The Facts and Fallacies of Verilog Event Scheduling: Is the IEEE 1364 Standard Right or Wrong?

Lee Tatistcheff, Charles Dawson, David Roberts, Rohit Rana
Cadence Design Systems, Inc
leet@cadence.com

Abstract

We examine both the ambiguities and inconsistencies of the IEEE 1364 and the behavior of Verilog-XL® in order to understand the controversy surrounding the precise scheduling behavior of tf_synchronize and cbReadWriteSynch. We propose two new unambiguous callback types to be added to the VPI library and we demonstrate methods of using these callbacks both within VPI and earlier TF/ACC PLI applications.

1. Introduction

As C modeling of hardware to interface with Verilog simulators has become more and more widespread, the requirements for the PLI interface have become better defined. Many of the earlier interface routines did not accommodate these requirements for precision, and some of the later interface routines were insufficiently clear about them. In this paper, we examine two specific callback types, reason_synch and cbReadWriteSynch, attempt to understand the origin of the ambiguity, and demonstrate an appropriate solution.

Why would a C model need to synchronize to a write-legal slot at the end of the time slice? Because the model needs to know that its inputs have finished changing and are stable before it calculates its response.

Why would a C model need to synchronize just before the non-blocking assignments are updated? Because it needs to capture the old values of certain registers and save them before the new values appear.

The Verilog-XL® and Cadence NC-Verilog® simulators execute callbacks for reason_synch at different times: Verilog-XL® executes them at some unspecified time before the non-blocking assignment updates, and NC-Verilog® executes them just before the tool is about to close the time slice down for writing (generally, just before callbacks for reason_rosynch). There are many who believe that NC-Verilog® is violating the IEEE 1364 in this. We believe instead that Verilog-XL® is incorrect in its behavior – some of us were actively implementing changes to this in the early 1990's – and that the 1364 simply attempted to capture that incorrect behavior.

By contrast, both Verilog-XL® and NC-Verilog® execute the newer VPI cbReadWriteSynch at the same time, when the tool believes it is done with the time slice, but before shutting down write capability. Some parties have asserted that both tools are incorrect in this, and that cbReadWriteSynch belongs before the non-blocking assignment updates, not after. They are not correct. The engineers who implemented VPI for both Verilog-XL® and NC-Verilog® (Charles Dawson and David Roberts) were members of the IEEE 1364-1995 and 1364-2000 task forces, as well as participating in the original specification by OVI; the behavior of NC-Verilog® and Verilog-XL® reflects both their intent and the needs of the users at that time.

2. Specification history

In 1985, Gateway Design Automation introduced the Verilog-XL® simulator and its Programming Language Interface (PLI). Both the Verilog and the PLI languages were proprietary to Gateway, defined in the Verilog-XL® Language Reference Manual and the PLI Manual [1][2]. In 1989, after merging with Gateway, Cadence Design Systems, Inc., released both Verilog and PLI to the public domain.

The Open Verilog International (OVI) organization formed in 1989, and released a specification for both languages in 1993 [3][4][5]. This specification included the first definition of second-generation PLI routines (initially ACC routines, but finalized as VPI routines in the IEEE 1364-1995) intending that these new routines would supersede the earlier TF and ACC routines completely.

In 1995, the IEEE general membership accepted the Verilog Language Reference Manual as specification 1364; this included definitions of the TF/ACC routines for backward compatibility, and definitions of the VPI routines [7]. The stated purpose of the 1364 working group was "not to spend a lot of time extending the

language, but […] to concentrate on clarifying the language" [8]. The section on scheduling semantics (Chapter 5) was first introduced in this version of the language and was not reviewed by the PLI task force.

A new version of the 1364 was approved early in 2001 [9]. It contains corrections and clarifications for some TF/ACC routines, but specifically excludes enhancements to them. According to one working group member, "All enhancements and new features in the Verilog PLI standard will only be made in the VPI routines. The older ACC and TF routines will continue to remain in the standard to provide backward compatibility, but no new ACC and TF routines will be added" [10].

