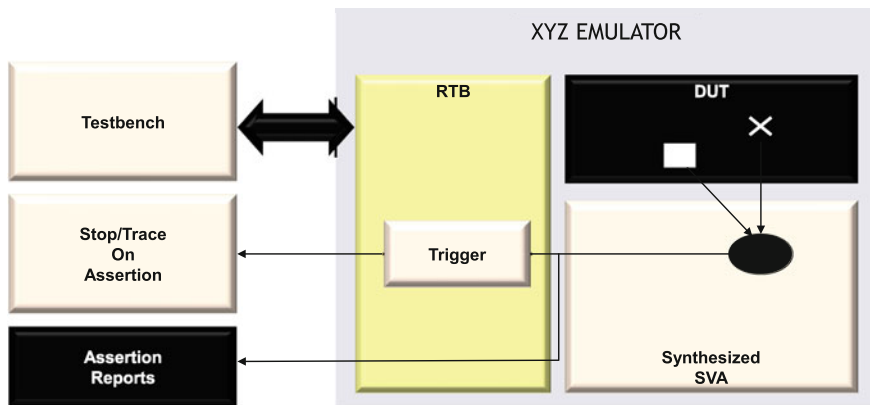
## 2.3 How do Assertions Work with an Emulator?

This section is to point out that assertions are not only useful in software based simulation but also hardware based emulation. The reason you can use assertions to fire directly in hardware is because assertions are synthesizable (well, at least the simpler ones). Even though assertion synthesis has ways to go, there is enough of a subset covered by synthesis and that is sufficient to deploy assertions in hardware.



In Fig. 2.5 a generic emulation system is shown. Synthesizable assertions are part of the design that get synthesized and get partitioned to the emulation hardware. During emulation, if the design logic has a bug, the synthesized assertion will fire and trigger a stop/trace register to stop emulation and directly point to the cause of failure.

Anyone who has used emulation as part of their verification strategy, very well know that even though emulator may take seconds to ‘simulate’ the design, it takes hours thereafter to debug failures. Assertions will be a great boon to the debug effort. Many commercial vendors now support synthesizable assertions.

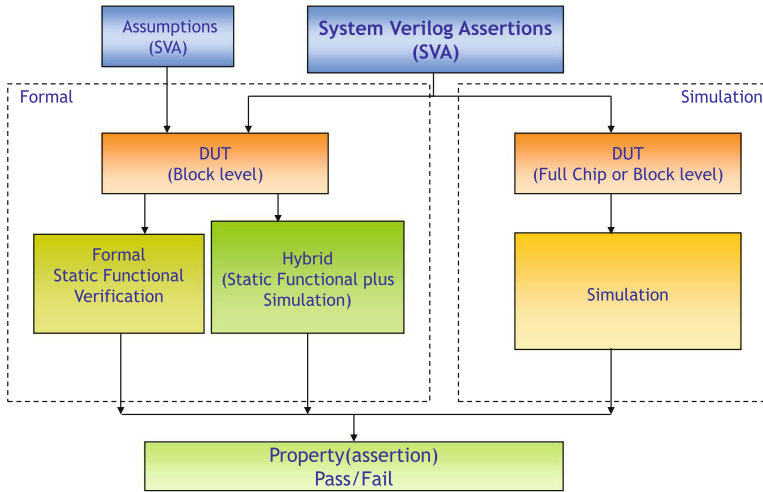


- Synthesizable assertions supported by XYZ Emulator
- Embedded and bound assertions are automatically synthesized and mapped into the emulator
- Assertion reporting and triggering at run-time, similar to simulation
- Compile time and Run time control over Assertion compilation and reporting/triggering

**Fig. 2.5** Assertions for hardware emulation

On the same line of thought, assertions can be synthesized in silicon as well. During post-silicon validation, a functional bug can fire and a hardware register can record the failure. This register can be reflected on GPIO of the chip or the register can be scanned out using JTAG boundary scan. The output can be decoded to determine which assertion fired and which part of silicon caused the failure. Without such facility, it takes hours of debug time to pin point the cause of silicon failure. This technique is now being used widely. The ‘area’ overhead of synthesized assertion logic is negligible compared to the overall area of the chip but the debug facilitation is of immense value. Note that such assertions can make it easier to debug silicon failures in field as well.

