IP Reuse Hardening via Embedded Sugar Assertions

Erich Marschner[1], Bernard Deadman[2], Grant Martin[1]

ABSTRACT

Rapid design of complex chips requires acquisition and integration of reusable IP blocks. Effective reuse of an IP block requires documenting the designer's understanding of the block, particularly its interface requirements and any assumptions about its internal operation. Assertions embedded in the design can capture this knowledge so that any errors in the configuration or use of the IP will be caught quickly. Sugar 2.0, an upcoming standard property specification language being developed by the Accellera Formal Verification Technical Committee [1], provides a powerful means of expressing such assertions.

1. INTRODUCTION

Design reuse has become mandatory as design complexity continues to increase and market windows continue to shrink. Rapid creation of new, ever more complex designs demands that significant portions of each design must be acquired from other sources and integrated to achieve the end goal. Reuse of intellectual property (IP), or virtual components (VC), whether developed by the same company or purchased from an IP supplier, has become a fact of life for System on Chip (SoC) development.

But IP reuse is challenging. IP must be explicitly designed for reuse, and it must be packaged with all the information necessary to enable its effective reuse, including configuration, synthesis, and especially verification information. The fact that "soft" IP may be further modified by the consumer makes verification even more challenging. Current guidelines for reuse, such as [4] and [6], are starting to reflect these new reuse requirements.

IP designed for reuse must be modular in all senses. It must perform a well-defined function, with well-defined interfaces. Knowledge about its function and its interfaces must be captured with the IP, so that the IP can be used in the design of chips that are far removed in time or space from the design team that originally built the IP.

Documenting such knowledge is necessary, but not sufficient--the knowledge must be understood and used correctly to be effective. Ideally, we would like to document this knowledge in a way that (a) travels with the design, (b) is both human- and machine-readable, to facilitate both IP-based design and verification of IP blocks and SoCs containing IP blocks, and (c) is based on standardized languages so that it might be rapidly inserted into design flows, and that a variety of tools will be available to comprehend it. Doing so effectively "hardens" the IP by fixing in place its (original) intended functionality and interface requirements, to facilitate verification both *in situ* and standalone, therefore enabling more reliable reuse, even when modified.

We can accomplish this through the use of assertions embedded within the design. Such assertions can specify both the functional characteristics of the environment (as visible through the interface) that must be satisfied for the IP to work correctly, as well as the designer's assumptions about the internal operation of the IP block. Embedded assertions appear in context in the design, so they are more easily understood by downstream consumers. At the same time, embedded assertions can be simulated and formally verified, so they can help verify both configurations or modifications of the IP block and the environment in which the block is inserted.

Various embedded assertion mechanisms have been used in the past. VHDL has always had combinational assertion statements, and various approaches have been used for Verilog design [5]. One particular assertion language, Sugar 2.0, is about to become an Accellera standard. Originally developed at IBM Haifa Research Laboratory, Sugar was selected over several other candidates and then extended to address a wide range of requirements for the specification of design properties. An embedding strategy that allows Sugar assertions to be included inline in the text of an HDL model enables use of Sugar to capture interface requirements, internal implementation assumptions, and related information required to harden IP for reuse.

Using Sugar, the designer can record his or her knowledge about the design while it is being created, so that this knowledge is available later when the IP is being reused. Interface information, such as clock relationships, reset sequences, and the protocol requirements for communication between the IP block and its environment can be expressed in Sugar assertions embedded in the design. HDL modules containing embedded assertions can be used to package sets of related assertions to create reusable "checkers", or monitors, for standard protocols. Internal details, such as encoding requirements, correct operation of queues, and even performance expectations can be represented as Sugar assertions.

Hardening an IP block with Sugar assertions helps both the designer and downstream consumers of the IP verify that the block works correctly by itself, both as originally designed and as modified or configured by the IP consumer. Interface assertions can provide constraints for use in general or directed-random simulation of the block, or can be interpreted as assumptions for formal verification of the block. Internal assertions will catch errors in simulation sooner, and closer to the actual source of the error, and long before the effects propagate to, and become observable at, a system output. This is particularly valuable for the consumer who has modified IP acquired from elsewhere, since such modifications could perturb the original functionality

[1] Cadence Design Systems, Inc.

