PLATFORM APPLICATION NOTE

cadence

The Role of Assertions in Verification Methodologies

USING ASSERTIONS IN A SIMULATION ENVIRONMENT

February 2003

TABLE OF CONTENTS

CADENCE INCISIVE VERIFICATION PLATFORM

Verifying today's complex ICs requires the speed and efficiency that can only be provided in a unified verification methodology. The Cadence Incisive verification platform enables the development of a unified methodology from system design to system design-in for all design domains. A unified verification methodology consists of many different tools, technologies and processes all working together in a common environment. The Incisive platform provides the tools, technologies, a common user environment and the support needed to develop a unified methodology. This application note details specific topics for using the tools and technologies in the Incisive platform to help create a unified methodology to verify your design.

1 ASSERTIONS OVERVIEW

Assertions play an important role in a unified verification methodology. Assertions allow the architect or designer to capture his or her design intent and assumptions in a manner that can be verified in the implementation. Assertions are captured during the development process and are continually verified throughout the process. Assertions, working in a unified verification methodology, reduce the verification time by detecting bugs earlier, isolating where a bug is located, and detecting protocol violations that may not cause functional errors to propagate to the outputs. In addition to bug detection, assertions improve the efficiency in a unified methodology by improving reuse, enhancing testbench checking, and capturing coverage information.

2 INTRODUCTION

Assertions express functional design intent in terms of behavior. Assertions can be used to express assumed input behavior, expected output behavior, or forbidden behavior. For example, if a processor has a read signal and a write interface signal, a designer might assume that "the read signal and the write signal should never be both active at the same time." With assertion-based verification (ABV) in a simulation environment, if during a particular simulation run, there was ever a cycle when both the read and the write signals were active, then the assertion would fire and report a message. This is an example of an interface assertion. Interface assertions are used to check the protocol of interfaces between blocks. Other types of assertions include architectural assertions and structural assertions, as shown in Figure 1. Application assertions are used to prove architectural properties such as fairness and deadlocks. For example, "after a request for transfer, eventually an acknowledge is granted." Structural assertions are used to verify low-level internal structures within an implementation, such as a FIFO overflow or incorrect FSM transitions. For example, "if the read and write pointers of a FIFO are the same, then the state of the FSM remains *FIFO_Overflow* until data is read."

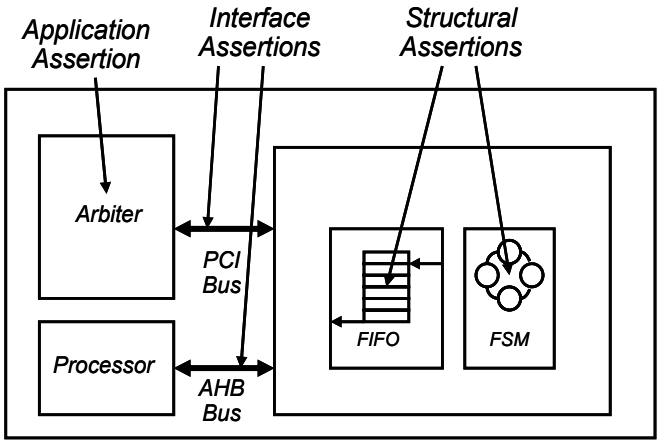

Figure 1. Types of Assertions

This document provides an introduction to the PSL/Sugar assertion language and to the PSL/Sugar-based implementation of ABV (Dynamic ABV) that is supported in the Cadence Incisive verification Platform. The benefits of using assertions are discussed, and a specific example of interface assertions for the AMBA AHB processor type interface is presented. The use models of Cadence's implementation of Assertion-Based Verification are also discussed, including how you run a simulation with assertions, debug them, and turn them on and off.

3 BENEFITS OF USING ASSERTIONS

Assertions are created in the unified verification methodology whenever design or architecture information is captured. These assertions are then used throughout the verification process to efficiently verify the design. The use of assertions improves the speed and efficiency of verification by bug catching and by enhanced testbench checking. Assertions also facilitate re-use and provide data that is useful in evaluating functional coverage.

3.1 BUG CATCHING

The primary focus of Cadence's implementation of assertion-based verification is to identify bugs as close to the source as possible. This reduces the debug time, especially after full chip integration of blocks when bugs take longer to propagate to the output and are more difficult to isolate. Assertion checking during simulation can identify internal errors within the module

sooner and closer to the source than is possible without assertions. Without assertions, errors must propagate to the outputs before being detected. Assertions reduce the time it takes to locate difficult bugs by identifying where in a design the bug first appears. Assertions also catch bugs that do not propagate to the output. When an assertion fires, it provides an immediate localization of the failure, which simplifies the insertion of intellectual property (IP) into design environments.

3.2 ENHANCING TESTBENCH CHECKING

Assertions help to supplement testbench checking. Often testbench checkers require added complexity to verify the correct operation of internal device under verification (DUV) behavior, such as verifying that an arbiter, buried deep in the design, is servicing requests in a fair manner. Assertions reduce the necessity for complexity in the checker as they can verify internal operations close to the source in a simple manner. By placing assertions throughout the design you are enhancing the overall checking ability of the testbench.

3.3 RE-USE

Time to market pressures, and the high cost and risk of new developments continues to increase the importance of re-use, both for design and verification IP. A key objection that designers have in adopting re-use methodologies is that it is difficult to understand and integrate someone else's code, especially when the developer is no longer available for consultation. Also, there is a general lack of confidence in how proven the code is and how accurate the documentation is.

Assertions help to instill confidence that the design is well documented and functions as specified. Embedded assertions provide a standard format of documented behavior that is checked by simulation vectors. Embedded assertions travel with the design, making the design easier to integrate. Assertions can flag when the assumptions of the original designer are not true, or when assumed interface protocol is violated. The error is caught at or near the source.

PSL/Sugar helps in providing good code documentation because it easy to express, and therefore interpret, both assumed and expected behavior. Each behavior is verified with the simulation vectors, so in a sense, the documentation becomes a standardized format that is proven and continually tested by simulation vectors.

3.4 COVERAGE

With Cadence's implementation of ABV, assertion statistics represent coverage information. This coverage information can be helpful in identifying boundary conditions, corner cases, or sequences that have not been tested.

4 OVERVIEW OF PSL/SUGAR

The Verilog and VHDL simulators of the Incisive verification platform support the use of assertions with native simulation support for the PSL/Sugar assertion language. This section provides an overview of the features commonly used. Refer to the *Simulation-Based Assertion Checking Guide* for complete details of which features the simulators in the Incisive verification platform support.

PSL/Sugar makes it easy to express very complex behavior. PSL/Sugar is based on Boolean expressions, and defines dialects that are specific to hardware languages. PSL/Sugar makes it straightforward to express the following behavior:

1. A behavior may always or never hold true
2. A behavior may always or never hold true only within some window
3. A behavior may express some specific sequence of events (including overlapping sequences)
4. A behavior may express an eventuality
5. Any behavior may have exceptions to the rule
6. Any behavior may only hold true for specific configurations

The Verilog and VHDL simulators in the Incisive verification platform provide native support of PSL/Sugar language. Native support is very important for optimal performance.