## 2.4 Assertions in Static Formal



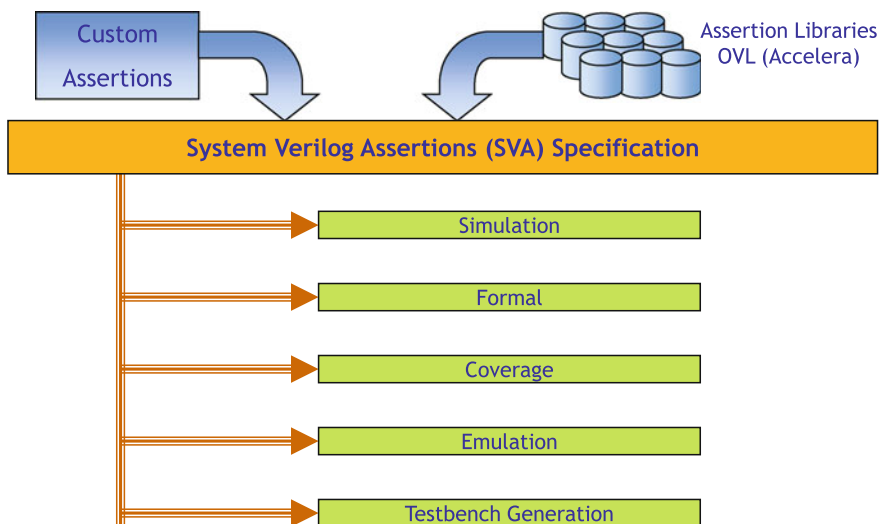
**Fig. 2.6** Assertions and assumptions in formal (static functional) and simulation

The same assertions that you write for design verification can be used with static functional verification or the so-called hybrid of static functional plus simulation algorithms. Figure 2.6 shows (on LHS) SVA *Assumptions* and (on RHS/Center) SVA *Assertions*. As you see the assumptions are most useful to *Static Formal Verification* (aka *Formal*) (even though assumptions can indeed be used in Simulation as well, as we will see in later sections) while SVA assertions are useful in both Formal and Simulation.

So, what is Static Functional Verification (also called Static Formal Functional)? In plain English, static formal is a method whereby the static formal algorithm applies all possible combinational and temporal domain possibilities of inputs to exercise all possible ‘logic cones’ of a given logic block and see that the assertion(s) are not violated. This eliminates the need for a test-bench and also makes sure that the logic never fails under *any* circumstance. This provides 100 % comprehensiveness to the logic under verification. So as a side note, why do we ever need to write a test-bench!!! The static formal (as of this writing) is limited by the size of the logic block (i.e. gate equivalent RTL) especially if the temporal domain of inputs to exercise is large. The reason for this limitation is that the algorithm has to create different logic cones to try and prove that the property holds. With larger logic blocks, these so-called logic cones explode. This is also known as ‘state space explosion’. To counter this problem, simulation experts came up with the *Hybrid Simulation* technique. In this technique, simulation is deployed to ‘reach’ closer to the assertion logic and then employ the static functional verification algorithms to the logic under assertion checking. This

reduces the scope of the # of logic cones and their size and you may be successful in seeing that the property holds. Since static functional or hybrid is beyond the scope of this book, we'll leave it at that.

## 2.5 One Time Effort, Many Benefits



**Fig. 2.7** Assertions and OVL for different uses

Figure 2.7 shows the advantage of assertions. Write them once and use them with many tools.

We have discussed at high level the use of assertions in Simulation, Formal, Coverage and Emulation. But how do you use them for Test-bench Generation/Checker and what is OVL assertions library?

*Test-bench Generation/Checker:* With ever-increasing complexity of logic design, the test benches are getting ever so complex as well. How can assertions help in designing test-bench logic? Let us assume that you need to drive certain traffic to a DUT input under certain condition. You can design an assertion to check for that condition and upon its detection the FAIL action block triggers, which can be used to drive traffic to the DUT. Checking for a condition is far easier with assertions language than with SystemVerilog alone. Second benefit is to place assertions on verification logic itself. Since verification logic (in some cases) is even more complex than the design logic, it makes sense to use assertions to check test-bench logic also.

