CHAPTER

4

PLI-BASED ASSERTIONS

In this chapter, we demonstrate how to create a set of Verilog Programming Language Interface (PLI) assertions. The PLI is a user-programmable, procedural interface that provides a means for interfacing *C* applications with a commercial Verilog simulator. The IEEE 1364-1995 and 1364-2001 standards contain three implementations of PLI library routines. These include the initial OVI *PLI 1.0* standard, which consists of the first generation **TF** and second generation **ACC** libraries, and the later OVI *PLI 2.0* standard, which consists of the third generation of PLI routines, the **VPI** library. The **VPI** library is a super set of the TF and ACC routines that provides additional capability and simplified syntax and semantics. At the time of this writing, not all commercial simulators support the PLI 2.0 VPI standard. Therefore, all examples in this chapter are coded using the older *PLI 1.0* standard to ensure compatibility with every reader's simulator. We encourage you to implement your PLI-based assertion methodology using the newer *PLI 2.0* standard if your simulator supports the VPI routines. *The Verilog® PLI Handbook* by Stuart Sutherland [2002] is a comprehensive reference manual and guide for learning both the *PLI 1.0* and *2.0* standards.

Even within the PLI 1.0, interpretations of the PLI standard differ between vendors, and vary from what is described in books that attempt to explain the PLI. PLI-based assertion users report that most of their porting problems when testing another vendor's simulator are not in the Verilog text itself, but in their assertion PLI interface. One example is that the status of the parameters seen by the *checktf* varies between Verilog simulation vendors (*checktf* routines are discussed later in this chapter).

Another consideration when implementing a PLI-based assertion solution is the PLI's impact on simulation. A negative impact on simulation can occur if the PLI-based assertion is called at every visit through procedural code to perform its check, particularly when the assertion is not violated.

Broad use of PLI-based assertions places a participating project in the middle of complex portability issues. Hence, we recommend that projects limit their use of PLI-based assertions to a very small set.

PLI-based assertion library

This chapter introduces concepts that enable designers to implement their own PLI-based assertion library. PLI-based assertions; unlike the concurrent assertion constructs introduced in Chapter 3, “Specifying RTL Properties”; may be used directly in procedural code. However, teams must address two issues to prevent false firing of procedural assertions during event-driven simulation. The first issue concerns the potential for evaluating the same procedural blocks multiple times within a *single* simulation time slot. In other words, the transient behavior of variables within a given time slot, prior to reaching a steady-state, could trigger a procedural assertion. The second issue concerns the transient behavior of variables across *multiple* time slots. In other words, generally the engineer is only interested in checking a procedural assertion at a clock edge (that is, the cycle-based semantics of assertions). This is problematic for procedural blocks that are not triggered by a clocking event (for example, modeling combinational logic between sequential elements). In this chapter, we demonstrate the false firing problem encountered by PLI-based procedural assertions in event-driven simulation, and then we present techniques to prevent these errors.

4.1 Procedural assertions

In Chapter 3, “Specifying RTL Properties”, we introduced the OVL. This library consists of a set of assertion modules that concurrently validate an RTL expression at every edge of a sample clock. While *OVL concurrent assertions* prevent false firings by sampling the assertion test expression at a clock edge, *procedural assertions* are only checked during procedural visits through the code. In this section, we present some of the strengths of procedural assertions; for example, expressiveness and convenience. We also discuss the weaknesses of procedural assertions; for example, over constraining and false firing if not properly constructed.

over-constraining procedural assertions

Experience has demonstrated that, when compared to declarative forms of assertions, procedural assertions that are deeply nested within RTL case and if statements run a higher risk of being over constrained. When this is the case, they can miss a bug. That is not to say declarative assertions are immune to over constraint. However, when designers embed assertions deeply within RTL code, they must seriously consider the effect on the assertion for all conditional expressions related to the nested case and if statements. Particularly as the design undergoes changes.

Example 4-1 demonstrates this point. In this simplified example, the engineer is using a PLI task to validate that the three one-bit variables (a, b, and c) are mutually exclusive. However, the mutually exclusive property only validates when d is true. If the true property of these variables is that they should *always* be mutually exclusive, then there is a potential to miss an error in the design during verification if d is not true during the previous visit. In actual RTL code, unlike this simple code in Example 4-1, assertions deeply nested within case and if statements are often quite complex. Therefore, the designer must be especially alert to the potential for over constraining assertions embedded in procedural code.

Example 4-1 Over constrained procedural assertion

```
always @(a or b or c or d) begin
   :
  if   (d)
    $assert_one_hot ({a,b,c});
end
```

Procedural assertion convenience

In spite of the potential problem of over constraining a procedural assertion, designers generally prefer the convenience of expressing assertions procedurally. For example, the assertion expression, created for an OVL concurrent check, can become quite complicated if it is necessary to qualify the assertion with an expression that represents the sensitized path down through the deeply nested procedural code. However, if the designer places the assertion directly in the procedural code, this reduces the amount of required coding.