4.1 BOOLEANS

PSL/Sugar is designed to be used in conjunction with hardware description languages (HDL) such as Verilog or VHDL. At the bottom level, the PSL/Sugar language is used to specify the conditions that define a behavior of interest, referring to signals, variables, and values within an HDL description of a 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 PSL/Sugar description of the same behavior. This also reduces the learning curve for being able to express a behavior. Cadence's implementation of ABV supports two dialects of PSL/Sugar. Boolean expressions take on the syntax of the language of the compiler. NCVerilog supports the Verilog dialect and NCVHDL supports the VHDL dialect.

4.2 CLOCKING AND EVALUATION

PSL/Sugar assertions are declarative in nature. This means that every assertion is evaluated on every verification cycle, and all assertions are evaluated concurrently. Activation of assertions is not dependent on the HDL code. In other words, activation of assertions are not affected by placement in a conditional statement, like a *case* statement or an *if* statement. Asynchronous or synchronous behavior can be expressed. You can also define a default clock that applies if no explicit sampling clock is attached to an expression. The clock can be defined as any other Boolean expression, and can make use of edge functions.

4.3 SEQUENCE EXPRESSIONS

Sequences can express a simple Boolean expression or a multi-cycle behavior of Boolean expressions, where each cycle of a sequence is separated by a ";".A sequence can be defined once as a sequence declaration and then be referenced, like an alias. The declaration syntax is:

```
Verilog:    sequence <name> = {<sequence_definition>};
VHDL:       sequence <name> is {<sequence_definition>} ;
```

For example, if a, b, c, d, and e are Boolean expressions, then the following define legal sequences:

```
sequence seq1 = { a };
sequence seq2 = { b;c };
sequence complex_seq = {{seq1}; {seq2}};
```

The complex sequence, `complex_seq`, evaluates to true when `a` is true followed by `b` is true followed by `c` is true.

Note the following:

1. Sequence names are enclosed in braces when referenced, indicating it is a sequence expression.
2. Successive cycles of a sequence are separated by a semicolon (`;`).
3. A boolean expression may be viewed as having a sequence length of one. Defining a sequence expression of length one allows you to give the expression a more meaningful name to improve readability.
4. The keyword `true` can be used to specify a *don't care* condition during any cycle.
5. You cannot apply the Verilog `!` or VHDL`not` operator on a sequence expressions. (for example, `!{sequence_expression}` is not valid).
6. Repetition specifiers can be used for any expression of a cycle. `inf` can be used to express infinity. Formats exist for each HDL dialect. The more commonly used operaters are shown below:

Verilog Dialect	VHDL Dialect	Definition
`[*]`	`[ * ]`	Indicates that the expression may occur zero or more consecutive times.
`[+]`	`[+]`	Indicates that the expression may occur one or more consecutive times.
`[ * n]`	`[* n]`	Indicates that the expression should occur exactly *n* consecutive times.
`[ *n :inf]`	`[*n  to inf]`	Indicates that the expression should occur a minimum of *n* consecutive times.
`[* : m]`	`[* to m]`	Indicates that the expression may occur a maximum of *m* consecutive times.
`[*n : m]`	`[*n to m]`	Indicates that the expression may occur *n* or *m* consecutive times.

PROPERTIES

A property defines a behavior to be checked. A property can be thought of as containing an optional *enabling* condition, a *fulfilling* condition, which is the behavior to be checked, an optional *discharging* condition, and an optional *clocking* condition. The most general form of a property and the definition of terms is shown below:

Verilog:

```
property <name> = [operator] [enabling_condition(s)]
    [implication_operator(s)] (fulfilling_condition)
    [until | until_ | abort discharging_condition] [@(clock_expression)];
```

VHDL:

```
property <name> is [operator] [enabling_condition(s)]
    [implication_operator(s)] (fulfilling_condition)
    [until | until_ | abort discharging_condition] [@(clock_expression)];
```

The following describes the key terms of a property expression:

1. ***property*** is a declaration that what follows is a behavior. The simulator, by default, will verify this behavior.
2. A property ***<name>*** is a unique identifier. This should be a very descriptive name because the failure message reports only this name and the time of failure. HDL naming conventions apply. The name must be unique to the module or entity in which it is used.
3. The ***operator*** is one of *always* or *never* and indicates whether the behavior should always occur or should never occur. Note that you may choose to omit the always or never term, in which case the check occurs at time zero only.
4. The ***enabling condition*** can be any Boolean expression or sequence expression. Multiple enabling conditions and implication operators can be strung together to form a complex enable. The assertion is considered to begin when the first sequence of the enabling condition evaluates to true.
5. The ***implication operator*** specifies the logical relationship between two expressions. The operator is one of the following symbols:

`->`	Evaluation of the right hand side(RHS) is started when the left hand side(LHS) is true. The behavior can be read as “if (enabling condition) is true, then the (fulfilling condition) must also be true”. The LHS must be boolean.
`-> next`	Evaluation of the RHS is evaluated in the next cycle after the LHS becomes true. The behavior can be read as “if (enabling condition) is true, then the (fulfilling condition) must be true in the next verification cycle”. The LHS must be boolean
`|->`	Evaluation of the RHS is started in the last cycle of the LHS. The LHS must be a sequence when this is used.
`|=>`	Evaluation of the RHS is started following the cycle when the LHS condition becomes true. The LHS must be a sequence when this is used.
`eventually!`	The RHS is true in some future cycle, and must be true before the end of simulation. The RHS must be a boolean or a sequence.

6. The ***fulfilling condition*** is the behavior to be tested. It is checked on every verification cycle by default. The fulfilling condition can be any Boolean expression or sequence. The fulfilling condition is the last expression prior to any discharging condition.
7. The ***discharging condition*** is a condition that indicates to stop the checking of the behavior. Discharging conditions include the following:

`until` (*Boolean expression*)	When all of the enabling conditions have evaluated to true, the fulfilling condition must be true until the Boolean expression becomes true. It then finishes immediately. `until` has no affect until the fulfilling condition is being checked.
`until_` (*Boolean expression*)	When all of the enabling conditions have evaluated to true, the fulfilling condition must remain true up to and including the cycle where the Boolean expression is true. It then finishes immediately. `until_` has no affect until the fulfilling condition is being checked
`abort` (*Boolean expression*)	Abort cancels the checking of an assertion. An abort that occurs during the checking of the enabling condition or the fulfilling condition will cancel the checking. This is different than the `until` or `until_` that requires that the fulfilling condition is being checked.

8. The *clock expression* is the condition that is used to sample the assertion. This is generally a clock edge, but can be any Boolean expression. A default clock can be specified, eliminating the need to specify it for every assertion.

