Verilog HDL, Powered by PLI: a Suitable Framework for Describing and Modeling Asynchronous Circuits at All Levels of Abstraction

Arash Saifhashemi
Department of Computer Eng.
Amirkabir University of Technology
424, Hafez Ave.
Tehran 15914, Iran
saif@ce.aut.ac.ir

Hossein Pedram
Department of Computer Eng.
Amirkabir University of Technology
424, Hafez Ave.
Tehran 15914, Iran
pedram@ce.aut.ac.ir

ABSTRACT
In this paper, we show how to use Verilog HDL along with PLI (Programming Language Interface) to model asynchronous circuits at the behavioral level by implementing CSP (Communicating Sequential Processes) language constructs. Channels and communicating actions are modeled in Verilog HDL as abstract actions.

Categories and Subject Descriptors
B.6.1 [**Logic Design**]: Design Styles–*Asynchronous circuits, Parallel circuits;* D.1.3 [**Programming Techniques**]: Concurrent Programming–*CSP;* B.6.3 [**Logic Design**]: Design Aids–*Hardware description languages, Simulation, Verilog, PLI.*

General Terms Design, Languages.

Keywords
Asynchronous circuits, CSP, CHP, Verilog, PLI, Channel

1. INTRODUCTION
Today, many asynchronous circuit design flows use CSP-derived languages, originally developed by Hoare [2], to model asynchronous circuits at the behavioral level.

The main motivation for using CSP is its two specific features, which standard HDLs have been said to lack. First, using ports and channels, CSP has made communication actions between two processes abstract actions. Secondly, in CSP language one can nest concurrent blocks within sequential blocks and vice versa without any limitation on the statements within them or the nesting level. This feature is so called fine-grained concurrency.

On the other hand, there are some problems with CSP. First, it has not been formally standardized. Hence, code exchanging is difficult. Second, it has little thing, if anything at all, to do with the lower levels of the design. Thus, in contrast to synchronous design flows, one cannot describe their circuits at different levels using a single language and platform. In fact, most design flows that use CSP in the behavioral level use Verilog or VHDL at lower levels. Besides, they can hardly use a single test bench at all levels of the design.

Several CSP modeling tools have been developed until now, which can be classified into two groups:

1. Developers of the first group have invented their ad-hoc languages, derived from CSP together with a simulator for them. LARD [5] is an example.
2. The second group wished to use standard HDLs and strengthen them to support CSP features. In this way, they would be able to use commercial simulators [3, 4].

Although the first group has developed practical tools and complex circuits, their tools have some shortcomings:

1. Since they have used CSP-derived languages, they share the same problems of CSP.
2. Comparing to commercial synchronous modeling tools, they are limited.
3. In many cases, they are not available to the public.

It can be claimed that if it becomes possible to model all CSP constructs in a standard HDL at an abstraction level equal to CSP, there would be no need to invent another ad-hoc language and develop its simulator. Additionally, in contrast to CSP, standard HDLs like Verilog and VHDL can describe circuits at lower levels very well. Thus, they can potentially be ideal for describing asynchronous circuits at all levels of abstraction.

Although Verilog HDL is popular among synchronous designers, most people who have developed a solution for simulating CSP have used or compared their solution with VHDL [3, 4, 5].

This paper mainly deals with channels and communication actions, i.e. we try to make channel communications abstract actions in Verilog HDL. Therefore, together with the fine-grained concurrency feature of Verilog, which VHDL lacks, most important features of CSP would be addressed.

The rest of this article is organized as follows: Section 2 describes how to use Verilog along with PLI to implement a channel as an abstract construct. Section 3 includes some other applications of PLI. Section 4 concludes the paper.