[2] Structured Design Verification, Inc.

of the block, and because the consumer may not understand the original implementation of the IP fully.

Sugar assertions also help verify the SoC into which an IP block has been integrated. During simulation, interface assertions within the IP block continually check that the SoC is interacting correctly with the IP block, and that the IP block in turn is interacting correctly with its environment. Failures that do occur are immediately observed and reported by the relevant embedded assertion, thus providing quick detection and direct diagnosis of the error. Coverage analysis of assertion activity can even contribute to determination of the degree to which the IP block has been verified in the context of the SoC.

This paper focuses on the use of embedded Sugar assertions to express interface requirements. Section 2 of this paper presents an overview of the Sugar 2.0 assertion language and how Sugar assertions can be embedded in designs. Section 3 describes the use of Sugar assertions to express clock and reset requirements, protocol requirements, legal transaction forms, and arbitration requirements.

2. Overview of Sugar 2.0

Sugar 2.0 supports a wide range of facilities for specification and verification of design behavior. This section provides a brief overview of the language. For a more complete definition, see [3].

The Sugar 2.0 language definition is segmented into four 'layers': boolean, temporal, modeling, and verification. These layers are further partitioned into three 'flavors': Verilog, VHDL, and IBM's internal Environment Description Language (EDL). The temporal layer is further partitioned into operators and statements with Linear Temporal Logic (LTL) semantics on the one hand, and operators and statements with Computation Tree Logic (CTL) semantics on the other. In this context we consider only the LTL portion of Sugar 2.0, and all the examples will be presented using the Verilog flavor of Sugar.

2.1 Booleans

Sugar 2.0 is designed to be used in conjunction with hardware description languages such as Verilog or VHDL. At the bottom level, Sugar specifications refer to signals, variables, and values within an HDL description of a design.

A Sugar specification identifies the conditions that define behavior of interest in the design. Those conditions are represented by HDL expressions that can be interpreted as having Boolean values. Using the underlying HDL syntax and semantics for such expressions ensures that the assertion language can cover the full range of behavior that can be described in the HDL, and that there is no chance of semantic mismatch between the HDL description of a behavior and the Sugar description of the same behavior.

2.2 Sequences

Sugar 2.0 includes *Sugar Extended Regular Expressions* (SEREs), which describe sequences of boolean conditions that occur at successive clock cycles. A boolean expression is by definition a SERE; more complex SEREs can be constructed as follows (where a,b,c,d are boolean expressions or subordinate SEREs):

successive conditions {a; b; c; d}

one-cycle overlap {{a; b} : {c; d}} (equal to {a; b *and* c; d})

alternate sequences {{a; b} | {c; d}}

parallel sequences[3] {{a; b} & {c; d}}

parallel sequences {{a; b} && {c; d}}

Terms in a SERE may be followed by a square-bracketed qualifier which indicates how many instances of the term must occur, and whether or not the instances must be contiguous. For example, for SERE r, and boolean b:

r[*] = a sequence of zero or more contiguous occurrences of r

r[+] = {r; r[*]}

r[*n] = a sequence of n contiguous occurrences of r

b[=n] = any sequence containing n occurrences of b

b[->n] = any sequence ending in the nth occurrence of b

In general, the repeat count n shown above can be replaced by a range of values (e.g., [3:5], or for an unbounded range, [0:inf]). The + and * qualifiers may stand alone, without a preceding SERE, in which case they represent the indicated number or range of cycles (i.e., the default SERE is 'True'). For example:

[*2] = True[*2]

2.3 Properties

Sugar supports standard LTL operations. Additional and more readable operators are defined in terms of the base operators as part of a "syntactic sugar" layer of the language, from which Sugar gets its name. The base operators and for some, their more readable equivalents, are as follows:

!f		(f does not hold)
f1 & f2		(f1 and f2 both hold)
f1 | f2		(f1 or f2 or both hold)
f1 -> f2		(f1 implies f2)
f1 <-> f2		(f1 -> f2 *and* f2 -> f1)
G f	always f	(f holds in every cycle)
G !f	never f	(f does not hold in any cycle)
X f	next f	(f holds in the next cycle, if any)
X! f	next! f	(f holds in the next cycle)
F f	eventually! f	(f holds in some future cycle)
[f1 U f2]	f1 until! f2	(f1 holds until f2 eventually holds)
[f1 W f2]	f1 until f2	(f1 holds until f2 holds, if ever)

The '!' character identifies "strong" operators, which require that the terminal condition of the property must occur, e.g. the strong form (a until! b) holds only if a is true until b is true, *and* b is eventually true, whereas the weak form (a until b) holds even if a is true forever and b is never true.

Sugar also supports operators build properties out of SEREs:

{r1} |-> {r2}	({r2} starts in the last cycle of {r1})
{r1} |=> {r2}	({r2} starts in the first cycle after {r1})
{r} (f)	(f holds in the last cycle of {r})

[3] There are two conjunction operators on SEREs: & and &&. The & operator allows one operand to be shorter than the other; the && operator requires that both operands be of the same length.

Sugar also defines both strong and weak forms of the following "syntactic sugar" operators (only the weak form shown here):

f1 before f2 (f1 holds before f2 holds)

within({r1},b) {r2} ({r2} occurs after {r1} and before b)

Also defined are variants of until, before, and within that require the termination condition to overlap one cycle with the fulfilling condition (indicated by a trailing underscore on the operator name) – e.g., within_({start}, stop) {busy[*]} is satisfied only if busy is still true in the cycle in which stop is asserted. Finally, Sugar defines always{r}, never{r}, and eventually{r} on SEREs.

2.4 Clocking and Evaluation

Boolean expressions involved in sequences and properties are evaluated at clock edges. Sugar 2.0 allows use of multiple clocks, with transparent synchronization between clocks. This enables Sugar assertions to easily express any kind of behavior, whether it is asynchronous, or single-clock synchronous, or multiple-clock synchronous, or a mixture. Sugar also allows definition of a default clock that applies if no explicit sampling clock is attached to an expression.

2.5 Declarations and Directives

Sequences and properties can be declared and given names, so that complex properties can be decomposed into, or composed from, simpler elements. Such declarations can include parameters, so that generic sequences or properties can be declared. When the name of a parameterized sequence or property is referenced, the relevant actual parameter values are provided along with the name, and the actual parameter values replace the parameter definitions in that instance of the sequence or property.

A sequence or a property by itself just describes behavior; there is no inherent obligation for the behavior to occur. Directives specify whether a given property is expected to hold (**assert**), assumed to hold (**assume**), or both. Similarly, other directives specify whether verification should exclude situations in which a given sequence occurs (**restrict**), or ensure that other sequences are encountered during verification (**cover**).

2.6 Embedded Assertions

Although Sugar 2.0 depends upon an underlying HDL for the syntax and semantics of its Boolean expressions, Sugar is nonetheless a separate language that is defined to exist side-by-side with the HDL description to which it refers. However, there is clearly a strong advantage to embedding Sugar specifications directly within the HDL description, so that they can be maintained along with the HDL source code, and so they are guaranteed to travel with the IP and be available for verification of the IP in subsequent applications.

To that end, Sugar 2.0 implementers have used a pragma approach to allow Sugar specifications to be written within comments in the HDL code, but in such a way as to be recognizable by verification tools that are aware of the pragma convention. This has enabled Sugar properties to be available for Sugar-cognizant verification tools, yet invisible to tools that should be unaffected by the Sugar specifications. It should be understood that the Sugar examples given in the remainder of this paper would typically be embedded in the design using this pragma convention.

3. INTERFACE REQUIREMENTS

One of the main problems associated with IP reuse is the challenge of ensuring that the environment in which an IP block is used drives inputs to the IP block appropriately. The required behavior of inputs to an IP block is a necessary part of the documentation needed for successful IP reuse. Defining such requirements as embedded assertions helps ensure that the requirements are indeed met, by checking for incorrect input behavior during verification.