Some examples are as follows:

Assuming read and write are interface signals, the following is an example of an interface assertion:

```
Verilog:    property NeverRdWrBothActive  = never ((read == 1) && (write == 1));
VHDL:     property NeverRdWrBothActive is never ((read = 1) and (write = 1));
```

An application assertion might be that every request is eventually granted unless a reset occurs:

```
Verilog:  property EveryRequestIsEventuallyGranted = always (
            Req -> eventually! (Granted == 1)) abort (reset == 1)@(posedge clk)
VHDL:     property EveryRequestIsEventuallyGranted is always (
            Req -> eventually! (Granted = 1)) abort (reset = 1) @(clk'event and clk = '1')
```

A structural assertion might be that if the memory task is *clear* (`m_task = 2'b00`), it must always be followed by a three-cycle sequence that must be repeated 256 times. The sequence is `write_n` low for one sample followed by a write high for two samples. The write signal is synchronous to the positive edge of `clk`. Reset is high and will cancel the write.

```
Verilog:
   sequence WRITE_PULSE = {(write_n == 0);(write_n == 1);(write_n == 1)};
   property CLEAR_MEM_WRITE_N = always
          {m_task == 2'b00} |=> {WRITE_PULSE[*256]}
           abort (reset == 1) @(posedge clk),
VHDL:
   sequence WRITE_PULSE is {(write_n = 0);(write_n = 1);(write_n = 1)};
   property CLEAR_MEM_WRITE_N is always
          {m_task = '00'} |=> {WRITE_PULSE[*256]}
           abort (reset = 1) @(clk'event and clk = '1');
```

4.4 DIRECTIVES

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`) or assumed to hold (`assume`). In addition, a `cover` directive specifies the desire to ensure that a sequence is encountered during verification. Properties are always *asserted* or *assumed*. For assertions used in simulation, `assert` and `assume` have the same meaning. (Static verification will prove properties that are asserted, and will use assumed properties to limit the state space of the search.) Sequences can be used in properties or they can be used explicitly for coverage. The format is as follows for both Verilog and VHDL:

```
assume | assert <property_name>;
cover <sequence_name>;:
```

In Cadence's implementation of simulation-based assertion verification, all properties are treated as if they are asserted. That is to say, the `assert` or `assume` directives are optional.

4.5 DEFINING ASSERTIONS

Cadence's implementation of ABV supports defining assertions embedded in the same module or entity/architecture as the HDL with which it is associated, or in a separate file. Embedded assertions are in comments and are identified using the special pragma identifier `sugar`. For example, the following is an embedded assertion that expresses that the read and write signals are never both active, assuming the active state is high:

```
Verilog:  // sugar property NeverRdWrBothActive = never ((read == 1) && (write == 1))
VHDL:     -- sugar property NeverRdWrBothActive is never ((read = 1) and (write = 1))
```

Because the assertions can be specified as comments, they will be ignored by tools that do not support PSL/Sugar. Assertions may also exist in an external file that references signals, variables, and values within an HDL description of a design. Using assertions from an external file is described in the *Simulation-Based Assertion Checking Guide* in Cadence documentation.

Structural assertions that are used to verify low level internal structures within an implementation will likely be inline with the HDL code. This is the recommended approach when the assertion is specific to a design, because the assertion is guaranteed to travel with the design.

Application assertions that are used to prove architectural properties may be embedded in the design HDL, or they may be in an external file that references the design HDL. The primary reason for defining assertions in an external file is to express behavior that references signals from multiple modules or entities. Another use for assertions in an external file is when assertions are added to legacy IP, where the IP cannot be modified.

Interface assertions that check the protocol of interfaces between blocks, are likely to be in a separate file, but that file is a standalone module or entity. The interface assertions are self-contained in the module/entity that contains HDL code, so they have the same effect as embedded assertions. The HDL code is minimal but can be used, for example, to calculate expected values that are then used in the assertions. Placing the assertions in a separate file allows the interface assertions to be instantiated in any block that makes use of that interface (for block level verification) or on the interface itself when the blocks are integrated. Defining a separate module or entity that is instantiated also allows the signal names to be mapped when the interface assertions are instantiated, making it easier to use. The AMBA AHB Interface assertions discussed in the next section provide an example.

5 PRACTICAL EXAMPLE - AMBA AHB INTERFACE ASSERTIONS

The AMBA AHP interface is a typical processor interface. This section details how to write PSL/Sugar compliance assertions using the Verilog dialect for an AHB interface. Each behavior is described and the PSL/Sugar representation is shown. The next section will show how to simulate with these assertions. The detailed AHB specification can be found at `www.arm.com/armtech/AMBA_Spec?OpenDocument`.

Signaling interface monitors will be HDL based assertions that use the PSL/Sugar language. This choice keeps the clock level analysis in the native language of the simulator so it does not require having to traverse a lower performance interface. You may not want to check that the contents of what was written were written correctly, you may just want to check that the protocol specified for the transfer was adhered to. Higher layer protocols are better checked using higher level languages like SystemC because of its extended capabilities, such as queuing, and other features for tracking expected results.