4.1.1 A simple PLI assertion

Example 4-2 demonstrates the source code for a simple PLI $assert_always() check. This PLI assertion validates the designer’s Verilog Boolean expression, which is passed in as its argument, is always true. The PLI *C* code for this assertion is divided into two functions: a *checktf* routine and a *calltf* routine.

Checktf routine

The assert_always_checktf() function, shown in Example 4-2, is automatically called by the simulator before the simulator starts running (that is, prior to simulation time 0). This is either at the Verilog source code compilation time or load time, depending on the simulator. The purpose of the assert_always_checktf() function is to verify that the arguments used in the PLI *system task* are used correctly when instantiated within the designer's RTL (for example, correct number of arguments, or correct expression width).

Example 4-2 PLI checktf routine for $assert_always

```
/******************************************
* checktf routine to validate arguments
******************************************/
int assert_always_checktf(char *user_data)
{
  if (tf_nump() != 1)
    tf_error("$assert_always only 1 argument.");
  else if (tf_sizep(1) != 1)
    tf_error ("$assert_always argument size!=1");
  return (0) ;
}
```

The details for the individual PLI routines used in the assert_always_checktf() routine shown in Example 4-12 are as follows:

tf_error The tf_error () routine, shown in Example 4-2, is similar to the *C* printf() function and can be used to print an error message to the simulator's output window. The tf_error () routine, when called from a user's *checktf* routine, prints an error message and then aborts the simulator process.

tf_nump The tf_nump() routine, shown in Example 4-2, returns the number of arguments passed into the Verilog-instantiated $assert_always PLI task. If the number of arguments is not equal to one in our example, then the *checktf* routine reports an error prior to the beginning of simulation.

tf_sizep The tf_sizep() routine, shown in Example 4-2, returns the bit width for an indexed, referenced argument in the instantiated PLI call. The *checktf* Aroutine, illustrated in Example 4-2, checks that the first argument used in the instantiated PLI call, either a variable or an expression, is of size 1 (for example, `$assert_always (a==0)`, in which the first argument is the expression `a==0` and is of size 1). If the bit width of the first argument is not equal to one, then the *checktf* routine reports an error prior to the beginning of simulation.

Example 4-3 PLI calltf routine for $assert_always

```
/**********************************************
* calltf routine to check assertions
************************************************/
int assert_always_calltf (char *user_data)
{
 /* read current value */
  if (tf_getp(1) == 0) {
    io_printf ("ASSERT ALWAYS ERROR at %s:%s\n",
           tf_strgettime(),  tf_spname());
    tf_dofinish() ; /* stop simulation */
  }
  return(0);
}
```

Calltf routine

The assert_always_calltf() *calltf* routine, shown in Example 4-3, is invoked each time an $assert_always() PLI task is encountered during the simulator's procedural visit through the Verilog code. This routine validates the user's assertion test expression.

The details for the individual PLI routines used in the assert_always_calltf routine shown in Example 4-2 are as follows:

tf_getp The tf_getp ( ) routine returns the specific instance's current value for a referenced argument. In our example, the tf_getp(1) is referencing the first argument in the instantiated PLI assertion, which is the Verilog expression we are asserting to be true.

io_printf The io_printf() routine is similar to the *C* printf() function and can be used to print formatted text for up to twelve arguments. The text message will be printed to both the simulator's output window and the simulator's output log.

tf_strgettime The tf_strgettime() routine returns the current simulation time as a string.

tf_spname The tf_spname ( ) routine returns a point to a string that contains the hierarchical path name for the instantiated PLI assertion.

tf_dofinish The tf_dofinish() routine performs the same function as the Verilog $finish ( ) built-in system task—which is to close all open files and cause the simulator to exit.

Sutherland [2002] provides an excellent description of techniques for linking the user's PLI application to various Verilog simulators.

4.1.2 Assertions within a simulation time slot

procedural assertions may encounter false firings in simulation

Unlike the OVL assertion modules, which sample the Verilog assertion expression at a clock edge, PLI-based procedural assertions validate the assertion expression each time an assertion is encountered during a procedural visit through the code. Hence, procedural assertions run the risk of false firing due to the transient behavior of variable assignments during event-driven simulation.

Example 4-4 illustrates this potential for false firing using a `$display` system task to report the assertion violation. That is to say, the ordering (or scheduling) of events within the same simulation time slot can cause the procedural always block to execute multiple times. Therefore, the transient behavior of the `a` and `b` variables within the same time slot can cause a procedural assertion to fire.