2. ABSTRACTING CHANNELS
Several implementations have been suggested for implementing CSP communication actions. Four-phase implementation [1] has been used most until now. The following code shows such an implementation for a WRITE action in Verilog:

DAC 2003, June 2-6, 2003, Anaheim, CA, USA.

```
module p (out, req, ack);
   input   req;
   output  out, ack;
   reg     data;
   always
   begin
      //Produce data;
      wait (req == 1); ack = 1; out = data;
      wait (req == 0); ack = 0;
   end
endmodule
```

Here, the designer has to specify the handshaking protocol together with handshaking signals at the definition part of the module. Hence, we can formally define our goals as follows:
- *Omit the need of mentioning and defining the handshaking signals from the definition part of the module.*
- *Hide the handshaking actions from the users notice.*

The ideal code would be like this:

```
module p (out);
   output out;
   reg  data;
   //Produce data;
   //out!data; ??
endmodule
```

Reaching these two goals is not possible through pure Verilog.

2.1 Programming Language Interface

Detailed description of the PLI is beyond the scope of this paper. However, in short terms, PLI is a procedural interface, which provides the ability to call precompiled C routines within the Verilog code. Thus, most of the C language features can be used in Verilog. As an example, consider this simple Verilog code:

```
for (i = 1; i <= 20; i = i + 1)
 @(posedge clk)
   $rnd_data (x_bus);
```

As can be seen, *rnd_data* is called to put random data on *x_bus*. Its body was written in C and compiled in advance. Simulators link this routine to the Verilog body at elaboration time.

Many library routines are provided to support interfacing the C language and Verilog. For example, for *x_bus* in PLI space, several features such as its length, the net to which it is connected, its type, its current value, etc. can be found. In addition, its value can be changed by the PLI routine.

A PLI feature that we mostly take advantage of is shared memory (global variables). That is, several PLI functions can access and change the value of a global variable.

2.2 A solution to communication actions

A possible way to hide handshaking variables is to place them in the global memory of PLI space instead of the Verilog body.

Two processes on a channel can communicate by calling PLI routines. Through these routines, they change and read values of some shared variables.

Instead of handshaking variables in Verilog body, we can define a structure for a channel in PLI space shared memory as follows:

```
struct t_channel{
   BOOL    bWriteDone, bReadDone;
   char    *buffer;
   handle  hSimulationNet; }
```

Each two req and ack signals are tied together to form a single variable: bWriteDone and bReadDone. Additionally, we have a handle, which functions like a pointer in C, to the net on which writer's and reader's ports are placed. Thus, by calling $Write(p,data) in one process and $Read(c,data) in another, we can check if ports p and c are on the same net, and hence have formed a channel.

Now, consider the example of a producer and a consumer module. One of them continuously produces data, while the other consumes that data. They communicate on channel (in, out).

At the first glance, it seems that we should implement the following algorithm:

	Producer	Consumer

Verilog	`module p(out);`
	`always`
	`begin`
	`//Produce data`
	`$Write(out,data);`
	`end`
	`endmodule`
		`module c(in);`
		`always`
		`begin`
		`$Read(in,data);`
		`//Consume data`
		`end`
		`endmodule`

PLI	`Write_Calltf(){`
	`buffer=data;`
	`bWriteDone=1;`
	`while(!bReadDone);`
	`bWriteDone=0;`
	`while(bReadDone);`
	`}`
		`Read_Calltf(){`
		`while(!bWriteDone);`
		`data=buffer;`
		`bReadDone=1;`
		`while(bWriteDone);`
		`bReadDone=0;`
		`}`

Unfortunately, this code does not work because while a PLI routine is executed, the simulator is blocked and cannot simulate other processes. So, when one process is blocked in a while loop within the PLI system call, the whole simulation will stop since in PLI routines the inherent Verilog concurrency is lost. Generally, it is not possible to have wait actions in a PLI routine.