3. Synchronization PLI

3.1. TF synchronization

The tf_synchronize() routine was introduced (some time before the language was made public) mainly for the purpose of modeling certain hardware units in C. The intent of these routines was to permit the model to be called at the end of a time slice, but still be permitted to modify values *in the current time slice*. This way the model can decide what values to place on its outputs based on the final values of its inputs or other criteria.

This in sharp contrast with the tf_rosynchronize() routine which is called after no more modifications to the current time slice are permitted; tf_rosynchronize is intended for monitoring functions, e.g., waveform tools or response checkers.

The PLI Reference Manual from 1989 describes these two routines as follows [11]: "Purpose: synchronize to end of simulation time unit … the routines tf_synchronize and tf_isynchronize allow the processing of parameters to be delayed until the end of the current simulation time slot. This is useful when the user wants to synchronize all parameter value changes and process them after all that will change at a particular time have changed." "Purpose: synchronize to end of simulation time slot … the routines tf_rosynchronize and tf_irosynchronize allow certain processing to be delayed until the end of the current simulation time." The language referring to when in the time slice these callbacks execute is consistent, referring only to the end of the time period, not some intermediate phase. This is consistent with the understanding of other members of the IEEE PLI Task Force, as well [12].

The IEEE 1364-1995 introduced a different interpretation of the time for reason_synch, however. These routines are discussed in several places. In chapter 21, where the TF details are described, it says: "The routine tf_synchronize() shall call the associated misctf application *at the end of the current simulation time step* with reason_synch […] The routine tf_rosynchronize() shall call the associated misctf *at the end of the current simulation time step* with reason_rosynch […]" [emphasis added]. Note that the language for when in the time slice each routine will be called is identical and vague, very similar to that used in the original Verilog-XL® PLI manual.

In chapter 20, where the overview of TF routine usage is presented, the language is much less vague, but makes reference to terms introduced in the new chapter 5: the active, inactive, and non-blocking assignment update sections of the stratified queue: "The misctf can be called at the end of the current time step or at some future time step. The tf_synchronize() routine shall place the callback at the end of the inactive event queue for the current time step. The tf_rosynchronize() callback shall occur after all active, inactive, and nonblocking assign events for a time step have been processed." Note that the first sentence conflicts with the second: the "end of the current time step" is not the same thing as "the end of the inactive event queue for the current time step".

The new chapter 5 of the 1364-1995 (which attempts to capture the behavior of Verilog-XL®, not modify it) mentions PLI synchronization in one place: "The callback procedures scheduled with PLI routines such as tf_synchronize() […] shall be treated as inactive events."

In the balloting process for the IEEE 1364-2000 proposal PTF-37 clarifying the language here was rejected. It would have said in section 5.3: "PLI callbacks scheduled with PLI routines tf_synchronize() […] shall create synchronize events. A software implementation may process synchronize events either before or after nonblocking assign update events." The discussion noted that neither Verilog-XL® nor NC-Verilog® behaved as the 1364-1995 described, but that it correctly described the behavior of the VCS® simulator from Synopsys.

3.2. VPI synchronization

The OVI PLI 2.0 specification introduced callbacks at defined points in the time slice, not just for defined reasons, including: cbAtStartOfSimTime, cbReadWrite-Synch, cbReadOnlySynch, cbNextSimTime, and cbAfter-Delay. The callbacks most of interest to us are defined as follows: cbReadWriteSynch "occurs after execution of events for a specified time" and cbReadOnlySynch "*Same as cbReadWriteSynch*, but writing values or scheduling events before the next scheduled event is disabled" [emphasis added] [6]. The IEEE 1364-1995 modified the language somewhat, substituting "shall occur" for "occurs", etc. Another author of this specification has described his intent for this callback quite clearly [13],

also pointing to the end of the time slice, *after* non-blocking assignment updates.

This language conflicts with that of IEEE 1364-1995 section 5.3: "The callback procedures scheduled with PLI routines such as […] vpi_register_cb(cb_readwrite) [sic] shall be treated as inactive events."