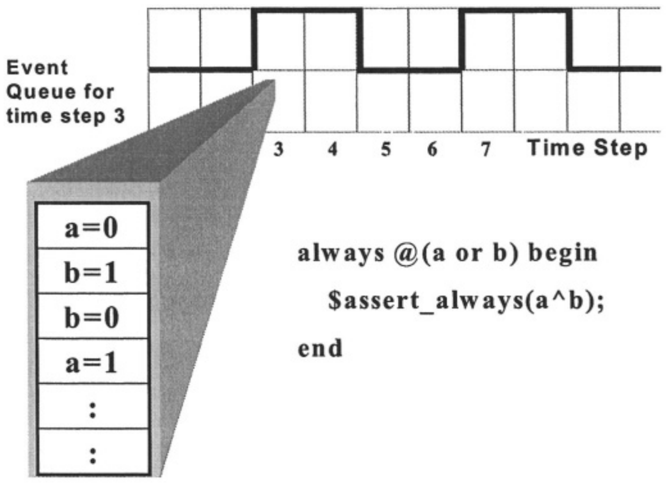
Example 4-4 False firing of procedural assertion

```
always @(a or b) begin
  $display (a^b);
end
```

Figure 4-1 illustrates multiple time slots of the simulator, and then shows the details of the simulator's event queue for time slot 3. For demonstration purposes, the simulation time slots are represented symmetrically, although in practice this would generally not be the case.

Notice the details for the event queue in time slot 3. As simulation progresses, new variable assignments are placed on the event queue and are scheduled for evaluation within the current time slot. This is due to the occurrence of new blocking assignments generated during the current simulation time slot. It is possible that the PLI assertion would fire when `a==0` and `b==0` during an early evaluation of the procedural code for time slot 3. However, later within this same time slot, variable `a` is scheduled for evaluation with a new value of 1. Any previous assertion violations are no longer valid for this time slot. Hence, to prevent false firings, the PLI routine must be constructed in such a way that it will wait to evaluate the assertion expression after the transient behavior of the simulated variables settles out, which is at the end of simulation time slot 3.

Figure 4-1 **Simulation event queue**



Example 4-5 illustrates an enhancement to the $assert_always() PLI code previously shown in Example 4-2. This enhancement prevents a false firing of the assertion by rescheduling the assertion evaluation to occur at the end of the time slot (after all transient evaluations have settled out).

The PLI *mistf* routine is used to execute a *C* application after a miscellaneous event occurs during simulation. Miscellaneous simulation events include: *end of compilation, end of simulation, end of a simulation time slot,* as well as many other simulation events. (See Sutherland [2002] for additional details on PLI *misctf* routines and simulation events.) In the following section, we will demonstrate how to perform a callback at the end of the current simulation time slot using a tf_rosynchronize routine combined with a *misctf* routine.

tf_rosynchronize The tf_rosynchronize () routine shown in the assert_always_calltf() code in Example 4-5 schedules a callback to a user-defined *mistf* routine (for example, assert_always_mistf()). This callback occurs at the end of the current simulation time slot.

The simulator generates the `reason` constant, shown in Example 4-5, and passes it to the *mistf* routine during its simulation call. If the *mistf* routine is called due to an *end of a time slot event,* the `reason` integer is set to a constant value of `REASON_ROSYNCH`, as defined in the simulator's *verisuser.h* file. If the assert_always_misctf routine is called for any simulation event, other than a `REASON_ROSYNCH`, then the *mistf* routine exits without performing the assertion check. By checking the assertion only during a REASON_ROSYNCH event, false firing of assertions within a simulation time slot is eliminated.:

Example 4-5 $assert_always with callback at end of current time slot

```
/*****************************************
* checktf routine to validate arguments
*****************************************/
int assert_always_checktf(char *user_data)
{
  if (tf_nump() != 1)
    tf_error ("$assert always only 1 argument.");
  else if (tf_sizep(1) != 1)
    tf_error("$assert_always argument size!=1");
  return(0);
}
/*****************************************
* calltf routine to schedule callback
*****************************************/
int assert_always_calltf(char *user_data)
{
  tf_rosynchronize();
  return(0);
}/*****************************************
 * misctf routine to check assertion
 *****************************************/
int assert_always_misctf(char *user_data,
                         int reason, int paramvc)
{
  if (reason != REASON_ROSYNCH)
    return(0);

/* read current value */
  if (tf_getp(1) == 0) {
    io_printf ("ASSERT ALWAYS ERROR at %s:%s\n",
            tf_strgettime(),tf_spname());
    tf_dofinish(); /* stop simulation */
   }
  return (0);
}
```

Nested PLI assertion problem

In the previous section, we demonstrated how to construct a PLI routine that can safely prevent a false firing within the procedural code by scheduling a callback evaluation that is performed at the end of the simulation time slot in which the PLI routine was called. However, this assertion can still encounter problems. Refer to and note that if there is a transient behavior of the c variable that initially causes the procedural block to execute by assigning one to c, this would schedule a callback at the end of the current simulation time slot. However, if the variables within the simulation reached a steady state and the final value of the variable c is zero, then the PLI assertion really should not have been visited during the current time slot. This could cause a false firing of the assertion.