One possible solution to this problem is to push wait actions back to Verilog body and leave the remained job to PLI. In this way, PLI routines do not block the simulation anymore. In other words, we let PLI just do those tasks that would not be blocked and can be executed in zero time. Instead, wait actions are moved to Verilog body. To do so, we have to make some alterations in our channel structure. The new one can be as follows:

```
struct t_channel{
BOOL       bWriteDone, bIsReadRequest;
char       *buffer;
handle     hReadDone, hReadClear,
           hSimulationNet;}
```

Note that here we changed the type of bReadDone signal to a handle (hReadDone) because we want to have a wait action on this signal in Verilog body. We define this variable in our Verilog module, but we store its handle in the PLI global memory. Through it, the other process can change the value of the actual signal. Also, we added a handle called hReadClear, on which, in consumer process we can have a wait action.

Now we can present the method as follows:

```
module p(out);
   ...
   reg bReadDone;
always
begin
   //Produce data
   bReadDone=1'b0; $Write(out,data);
   $RegisterReadDoneFlag(out,bReadDone);
   wait(bReadDone==1'b1);
   $ResetWriteRequest(out);
end
endmodule
```

Below, the pseudo code description of each PLI function is given.

```
Write(out,data){
   buffer = data; bWriteDone = TRUE;
   if (bIsReadRequest)
      change the value of bReadClear(using
      hReadClear) to TRUE;}


RegisterReadDoneFlag(out,bReadDone){
   hReadDone = handle(bReadDone)}

ResetWriteRequest(out){
      bWriteDone=FALSE;}
```

Obviously, none of the above functions blocks the simulation. The consumer process is as follows:

```
module c(in);
...
   reg bReadClear;
always
begin
   bReadClear=1'b0;
   $RegisterReaderFlag(in,bReadClear);
   wait(bReadClear==1);
   $Read (in,data); $ResetReadRequest(in);
   //Consume data;
end
endmodule
```

The new PLI functions are described below using pseudo code:

```
RegisterReaderFlag(in,bReadClear){
   if(bWriteDone==TRUE)
      Change bReadClear to TRUE;
   else{
      store the bReadClear handle in
      hReadClear;
      bIsReadRequest = TRUE;}
}


Read(in,data){
   data = buffer;
   Change the value of bReadDone(using
   hReadDone) to TRUE;}


ResetReadRequest(in){
      bIsReadRequest=FALSE;}
```

We introduced two registers in Verilog body, bReadDone and bReadClear. Notice that their handles are stored in PLI body.

What happens is simple: the producer module writes data into the buffer, then stores the handle of bReadDone signal in the shared memory of the PLI (by calling RegisterReadDoneFlag). Later, the consumer will use this handle to unblock the producer process. Therefore, the producer can wait on the signal. Since this wait does not block the simulation, the simulation can go on.

On the other side, the reader first resets the value of bReadClear. Then, it stores its handle into the shared memory of the PLI interface. Later, using its handle the writer would set this flag. Next, the consumer waits on that signal to become TRUE.

Observe that if RegisterReaderFlag function is called sooner than Write, Write function changes the value of bReadClear. However, if Write function is called before RegisterReaderFlag, RegisterReaderFlag changes the value of bReadClear;

Whoever starts first, will wait for the other to finish, and both communication actions will finish in parallel.

Next, a routine for implementing probes is presented:

```
Probe_Calltf(port){
if(m_bIsWriteDone || m_bIsReadRequest)
   return TRUE;
else return FALSE;}
```

This routine can be used in Verilog body as follows:

```
if ($Probe(prt) ) …
```

Next, to make the communication action abstract, we define them as macros.

The final code for producer can be considered like this:

```
`define USES_CHANNEL reg bReadDone;\
                     reg bReadClear;


`define WRITE(prt,d) begin\
   bReadDone=1'b0; $Write(prt,d);\
   $RegisterReadDoneFlag(prt,bReadDone);\
   wait(bReadDone==1'b1);
   $ResetWriteRequest(prt);\
   end


module p(prt);
   `USES_CHANNEL
always
begin
   //Produce data
   `WRITE(out,data)
end
endmodule
```

The same thing can be done for the consumer module. Notice that a top module should connect *in* and *out* ports on a single net.