The AHB assertions will be embedded in an interface assertion monitor that can be instantiated in the design. This allows the assertion monitor to be instantiated within any AHB interface device, or once on any AHB bus interface. The PSL/Sugar code is shown below. The top of the file simply declares all the signals. It also defines parameters used to improve readability. Some HDL code is required to create variables used in the assertions. The assertions are then defined. For each assertion, the behavior is described and then defined in PSL/Sugar. Note that all properties are asserted by default in the simulators of the Incisive verification platform, so the assume directive is not used.

```
/****************** AHB Interface PSL Sugar Assertions ****************/
`timescale 1 ns / 100 ps

module ahbCompliance (ahbAddr_i, ahbTrans_i, ahbWrite_i, ahbSize_i,
    ahbBurst_i, ahbProt_i, ahbWData_i, ahbRData_i, ahbReady_i, ahbResp_i,
    ahbReq_i, ahbLock_i, ahbGnt_i, ahbResetn_i, ahbSel_i, ahbMaster_i,
    ahbMastLock_i, ahbSplit_i, ahbClk_i);

parameter dataBusWidth    = 32;
parameter numSlaves       = 32;
parameter busMaster       = 0;


/* these parameters define the Transfer Type as specified by HTrans */
parameter IDLE = 2'b00;
parameter BUSY = 2'b01;
parameter SEQ  = 2'b11;
parameter NONSEQ = 2'b10;

/* these parameters define the Burst Mode as specified by HBurst */
parameter SINGLE = 3'b000;
parameter INCR   = 3'b001;
parameter WRAP4  = 3'b010;
parameter INCR4  = 3'b011;
parameter WRAP8  = 3'b100;
parameter INCR8  = 3'b101;
parameter WRAP16 = 3'b110;
parameter INCR16 = 3'b111;

/* define the amount of data in each beat as specified by HSize*/
parameter bits8    = 3'b000;
parameter bits16   = 3'b001;
parameter bits32   = 3'b010;
parameter bits64   = 3'b011;
parameter bits128  = 3'b100;
parameter bits256  = 3'b101;
parameter bits512  = 3'b110;
parameter bits1024 = 3'b111;

/* define the transfer response signals as specified by HResp*/
parameter OKAY = 2'b00;
parameter ERROR = 2'b01;
parameter RETRY = 2'b10;
parameter SPLIT = 2'b11;

input [31:0]          ahbAddr_i, ahbWData_i, ahbRData_i;
input [1:0]           ahbTrans_i, ahbResp_i;
input                 ahbWrite_i, ahbReady_i, ahbReq_i;
input [2:0]           ahbSize_i, ahbBurst_i;
input [3:0]           ahbProt_i, ahbMaster_i;
input                 ahbLock_i, ahbGnt_i, ahbMastLock_i;
input [numSlaves-1:0] ahbSel_i;
input [15:0]          ahbSplit_i;
input                 ahbResetn_i, ahbClk_i;
```

```
reg [31:0] prev_ahbAddr, prev_ahbAddrPlusSize;
reg prev_ahbWrite, prev_ahbReady;
reg [numSlaves-1:0] prev_ahbSel;
reg [3:0] prev_ahbProt;
reg [2:0] prev_ahbSize, prev_ahbBurst;
reg [dataBusWidth-1:0] prev_ahbRData, prev_ahbWData;
reg [1:0] prev_ahbTrans, prev_ahbResp;
integer AddrIncrement;
integer NumberBeats=0;

// synopsys translate_off

/* capture data that is needed in the following assertions */
always @(posedge ahbClk_i or negedge ahbResetn_i)
   begin
     prev_ahbAddr  <= ahbAddr_i;
     prev_ahbReady <= ahbReady_i;
     prev_ahbWrite <= ahbWrite_i;
     prev_ahbRData <= ahbRData_i;
     prev_ahbWData <= ahbWData_i;
     prev_ahbTrans <= ahbTrans_i;
     prev_ahbBurst <= ahbBurst_i;
     prev_ahbResp  <= ahbResp_i;
     prev_ahbSel   <= ahbSel_i;
     prev_ahbSize  <= ahbSize_i;
     prev_ahbProt  <= ahbProt_i;

     case (ahbSize_i)    /* AddrIncrement is used to check the address */
               bits8:    AddrIncrement=1;
               bits16:   AddrIncrement=2;
               bits32:   AddrIncrement=4;
               bits64:   AddrIncrement=8;
               bits128:  AddrIncrement=16;
               bits256:  AddrIncrement=32;
               bits512:  AddrIncrement=64;
               bits1024: AddrIncrement=128;
               default:  ;
     endcase
/* NumberBeats is used to check packet lengths */
     if (ahbTrans_i == IDLE)
        NumberBeats <= 0;
     else if ( ahbTrans_i == NONSEQ)
        NumberBeats <= 1;
     else if ( (ahbTrans_i == SEQ) && (ahbReady_i == 1) )
        NumberBeats <= NumberBeats + 1;
     else
        NumberBeats <= NumberBeats;
/* determine the predicted address */
     if ((ahbTrans_i != BUSY) && (ahbReady_i != 0))
          prev_ahbAddrPlusSize <= ahbAddr_i + AddrIncrement;
   end

/*************************** SUGAR ASSERTIONS ************************/
/* define default clock that will be used to sample all subsequent asserts */
/* NOTE: it is important that data does not change on the same edge as clock
    because, like HDL, assertions can be sensitive to races */
// sugar default clock = (posedge ahbClk_i);

/* Behavior: If reset is active then Transfer Type is IDLE and Resp is OKAY
   (default clock) */
// sugar property IdleAndOkayDuringReset = always
//    {(ahbResetn_i == 0);(ahbResetn_i == 1)} |->
//         {(ahbTrans_i == IDLE) && (ahbResp_i == OKAY)};

/* Behavior: If the transfer type is NONSEQ and Burst Mode is SINGLE,
   then on the next clock cycle, the transfer type is not SEQ and not BUSY
   (default clock) */
// sugar property BusyAndSeqNeverFollowNonseqOnSingleTransfer = always (
//       (ahbTrans_i == NONSEQ) && (ahbBurst_i == SINGLE ) ->
//           next ((ahbTrans_i != SEQ) && (ahbTrans_i != BUSY)) );

/* Behavior: If the transfer type is BUSY and slave is ready,
   then on the next clock cycle, the transfer type must not be IDLE and
   must not be NONSEQ unless grant is 0 and ready is 1 (default clock) */
// sugar property NeverGoFromBusyToIdleOrNonseqUnlessNoGrant = always (
//       ((ahbTrans_i == BUSY) && (ahbReady_i == 1)) ->
//            next ((ahbTrans_i != IDLE) && (ahbTrans_i != NONSEQ))
//              abort ((ahbGnt_i == 0) && (ahbReady_i == 1)) || (ahbResetn_i == 0));

/* Behavior: if the transfer type is IDLE, then on the next clock
   cycle, the transfer type must be either IDLE or NONSEQ (default clock)*/
// sugar property AlwaysGoToNonseqFromIdle = always (
//    (ahbTrans_i == IDLE) ->
//           next ((ahbTrans_i == IDLE) || (ahbTrans_i == NONSEQ)) );
```

```
/* Behavior: if the transfer type is BUSY then on the corresponding
   data transfer , the slave must provide a zero wait state OKAY response
   (default clock) */
// sugar property ResponseToBusyMustBeZeroWaitOKAY = always (
//       ((ahbTrans_i == BUSY) && (ahbReady_i == 1)) ->
//            next ((ahbResp_i == OKAY) && (ahbReady_i == 1)) );

/* Behavior: if the transfer type is IDLE then on the corresponding data
   transfer, the slave must provide a zero wait state OKAY response
   (default clock) */
// sugar property ResponseToIdleMustBeZeroWaitOKAY = always (
//       ((ahbTrans_i == IDLE) && (ahbReady_i == 1)) ->
//         next ((ahbResp_i == OKAY) && (ahbReady_i == 1)) );

/* Behavior: if the previous response was OKAY, and the current response
   is not OKAY, then Ready must be 0 during the second not OKAY response
   (default clock) */
// sugar property ReadyMustBeZeroDuringFirstNotOkayResponse = always
//       {(ahbResp_i == OKAY);(ahbResp_i != OKAY)} |->
//          {ahbReady_i == 0} ;

/* Behavior: if the response is SPLIT or RETRY, the master must drive IDLE
   (default clock) */
// sugar property FirstNotOkayResponseCausesNextIdle = always (
//       ((ahbResp_i == SPLIT) || (ahbResp_i == RETRY)) && (ahbReady_i == 0) ->
//            next (ahbTrans_i == IDLE));

/* Behavior: if the transfer type is SEQ or BUSY, then the controls of the
   corresponding data transfer must be the same as their previous control
   (default clock)  NOTE: the equivalence operator is used with ahbProt
   because this is an optional signal and may be an "X" value if not
   connected */
// sugar property ControlMustBeConstantDuringABurst = always (
