Verilog Plus C Language Modeling with PLI 2.0: The Next Generation Simulation Language

Steve Meyer
Pragmatic C Software Corp.
220 Montgomery Street, Suite 925
San Francisco, CA 94104
(415) 296-7017 - FAX (415) 296-0946
Email: sjmeyer@pragmatic-c.com

Abstract: Verilog 98 Programming Language Interface (PLI) 2.0 vpi_ routine API is here advocated for high level electronic design. PLI 2.0 algorithms and source code examples are presented for enhancing Verilog and for modeling abstract hardware. It is argued that because Verilog PLI 2.0 implements high level programming iterators and object abstraction, no new HDL is needed. Verilog PLI 2.0 provides a superior framework for hardware design abstraction and system architecture expressed as computer programs.

1. Introduction

During the last fifteen years, the Verilog Hardware Description Language (HDL) has revolutionized digital circuit design. It is used in nearly every facet of digital design from high level system design to accurate ASIC modeling. Verilog is by far the most popular HDL (EETimes[1997b]). Verilog HDL is standardized as IEEE P1364 standard and defined by P1364 Language Reference Manual (LRM) (P1364LRM[1996]). One reason for Verilog longevity and popularity is its versatility. Verilog combines a good behavioral language for system modeling, RTL circuit description and test bench writing with an accurate gate level simulator. Additional Verilog versatility is provided by a programming language interface (PLI) application programming library (API) for user enhancement and for interfaces to third party tools (P1364LRM Ch. 19-22).

Electronic systems and circuit design now permeate every facet of society. As is common with any widespread activity, there is need for change and improvement. As the number of transistors available on integrated circuits increases, new design approaches become necessary. Currently, the most widely acknowledged hardware design problem is need for improved abstraction and programmability.

Because languages express thought (Whorf[1956]), one important aspect of hardware design is HDL selection. There are two possible approaches to HDL improvement. One approach advocates designing a new HDL to replace Verilog. The proposed SLDL language (EETimes[1997b]) illustrates this approach. Another approach is to enhance Verilog HDL.[1] Verilog enhancement is currently in progress under auspices of IEEE P1364 Verilog 98 working group (need reference?). This paper argues that new Verilog 98 PLI 2.0 API and HDL solve current high level design problems. No new HDL is needed. The main section discusses various high level design and abstraction problems and shows how to use PLI 2.0 for their solution.

1.1 Computer Programming in Hardware Design

Programming languages such as C (Kernighan[1988]) have long been used for conceptual and architectural hardware design because of compiler output execution efficiency and because of the superior expressibility of C style data structures. Hardware design using programming languages is usually accomplished either by adding a library or enhancing a programming language to implement scheduling, synchronization and parallel execution (see EETimes[1997a] for one project description). The Verilog PLI 2.0 API is superior to other programming language APIs for hardware

1. Another alternative might be to replace Verilog with the VHDL hardware description language (P1076LRM[1993]. That alternative is not discussed here because of shrinking market acceptance of VHDL (EETimes[1997b]) compared to Verilog.

description because: 1) it provides full access to Verilog scheduling and synchronization mechanism, 2) it allow mixing low level Verilog gate level models, RTL models, and high level C models, 3) it is backward compatible with design in Verilog by allowing enhancements to be added to traditional Verilog HDL and 4) PLI 2.0 is an open public standard.

There are two main rationales for new hardware design methods. The first argues that hardware verification using simulation needs to be replaced by some other verification method. Because simulation provides the only link from real circuit behavior to modeling, it is unlikely to disappear, and in any case PLI 2.0 API Verilog source scanning ability provides full programming language expressibility for implementation of any such new verification method. The second argues that Verilog (and for that matter all current HDLs) are deficient in expressibility. This papers shows that Verilog along with new PLI 2.0 API provides sufficient expressibility (capability for abstract expression) for all foreseeable hardware design problems.

It examines the three most common perceived current design expressibility problems and shows how to use Verilog in conjunction with PLI 2.0 API for their solution. The three are: 1) inability to express desired behavior, 2) inability to model deep submicron ASIC delays accurately, 3) inability to model high level function (utilize functional computer program models). Most discussions of Verilog modeling problems and IEEE P1364 Verilog 98 standard committee enhancement requests fall into one of these three areas. See Becker[1992], Becker[1996] and Hahn[1997] for discussions of hardware design problems using Verilog PLI.