3.1 Clock and Reset Requirements

Assertions can be valuable even when they express the most basic requirements on the inputs to an IP block. Clock and reset input requirements, in particular, are so simple that they can be forgotten in the process of documenting the more interesting aspects of an IP block, yet they are so fundamental that any errors in driving them can cause pervasive and difficult-to-diagnose problems.

Suppose an IP block requires a two-phase, non-overlapping clock, and both c1 and c2 are inputs of the block. We can assert the following to ensure that the 'non-overlapping' requirement is satisfied:

```
assert never (c1 && c2);
```

This assertion will fail if c1 and c2 ever occur at the same time. In particular, it will fail immediately if the IP user is not aware of the clocking requirement and plugs the IP block into an environment that generates c1 and c2 in such a way that they overlap. If such a clocking error were not caught right away, the IP block might still function, but exhibit incorrect behavior that could be difficult to diagnose. The assertion ensures that this will not happen.

This assertion illustrates one style of interface specification, in which we disallow (e.g., with 'never') incorrect behavior. An alternative style involves require (e.g., with 'always') correct behavior. For example, we might require a two-phase, non-overlapping clocking scheme by writing the following assertion:

```
assert always {c1} |->
  {(c1 && !c2)[+]; (!c1 && !c2)[*];
    (!c1 && c2)[+]; (!c1 && !c2)[*];
      (c1)};
```

This says that whenever clock c1 is high, c1 and c2 go through one two-phased, non-overlapping clocking cycle, with c1 and c2 each high for one or more evaluation cycles, separated by zero or more cycles in which both c1 and c2 are low, and ending with c1 being high again. The assertion both documents the required behavior of the clock input and implicitly precludes incorrect behavior (e.g., overlap of c1 and c2).

The same approach can be used to define required behavior of a reset input. Suppose the same IP block has a reset input, and once that input goes low, it must be held low for 3 cycles of c1, and raised high by the following occurrence of c2, in order to reset the block correctly. We can express this as follows:

```
assert always {reset; !reset} |-> {!reset@c1 [*3]; reset@c2};
```

This assertion will fail if the IP block is reset incorrectly – e.g., if, once reset goes low, it is not held low long enough, or if it does not rise at the appropriate time.

3.2 Protocol Requirements

While some aspects of IP block interfaces involve relatively independent signals such as clock and reset inputs, most communication between an IP block and its environment takes place via standard or proprietary communication protocols that involve multi-cycle, coordinated interaction of many interdependent signals. Such protocols impose many requirements on both the IP block and on its environment. Unless those requirements are met, communication between the block and the environment may fail, and the block may not work correctly as a result.

Suppose an IP block interacts with memory in the SoC via the Advanced High-Performance Bus (AHB) protocol supported by the ARM Advanced Microcontroller Bus Architecture (AMBA) [2]. The AHB protocol specification specifies how an AHB master and slave must interact. Assertions can capture these requirements, both for human-readable documentation and for verification via simulation or formal verification tools.

The AHB protocol samples signals at the rising edge of signal HCLK. Sugar allows the relevant clock edge to be specified as part of the property, but it is more convenient to define a default clock, especially for a single-clocked system such as the AHB protocol. The following default clock declaration specifies that all properties presented following it are clocked by the rising edge of HCLK:

```
default clock = rose(HCLK);
```

The AHB protocol imposes many requirements on the manner in which the various components (master, slave, arbiter) attached to an AHB bus must interact. For example, the AHB specification states that if a master inserts a cycle delay into a burst transfer by specifying a transfer type of BUSY in a given cycle, then in the next cycle, the slave must assert HREADY and set HRESP to "OKAY". The required relationships between the master's HTRANS output and the slave's HRESP and HREADY outputs are illustrated in Figure 1.