//     (ahbTrans_i == SEQ) || (ahbTrans_i == BUSY) ->
//         ((ahbSize_i == prev_ahbSize) && (ahbProt_i === prev_ahbProt) &&
//            (ahbWrite_i == prev_ahbWrite)) );

/* Behavior: if Ready is 0 and the response is OK or ERROR, the address
   and control, and data from the master are held constant (default clock)
   NOTE: the equivalence operator is used with ahbProt because this is an
   optional signal and may be an "X" value if not connected */
// sugar property ControlMustBeConstantWhenSlaveNotReady = always (
//     (ahbReady_i == 0) && ((ahbResp_i == OKAY) || (ahbResp_i == ERROR))  -> next
//       (ahbTrans_i == prev_ahbTrans) && (ahbSize_i == prev_ahbSize) &&
//           (ahbProt_i === prev_ahbProt) && (ahbWrite_i == prev_ahbWrite) &&
//            (ahbAddr_i == prev_ahbAddr) && (ahbBurst_i == prev_ahbBurst) )
//              abort (ahbResetn_i == 0);

/* Behavior: if ready is 0 and the previous transfer type was SEQ or NONSEQ,
   then on the next cycle the write data is the same as the previous write data
   (default clock) (This assertion is designed to fail!!!) */
// sugar property WriteDataMustBeConstantWhenSlaveNotReadyAndDataBeingTransferred = always (
//     (ahbReady_i == 0) && (ahbWrite_i == 1) &&
//          ((prev_ahbTrans == SEQ) || (prev_ahbTrans == NONSEQ)) -> next
//          (ahbWData_i == prev_ahbWData) )
//             abort (ahbResetn_i == 0);

/* Behavior: if transfer type is BUSY then address does not change
   (default clock) */
// sugar property AddrHeldWhenMasterBusy = always (
//        (ahbTrans_i == BUSY)  ->  next (ahbAddr_i == prev_ahbAddr) );

/* Behavior: if transfer is in progress, then 1K boundary is not exceeded
   (default clock) */
// sugar property PageAddressNeverExceeds1kBoundary = always (
//        ((ahbTrans_i == SEQ) || (ahbTrans_i == BUSY)) ->
//           (ahbAddr_i[31:10] == prev_ahbAddr[31:10]) );

/* Behavior: If the burst type is not INCR and the transfer type is not
   IDLE , then the NumberBeats for the Burst Mode is not exceeded
   (default clock) */
// sugar property BurstIsNotTooLong = always (
//       (ahbBurst_i != INCR) && (ahbTrans_i != IDLE) ->
//          ((ahbBurst_i == SINGLE) &&  (NumberBeats <= 1)) ||
//          ((ahbBurst_i == WRAP4)  &&  (NumberBeats <= 4)) ||
//          ((ahbBurst_i == WRAP8)  &&  (NumberBeats <= 8)) ||
//          ((ahbBurst_i == WRAP16) &&  (NumberBeats <= 16)) ||
//          ((ahbBurst_i == INCR4)  &&  (NumberBeats <= 4)) ||
//          ((ahbBurst_i == INCR8)  &&  (NumberBeats <= 8)) ||
//          ((ahbBurst_i == INCR16) &&  (NumberBeats <= 16)) );

/* Behavior: For all but INCR Burst mode, if the end of the packet is being
   transferred as indicated by a transition from SEQ to IDLE when Resp is ok
   then the NumberBeats for the Burst Mode is the max number unless grant is 0
   (default clock) */
```

```
// sugar property BurstIsNotTooShort = always
//       {((ahbBurst_i != INCR) && (ahbResp_i == OKAY) && (ahbTrans_i == SEQ));
//         (ahbTrans_i == IDLE) } |->
//          {((prev_ahbBurst == SINGLE) &&  (NumberBeats == 1)) ||
//           ((prev_ahbBurst == WRAP4)  &&  (NumberBeats == 4)) ||
//           ((prev_ahbBurst == WRAP8)  &&  (NumberBeats == 8)) ||
//           ((prev_ahbBurst == WRAP16) &&  (NumberBeats == 16)) ||
//           ((prev_ahbBurst == INCR4)  &&  (NumberBeats == 4)) ||
//           ((prev_ahbBurst == INCR8)  &&  (NumberBeats == 8)) ||
//           ((prev_ahbBurst == INCR16) &&  (NumberBeats == 16))}
//    abort (ahbGnt_i == 0) ;

/* Behavior: if the transfer type is IDLE , then on the next clock cycle,
   the transfer type is not SEQ and the transfer type is not BUSY
   (default clock) */
// sugar property NeverGoFromIdleToSeqOrBusy = always (
//        (ahbTrans_i == IDLE) ->
//          next ((ahbTrans_i != SEQ) && (ahbTrans_i != BUSY)) );

/* Behavior: RETRY is always asserted for two cycles unless reset is asserted
   (default clock)*/
// sugar property RetryResponseMustPersistTwoCycles = always
//       {(ahbResp_i != RETRY);(ahbResp_i == RETRY)} |=>
//        {(ahbResp_i == RETRY)}
//            abort (ahbResetn_i == 0);

/* Behavior: SPLIT is always asserted for two cycles unless reset is asserted
   (default clock) */
// sugar property SplitResponseMustPersistTwoCycles = always
//        {(ahbResp_i != SPLIT);(ahbResp_i == SPLIT)} |=>
//            {(ahbResp_i == SPLIT)}
//              abort (ahbResetn_i == 0);

/* Behavior: ERROR is always asserted for two cycles unless reset is asserted
   (default clock) */
// sugar property ErrorResponseMustPersistTwoCycles = always
//        {(ahbResp_i != ERROR);(ahbResp_i == ERROR)} |=>
//           {(ahbResp_i == ERROR)}
//              abort (ahbResetn_i == 0);

/******* incorrectAddressCalculation for 3 example WRAP modes: This calculates
    whether the next address is correct for all wrap modes*******/
/* Behavior: if Burst mode is WRAP4 and Size is 8 bits, then if
   transfer mode is SEQ or BUSY, then if previous transfer mode is not
   busy and previous Ready is not zero,  then check that the address is as predicted */
// sugar property CorrectAddressDuringPageSize4BurstWrap = always (
//     ((ahbBurst_i== WRAP4) && (ahbSize_i == bits8)) ->
//         ((ahbTrans_i == SEQ || ahbTrans_i == BUSY)) ->
//              ((prev_ahbTrans != BUSY) && (prev_ahbReady != 0)) ->
//                  ((ahbAddr_i[31:2] == prev_ahbAddr[31:2]) &&
//                    (ahbAddr_i[1:0] == prev_ahbAddrPlusSize[1:0])) );
// sugar property CorrectAddressDuringPageSize8BurstWrap = always (
//     ((ahbBurst_i== WRAP4) && (ahbSize_i == bits16)) ||
//     ((ahbBurst_i== WRAP8) && (ahbSize_i == bits8)) ->
//         ((ahbTrans_i == SEQ) || (ahbTrans_i == BUSY)) ->
//              ((prev_ahbTrans != BUSY) && (prev_ahbReady != 0)) ->
//                  ((ahbAddr_i[31:3] == prev_ahbAddr[31:3]) &&
//                    (ahbAddr_i[2:0] == prev_ahbAddrPlusSize[2:0])) );

// sugar property CorrectAddressDuringPageSize2048BurstWrap = always (
//     ((ahbBurst_i== WRAP16) && (ahbSize_i == bits1024)) ->
//         ((ahbTrans_i == SEQ) || (ahbTrans_i == BUSY)) ->
//              ((prev_ahbTrans != BUSY) && (prev_ahbReady != 0)) ->
//             ((ahbAddr_i[31:11] == prev_ahbAddr[31:11]) &&
//                  (ahbAddr_i[10:0] == prev_ahbAddrPlusSize[10:0])) );

/* Behavior: the address must increment by size during all INCR beats */
// sugar property AddressIncBySizeDuringAllBurstIncrBeats = always (
//     ((ahbBurst_i == INCR16) || (ahbBurst_i == INCR8) ||
//     (ahbBurst_i == INCR4)) ->
//        ((ahbTrans_i == SEQ) || (ahbTrans_i == BUSY)) ->
//         ((prev_ahbTrans != BUSY) && (ahbReady_i != 0)) ->
//              (ahbAddr_i == prev_ahbAddrPlusSize) );

/* Behavior: always address must be aligned to the transfer size */
// sugar property AddressNotAlignedToTransferSize = always (
//        ((ahbSize_i == bits8) ||
//        ((ahbSize_i == bits16) && (ahbAddr_i[0]   == 1'b0)) ||
//        ((ahbSize_i == bits32) && (ahbAddr_i[1:0] == 2'b00)) ||
//        ((ahbSize_i == bits64) && (ahbAddr_i[2:0] == 3'b000)) ||
//        ((ahbSize_i == bits128) && (ahbAddr_i[3:0]  == 4'b0000)) ||
//        ((ahbSize_i == bits256) && (ahbAddr_i[4:0]  == 5'b00000)) ||
//        ((ahbSize_i == bits512) && (ahbAddr_i[5:0]  == 6'b000000)) ||
//        ((ahbSize_i == bits1024) && (ahbAddr_i[6:0] == 7'b0000000)) ) );
```

```
/* Behavior: always the maximum number of wait states is 16 */
// sugar property NeverMoreThan16WaitStates = always
//     {(ahbReady_i == 1);(ahbReady_i == 0)} |=>
//       {{(ahbReady_i == 0)[*0..15]};{ahbReady_i == 1}}
//      abort (ahbResp_i != OKAY);

// synopsys translate_on
endmodule
```