There is a forth area of enhancement requests involving problems with data handling of design data inside computers that is not discussed here because the new Verilog 98 standard addresses such problems. The most important are addition of a standard generate feature to allow preprocessor generation of "arrayed" Verilog source that is standardized and part of Verilog itself and addition of a "uselib" capability to facilitate library component selection data processing. These enhancements work in conjunction with PLI 2.0 API model generation that is used during PLI 2.0 design verification.

2. Verilog PLI Evolution

PLI 2.0 vpi_ routine API is the culmination of a decade of Verilog PLI development (P134LRM[1996], ch. 17-23). First the tf_ (task/function) PLI API was developed to allow adding user coded Verilog system tasks and functions written in C. The tf_ interface was limited because design information could only be passed to and from C code by system task arguments. Only changes to tf_ PLI system task and function arguments could be monitored using inefficient value change flags (VCLs). Verilog models needed to be written with explicit knowledge of tf_ PLI system tasks and functions.

Second, in the early 1990s the acc_ (access) PLI API was defined to remedy problems with the tf_ interface. The acc_ PLI interface was designed to allow accessing Verilog source information from within PLI API. Designs could be scanned using acc_, **acc_get_next** source object grouping routines. Also, delays could be annotated and values could be set using object handles determined from source scanning. Acc_ interface was a significant improvement but was difficult to use. It grew over time with new features (routines or access paths) added haphazardly as needed. New routines were added with little concern for compatibility with other routines. The acc_ API provides hundreds of routines, access methods and C data structures (P1364LRM[1996], ch. 19). More design objects could be monitored (flag set on change) using the same inefficient VCL mechanism used by tf_ routines.

The new PLI 2.0 vpi_ routines API has been defined to simplify and improve Verilog PLI (P1364LRM[1996], ch. 22, 23, also Dawson[1996]). The API is called vpi_ because routine and constant names start with "vpi" prefix to distinguish names from tf_ and acc_ names and to allow mixing of various PLI interfaces. The PLI 2.0 vpi_ API replacement simplifies and improves tf_ and acc_ features. It was designed by analyzing complexities and inconsistencies in acc_ PLI API.

Until recently use of PLI 2.0 has been limited by lack of availability. Currently only Cadence Verilog XL and NC (Cadence[1997]) and my companies’ Cver (PragmaticC[1997]) simulators implement full PLI 2.0 interface. However, nearly all simulator vendors have announced plans to support PLI 2.0. The IEEE P1364 Verilog working group has decided that all future PLI enhancements will be made to PLI 2.0 only.

3. PLI 2.0 API Overview

Verilog PLI 2.0 defines a set of routines and data structures that interact with user applications. Communication between a Verilog simulator and user coded PLI 2.0 programming language routines is bidirectional. In one direction, user routines call PLI 2.0 routines to access and set simulation state values or to request services such as future call back routine registering or system I/O. In other direction, user Verilog PLI 2.0 API routines are invoked as PLI 2.0 user coded system tasks or functions or registered (added to an internal table and scheduled) and called when associated events occur. There are three types of call backs: 1) at specific time (alarm clock), 2) simulation stage completion (for example after design loading just before time 0 simulation) and on event such as variable or wire driver change. User routines execute, call other routines and schedule future activity by registering call backs. If user PLI routines do not explicitly schedule future activity, PLI activity ceases unless Verilog source contains explicit PLI user system task or function invocations.

3.1 Building Enhanced PLI Binaries

Verilog’s normal prepare source, prepare memory data, execute simulator verification process requires an added operating system link step to create PLI extended simulator binaries. A Verilog simulator in object form is linked with all user PLI 2.0 routines to produce PLI enhanced executable. Any number of unrelated PLI 2.0 models can be combined by linking object (.o) files from each into final PLI simulator binary. Because PLI 2.0 uses abstract handles, there will be no conflict because each PLI model gets and stores it own handles (references to simulation objects) and registers its own call backs. For example, it is possible to run two different wave form viewers during one simulation. Each wave viewer opens its own windows and adds its own monitoring call backs. Such as an approach would not be efficient because each value change would require that two different user routines (one for each wave viewer) be called, but it would work. It is also possible to link in any API library such as a library of calls to evaluate VSI standard models (need ref.) or a mathematical library to compute values for comparison with hardware computed results.