*OVL Library :* Open Verification Library. This library of predefined checkers were written in Verilog before PSL and SVA became mainstream. Currently the

library includes SVA (and PSL) based assertions as well. The OVL library of assertion checkers is intended for use by design, integration and verification engineers to check for good/bad behavior in simulation, emulation and formal verification. OVL contains popular assertions such as FIFO assertions, among other. OVL is still in use and you can download the entire standard library from Accellera website: <http://www.accellera.org/downloads/standards/ovl>.

We will not go into the detail of OVL since there is plenty of information available on OVL on net. OVL code itself is quite clear to understand. It is also a good place to see how assertions can be written for ‘popular’ checks (e.g. FIFO) once you have better understood assertion semantics.

## 2.6 Assertions Whining

Maybe the paradigm has now shifted but as of this writing there is still a lot of hesitation on adopting SVA in the overall verification methodology. Here are some popular objections.

- *I don't have time to add assertions. I don't even have time to complete my design. Where am I going to find time to add assertions?*
  - That depends on your definition of “completing my design”. If the definition is to simply add all the RTL code without –any– verification/debug features in the design and then throw the design over the wall for verification, your eventual ‘working design’ will take significantly longer time.
  - During design you are already contemplating and assuming many conditions (state transition assumptions, inter-block protocol assumptions, etc.). Simply convert your assumptions into assertions as you design. They will go a long way in finding those corner case bugs even with your simple sanity test benches.
- *I don't have time to add assertions. I am in the middle of debugging the bugs already filed against my design.*
  - Well, actually you will be able to debug your design in shorter time, if you *did* add assertions as you were designing (or at least add them now) so that if a failing test fires an assertion, your debug time will be drastically short.
  - Assertions point to the source of the bug and significantly reduce time to debug as you verify your block level, chip level design.
  - In other words, this is a bit of chicken & egg problem. You don't have time to write assertions but without these assertions you will spend a lot more time debugging your design!
- *Isn't writing assertions the job of a verification engineer?*
  - Not quite. Design Verification (DV) engineers do not have insight into the micro-architectural level RTL detail. But the real answer is that BOTH

Design and DV engineers need to add assertions. We will discuss that in detail in upcoming section.

- *DV engineer says: I am new to assertions and will spend more time debugging my assertions than debugging the design.*
  - Well, don't you spend time in debugging your test bench logic? Your reference models? What's the difference in debugging assertions? If anything, assertions have proven to be very effective in finding bugs and cutting down on debug time.
  - In my personal experience (over the last many SoC and Processor projects), approximately 25 % of the total bugs reported for a project were DV bugs. There are significant benefits to adding assertions to your test-bench that outweigh the time to debug them.
- *The designer cannot be the verifier also. Doesn't asking a designer to add assertions violate this rule?*
  - As we will discuss in the following sections, assertions are added to check the 'intent' of the design and validate your own assumptions. You are not writing assertions to duplicate your RTL. Following example make it clear that the designer do need to add assertions.
  - For example,

For every 'req' issued to the next block, I will indeed get an 'ack' and that I will get only 1 'ack' for every 'req'. This is a cross module assumption which has nothing to do with how you have designed your RTL. You are not duplicating your RTL in assertions.

My state machine should never get stuck in any 'state' (except 'idle') for more than 10 clocks.

### ***2.6.1 Who Will Add Assertions? War Within!***

*Both Design and Verification engineers need to add assertions...*

- Design Engineers:
  - *Micro architectural level decisions/assumptions are not visible to DV engineers.* So, designers are best suited to guarantee uArch level logic correctness.
  - *Every assumption is an assertion.* If you assume that the 'request' you send to the other block will always get an 'ack' in 2 clocks; that's an assumption. So, design and assertion for it.
  - *Add assertions as you design your logic, not as an afterthought.*

- DV Engineers:

- *Add assertions to check macro functions and Chip/SoC level functionality.*

Once the packet has been processed for L4 layer, it will indeed show up in the DMA queue.

A machine check exception indeed sets PC to the exception handler address.

- *Add assertions to check Interface IO logic*

After Reset is de-asserted none of the signals ever go 'X'.

If the processor is in Wait Mode and no instructions are pending that it must assert a SleepReq to memory subsystem within 100 clocks.