Example 4-6 False firing of procedural assertion

```
always @ (a or b or c) begin
  if (c)
    $assert_always (a^b);
end
```

To prevent a false firing, we recommend that you associate a sampling clock with the PLI-based assertions that will cause the assertion to be scheduled for a future evaluation on an edge event associated with the change of the clock. This technique is discussed in the next sections.

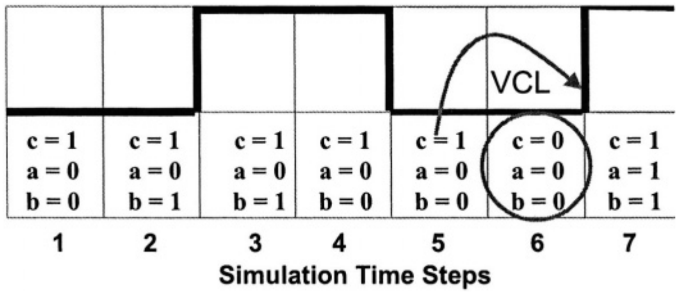
4.1.3 Assertions across simulation time slots

Often, procedural blocks represent purely combinational logic that is bounded between sequential elements that are described in separate clocked procedural blocks. However, the designer generally focuses on the cycle-based semantics for any assertions within these combinational blocks. In this section, we demonstrate a technique of associating a clock with a PLI-based assertions to achieve assertion cycle-based semantics.

false firing example across multiple time slots

Figure 4-2 illustrates seven time slots of the simulator for the procedural block shown in Example 4-6. For demonstration purposes, the simulation time slots are represented symmetrically, although in practice this would generally not be the case. At time slots 2 and 3, no assertion fires since the a and b variables are mutually exclusive. For time slots 1, 4, 5, and 7; the procedural assertion fires due to the non-mutual exclusivity of the a and b variables. At time slot 6, the c variable should prevent evaluation of the procedural assertion. However, if the combinational logic feeds into sequential elements, we are generally only interested in the cycle-based semantics for these procedural assertions. In other words, we should evaluate the procedural assertion at the rising edge of the clock.

Figure 4-2 **Assertions across multiple simulation time slots**



cycle-based semantics for procedural assertions

Example 4-7 illustrates a VCL version of the PLI $assert_always_ck assertion that attempts to prevent a false firing of the assertion across multiple simulation time slots by introducing a sampling clock to the assertion. Note that the clock is not part of the procedural block's event sensitivity list (for example, `@ (a or b or c))`. In other words, we do not want to change the procedural block's simulation behavior by having it trigger on the rising and falling edge of the clock. However, we want to ensure that the assertion schedules an evaluation of the PLI routine on the next rising edge of a clock

Example 4-7 **Procedural assertion with a Value Change Link on the clock**

```
always @ (a or b or c) begin
   :
  if  (c)
     $assert_always_ck (ck, a^b) ;
 end
```

a simple clock edge detection approach will not work

Note that we cannot be assured that the procedural code in Example 4-7 will execute prior to and after a rising edge of the clock. For example, the variables in the event sensitivity list do not change values between time slot 2 and time slot 3 (as shown in Figure 4-2), which would mean that the procedural block would not be evaluated during time slot 3. Hence, we cannot simply detect an edge of the clock for assertion evaluation by keeping a record of the clock value within the PLI routine for previous procedural visits of the assertion. However, by using PLI Value Change Link (VCL) routines, the simulator can schedule an evaluation of the assertion to occur on the appropriate edge of the clock.

Controlling assertion evaluations by a clock

using the VCL to monitor clock edges for assertions

The PLI assertion shown in Example 4-7 can be constructed utilizing the VCL routines within the ACC library. These VCL routines allow the Verilog simulator to schedule an evaluation of the user's PLI *consumer* routine by monitoring a specific simulation object for value changes. For our example, this enables us to delay the evaluation of the assertion until a positive edge of the clock occurs at the start of steps 3 and 7 as shown in Figure 4-2.

Example 4-8 (below) demonstrates $assert_always_ck assertion coding that uses the VCL to provide cycle-based semantics for the assertion evaluation.