In Cver, one would use the following make file dependency line (Peek[1993]) to create an enhanced simulator that contains the other side of an async protocol (probably to test a protocol chip):

```
asyncsim:    async.o veriuser.o
    $(CC) $(LFLAGS) cverobj.o async.o veriuser.o \\
    $(LIBS) -o asyncsim
```

The user PLI 2.0 routines invoked by asyncsim (name of new Cver binary) are coded in the async.o file compiled from an async.c file. LIBS is assigned all libraries async.o file requires such as -lm for math library or -ldl if dynamic linking is used. File veriuser.o provides a dummy PLI 2.0 stub. Both PLI 1.0 and PLI 2.0 models can be combined in one simulation for backward compatibility. However because asyncsim only uses PLI 2.0, dummy veriuser.o stub is required.

3.2 PLI 2.0 Modeling Outside Verilog Source

PLI 2.0 API can be used without changing Verilog source. PLI 2.0 allows scanning internal design representation, finding objects of interest, and setting call backs on them. Normal model construction involves first building and testing Verilog modules separately then in final system simulation combining all Verilog models and linking in all PLI 2.0 C routines. The entire process is started by coding a table of functions in an extern pointer to function table defined within Verilog simulator (declaration is "extern void (*vlog_startup_routines[]) ();". One of the linked C (or C++) [2] files (usually vpi_user.c by convention) must define a startup routine for each included PLI 2.0 model. The following function pointer table might be used for asyncsim model:

```
void (*vlog_startup_routines[]) () =
 {
     register_async_systfs,
     register_async_cbs,
     register_wave_viewer_callbacks,
     setup_OS_pipes,
     0
  };
```

Routines that implement user PLI 2.0 models can appear in any order. They are automatically invoked during Verilog source translation.

2. Because C++ (Strousoup[1987]) can be linked with C programs (Kernighan[1988]), in this paper examples and references will use C. It is possible to also use C++ or other language that can be linked with C. Choice of C or C++ depends upon one’s preference for routine call or class invocation calling mechanism.

Normally these routines just register a cbStartOfSim call back at which time actual setup occurs. This is needed for setup routines to be able to access simulation variables and translated (loaded) Verilog source. The "register_OS_pipes" startup routine defines a PLI model to support Unix style pipe opening and closing that happens to be required by the C coded async model.

3.3 PLI 2.0 Routines

Because PLI 2.0 operates on self identifying HDL objects, only eight routines in three operation type classes are needed (additional utility routines are also defined) (P1364LRM[1996], ch. 23).

3.3.1 Source Object Access Routines

Verilog source connectivity is traversed using **vpi_handle** for one to one access methods (such as vpiHighConn to access instance connection to a port) or **vpi_iterate** to build a collection of object using a one to many access method (such as vpiOperand to access operands of an expression operator) (P1364LRM[1996], ch. 22).

3.3.2 Scheduling Routines

Routine **vpi_register_cb** is used to register call backs (P1364LRM[1996], pp. 584-588). Call backs can cause wake up at a future time, can cause C routine call on value change or can cause routine call when a simulation stage is reached (such as vpiEndofSimulation to allow clean up when simulation finished). Routine **vpi_register_systf** to register user coded PLI system tasks and functions (must start with dollar sign) that are invoked from Verilog source. Here Verilog source is coded assuming a particular PLI 2.0 programming language model will be linked to produce PLI enhanced simulator binary.

3.3.3 Value Access or Change Routines

Routines **vpi_get_value** and **vpi_put_value** are used to access and set values respectively. Values can be set immediately (vpiNoDelay mode flag) or scheduled (using flag such as vpiInertialDelay scheduling mode). Assignment to registers changes old value using procedural assignment semantics. For wires, assignment is a two step process. First a connector (vpiConnector object) is added to a wire or a bit of a wire and then the vpiConnector object is passed to **vpi_put_value** routine. Because wires can have multiple drivers, **vpi_put_value** to wires follows continuous assignment semantics. The added vpi_ driver is resolved against other wire drivers to determine net value. **vpi_get_value** can read pre and post delay driving values. Also multiple connectors can be added so, for example, ten different PLI 2.0 models can drive the same bus. The models must, of course, implement some bus contention protocol. **vpi_get_delays** and **vpi_put_delays** respectively read and set delays.