On Critical Interrupt, the external clock/control logic block must assert CPU\_wakeup within 10 clocks.

## 2.7 A Simple PCI Read Example: Creating an Assertions Test Plan

Let us consider a simple example of PCI Read. Given the specification in Fig. 2.8, what type of assertions would the design team add and what type would the verification team add? The tables below describe the difference. I have only given few of the assertions that could be written. There are many more assertions that need to be written by verification and design engineers. However, this example will act as a basis for differentiation.

Designers add assertions at micro-architecture level while verification engineers concentrate at system level, specifically the interface level in this example.

We will model the assertions for this PCI protocol later in the book under LAB6 exercise. It is too early to jump into writing assertions without knowing the basics at this stage.

The PCI protocol is for a simple READ. With FRAME\_ assertion, AD address and C\_BE\_ have valid values. Then IRDY\_ is asserted to indicate that the master is ready to receive data. Target transfers data with intermittent wait states. Last data transfer takes place a clock after FRAME\_ is de-asserted.

Let us see what type of assertions need to be written by design and verification engineers (Tables 2.1 and 2.2).

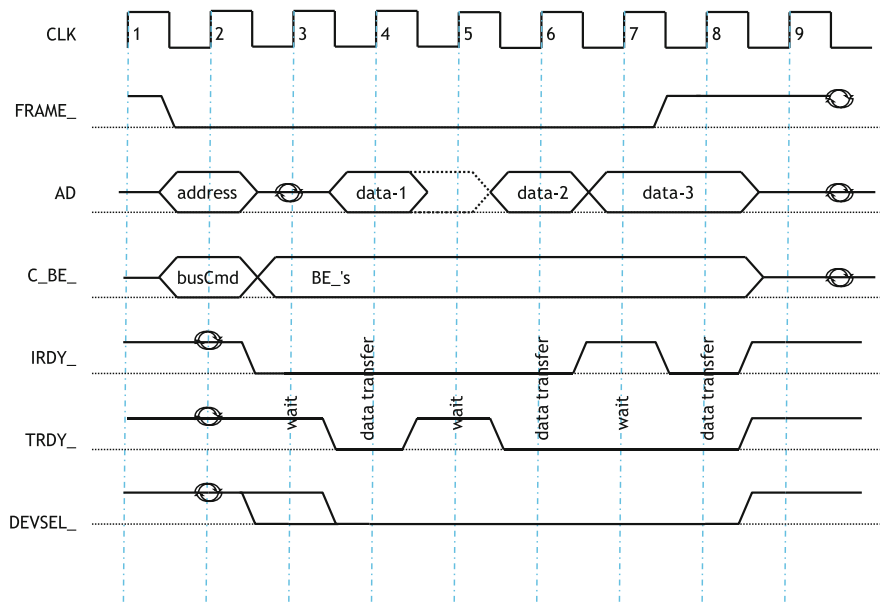


Fig. 2.8 A simple PCI read protocol

Table 2.1 PCI read protocol test plan by functional verification team

Property name	Description	Property fail?	Property covered?
Protocol interface assertions			
checkpci_AD_CBE (check1)	On falling edge of FRAME_, AD and C_BE_ bus cannot be unknown		
checkpci_DataPhase (check2)	When both IRDY_ and TRDY_ are asserted, AD or C_BE_ bus cannot be unknown		
checkPCI_Frame_Irdy (check3)	FRAME can be de-asserted only if IRDY_ is asserted		
checkPCI_trdyDevsel (check4)	TRDY_ can be asserted only if DEVSEL_ is asserted		
checkPCI_CBE_during_trx (check5)	Once the cycle starts (i.e. at FRAME_ assertion) C_BE_ cannot float until FRAME_ is de-asserted		

Table 2.2 PCI read protocol test plan by design team

Property name	Description	Property fail?	Property covered?
Microarchitectural Assertions			
check_pci_adrcbe_St	PCI state machine is in 'adr_cbe' state the first clock edge when FRAME_ is found asserted		
check_pci_data_St	PCI state machine is in 'data_transfer' state when both IRDY_ and TRDY_ are asserted		
check_pci_idle_St	PCI state machine is in 'idle' state when both FRAME_ and IRDY_ are de-asserted		
check_pci_wait_St	PCI state machine is in 'wait' state if either IRDY_ or TRDY_ is de-asserted		

Note that in the table there are two columns. (1) Did the property FAIL? (2) Did the property get covered? There is no column for the property PASS. That is because, ‘cover’ in an assertion triggers only when a property is exercised but does not fail; in other words it passes. Hence, there is no need for a PASS column. This ‘cover’ column tells you that you indeed covered (exercised) the assertion and that it did not fail. When the assertion FAILs, it tells you that the assertion was exercised (covered) and that it Failed during the exercise.