Each time the $assert_always_ck task is encountered during a procedural visit through the Verilog code, the *calltf* routine, shown in Example 4-8, activates a VCL monitor for the assertion clock. The `assertion->active` flag tracks previously activated VCL monitors to prevent activating multiple monitors between clock edges.

The VCL *consumer* routine is called whenever the assertion's clock changes values (that is, the *clock object*). This routine retrieves the previously recorded clock value, captured in the *calltf* routine, and uses this value to determine rising edges of the clock. If a rising edge is detected, the *consumer* routine retrieves and validates the assertion test expression. After validating the assertion, the *consumer* routine removes the VCL monitor from the clock object. Future evaluations of the assertion now depend on encountering new procedural visits to the PLI assertion task; at which time, a new VCL monitor is activated for the clock object.

The details for the individual PLI routines used in Example 4-8 are as follows:

acc_initialize

The acc_initialize () routine is required when using the ACC library routines. This function resets the ACC environment to default values.

tf_setworkarea

The tf_setworkarea () routine enables a specific instance of a system task to store a pointer for working area PLI application data. The stored data can be retrieved for use within a *mistf* or *consumer* routine. In our $assert_always_ck example, we must store one handle for each instance to the clock object, which will be used during the VCL evaluation of the assertion's *consumer* routine. In addition, we must store a string that represents the instance path and the value of the clock during the last PLI call, to determine a rising edge of the clock within the *consumer* routine.

Example 4-8 $assert_always_ck with VCL callback to consumer routine

```
typedef struct always_t {
  int active;
  handle clk_handle;
  char *instance;
  int clk;
} Always;
/*****************************************
* checktf routine to validate arguments
*****************************************/
int assert_always_ck_checktf(char *user_data)
{
  if (tf_nump() != 1)
    tf_error ("$assert_always_ck only 1 argument.");
  else if (tf_sizep(1) != 1)
    tf_error ("$assert_always_ck argument size!=1");
  return(0);
}
/*****************************************
* calltf routine setup VCL for clock
*****************************************/
int assert_always_ck_calltf(char *user_data)
{
    struct always_t * assertion;

    acc_initialize();
    assertion = (Always *) tf_getworkarea();

    if (assertion == NULL) {
      assertion = (Always *) calloc(1, sizeof(Always));
      assertion->clk_handle = acc_handle_tfarg(1);
      assertion->instance = tf_getinstance();
      assertion->active = 0;
    }
    assertion->clk = tf_getp(1) & 1;
    tf_setworkarea((char *) assertion);
    if (assertion->active == 0) {
      assertion->active = 1;
     acc_vcl_add(assertion->clk_handle, assert_always_ck_consumer,
                    char *) vcl_verilog_logic);
    }
}
/*****************************************
* checktf routine to validate arguments
*****************************************/
int assert_always_ck_checktf(char *user_data)
{
  if (tf_nump() != 1)
    tf_error ("$assert_always_ck only 1 argument.");
  else if (tf_sizep(1) != 1)
    tf_error ("$assert_always_ck argument size!=1");
  return(0);
}
```

Example 4-8 $assert_always_ck VCL callback to consumer routine

```
/***************************************
* calltf routine setup VCL for clock
****************************************/
int assert_always_ck_calltf(char *user_data)
{
    struct always_t * assertion;

    acc_initialize();
    assertion = (Always *) tf_getworkarea ();

    if  (assertion == NULL) {
      assertion = (Always *) calloc(1, sizeof(Always));
      assertion->clk_handle = acc_handle_tfarg(1);
      assertion->instance = tf_getinstance ();
      assertion->active = 0;
    }
    assertion->clk = tf_getp(1)  & 1;
    tf_setworkarea( (char *) assertion);
    if  (assertion->active == 0) {
      assertion->active = 1;
      acc_vcl_add(assertion->clk_handle,
             assert_always_ck_consumer,  (char *)  vcl_verilog_logic);
    }
    acc_close () ;
    return(0) ;
}
/***************************************
 * consumer routine to perform check
****************************************/
int assert_always_ck_consumer(p_vc_record vc_record)
{
  int clk;
  struct always_t *assertion;
  assertion =  (Always *) vc_record->user_data;
  switch  (vc_record->vc_reason) {
    case  logic_value_change:
    case  sregister_value_change: {
      clk = tf_igetp(1, assertion->instance)  & 1;
      if (assertion->clk==0 && clk==1)  {
       if  (tf_igetp(2, assertion->instance)==0  {
         io_printf  ("ASSERT ERROR at %s:%s\n",
             tf_strgettime () , tf_ispname(assertion->instance)) ;
         tf_dofinish();  /* stop simulation */
       }
       assertion->active = 0;
       acc_vcl_delete(assertion->clk_handle,
                         assert_always_ck_consume, (char *)  assertion,
                         vcl_verilog_logic) ;
     }
    assertion->clk = clk;
   }
  }
 return  (0);
}
```

tf_getworkarea — The tf_getworkarea () routine retrieves a pointer to previously stored work area data for each PLI instance.