5.1 TIPS FOR WRITING ASSERTIONS

Here are a few things to remember that will help you to get started with writing assertions:

1. Performance is adversely affected by eventualities and by unbounded repetition in the right hand side of the assertion.
2. Assertions are sensitive to race conditions just like HDL, so it will be appropriate to evaluate most assertions relative to a clock.
3. You should avoid using `never` or `not` applied to an implication. The semantics are non-intuitive, because the implication (`a -> b`) is true if a is false, so `property example = never (a->b)` can only be true if a is always true. In addition, the statistics will not indicate that a *never* assertion ever *finishes* because it can only *fail*.
4. Assertions inside functions and Verilog tasks and VHDL procedures are ignored. However, an assertion can utilize a function as part of a Verilog or VHDL expression.
5. Remember that assertions are evaluated at every verification cycle and all are evaluated concurrently. An assertion that expresses a sequence of length 3 can have three concurrent evaluations ongoing simultaneously if the initial expression remains true for extended cycles. This is very powerful because it allows for detection of overlapping conditions.
6. As you will see in the next section, it is important to give your properties descriptive names because the name is the message that is provided when the assertion fires.

6 ASSERTION USE MODEL IN THE INCISIVE VERIFICATION PLATFORM

This section describes how to run a Verilog simulation with assertions and how to debug assertion failures. Most assertions should be enabled all the time because the overhead is very low relative to the overall simulation times and they provide tremendous value in isolating the cause of bugs. It is not necessary to always probe assertions because failures will be reported regardless of whether or not the assertion is probed.

6.1 RUNNING A VERILOG SIMULATION

The first step is to compile and elaborate the monitor and test fixture. Figure 2 shows the command used to compile the monitor and test fixture, and the text output that is sent to the terminal. (Note: all design files and simulation messages are not shown.) The `ncverilog +assert` directs the compiler to enable assertion processing. Without this compiler option, all assertions are ignored. If there are assertions in the design, and the `+assert` compile option was specified, then the design hierarchy will show the number of assertions. The simulation is run in command line mode so there are no breakpoints and the simulator runs to the finish. In the GUI mode, the default is to break on assertion failures, which makes the debug easier because you can query current signal values at the time of failure. Assertion errors are printed both to the terminal and to the log file. Assertions can also be directed to their own log file.

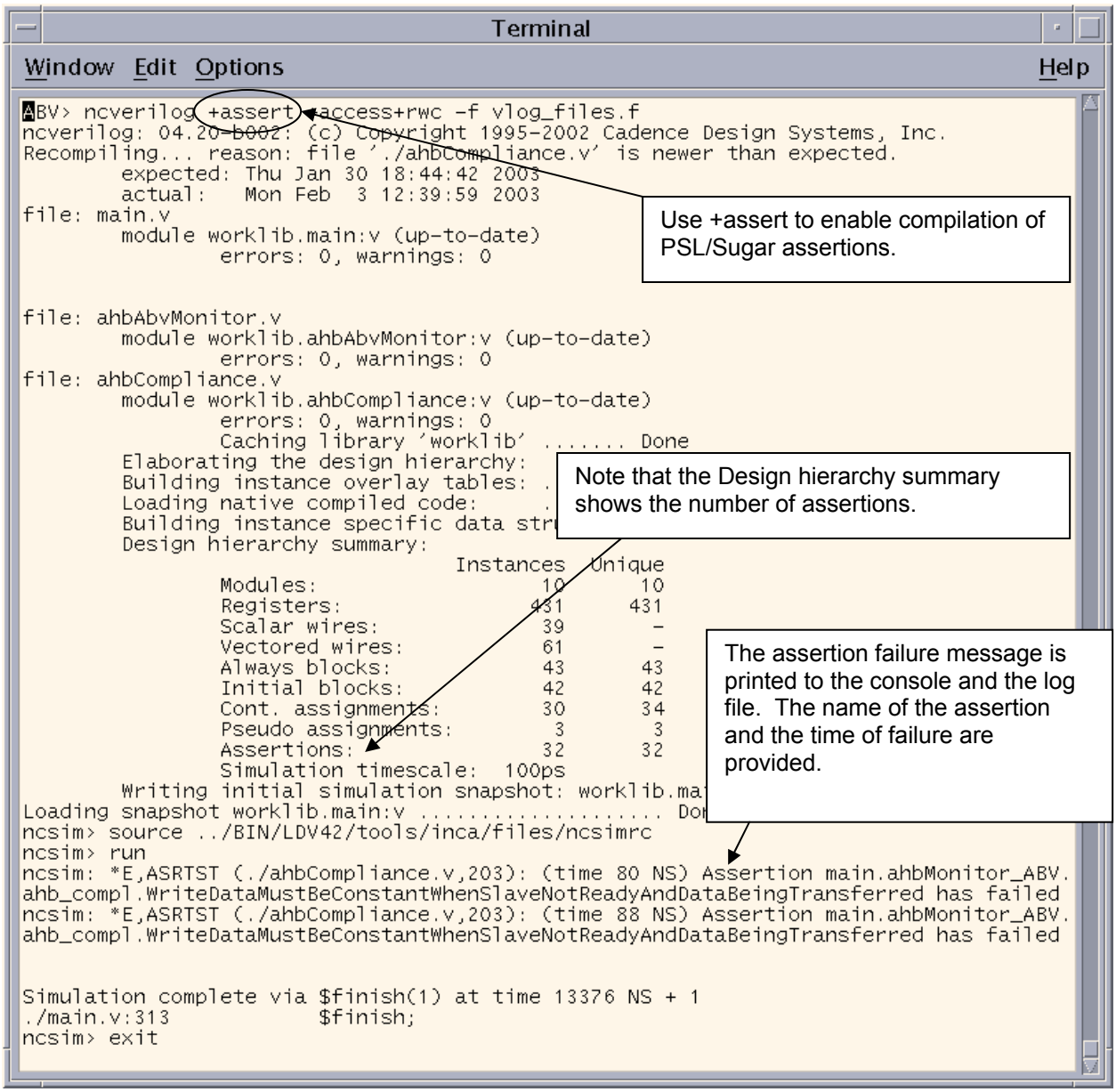

Figure 2. Running a Verilog Simulation with Assertions

6.2 DEBUGGING ASSERTION FAILURES

In general, for debug, you want to use `+access+wrc` to elaborate the design so you can see all signals. Figure 3 shows the waveform that is generated when the test case is run. All input signals were probed and all the assertions were probed to one event counter. Notice that this event counter has a red x anytime a failure of any of the assertions occurs. You can determine which assertion failed by selecting the probe name, then placing the curser on the x and choosing the *Explore—GoTo—Cause* menu command. This will open the source browser with a pointer to the location of the assertion definition, as shown in Figure 4. Placing your curser over a signal or variable name will show the current value of that signal or variable. If multiple assertions fail, use the *View—Expand Sequence* menu command to select only one assertion at a time.