4. Enhancing Verilog Using PLI 2.0

Any HDL used for as wide a variety of different applications as Verilog will necessary require enhancement and change. This section discusses some common perceived problems with Verilog and shows how PLI 2.0 might be used to enhance Verilog HDL to solve them. Because PLI 2.0 API provides a high level interface based on a general purpose programming language, PLI 2.0 provides a meta solution to Verilog enhancement.[3] A major advantage of PLI 2.0 API feature addition is that it solves the feature generality problem. For example, many users believe Verilog HDL timing checks do not provide sufficient functionality. As a result, new timing checks are proposed for Verilog 98, but because each semiconductor vendor uses different manufacturing and modeling techniques, there is no consensus on new timing check functionality. The proposed solution for Verilog 98 adds many new timing checks with many additional flag arguments to select functionality. PLI 2.0 coded timing checks can be vendor specific (see 4.2 below).

4.1 Adding Direct Memory Bit Access

One of the most common Verilog HDL enhancement requests is addition of direct memory bit access. Below is a PLI 2.0 user coded system function to access a bit of a vectored memory directory. It is not quite as esthetic as double selection operator but it is just as usable. PLI 2.0 user coded system task is registered using **vpi_register_systf** routine that registers this C routine:

3. Since the original paper (Earley[1972]) on high level programming, high level programming has meant iterators and generalized object (handle) access methods both of which are central to PLI 2.0 API. Also at least since 1990, high level electronic design has meant design using computer programming (Wilkes[1990]).

```
 1  int bsel_from_mem(void) {
 2   int memi, biti, scalval;
 3   vpiHandle systfref, iter, memref, memiref;
 4   vpiHandle bitiref, memcell;
 5   struct t_vpi_value valrec;
 6
 7   systfref = vpi_handle(vpiSysTfCall, NULL);
 8   if ((iter = vpi_iterate(vpiArgument, systfref))
 9     == NULL) goto error;
10   memref = vpi_scan(iter);
11   memiref = vpi_scan(iter);
12   bitiref = vpi_scan(iter);
13   valrec.format = vpiIntVal;
14   vpi_get_value(memiref, &valrec);
15
16   memi = valrec.value.integer;
17   vpi_get_value(bitiref, &valrec);
18   biti = valrec.value.integer;
19   memcell = vpi_handle_by_index(memref,
20     memi);
21   valrec.format = vpiBinStrVal;
22   vpi_get_value(memcell, &valrec);
23
24   if ((scalval = get_bitstr_ndx(valrec.value.str
25     memcell, biti)) == -1) goto error;
26   valrec.format = vpiScalarVal;
27   valrec.value.scalar = scalval;
28   vpi_put_value(systfref, &valrec, NULL,
29    vpiNoDelay);
30   return(0);
31  error:  ... error handling code goes here ...
32  }
```

This example shows some PLI 2.0 idioms. It uses standard PLI 2.0 vpi_user.h include file (P1364LRM[1996], Annex E, pp. 622-634). Line 7 accesses calling instance. Passed arguments are then accessed 10, 11 and 12 by scanning iterator built on line 8. **vpi_get_value** on lines 14 and 17 access first memory cell index and then bit within cell index. Line 19 accesses the selected cell using **vpi_handle_by_index** utility routine. Entire memory value is accessed on line 22 then get_bitstr_ndx routine on line 24 written by user accesses scalar bit value from passed object (here memcell) and object value in vpiBinStrVal format. It uses the passed index to select needed bit from the passed value and build a s_vpi_value record using vpiScalarVal format. Finally **vpi_put_value** on line 28 sets system function return value (assign to handle systfref).

