The CSI Journal on
Computer Science and Engineering
Vol. 1, No. 4 (b), Winter 2003
Pages 1-10
Reguler Paper

Using Standard HDLs and CAD Tools for the Design and Simulation of Asynchronous Circuits

Arash Saifhashemi [1] Mohsen Naderi [1] Hossein Pedram [1] Arash Farhoodfar [2]

[1]Computer Engineering Department, Amirkabir University of Technolosy, Tehran, Iran
[2]Applied Micro Circuit Corp, Ontario, Canada

Abstract

In this paper we will show that it is both possible and easy to use a standard HDL language like Verilog HDL, along with PLI to model asynchronous circuits at all levels of abstraction, including the behavioral level (CSP level). Our method allows CSP (Communicating Sequential Processes) codes to be simulated on ordinary Verilog simulators. The suggested algorithm is easy to implement (less than 100 lines of code in our case), thus other asynchronous designers not only can implement their own codes, but also they can further improve and optimize the method. In this way, one can customize the way that CSP features are added to Verilog HDL. Furthermore, it does not need any preprocessing or extra tool. The designer can write his code in Verilog language from the very beginning steps of the design, while channel communications become like atomic actions, and fine-grained concurrency within processes is available.
We believe that this method has the potential to override methods that involve CSP-like languages and new simulators for asynchronous circuits which are described by CSP-like languages, and enables designers to exchange their codes easily.

Keywords: asynchronous circuits, Verilog, modeling

1. Introduction

Several advantages and motivations for asynchronous circuits are described elsewhere, such as [1]. However, there are some problems that avoid asynchronous design to become popular. One of the most important problems is the lack of appropriate CAD tools.
Today, each methodology for asynchronous design uses its own language and design flow. However, most asynchronous circuit design flows use CSP-like languages, developed by Hoare [2], to model asynchronous circuits at the behavioral level. This language helps designers to describe circuits without regarding to handshake protocols and signals.
The main motivation for asynchronous designers to use this language is the existence of features which standard HDL languages like VHDL and Verilog HDL have seemed to lack. By introducing ports and channels, CSP has made the communication action between two processes become an abstract action. It means that when a designer wants a process to read from or write to another process, he doesn't need to specify the communication protocol and handshaking signals. Instead, he just issues read and write actions which are considered to be atomic.
Because of these great features of the CSP language, a CSP simulator is the first thing that most asynchronous designers wish to have. Unfortunately, however, despite the popularity of the CSP language, CSP simulators are not yet supported by commercial CAD-tool developers. In fact, almost none of the commercial simulators support these languages. This fact caused many designers to develop their own simulation, modeling, and synthesis tools. Thus, every new designer who wants to design an asynchronous circuit should first define his own CSP-like language and develop a simulator for it. This is why many universities and research groups began to develop new languages and modeling CAD-tools. As a result of lacking a standard, however, although they are common in the basics, none of these CAD-tools is compatible with the other one. Even the simulator tools, which should simulate the same concepts, are not compatible because the syntax of languages differs.

The rest of this article is organized as follows: Section 2 reviews currently available tools for asynchronous design and simulation. Section 3 describes the goals of designing a new approach. A brief description of the PLI is presented in section 4. Section 5 describes how it is possible to use Verilog to describe channels and communications asynchronous circuits. At section 6 we will show how we have implemented fine-grained concurrency by Verilog and Section 7 includes a real circuit description described in this way. Section 8 discusses about the advantages and limitations of this approach and presents how much we have reached our goals. In addition, we will make a comparison of our approach with other tools.

2. Available Asynchronous Tools

At the time that this paper is written, few CSP CAD-tools have been developed. In most cases, however, they are neither standard, nor available to the public.

One can classify these tools into two general groups:

1. The first group includes tools that were designed from scratch. A new language was derived from CSP, and a simulator was developed for that. CAST [3] [2], which is both a simulator and synthesis tool, and LARD [7], a simulator which is available to the public for free, are two examples.
2. In the second group standard HDL languages like VHDL were strengthened to support CSP language. VHDL++ [6], and CHP_2VHDL [8] are two examples of this approach. While developers of VHDL++ added some macros to VHDL to support channels and concurrency within processes, CHP_2VHDL developers decided to define a completely new language, whose syntax is like CSP. The code in that language is then converted to standard VHDL.

The common drawbacks of the two approaches are as follows:

1. They are not standard and they are hardly compatible. They also face portability problems.
2. They do not consider lower levels of the design. One of the best features of the standard HDL languages is their ability to describe a circuit in several levels of abstraction, from behavioral to the switch level, using a single simulator. For CSP CAD-tools, after the behavioral modeling and simulation, standard Verilog or VHDL is used for the rest of the design. For example, in LARD, modeling is done in LARD language, and the lower levels Verilog or VHDL is used.
3. These approaches are OS-dependent, and different compilers are required for different environments.
4. Support is limited. Since each of these tools is provided by a single provider, the users are strongly dependent on that provider.
5. Because these tools are not provided commercially, they have some limitations and inflexibilities.