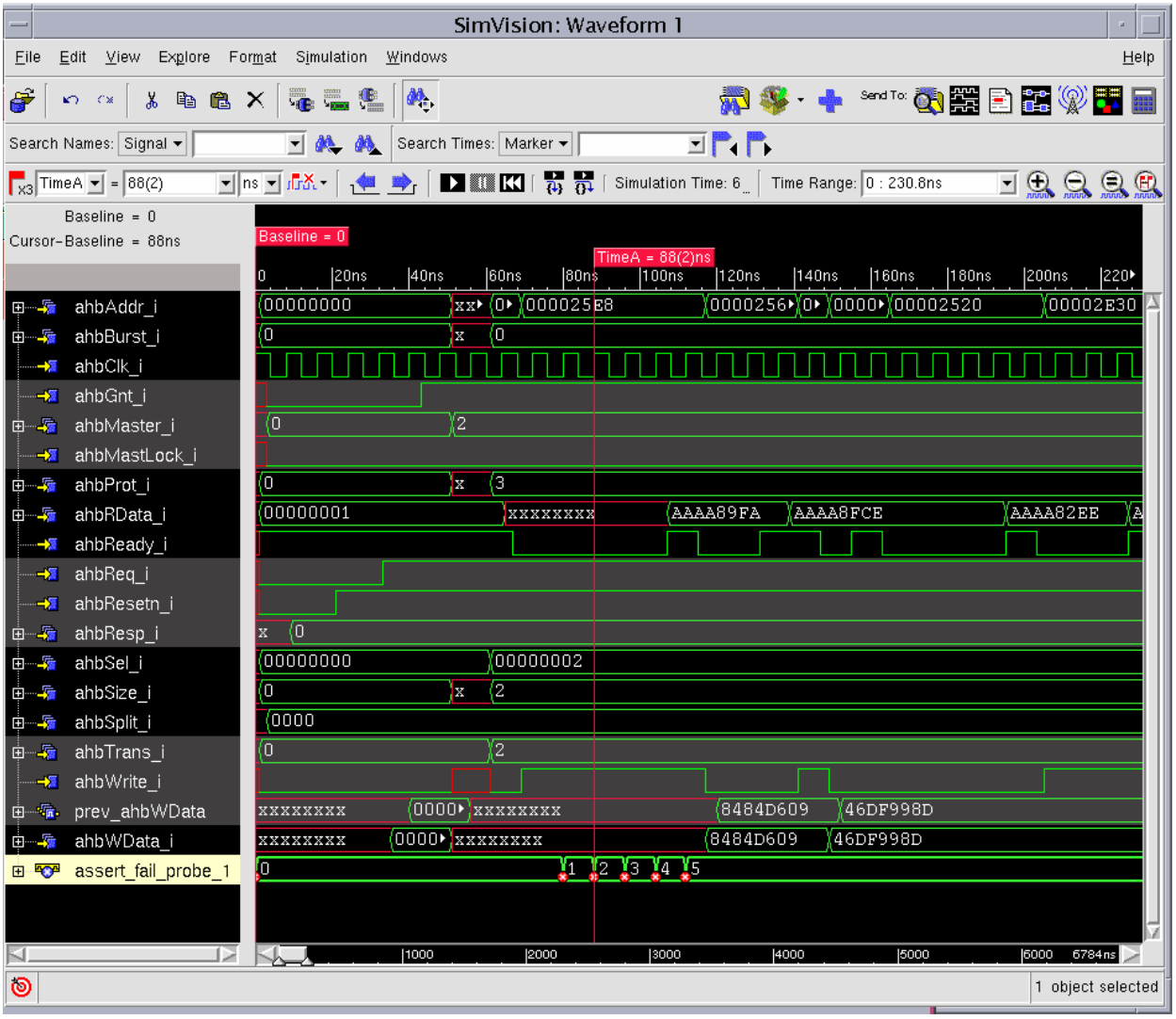

Figure 3 Simulation Waveform Shown with all Assertions Probed to One Event Counter.