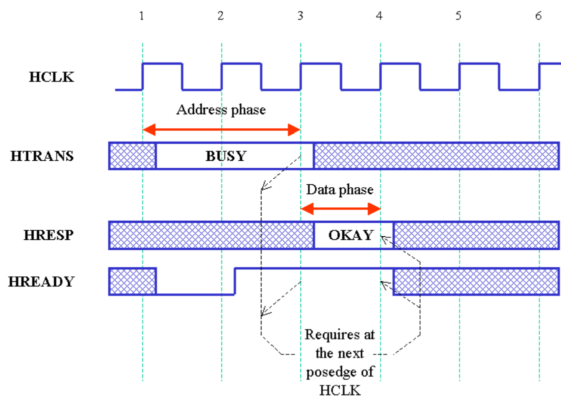

Figure 1. Relationship of HTRANS, HRESP, and HREADY.

This required relationship can be expressed with the following assertion:

```
assert always (HREADY && HTRANS==`BUSY) ->
  next (HREADY && HRESP==`OKAY);
```

This assertion will fail if the memory subsystem in the SoC into which this IP block is inserted does not implement correctly this aspect of the AHB protocol.

It is often the case that a protocol allows for multiple alternative behaviors. Which behavior actually occurs at any given time depends upon dynamic factors such as timing, loading, etc., but all of the behaviors are legal. For example, the AHB specification defines all the legal slave responses to a master request. The AHB protocol requires a slave to respond to a transfer request in any of four ways:

- signal successful completion
- signal an error
- signal that the master should request it again immediately
- signal that the master should request it again later

The response may be preceded by a series of wait cycles. An error, retry, or split response must be held for two cycles in order to flush the 2-stage address/data pipeline.

These are the only valid responses. By asserting that every request issued by the IP block receives one of these four valid responses, we can exclude all possible illegal behavior. We can express such alternative behaviors in Sugar 2.0 using disjunction of SEREs. The following assertion can express this requirement:

```
assert always { HREADY } |=>
   { ( !HREADY && (HRESP==`OKAY) )[*];
     {
       { HREADY && (HRESP==`OKAY)}
     | { !HREADY && (HRESP==`ERROR);
         HREADY && (HRESP==`ERROR) }
     | { !HREADY && (HRESP==`SPLIT);
         HREADY && (HRESP==`SPLIT) }
     | { !HREADY && (HRESP==`RETRY);
         HREADY && (HRESP==`RETRY) }
     }
   };
```

This assertion says that, following a cycle in which the HREADY signal is asserted (i.e., driven high) by the slave, the slave may insert any number of wait cycles (HREADY low and HRESP set to "OKAY"), and then any one of four alternative responses must occur: the one-cycle "completed successfully" response (with HREADY high and HRESP set to "OKAY"), or the two-cycle ERROR, SPLIT, or RETRY responses in which HREADY is low in the first of two cycles and high in the second.

The above assertion illustrates a "time-slice"-oriented style of expressing required behavior, in which each alternative behavior is described as a single sequence, and for each cycle of the sequence the required values of all relevant (i.e., non-don't care) signals are specified. Another style is more "signal-slice"-oriented, in which each signal involved is described as a separate

sequence, and various sequences are conjoined, in parallel. This is illustrated in the following assertion, which is equivalent to the above:

```
assert always { HREADY } |=>
 { ( !HREADY && (HRESP==`OKAY) )[*];
   {
     { HREADY && (HRESP==`OKAY)}
   | { {!HREADY;HREADY} && {(HRESP==`ERROR)[*2]} }
   | { {!HREADY;HREADY} && {(HRESP==`SPLIT)[*2]} }
   | { {!HREADY;HREADY} && {(HRESP==`RETRY)[*2]} }
   }
 };
```

In either case, the assertion characterizes the required behavior of an AHB slave with respect to the HREADY and HRESP signals. The assertion would catch any failure of the environment to correctly implement this aspect of the AHB protocol, and would detect such failures in the first cycle in which the behavior of the system diverges from that required by the assertion.