This 32 line routine provides a system task for bit selecting from memories. In Verilog source one might write:

```
if ($membsel(ram1, base1 + celli, bitk) == 1) ...
```

4.2 And Gate Model with PLI 2.0 Program Replacing Timing Checks

As process line widths narrow, deep submicron modeling has become more difficult. Numerous specify section enhancements have been proposed because current Verilog ASIC modeling features do not allow sufficient accuracy and time checking. Because each vendor uses different modeling conventions, general Verilog features do not match any vendor needs exactly. PLI 2.0 can be used for general cell modeling and time checking.

One important deep submicron problem involves cell element dependencies such as related input transition cross talk or relative switching time dependencies between ports. PLI 2.0 is used to record changes and implement arbitrarily complex switching behavior. PLI functional modeling works by adding new PLI drivers to cell outputs and adding vpiChangeValue call backs to monitor cell inputs. The call backs are added usually during cbStartOfSim processing. To illustrate the method, code to mimic a complicated "and" gate model is given below. First call backs are placed on both inputs for each instance and a new PLI 2.0 driver is added to the output wire or output wire vector bit (setup is C code is not shown). Call back user_data field is used to point to the value and time change history for each instance. The state recording structure might be:

```
struct achgrec_t {
 unsigned i1val : 8;   /* input 1 value */
 unsigned i2val : 8;   /* input 2 value */
 double lasti1chg;     /* input 1 last change time */
 double lasti2chg;     /* input 2 last change time */
 double and_del;       /* delay for this instance */
 vpiHandle outdrv_ref; /* added vpi_ out driver */
};
```

The routine that processes each input change determines which input changed (built into register call back) and then calls a general cell evaluation routine. There will be only one routine for all instances of the cell. Call back record user_data field is used to pass per instance data. State update code might be:

```
 1  tmptim.type = vpiScaledRealTime;
 2  vpi_get_time(cbp->obj, &tmptim);
 3  achgp = (struct achgrec_t *) cbp->user_data;
 4  if (chgi == 1)
 5   {
 6    achgp->lasti1chg = tmptim.real;
 7    lastinv1 = cbp->value->value.scalar;
 8    lastinv2 = achgp->i2val;
 9    chg_delta = tmptim.real - achgp->lasti2chg;
10    }
11   else
12    {
13     achgp->lasti2chg = tmptim.real;
14     lastinv1 = achgp->i1val;
15     lastinv2 = cbp->value->value.scalar;
16     chg_delta = tmptim.real - achgp->lasti1chg;
17    }
```

Line 2 accesses current time for storing change time stamps. Line 4 uses passed changed input pin number to determine state to update. On line 9 and 16 the delta change difference between the two input pins is computed. Then later in the same call back routine, the following code might be used to compute and schedule output change:

```
tptr = localtime(time(&now));
/* if today is Monday - gate is unknown <g> */
if (tpr->tm_wday == 1)
 {
  outval.format = vpiScalarVal;
  outval.value.scalar = vpiX;
  vpi_put_value(achgp->outdrv_ref, &outv,
   NULL, vpiNoDelay);
 }
 else
  {
   /* if ins changed within 10 ticks no change */
   if (chg_delta > 10.0)
    {
     if (lastinv1 == vpi1) outv = lastinv2;
     else if (lastinv1 == vpi0) outv = vpi0;
     else
      {
       if (lastinv2 == vpi0) outv = vpi0;
       else outv = vpiZ;
      }
     tmptim.type = vpiScaledRealTime;
     tmptim.real = achgp->and_del;
     outval.format = vpiScalarVal;
     outval.value.scalar = outv;
     vpi_put_value(outdrvh, &outval, &tmptim,
      vpiInertialDelay);
    }
  }
```

Notice the extremely complicated gate evaluation algorithm. It could be more complicated and even setup and execute a spice (Nagel[1972]) simulation although such a model would simulate slowly. Simulation time and memory usage may seem large at first glance for PLI 2.0 models but when a larger cell is modeled, storage will probably be less than the amount needed by numerous timing checks and path delays required by a Verilog source model. Verilog source models also require per instance state stored in achgrec_t record in the PLI 2.0 model. There will always be some additional procedure call overhead in PLI 2.0 modeling.