## 2.8 What Type of Assertions Should I Add?

It is important to understand and plan for the types of assertions one needs to add. Make this part of your verification plan. It will also help you partition work among your team.

Note the ‘performance implication’ assertions. Many miss on this point. Coming from processor background, I have seen that these assertions turn out to be some of the most useful assertions. These assertions would let us know of the (e.g.) cache read latency upfront and would allow us enough time to make architectural changes.

- RTL Assertions (design intent)
  - Intra Module;
    - Illegal state transitions; deadlocks; livelocks;
    - FIFOs, onehot, etc.
- Module interface Assertions (design interface intent)
  - Inter-module protocol verification; illegal combinations (ack cannot be ‘1’ if req is ‘0’); steady state requirements (when slave asserts write\_queue\_full, master cannot assert write\_req);
- Chip functionality Assertions (chip/SoC functional intent)
  - A PCI transaction that results in Target Retry will indeed end up in the Retry Queue.
- Chip interface Assertions (chip interface intent)
  - Commercially available standard bus assertion VIPs can be useful in comprehensive check of your design’s adherence to std. protocol such as PCIX, AXI, etc.
  - Every design assumption on IO functionality is an assertion.
- Performance Implication assertions (performance intent)
  - Cache latency for read; packet processing latency; etc. to catch performance issues before it’s too late. This assertion works like any other. For example, if the ‘Read Cache Latency’ is greater than 2 clocks, fire the assertion. This is an easy to write assertion with very useful return.



## 2.9 Protocol for Adding Assertions

- Do not duplicate RTL
  - White box observability does not mean adding an assertion for each line of RTL code. This is a very important point, in that if RTL says ‘req’ means ‘grant’, don’t write an assertion that says the same thing!! Read on.
  - Capture the intent

For example, a Write that follows a Read to the same address in the request pipe will always be allowed to finish before the Read. This is the intent of the design. How the designer implements reordering logic is not of much interest. So, from verification point of view, you need to write assertions that verify the chip specifications.

A note here that the above does not mean you do not add low-level assertions. Classic example here is FIFO assertions. Write FIFO assertions for all FIFOs in your design. FIFO is low-level logic, but many of the critical bugs hang around FIFO logic and adding these assertions will provide maximum bang for your buck.

- Add assertions throughout the RTL design process
  - They are hard to add as an after-thought.
  - Will help you catch bugs even with your simple block level test bench.
- If an assertion did not catch a failure.
  - If the test failed and none of the assertions fired, see if there are assertions that need to be added which would fire for the failing case.
  - The newly added assertion is now active for any other test that may trigger it. Note: This point is very important towards making a decision if you have added enough assertions. In other words, if the test failed and none of the assertions fired, there is a good chance you still have more assertions to add.
- Reuse
  - Create libraries of common ‘generic’ properties with formal arguments that can be instantiated (reused) with ‘actual’ arguments. We will cover this further in the book.
  - Reuse for the next project.

## 2.10 How do I Know I have Enough Assertions?

- It’s the “Test plan, test plan, test plan...”
  - Review and re-review your test plan against the design specs.
  - Make sure you have added assertions for every ‘critical’ function that you must guarantee works.

- If tests keep failing but assertions do not fire, you do not have enough assertions.
  - In other words, if you had to trace a bug from primary outputs (of a block or SoC) without any assertions firing that means that you did not put enough assertions to cover that path.
- ‘formal’ (aka static formal aka static functional verification) tool’s ability to handle assertions
  - What this means is that if you don’t have enough ‘assertion density’ (meaning if a register value does not propagate to an assertion within 3 to 5 clocks—resulting in assertions sparsely populated within design), the formal analysis tool may give up on the state/space explosion problem. In other words, a static functional formal tool may not be able to handle a large temporal domain. If the assertion density is high, the tool has to deal with smaller cone of logic. If the assertion density is sparse, the tool has to deal with larger cone of logic in both temporal and combinatorial space and it may run into trouble.