For example, Figure 2 illustrates a situation in which an AHB protocol error would be detected. At time 2, HREADY is high; this is followed by a slave-inserted wait cycle (HREADY low and "OKAY") and then by an ERROR response – but in contrast to the AHB protocol requirements, as reflected in the assertions above, HREADY stays low for two cycles of "ERROR" response. This is not allowed by the protocol, and this error would be caught at time 5 and reported as a violation of the assertion.

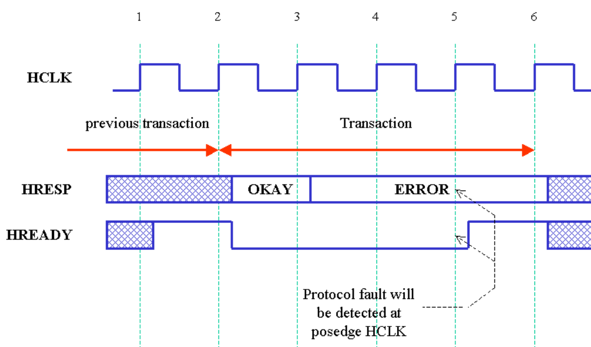

Figure 2. Incorrect "Error" Response

Similarly, Figure 3 illustrates a case in which a legal response (a wait cycle) occurs in the middle of another legal response (an error response). Here again, the assertion will fail, because this combination of responses is not among the legal alternatives listed in the assertion.

The ability to specify all legal behavior, and in doing so to rule out any illegal behavior, is clearly a very powerful method of defining verification goals for the design or its environment. Typically, there are fewer legal behaviors than there are illegal behaviors, so specifying alternative legal behaviors, and thereby implying that all other behavior is illegal, often can be much more effective than identifying and disallowing individual error cases.

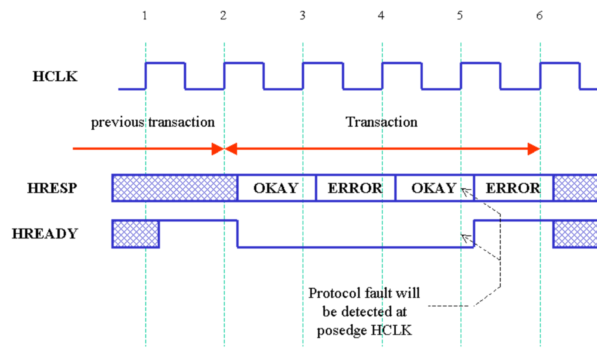

Figure 3. Spurious "Wait" Cycle

3.3 Transaction Modeling

We can take this approach one step further, by using Sugar to model all the alternative behaviors in an entire transaction. For example, the AHB protocol supports burst-mode read operations. Although burst-mode operations can be somewhat complex, nonetheless they can be defined using Sugar 2.0 sequences and assertions.

Consider the case of a burst-mode read of undefined length. This involves a sequence of data transfers from slave to master, during which either the slave or master or both can insert wait cycles if they are not ready to continue. Address and data "tenures" are pipelined, so that the control signals for the next transfer are set up while the previous data is being transferred, and they are held until the slave signals completion of the previous data transfer request. A simplified description of such an operation is as follows:

```
while slave is handling previous data,
  master requests first data transfer;
repeat
  while slave is handling previous data,
    master requests next data transfer, or signals 'busy';
until done;
```

The description above can be expressed in Sugar 2.0 as a combination of sequences of conditions, as follows. First, assume the slave response requirement discussed in section 3.2 above is now defined as the following sequence (where NotReady, Ready, Error, Split, and Retry are defined as the obvious sub-sequences):

```
sequence SlaveResponse = {
  NotReady[*];
  { Ready | Error | Split | Retry }
};
```

Next, we can define the control conditions required for the various aspects of a burst-mode read operation:

```
`define FirstTransfer (HTRANS==`NONSEQ)
`define NextTransfer (HTRANS==`SEQ)
`define MasterBusy   (HTRANS==`BUSY)
`define ReadIncr     (!HWRITE && (HBURST==`INCR))
```