The above method offers a programmable solution to some of the most serious Verilog deep submicron problems. Negative timing checks (coded so reference event is later edge) are needed in Verilog. Because PLI 2.0 state recording (see above) allows total generality any state information can be recorded. In other words, any general algorithm can be implemented to check timing.

Another possibility allows using actual connected drivers to determine delays. PLI 2.0 input change call back routines can scan driving connectors both on cell inputs and outputs, calculate loading and transition times and recompute cell output delays. This allows more accurate delay modeling than is currently possible in Verilog because drivers that are tristated and therefore make smaller (or no) contribution to cell delay can be compensated for. This method would also be used for off chip connections where actual off chip power usage can be calculated for each I/O pad transition. Per transition power draw modeling replaces statistical power draw modeling, Additional accuracy is possible at cost of increased computation for each transition.

4.3 PLI 2.0 End of Cycle Change Only Models

Another common enhancement request is for Verilog HDL changes to allow cycle based modeling. Features are desired that reduce simulation accuracy in exchange for increased simulation speed (faster system cycle test rate). The problem with changes to Verilog source is that it may require two different Verilog models. One for accurate modeling and one for cycle based. By using PLI 2.0 to suppress cell output changes but instead sample output from a PLI 2.0 call back only at cycle ends, PLI 2.0 can be used to speed up simulation. The PLI 2.0 can be used to reduce event activity by working behind the scenes on an accurate delay Verilog source model assuming only system clock edge output values need to be propagated. Setup routines called from cbStartOfSim call backs first set cell output delays to infinite values. Time call backs using reason cbAfterDelay are registered to read out and drive cell outputs (either change blocked internal wire or using added outside cell driver) only on system clock edges. Each cycle edge call back would register a delay call back for next cycle edge. This can produce large simulation speed up by reducing number of simulated events.

4.4 PLI 2.0 for Non Simulation Circuit Verification

Because simulation is inherently computer resource intensive, there is a desire to verify circuits without simulation. Here PLI 2.0 provides an alternative to reading Verilog source. The PLI 2.0 standard allows source reconstruction using **vpi_handle** for one to one connections (such as assignment left hand side expression vpiLhs selector) and **vpi_iterate** for one to many connections (for example vpiNet selector for all nets in module). Alternative to PLI 2.0 source access would be for each tool to directory read source. The advantage of PLI 2.0 is that it is standardized. Also as Verilog changes, PLI access methods are automatically added reducing maintenance requirements, and standardized PLI 2.0 connectivity extraction will reproduce simulator library selection exactly.

4.5 Adding PLI 2.0 Timing Wrapper to Functional Model

It would be valuable to be able to just plug architectural models expressed as self contained computer programs into Verilog simulations. This enhancement request is usually phrased by asking "is there a program that will convert my C program into Verilog source?". PLI 2.0 provides a better and more general method for solving this problem. One especially important need allows a functional processor instruction set model to drive timing dependent glue logic simulation. Here an empty Verilog module is defined. Only I/O ports are declared. If no intra cycle timing detail is required, cbAfterDelay or maybe cbReadOnlySync (to make sure inputs for next cycle have changed) call back is scheduled at beginning of each cycle. Called PLI 2.0 routine first drives current cycle output values onto output ports (using vpi_put_value to added PLI drivers for wires or to regs if wire resolution is not required). Next cycle processor outputs are also computed and stored during call back.

If input ports must be read and output ports driven within a cycle according to processor timing data sheet, the added PLI call back processing routines must be more complicated. Easiest solution would schedule additional timer call backs during sections (cycle clock broken down into a more fine grained intra cycle clock). Evaluation (calling) of timing independent processor model occurs just after call back where input could still change and outputs would be driven in different call backs depending on timing specification. A more complicated and more accurate PLI 2.0 wrapper around a processor model would require attaching cbValueChange call backs on input ports and scheduling output port changes from those value changes.

A programmer needs to write PLI 2.0 wrapper code, but understanding and modeling embedded processor timing seems an important part of system verification. Also, adding a synchronous PLI 2.0 interface is quite easy probably requiring only a few hundred lines of code. If more modeling effort is possible, timing could be added to processor instruction set model by rewriting internals of processor model using PLI 2.0 call backs and simulation value assignments.