It seems that the ideal situation is to have a standard CSP-like language which is accepted by all, and have some commercial CAD-tool providers provide the necessary tools. However, even by standardization of a CSP-like language still the problem will not be solved, unless this new standard language is able to describe the design at all levels of abstraction (like Verilog and VHDL). May be the better way is to add some standard features to Verilog and VHDL. Unfortunately, this may take a long time. So, what should be done until then? A new designer may become so confused by these non-standard asynchronous CAD-tools that he may decide to develop his own! Therefore, in future we may see that each design group has its own modeling tool

3. Goals of Developing a New Tool

Considering the above problems, we decided to choose a different approach. We have developed our own modeling method considering these goals:

1. To use standard HDL languages, Verilog HDL in our case, because they are supported by a wide range of CAD-tool providers. And these tools are available for many operating systems.
2. It should be possible to use the same CAD-tool at all levels of abstraction, i.e., the user should be able to use the same simulator from beginning to end.
3. Additions to the standard languages should be minimal so that any new user be able to learn it very fast without reading tens of pages of documentations.
4. Make the need for preprocessing as little as possible. We wanted the designer to debug the design using the same code that was written before.
5. Make the method become easily available to all, and be customizable. The user should be able to change the functions of the tool and maybe add something to it.

In the first approach described in section 2 the designers believed that standard HDL languages lack two CSP features, i.e. channels and fine grained concurrency. In LARD for example, it is said that in case of communications on channels in VHDL language, one has to describe the handshaking variables explicitly. Figure 1 shows an example that was brought to show this weakness. However, we will later show that it is possible to solve this problem in Verilog. Our equivalent suggested code is shown in the same figure.

VHDL	LARD	Verilog

data <= x;
req <= 1;
wait until ack=1;
req <= 0;
wait until ack=0;
	C!x	`Write(C,x)

wait until req=1;
y:=data;
ack <= 1;
wait until req=0;
ack <= 0
	C?(y:=?C)	`Read(C,x)

Figure 1. Explicit description of handshaking

In the following sections we will show that it is both possible and easy to use Verilog-HDL language enhanced by Programming Language Interface (PLI) routines for modeling the circuits at all levels of abstraction, including the CSP level. Implementing this approach is trivial, and designers can do the same by writing their own routines. Our case included less than 100 lines of C routines. Therefore, it is possible for designers to exchange their codes. At the

worst case, instead of the whole tool, they have to attach their PLI routines. We believe that in this way there would be no need for a new modeling language and modeling tool.

4. Programming Language Interface

Detailed description of the PLI is beyond the scope of this article. However, a brief description is presented. In [4] PLI is defined as follows: "a procedural interface, known as the Programming Language Interface, which provides a means for Verilog HDL users to access and modify data in an instantiated Verilog HDL data structure dynamically. An instantiated Verilog HDL data structure is the result of compiling Verilog HDL source descriptions and generating the hierarchy modeled by module instances, primitive instances, and other Verilog HDL constructs that represent scope. The PLI procedural interface provides a library of C-language functions that can directly access data within an instantiated Verilog HDL data structure."

Among some applications of PLI which are described in [4] the most important ones we used are:

1. Interfaces to actual hardware, such as a hardware modeler, that dynamically interact with simulations
2. Simulation models written in the C language and dynamically linked into Verilog HDL simulations
3. C-language delay calculators for Verilog modules
4. C-language applications that dynamically read test vectors or other data from a file and pass the data into a Verilog software product.

PLI is standardized now by IEEE and has been in use since the mid-1980s. However, at the time of this article there are three generations of PLI:

1. Task/function routines, (aka. TF routines) along with utility routines. In [4] these routines are classified into 12 groups, where most important ones are:
   a. Reading or modifying parameter value
   b. Detecting parameter value change
   c. Displaying messages
   d. Saving or restoring data from files

2. Access routines: these routines provide an object-oriented access into a Verilog HDL description. ACC routines are used to access and modify information, such as delay values and logic values on a wide variety of objects that exist in a Verilog HDL description. There is some overlap in functionality between access routines and TF routines. The most important groups of these functions are for [4]:
   a. Getting a handle to a given object
   b. Fetch a property of the object
   c. Modify the property, if required
   d. Monitor the object, if required
   e. Do some housekeeping jobs, such as printing messages etc.
   f. Move to the next object

3. Verilog Procedural Interface routines which are called VPI routine, or PLI 2. These routines provide an object-oriented access for both Verilog HDL structural and behavioral objects. The VPI routines are a superset of the functionality of the TF routines and ACC routines.