Although "same as cbReadWriteSynch" could lead to other confusion, this much of the authors' intent is clear: cbReadWriteSynch is called when the simulator believes it has completed the time slice but PLI is still permitted to put values in the current time, and cbReadOnlySynch is called under the same conditions but putting values in the current time is disabled.

4. Discussion

The reason for the ambiguity in the original descriptions of reason_synch is that the Verilog simulator that provided the first definition of the routines, Verilog-XL®, executed these callbacks at a variety of points in the time slice.

Consider the example in section 7.1 of this paper: no pair of the command lines listed actually produces the same results for both calltf and callback for reason_synch. Note that this example does not conform – in numerous ways – to the description of the event queue in Chapter 5 of the IEEE 1364-1995. Note also that in this case, Verilog-XL® *does* execute the synch callback some time before the next non-blocking assignment updates, even though our desire was to execute them at the *end* of the time slice.

Over time, Cadence has received numerous requests for clarification of precisely when the callbacks will occur, as well as requests for corrections when the user believes that the callback is occurring at the wrong time or not being called when it should [14][15][16][17][18]. Both our customers and we ourselves considered the fact that Verilog-XL® executed the reason_synch callback before the non-blocking assignment updates to be a bug.

Indeed, until 1995, the scheduling of these callbacks was actively being modified, to get the behavior in-line with our original intent of having the reason_synch callback fire only when Verilog-XL® believed it had finished with the time slice, but before it had closed that slice down (just as NC-Verilog® has always done).

We still consider any other behavior a bug, despite the fact that it has been captured in the IEEE 1364, but have had to acknowledge that significant modifications to the scheduling behavior of Verilog-XL® are no longer acceptable to the user community, and abandoned that effort in 1995.

For several years, NC-Verilog® and Verilog-XL® were the only commercial simulators that had implemented the VPI routines, and since they behaved the same way, the imprecise language in the IEEE 1364-1995 did not matter. Now that the other major commercial Verilog simulators have or are introducing VPI implementations, the discussion over when precisely cbReadWriteSynch should be called is beginning to be as heated as the discussion of reason_synch.

The solution most often proposed to Cadence is "change the behavior to <something>". The problem is that some of the requestors want the callback moved to the end of the time queue, and others want it moved to just before the non-blocking assignment updates. The NC-Verilog® product even has a run-time switch, -nbasync, which forces tf_synchronize callbacks to occur just before the non-blocking assignment updates. Using that switch, however, breaks models that were designed to run with NC-Verilog® in its default operation. Thousands of existing VPI applications have functioned as intended in NC-Verilog® and Verilog-XL® for years; modification of their behavior at this late date is entirely unacceptable to our users.

5. New callbacks

Although we do believe our interpretation of the existing callback times is correct, we also believe that *correctness here is irrelevant*. We are confident both needs are real – our users have demonstrated that need repeatedly – and so PLI needs to accommodate both of them. PLI cannot do that with a single callback reason. We propose two new VPI callback reasons, cbNBASynch and cbAtEndOfSimTime. Once we have finalized discussion with the other major simulation providers (in progress), these will be proposed to the IEEE 1364 for inclusion in the next revision of the specification.

The complete definition of these callbacks cannot be contained within this paper, but we will describe here our intent, as clearly as possible. This proposal is still being debated, and is subject to change before implementation in NC-Verilog® or submission to the IEEE.

The callback, cbAtEndOfSimTime, will fire when the simulator believes that *only* "monitor" events remain to be performed in the current time. Examples of monitor events include, but are not limited to: callbacks for reason cbReadOnlySynch, $strobe, $monitor. Our intention here is that this callback occurs when the simulator has *no* events that can alter simulation values in the current time slice.

The callback, cbNBASynch, will fire when *the next* event is either the non-blocking assignment updates or when it is the callback for reason cbAtEndOfSimTime, whichever comes first. This callback will fire as if it were the first of the non-blocking assignment updates in an associated non-blocking assignment update queue. This

means that if the application modifies any values without delays (e.g., vpi_put_value with vpiNoDelay, acc_set_value with accNoDelay), then the values are modified immediately, but propagated when the event processing for the time slice resumes, after completion of the associated non-blocking assignment update queue. All delayed modifications for the current time (e.g., vpi_put_value with a delay of zero) will occur and be propagated after completion of the associated non-blocking assignment update queue.