acc_handle_tfarg The acc_handle_tfarg () routine returns a handle for the object referenced by the argument number in the instantiated PLI system task. In our example, we reference the first argument in the instantiated PLI call, which is the clock.

tf_getinstance The tf_getinstance () routine returns a pointer to the instance of a PLI system task. In our example, this instance pointer is used in our *consumer* routine to obtain the values of the assertion test expression and clock, as well as hierarchical path name to the instantiated PLI assertion.

acc_vcl_add The acc_vcl_add () routine adds a VCL monitor for the assertion clock defined by our specific, instantiated PLI assertion. When the clock changes values, the *consumer* routine executes.

acc_close The acc_close () routine frees all memory allocated by the acc_initialize () routine and resets all configuration parameters to their default values.

acc_vcl_delete The acc_vcl_delete () routine removes any previously activated VCL monitors on a specific instance object. In our example, we want to stop monitoring the assertion clock as soon as a rising edge of the clock occurs.

tf_igetp The tf_igetp () routine is similar to the previously-defined tf_getp routine. This routine requires a pointer to the instance of the PLI assertion for which we wish to retrieve the argument value. The instance pointer was captured by the *calltf* routine and stored within the instance's working area.

tf_ispname The tf_ispname () routine returns a pointer to a string that contains the hierarchical path name for the instantiated PLI assertion. This routine requires a pointer to the instance of the PLI assertion.The tf_igetp () routine is similar to the previously defined tf_getp routine. This routine requires a pointer to the instance of the PLI assertion for which we wish to retrieve the argument value. The instance pointer was captured by the *calltf* routine and stored within the instance's working area.

4.1.4 False firing across multiple time slots

The problem with the VCL PLI assertion approach, shown in Example 4-7, is that the procedural path down to the assertion is no longer valid (or sensitized) at simulation time slot 6, as demonstrated in Figure 4-2, since the c variable is false at this point in time. Therefore, the $assert_always_ck should not evaluate at the next clock edge. However, a VCL was previously activated due to a visit by the assertion at time slots 3, 4 and 5 to monitor the clock. To solve this problem, the procedural always block shown in Example 4-7 requires a method of deleting any previously scheduled VCL clock monitors for all PLI assertion routines during each new visit to the procedural block.

Example 4-9 demonstrates one technique that could be used to delete all previously scheduled VCL clock monitors within the labeled procedural block. When an $assert_always_ck assertion is encountered during a procedural visit, the $assert_always_ck PLI code records the activation of an assertion in a globally accessible data structure, which identifies the specific assertion in the labeled procedural block that was activated. The $assert_delete user-defined task has access to this data structure. Upon future visits to the procedural block, any previously activated VCL PLI assertion is disabled by the $assert_delete task, which calls an acc_vcl_delete() routine for any previously initiated VCL assertions.

Example 4-9 Procedural assertion with a Value Change Link on the clock

```
always @(a or b or c) begin : my_block
  $assert_delete ();
  :
  if (c)
     $assert_always_ck (ck, a^b);
end
```

In general, procedural assertions can be deeply nested within case and if statements and should only be evaluated when the procedural path down to the assertion is valid during the last time slot prior to the clock. Otherwise, a false firing of the assertion can occur. Semantically, the PLI procedural assertion should behave as shown in Example 4-10, below.

Example 4-10 Required semantics for safe RTL assertion

```
reg test_assertion;
always @ (a or b or c) begin
  test_assertion = 1'b1
   :
  if (c) test_assertion = a^b;
end
assert_always ovl_assert (ck, reset_n,
                              test_assertion);
```

For this example, we introduce a variable to test the assertion within the procedural code. The variable is initialized to true at the beginning of the procedural code. Later, the test expression is set based on the evaluation of the a^b expression. To complete our example, an OVL assert_always module is instantiated to sample the *test_assertion* variable on a rising edge of a clock.

Hence, this example illustrates the semantics required when creating a set of PLI-based assertions.

Note that the OVL assertion module has a reset signal as an argument. The PLI-based $assert_always_ck task in our Example 4-8 and Example 4-9 could be modified to include a reset signal as an argument. In addition, an optional assertion message could be passed in as an argument, which can be displayed if the assertion fires. The PLI application determines if the optional parameter has been passed in by calling the tf_nump() routine. (Refer to "tf_nump" on page 4-106.) If the number of arguments in our example is equal to three, then only the *clock, reset,* and assertion *test expression* have been passed in as arguments. If the number of arguments is four, then the optional error message was specified in the instantiated PLI task. The tf_typep() routine can be used to ensure that the optional argument is a literal string (for example, `tf_type (4) ==TF_STRING`), which is used in the *checktf* routine for the $assert_always_ck task. The assertion would be instantiated as shown in Example 4-11, below.