4.6 PLI 2.0 Controlled Co-Simulation

Because of proliferation of computer networks, there is desire to enhance Verilog to allow parallel simulation on multiple computers and co-simulation so that parts of models can be simulated on other computers. Both simulations must be synchronized and common ports must be connected. Signals values must be sent back and forth across co-simulation interface. If one makes the simplifying assumption that Verilog because of its universality controls a co-simulation, PLI 2.0 can be used to execute, read and write to another simulation either running as a parallel process or on another computer. PLI 2.0 Synchronous co-simulation works by setting cbAfterDelay call backs at cycle boundaries. The call back routine then can use any C Language or operating system calls to read values from co-simulation, set needed Verilog side regs and wires and write output values to co-simulation engine for its processing. Operating system level synchronization would require that Verilog controlling simulation would need to wait for co-simulation operating system events (and vice versa) so it may not be more efficient than all

Verilog simulation, but it would allow easier modeling.

A better type of co-simulation would allow co-simulation asynchronous events to cause a PLI 2.0 call back so both simulations could be arranged to continue (PLI 2.0 program would need to be written to handle scheduling). PLI 2.0 currently does not support asynchronous PLI 2.0 interrupt call backs because controlling Verilog simulator would need to field interrupts and make call backs once a known safe simulation time state were entered. Such a call back type is planned for Verilog 98.

5. Acknowledgements

This paper benefitted from discussions with other members of the P1364 Verilog 98 PLI task force. Also Mr. Kikutani Makoto's improvements to some of my PLI 2.0 models is reflected in this paper.

6. References

Becker[1992] Becker, D., Singh, R., and Tell S. Software/hardware co-simulation with Verilog and C++. Proceedings 1st International Verilog HDL Conference, 1992, 33-37.

Becker[1996] Becker, M. Faster Verilog simulation using a cycle based programming methodology. Software/Hardware Co-Simulation with Verilog and C++. Proceedings 1st International Verilog HDL Conference, 1996, 24-31.

Cadence[1997] Verilog XL and Verilog NC reference manuals. Cadence Design Systems, Inc., 1997.

Dawson[1996] Dawson, C., Pattanam, S., and Roberts, K. The Verilog procedural interface for Verilog hardware description language. Proceedings 5th International Verilog HDL Conference - IVC'96, 1996, 17-23.

Earley[1975] Earley, J. High level iterators and a method for automatically designing data structure representation. *J. Computer Languages*, vol. 1 (1975), 321-342.

EETimes[1997a] Goering, R. 'Cycle-accurate' model built. *Electronics Engineering times.*, October 13, 1997, 80.

EETimes[1997b] Goering, R. VHDL faces shaky system-level future. *Electronics Engineering times.* October 27, 1997, 1, 134.

Hahn[1997] Hahn, D., and Russack, J. Implementation of a PCI Bus Virtual Driver Using PLI, Named Pipes, and Signals. Proceedings 6th International Verilog HDL Conference - IVC'97, 1997, 10-13.

Kernighan[1988] Kernighan, B., and Ritchie, D. *The C programming Language.* 2nd edition, Prentic Hall, 1988.

Nagel[1975] Nagel, L. SPICE2: a computer program to simulate semiconductor circuits. ERL Memo ERL-520, University of California, Berkeley, May 1975.

Peek[1993] Peek, J., O'Reilly, T., and Loukides, M. *Unix Power Tools.* O'Reilly & Associates, 1993, 519-521.

PragmaticC[1997] Cver Verilog simulator reference manual. Pragmatic C Software Corp., 1997.

P1076LRM[1993] IEEE Standards Board. IEEE Std 1076-1993 VHDL Language Reference Manual. IEEE: New York, 1993.

P1364LRM[1996] IEEE Standards Board. IEEE Std 1364-1995 Verilog Hardware Description Language Reference Manual. IEEE: New York, 1996.

Stroustrup[1987] Stroustrup, B. *The C++ programming Language.* Addison Wesley, 1987.

Wilkes[1990] Wilkes, M. It's all software, now. *CACM.* 33, 10 (October 1990) 19-21.

Whorf[1956] Whorf, B. *Language, Thought and Reality.* MIT Press, 1956.