Like cbReadWriteSynch, these two callbacks are "one-shots", which is to say, they will execute once per registration (even though both of these callbacks can occur more than once in any given time slot). If the application wants to be called each time these callbacks fire, the application will need to re-register the routine. Similarly, registering a callback for either of these reasons *creates new events*. For example, if an application executing for reason cbAtEndOfSimTime registers a callback for reason cbAtEndOfSimTime, that will create a *new* callback event; this must be used with care, to avoid infinite loops. The final definition of these callbacks must also consider VPI time queue objects, IEEE 1364-2000 section 26.6.40.

This description is intended to accommodate other simulators, which may have implemented callbacks for reason cbReadWriteSynch to occur just before the non-blocking assignment updates, as well as those who, like Cadence, implemented them to occur at the end of the time slice. For both situations, cbNBASynch will occur before cbReadWriteSynch, and cbAtEndOfSimTime will occur after cbNBASynch. This permits simple insertion of the new callbacks, without requiring any simulator to modify its implementation of the existing callbacks. We believe the overarching need of our users is predictable, consistent results, regardless of simulation provider, and that these new callback types will provide that.

For users who have not transitioned their applications to use VPI routines, we demonstrate how to use these callbacks instead of tf_synchronize in a TF/ACC application in section 7.2 of this paper.

6. Conclusions

We have examined the definition of reason_synch and cbReadWriteSynch from their origins through the current day specification, demonstrating where we believe incorrect language was inserted in the IEEE 1364. We believe that users have articulated a strong need for two callbacks types, and so have proposed a cbNBASynch and cbAtEndOfSimTime to supplement the specification; an illustration of how to use these callbacks within a TF/ACC application was provided. We encourage our users to take advantage of these new callbacks when writing their applications, and we encourage other simulation providers to implement these callback types in their VPI libraries.

7. Examples

7.1. Verilog-XL® and tf_synchronize

```
module test;
reg a;
initial begin
  a = 1'b0;
  #2;
  a = 1'b1;
  #2;
 end
dut someinst(a);
endmodule

module dut(myin);
input myin;
wire myin;

reg bb;
wire aa, cc, dd, ee;
wire dummy = 1'b1;

assign cc = myin;
always @ (aa) bb <= aa;

not (aa,cc);
tranif1 (dd,aa,dummy);
nmos(ee,dd,dummy);

always @(myin)
  #0 $sync_call(myin,aa,bb,cc,dd,ee);
always @(aa)
  $display("VLOG: aa fired %b",aa);
always @(bb)
  $display("VLOG: bb fired %b",bb);
endmodule

#include "veriuser.h"

int sync_call(){
  myprint("CALL");
  tf_synchronize();
  return(0);
}

int sync_misc(int user_data, int
reason){
  if (reason == REASON_SYNCH)
    myprint("SYNC");
  return(0);
}

static int myprint(char *where){
  io_printf("%s: myin+abcde =
",where);
```

```
  io_printf("%s, ",tf_strgetp(1,'b'));
  io_printf("%s, ",tf_strgetp(2,'b'));
  io_printf("%s, ",tf_strgetp(3,'b'));
  io_printf("%s, ",tf_strgetp(4,'b'));
  io_printf("%s, ",tf_strgetp(5,'b'));
  io_printf("%s\n",tf_strgetp(6,'b'));
  return(0);
}
```

If this test case is run in Verilog-XL® with the following options, it gets wildly different results, due to the different scheduling variations:

verilog test.v
verilog test.v +noxl
verilog test.v +turbo+3
verilog test.v +caxl
verilog test.v +switchxl

7.2. Using VPI callbacks with TF/ACC applications

VPI callbacks are not associated with a particular system task or function. Therefore, they do not call a misctf routine. Instead, you must setup a new routine to call, and as part of the setup of the callback provide the system task or function instance. The easiest way to do this is to put it in the user_data field. Then all PLI1.0 calls that rely on there being a current system task or function instance need to be changed to their instance specific equivalent (tf_ routines become tf_i routines). To get the prototypes and typedefs for VPI, include the vpi_user.h and vpi_user_cds.h files.