Example 4-11 PLI assert always check

```
$assert_always_ck (ck, reset_n, expression,
          "My optional assertion error message" )
```

4.2 PLI-based assertion library

In the previous section, we demonstrated a PLI-based implementation for an $assert_always procedural assertion. You can construct a library of PLI procedural assertions, with functionality similar to the OVL described in Chapter 3, "Specifying RTL Properties", just as we constructed the previous $assert_always assertion example.

We suggest you begin by constructing a simple procedural $assert_error task. This is useful for the default alternative error branch in case statements, or the else error branch in if statements. Example 4-12 (below) demonstrates the use of a simple $assert_error task within a case statement.

Example 4-12 $assert_never within a CASE default branch

```
case({a,b}) begin
  2'b01: s = c+4'b0001;
  2'b10: s = c-4'b0001;
  default: $assert_error("Case default error") ;
endcase
```

When creating a PLI-based assertion library, the target type of procedural block must be considered (for example, a clocked procedural block versus a non-clocked procedural block). Hence, the designer might decided to implement two sets of PLI-based assertions, as shown in Example 4-13. Note that the first assertion applies to clocked procedural block, while the second applies to a non-clocked procedural block.

Example 4-13 Clocked versus non-clocked procedural PLI assertions

```
always @ (posedge ck) begin
  $assert_always (a^b);
end

always @(a or b) begin
  $assert_delete ();
  $assert_always_ck (ck, a^b);
end
```

In the following section, we demonstrate another example of implementing a PLI-based assertion that could be included in the designer's PLI-based assertion library.

4.2.1 Assert quiescent state

After you have tried your hand at the PLI-based assert_always, the next one you will want to attempt is the assert quiescent state, which is useful for validating state machines after the completion of a transaction or sequence of events. This PLI-routine performs a new and powerful function not supported by the released version of the OVL assert_quiescent_state module (that is, our PLI routine executes an automatic callback at the *end ofsimulation* to perform a consistency checking). The *end of simulation* assert quiescent state evaluation is useful for identifying bus transactions that have stalled, deadlock states, and inconsistent behavior between multiple state machines.

Example 4-14 PLI quiescent state assertion

```
#ifndef MAXINT
#define MAXINT 32
#endif
#ifndef NUMWORDS
#define NUMWORDS (x) (x-1) /MAXINT
#endif
/**********************************************
 * checktf application to validate interface
 **********************************************/
int quiescent_state_checktf(char *user_data)
{
  s_tfexprinfo expr_info;
  int current_size, end_size;
  if (tf_nump() != 2)
    tf_error(
      "$assert_quiescent_state requires 2 arguments");
  current_size = tf_sizep(1);
  end_size = tf_sizep(2);
  if (current_size != end_size)
    tf_error(
     "$assert_quiescent_state arg(1) and arg(2) size mismatch.");
    return(0);
 }

 /*******************************************
  * calltf routine
  *******************************************/
int quiescent_state_calltf (char *user_data)
{
  tf_rosynchronize ();
  return(0);
}
/*************************************************
 * misctf routine to perform end of sim. check
 *************************************************/
int quiescent_state_misctf(char *user_ata,
                         int reason, int paramvc)
{
  int current_quiescent_state,
  int end_quiescent_state;
  int current_high, end_high;
  int i, size, ok=1;
  s_tfexprinfo current_expr_info, end_expr_info;

  if (reason != REASON_FINISH)
    return(0);
  size = tf_sizep(1);
  if (size <= 32) {
    current_quiescent_state = tf_getp(1);
    end_quiescent_state = tf_getp(2);
    if (current_quiescent_state !=
                              end_quiescent_state)
      ok = 0;
   }
```

Example 4-14 PLI quiescent state assertion

```
    else if (size <= 64) {
      current_quiescent_state =
                      tf_getlongp(&current_high, 1);
      end_quiescent_state =
                      tf_getlongp(&end_high, 2);    if
  (!((current_quiescent_state ==
            end_quiescent_state) &&
          (current_high == end_high)))
        ok = 0;
    }
    else {
       (void) tf_exprinfo(1, &current_expr_info);
       (void) tf_exprinfo(2, &end_expr_info);
       for (i=
         NUMWORDS(current_expr_info.expr_vec_size);
         i >= 0; i-=1) {
         if (current_expr_info.expr_value_p[i].avalbits
             != end_expr_info.expr_value_p[i].avalbits) {
           ok = 0;
           break;
          }
       }
    }
    if (ok==0) {
      io_printf ("$assert_quiescent_state error: %s : %s\n",
                           tf_spname(), tf_getp(1));
    }
    return (0);
  }
```