Next, we can define the initial and subsequent read operations, showing the slave response to the previous data in parallel with the master's assertion of the control signals for the next address:

```
sequence ReadFirst = {
   {SlaveResponse} &&
   {(`FirstTransfer && `ReadIncr)[*]}
};
sequence ReadNext ={
   {SlaveResponse} &&
   { {(`NextTransfer && `ReadIncr)[*]} | {`MasterBusy[*]} }
};
sequence BurstModeRead = {
   {ReadFirst} ;
   {ReadNext}[*]
};
```

If all valid transaction types were defined in this way, then we could write a Sugar 2.0 assertion that requires only valid transactions to occur following the completion of any previous transaction (indicated by HREADY going high), e.g.:

```
assert {HREADY} |=> {
   BurstModeRead
  | BurstModeWrite
  | SingleRead
  | SingleWrite
  | Inactive
  | Reset
};
```

3.4 Arbitration Requirements

The requirements imposed by a protocol do not affect only the blocks that communicate via the protocol. Just as important are the requirements imposed on the arbitration logic that governs which block can act as master at a given time. While these requirements tend to be more global in nature, rather than applying only to a particular block (such as an IP block being reused in a new context), correct operation of a given IP block may depend upon some aspects of the arbitration logic provided by the environment. Verification that these requirements are met by the environment therefore becomes relevant to reuse of such an IP block.

The AHB protocol specifies that, once a SPLIT has been requested by a Slave, the Master that initiated the transfer can not be granted access to the bus, and cannot therefore complete any transfers until the Slave has signaled for and completed the SPLIT transaction. This limits the memory required in the implementation of a slave, and therefore any IP block that acts as a slave on the AHB bus may want to ensure that this aspect of the arbitration logic is correctly implemented by the environment. This requirement can be expressed as follows:

```
assert forall m in {0:15} :
  always (HREADY &&
        (HMASTER == m) && (HRESP==`SPLIT)) ->
    next (HMASTER != m) until! (HSPLIT[m]);
```

The assertion says that, for each of the possible masters (up to the 16 allowed by the AHB protocol) if a SPLIT response occurs for a request from master m, then starting from the next cycle, unit m is not allowed to be master again, until the split transaction is completed (indicated by HSPLIT[m] being set), and furthermore (as indicated by use of the strong operator 'until!') that HSPLIT[m] must eventually be set, i.e., the split must eventually complete.

We might also include an assertion requiring that the correct master must eventually complete a transfer after the slave (i.e., this IP block) has asserted HSPLIT[m]. This could be expressed as follows:

```
assert forall m in {0:15} :
  always (HSPLIT[m]) ->
    eventually ((HMASTER == m) && HREADY && HSEL);
```

With assertions such as these embedded in the IP block, any incorrect behavior of the arbitration logic can be caught at its source, before such incorrect arbitration behavior causes the IP block to appear to function incorrectly.

4. SUMMARY

Embedded Sugar assertions enable reuse hardening of design IP. Assertions capture the original designer's knowledge about the design, especially about the interface requirements of the design, and embedded assertions record this information directly within the IP itself, so that such knowledge is guaranteed to be available to downstream consumers and their verification tools. The result is more reliable reuse of IP in the design of large, complex chips.

Various styles of specification are supported by Sugar, ranging from explicitly excluding error conditions to modeling all legal behavior and requiring only legal behavior. Which style is more appropriate depends upon the situation and the goals of the user, but the ability to model all, and require only legal behavior can be a very powerful way of defining error conditions by implication.

5. REFERENCES

[1] Accellera Formal Verification Technical Committee. http://www.eda.org/vfv.

[2] AMBA™ Specification (Rev 2.0), ARM Limited, 1999.

[3] Eisner, C., and Fisman, D. Sugar 2.0 Proposal Presented to the Accellera Formal Verification Technical Committee, March 20, 2002.

[4] Keating, M., and Bricaud, P., Reuse Methodology Manual, 3rd ed., Kluwer, 2002.

[5] Sathianathan, R., and Anderson, T., Assertion Methodologies for Verilog Design, Integrated System Design, April 2001.

[6] Virtual Socket Interface Alliance, http://www.vsi.org.