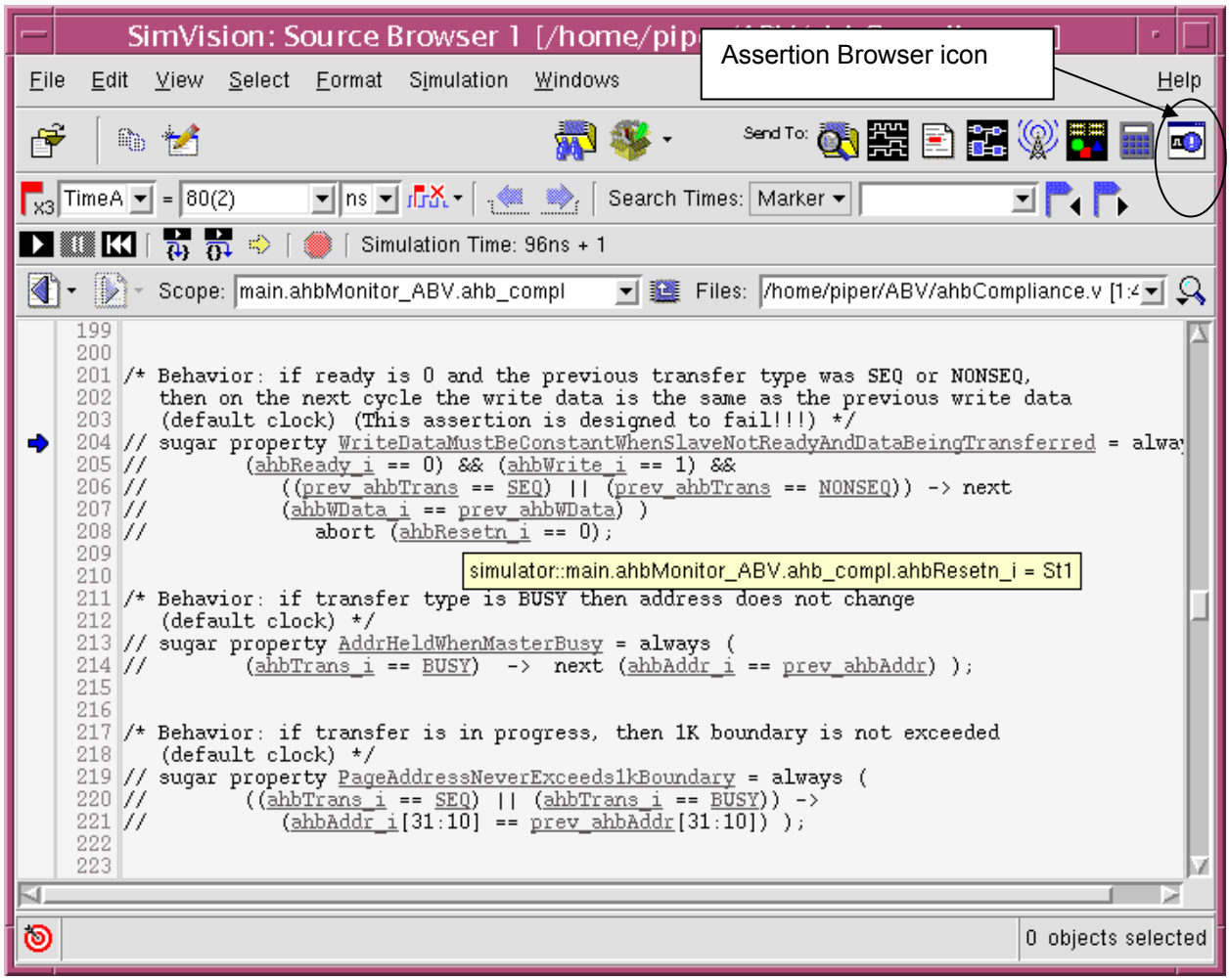

Figure 4 Source Browser Points to the Definition of the Failing Assertion

To debug your assertions in the GUI without using waveforms invoke the assertion browser by selecting the button with the assertion browser icon or through the menu: *Windows -> New -> Assertion Browser*. The assertion browser shows a listing of all assertions in the design. The colors reflect the *Current State*. When an assertion fails, it is easy to identify by the color red. Another way to locate an assertion is to use the filtering at the bottom. Assertions can also be sorted by clicking on any column heading.

Cadence's implementation of assertion-based verification defines four states: `begun`, `finished`, `inactive`, and `failed`. The assertion is `begun` when the first sequence of the enabling condition is satisfied, and remains begun until the assertion goes inactive, finishes or fails. The `failed` state indicates that the fulfilling condition for an assertion has evaluated to false. The `finished` state indicates that the fulfilling condition for an assertion has evaluated to true. The `inactive` state is when the assertion has not begun, and did not fail or finish during the currect clock cycle. The assertion statistics provide some feel for what the simulation vectors have exercised, but interpretation can be tricky due to overlapping conditions. A summary of statistics is provided for all assertions at the top, and for those displayed (filtered) at the bottom of the window.

You can double click on any assertion in the Assertion Browser and it will take you to the assertion definition in the source browser. To debug a failed assertion, double click on the failed assertion to view the source. The source browser shows the definition of the failed assertion, exactly as shown in Figure 4 above. When you run in GUI mode and use the default *break on assertion failure*, you can place the cursor over the terms of the fulfilling condition of the assertion definition to determine what fired the assertion. This provides the current values only. You will not be able to see sequence values from previous simulation time steps.

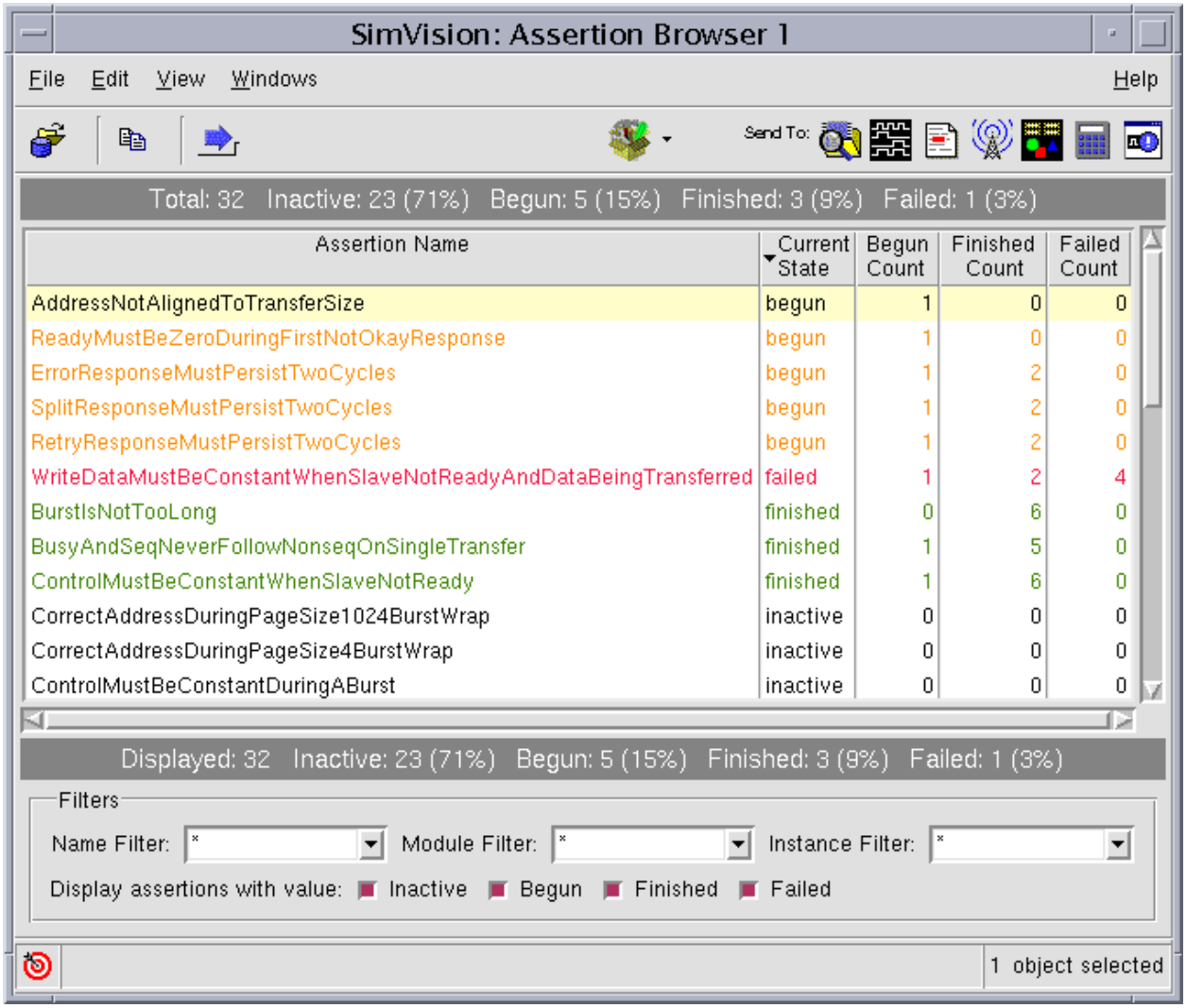

Figure 5 Assertion Browser shows Assertion Statistics

7 SUMMARY

Assertions within a design simplify the detection and diagnosis of errors by making internal test points visible when an error is detected. Assertions document the designer's assumptions and expected behavior in a standardized way that is tested and travels with the design, which is invaluable to any designer who must maintain or extend the code in the future.

This document has provided an overview of the Incisive verification platform's implementation of assertion based verification, including the language supported, example assertions, and the Verilog simulation use model. Additional information is available in the *Simulation-Based Assertion Checking Guide* in Cadence documentation.

8 REFERENCES

1. *Simulation-Based Assertions Verification Tutorial* – Cadence Documentation
2. *Simulation-Based Assertion Checking Guide* – Cadence Documentation
3. Sugar 2.0 Definition, March 20, 2002. http://www.haifa.il.ibm.com/projects/verification/sugar/literature.html
4. Accellera Property Specification Language Reference Manual. January 31, 2003. http://www.eda.org/vfv/docs/psl_lrm-1.0.pdf
5. AMBA AHB specification: www.arm.com/armtech/AMBA_Spec?OpenDocument