## 2.11 Use Assertions for Specification and Review

- Use assertions (properties/sequences) for specification
  - DV (Design Verification) Team:
 

Document as much of the ‘response checking’ part of your test plan as practical directly into executable properties.  
Use it for verification plan review and update
  - Design Team:
 

Document micro-arch. level assertions directly into executable properties.  
Use it for design reviews.
- Assertions Cross-Review
  - Review:
 

DV team reviews macro, chip, interface level assertions with the design team.
  - Cross Review
 

Block A designer reviews module B interface assertions  
Block B designer reviews module A interface assertions
  - Mis-assumptions, incorrect communication are detected early on.

## 2.12 Assertion Types

There are three kinds of assertions supported by SVA. In brief, here's their description. We will discuss them in plenty detail throughout this book.

- Immediate Assertion
- Concurrent Assertion
- Deferred Assertion (introduced in IEEE 1800-2009)

### Immediate Assertions

- Simple *non-temporal domain assertions* that are executed like statements in a procedural block,
- Interpreted the same way as an expression in the conditional of a procedural 'if' statement
- Can be specified only where a procedural statement is specified.

### Concurrent Assertions

- These are *temporal domain assertions* that allow creation of complex sequences using clock (sampling edge) based semantics.
- They are edge sensitive and not level sensitive. In other words, they must have a 'sampling edge' on which it can sample the values of variables used in a sequence or a property.



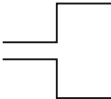



### Deferred Assertions (introduced in IEEE 1800-2009)

- Deferred assertions are a type of Immediate assertions. Note that immediate assertions evaluate immediately without waiting for variables in its combinatorial expression to *settle* down. This also means that the immediate assertions are very prone to glitches as the combinatorial expression settles down and may fire multiple times. On the other hand, deferred assertions do not evaluate their sequence expression until the end of time stamp when all values have settled down (or in the reactive region of the time stamp).

If some of this does not quite make sense, which is OK. That is what the rest of the book will explain. Let us start with Immediate assertions and understand its semantics. We then move on to Concurrent assertions and lastly Deferred assertions. The book focuses on concurrent assertions because that is really the main gist of SystemVerilog Assertion Language.

## 2.13 Conventions Used in the Book

**Table 2.3** Conventions used in this book

	LEVEL SENSITIVE HIGH: This symbol means that the signal is detected HIGH (level sensitive) at the clock edge noted in a timing diagram. It could have been high or low the previous clock and may remain high or low after the clock edge. <i>It does NOT however mean that a 'posedge' is expected on this signal at the noted clock edge</i>
	LEVEL SENSITIVE LOW: This symbol means that the signal is detected LOW (level sensitive) at the clock edge noted in a timing diagram. It could have been high or low the previous clock and may remain high or low after the clock edge. <i>It does NOT however mean that a 'negedge' is expected on this signal at the noted clock edge</i>
	EDGE SENSITIVE HIGH: This symbol means that a posedge is expected on this signal
	EDGE SENSITIVE LOW: This symbol means that a negedge is expected on this signal
	PROPERTY PASSES: This symbol means that a sequence/property match is detected here (i.e. the sequence/property PASSES)
	PROPERTY FAILS: This symbol means that a sequence/property did not match here (i.e. the sequence/property FAILS)

***Note that the level sensitive attribute of a signal is shown as a 'fat' High and Low symbol.*** I could have drawn regular timing diagrams but saw that they look very cumbersome and does not easily convey the point. Hence, I chose the fat arrow to convey that ***when the fat arrow is high, the signal was high before the clock; at the clock and after the clock. The same applies for the fat low arrow.***

For edge sensitive assertions, I chose the regular timing diagram to distinguish them from the level sensitive symbol.

A high (thin) arrow is for PASS and a low (thin) arrow is for FAIL (Table 2.3).

SystemVerilog Assertions and Functional Coverage  
Guide to Language, Methodology and Applications  
Mehta, A.B.

2014, XXXIII, 356 p. 260 illus., Hardcover

ISBN: 978-1-4614-7323-7