The above code is at the same level of abstraction as it is in CSP.

It is worth mentioning that one can write similar PLI routines to implement communication actions using other handshaking protocols. The only change would be to include the file containing new macros.

Finally, it is also possible to build a synthesis tool based on this method just like any other synthesis tool that synthesizes a CSP description. For example, a simple tool can replace the macros with their equivalent handshaking protocol actions. Some synthesis directives can be used for specifying the protocol for each communication action, and the ports' kinds.

2.3 Generalize the method to handle more than one channel

The presented method can be generalized to enable managing more channels. The approach that we used is this: there is a list of channels in the PLI body. Each member of the list is of channel structure type. When a PLI routine is called by a communication action on a port, if the handle of the net on which the port is placed is not in the list (this handle is stored in hSimulationNet field of the channel struct), the PLI routine adds a new member to the list, and fills the new hSimulationNet field.

On the other hand, if the handle of the net was previously placed in the list, the communication action will be done as before. Observe that in this way if we generalize ports for more than one process, like when we need to have a shared bus, as far as all the ports are on a single net, the method works and there is no need of any further change.

3. OTHER APPLICATIONS OF PLI

Here are some other usages of PLI in asynchronous design:

1. **Pure handshaking:** In [1], a type of communication action is defined for synchronization between two processes. This can be implemented by a new macro.
2. **Arbitration:** Although not as abstract as it is in CSP, here is an example a non-deterministic choice in Verilog:

   Arb =*[[$\overline{A}$ -> A!x [] $\overline{B}$ -> B!x]]

   In the above CSP code, A and B ports are probed and the one which returns true is selected. However, if both return true, the functionality is not deterministic and arbitration should be done. One form of equivalent Verilog code can be as follows:

```
arbNumber = $Arbitrate(A,B)
if (arbNumber == 1)   `WRITE(A, x)
if (arbNumber == 2)   `WRITE(B, x)
```

   The PLI function *Arbitrate* can probe both A and B. If both return true, it can arbitrarily select one.
3. **Statistical measurements:** For example, after a slight change in PLI routines of communication actions, one can make them count each communication action. Later, this number can be used as an approximate assessment of the circuit's power consumption.
4. **Bullet operator:** in [1] bullet operator is defined to relate two communication actions in a way that they finish together. To implement, one can write a new macro, e.g., A!x●B!y can be implemented by `WRITE_WRITE(p1, v1, p2, v2), where A, x, B, y are the parameters respectively. Any handshaking protocol, but now for two interleaved actions, can be used to make this macro work. This method and the second one are similar to the ones in [3].
5. **Using a single test bench (mixed mode simulation):** A simple interface module can be added to the test bench to convert the abstract communication actions to the implemented handshaking protocol to let mixed mode simulation possible.

4. Conclusions

In conclusion, Verilog HDL together with PLI some routines can be considered as a perfect alternative for asynchronous designers because it is a standard HDL and supports lower levels of the design, hence, mixed mode simulation would be possible. Besides, many competent simulators are available for it. In contrast to VHDL, it supports fine-grained concurrency. This kind of modeling is not restricted to a single asynchronous design flow and can be used in any asynchronous design flow that a CSP-derived language is used.

5. REFERENCES

[1] Alain J. Martin. Synthesis of Asynchronous VLSI Circuits. Internal Report. Caltech-CS-TR-93-28. California Institute of Technology, Pasadena, CA. 1993

[2] C.A.R. Hoare. Communicating Sequential Processes. CACM 21, 8, pp 666-677, 1978

[3] Chris J. Myers. Asynchronous Circuit Design. John Wiley & Sons, INC.

[4] Jens Sparsoe and Steve Furber. Principles of Asynchronous Circuit Design. Kluwer Academic Publishers.

[5] Philip Endecott and S. Furber. Modelling and Simulation of Asynchronous Systems using the LARD. http://www.cs.man.ac.uk/amulet/projects/lard