This example illustrates how this would be done for the functions shown in section 7.1. The Verilog remains unchanged, and the new sync.c file looks as follows:

```
#include "veriuser.h"
#include "vpi_user.h"
#include "vpi_user_cds.h"

int sync_callback(p_cb_data cback_p) {
  imyprint("VPCB",cback_p->user_data);
  return(0);
}

int sync_call() {
s_cb_data cback;
s_vpi_time mytime;
vpiHandle cbhdl;
char *tfinst;

  tfinst = tf_getinstance();
  imyprint("CALL", tfinst);

  cback.reason = cbNBASynch;
  cback.user_data = (char *)tfinst;
  cback.cb_rtn = sync_callback;
  mytime.low = 0;
  mytime.high = 0;
  mytime.type = vpiSimTime;
  cback.time = &mytime;

  cbhdl = vpi_register_cb(&cback);
  vpi_free_object(cbhdl);
  return(0);
}

int sync_misc(int user_data, int
reason) {
  return(0);
}

static int imyprint(char *where, char
*tfinst) {
  io_printf("%s: myin+abcde =
",where);
  io_printf("%s, ",tf_istrgetp(1, 'b',
    tfinst));
  io_printf("%s, ",tf_istrgetp(2, 'b',
    tfinst));
  io_printf("%s, ",tf_istrgetp(3, 'b',
    tfinst));
  io_printf("%s, ",tf_istrgetp(4, 'b',
    tfinst));
  io_printf("%s, ",tf_istrgetp(5, 'b',
    tfinst));
  io_printf("%s\n",tf_istrgetp(6, 'b',
    tfinst));
  return(0);
}
```

8. References

[1] Verilog-XL® Reference Manual, Version 1.6, Cadence Design Systems, Inc., USA, 1991.

[2] Programming Language Interface Reference Manual, Version 1.5, Cadence Design Systems, Inc., Lowell, MA, December 1989.

[3] Verilog Hardware Description Language Reference Manual (LRM), Version 2.0, Open Verilog International, Los Gatos, CA, March 1993.

[4] Programming Language Interface (PLI) Manual, Version 1.0, Open Verilog International, Los Gatos, CA, February 1991.

[5] Programming Language Interface (PLI) Manual, Version 2.0, Open Verilog International, Los Gatos, CA, February 1993.

[6] Ibid. p 217.

[7] IEEE Standard Hardware Description Language Based on the Verilog® Hardware Description Language, IEEE Std 1364-1995, Institute of Electrical and Electronics Engineers, Inc., New York, NY, 1996.

[8] Ibid. p iii.

[9] IEEE Standard Hardware Description Language Based on the Verilog® Hardware Description Language, IEEE Std 1364-2000, Institute of Electrical and Electronics Engineers, Inc., New York, NY, 2001.

[10] Sutherland, S., The Verilog PLI Handbook, Kluwer Academic Publishers, Massachusetts, 1999, p 5.

[11] Programming Language Interface Reference Manual, Version 1.5, Cadence Design Systems, Inc., Lowell, MA, December 1989, pp 3-19, 3-20.

[12] Sutherland, S., The Verilog PLI Handbook, Kluwer Academic Publishers, Massachusetts, 1999, pp 397-98.

[13] Ibid, pp 208-209.

[14] Customer Support Case 30001327, Cadence Design Systems, Inc., January 1996.

[15] Customer Support Case 30011194, Cadence Design Systems, Inc., June 1996.

[16] Customer Support Case 30024357, Cadence Design Systems, Inc., January 1997.

[17] Product Change Request 59567, Cadence Design Systems, Inc., May 1992.

[18] Product Change Request 234414, Cadence Design Systems, Inc., October 1998.