The details for the individual PLI routines used in Example 4-14 are as follows:

end of simulation automatic check

The `reason` constant is generated by the simulator and passed to the *mistf* routine during its simulation call. If the *mistf* routine is called due to an *end of a simulation event,* the `reason` integer is set to a constant value of `REASON_FINISH,` as defined in the simulator's *verisuser.h* file.

processing arguments greater than 32 bits

If the quiescent state arguments are greater than 32 bits, but less than 64 bits, then you must use the tf_getlongp() routine to retrieve the argument's value. If the quiescent state arguments are greater than 32 bits, then you must use the tf_exprinfo() function to obtain detailed information about the system task's argument. This information is retrieved into an s_tf_exprinfo structure. The value of the argument is stored in an array of s_vecval structures and is referenced by the expr_value_p pointer from the s_tf_exprinfo structure. The quiescent state is validated by looping through the s_vecval structures and comparing the existing expression value (argument 1) with the expected end of simulation value (argument 2).

Example 4-15 OVL assert_quiescent_state enhanced with PLI task

```
module assert_quiescent_state (clk, reset_n,
     state_expr, check_value, sample_event);
// rtl_synthesis template
  parameter severity_level = 0;
  parameter width=1;
  parameter options = 0;
  parameter msg="VIOLATION";
  input clk, reset_n, sample_event;
  input [width-1:0] state_expr, check_value;

//rtl_synthesis translate_off
`ifdef ASSERT_ON
  parameter assert_name = "ASSERT_QUIESCENT_STATE";

  integer error_count;
  initial error_count = 0;

  `include "ovl_task.h"
  `ifdef ASSERT_INIT_MSG
    initial ovl_init_msg;
  `endif

  reg r_sample_event;
  initial r_sample_event=1'b0;

  always @ (posedge clk)
    r_sample_event <= sample_event;

  reg r_PLI_active;
  initial r_PLI_active=1'b0;
  always @ (posedge clk)
    if (r_PLI_active==1'b0) begin
      r_PLI_active=1'b1;
      $assert_quiescent_state (state_expr, check_value);
    end
  always @(posedge clk) begin
    `ifdef ASSERT_GLOBAL_RESET
      if (`ASSERT_GLOBAL_RESET != 1'b0)
    `else
      if (reset_n != 0)
    `endif
      begin
        if ((r_sample_event == 1'b0 && sample_event == 1'b1) &&
              (state_expr != check_value))
        begin
          ovl_error("");
        end
      end
  end
`endif
//rtl_synthesis translate_on
endmodule
```

OVL enhancement with PLI call In addition to building your own PLI-based assertion library, you can use PLI assertions to enhance the existing OVL modules. Example 4-15 (below) shows our enhancement to the OVL assert_quiescent_state module, which adds the PLI procedural $assert_quiescent_state assertion to the OVL module. This PLI routine validates the user's specified state automatically at the end of simulation. The r_PLI_active signal enables us to visit the PLI routine once, setting up the callback to the *misctf* routine to occur at an *end of simulation event,* and then ignore the PLI call for the remainder of the simulation run.

The designer might decide to instantiate an enhanced version of the OVL assertion shown below in Example 4-16 to check for only an *end of simulation* violation with our new PLI routine. In this case, the OVL `sample_event` argument is set to 1'b0. When simulation completes, the PLI's *mistf* routine is invoked to automatically check the `controller_state` bits 7 through 0 for the value specified by the `` `CNTRL_START_STATE `` macro. If any state other than the expected state is encountered at the end of simulation, the assertion will fire.

Example 4-16 OVL instantiated assertion to check for an end of simulation violation

```
    assert_quiescent_state valid_state (ck,
          reset_n, controller_state[7:0],
          'CNTRL_START_STATE, 1'b0);
```

4.3 Summary

In this chapter, we demonstrated how to create a set of Verilog Programming Language Interface (PLI) assertions. We then discussed considerations to take when implementing a PLI-based assertion solution and their impact on simulation. For example, a negative impact on simulation can occur if the PLI-based assertion is called at every visit through procedural code to perform its check, particularly when the assertion is not violated. We then presented some of the strengths of PLI-based procedural assertions; for example, expressiveness and convenience. We also discussed the weaknesses of procedural assertions; for example, over constraining and false firing if not properly constructed. Broad use of PLI-based assertions places a project in the middle of complex portability issues. Hence, we recommend you limit the use of PLI-based assertions to a very small set.