In fact, PLI routines are written in C language, and they are linked to the actual Verilog code. In the Verilog code, They can be called using a system call. Therefore, one can take advantage of what can be done in the C language while writing his code in Verilog HDL. Here is an example of using a PLI routine called *$get_vector* which reads a test vector from a file called test_vector.pat, and assigns the values to input bus.

```
for (i = 1; i <= 1024; i = i + 1)
  @(posedge clk)
  $get_vector("test_vector.pat", input_bus);
```

One of the most important advantages of this interface is its ability to store a variable in the shared memory of PLI interface from one routine, and later reuse it in another routine. On the other hand, the main drawback of PLI is that by using the PLI interface one loses the concurrency which is available in Verilog HDL since C routines are executed sequential, not in parallel. That is, when a PLI routine is called, the simulator is interrupted and does nothing until the call is ended.

We will show the usage of the shared memory, and will describe how we challenged the problem of losing concurrency while using PLI in the next section.

5. Modeling Channels and Communications

Surprisingly, although Verilog HDL is very powerful and it is older than VHDL, most people who have designed a new CSP-based language have compared it with VHDL, not Verilog [7], [8], [6].

In this section we will show how it is possible to use Verilog HDL to describe an asynchronous circuit in all levels of abstraction. We believe that by using Verilog there is no need for another new language.

In the first subsection we will show how we added communication actions to Verilog as an abstract concept. Next, we will describe how we implemented fine-grained concurrency within sequential processes. Finally, an example is given in which we will show how one can write a code at CSP level using our method.

5.1 Channels as an abstract concepts

As mentioned above, one of the most important features of the CSP language is channel. Using this concept, different processes can asynchronously communicate with each other. In addition, they can synchronize each other. Channels do not have buffers [2].

In [8] it is said that Read and Write actions should correspond (or match) with each other, i.e. they should execute simultaneously. If one of the processes is not ready, the other one should be delayed until the first process becomes ready.

Therefore, we can define read and write actions in the following fashion:

The write Algorithm:

4. Wait until the other process is ready.
5. Place data into the channel.
6. Wait until the other process finishes reading.

The read Algorithm:

7. Wait until the other process is ready.
8. Take data from the channel.
9. Wait until the other process finishes writing

Several implementations have been suggested for such an algorithm, including Two and Four-phase hand shakings [2]. The code in figure 2 shows an example of four-phase

handshaking using four extra signals. It shows an implementation for a communication between the processes P and C through a channel. Each process has two local req and ack signals.

Producer (out) variable data	Consumer(in) variable data
channel (out, in)	
[req]; ack↑; out = data; [~req]; ack↓;	req↑; [ack]; data = in; req↓; [~ack];

Figure 2. Communication between processes P and C

In this algorithm we say that P is passive and C is active since the starter is C. However, if we consider an algorithm where P is active and C is passive, passiveness and activeness is specified at the CSP level, that is not desirable. At least, we should make it optional for the designer.
To the extents that we know, this form of implementation has been the best and most popular one until now. However, it cannot be done in the Verilog language without explicitly adding extra signals to the design. The following code shows such a try.

```
module p (out, req, ack);
input       req;
output out, ack;
reg         data;
wait (req==1);
ack = 1;
out = data;
wait (req == 0);
ack = 0;
endmodule
```

In this piece of code, the designer not only has to specify the activeness and passiveness, but also he has to specify the signaling ports at the definition of the module. Therefore, we can formally say that we should follow two goals:

10. To omit the need of mentioning and defining the signaling nets from the definition of the module.
11. To hide the signaling actions.

Therefore, the ideal code would be something like this:

```
module p (out);
        output out;
        reg    data;
        //WRITE(out,data); ??

endmodule
```

Reaching the two goals is not possible through pure standard Verilog, however, PLI can help us to implement such a code.

5.2 The solution to communication actions

A possible solution avoid those signaling variables is to place them in the shared memory of the PLI interface (as global variables) instead of the main Verilog code. Therefore, two processes can communicate with each other through the PLI interface. Figure 3 shows the idea.

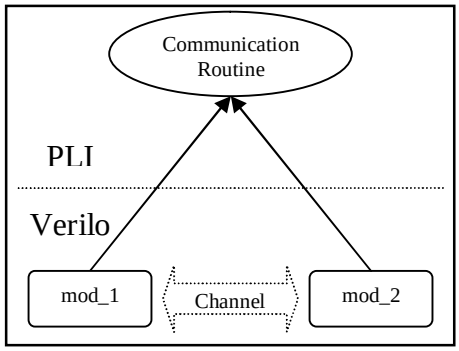

Figure 3. Communication through PLI

As can be seen, the two processes do not actually know each other. Instead, a PLI interface can do the communication action. A PLI routine can find the net (not the port) to which a given port is connected. Therefore, it can tie one module to another one to which its port is connected in order to form a channel.
Note that in Verilog, since we have two separate modules that do not know each other, we should have four different signals for handshaking. However, there are only two actual signals from the view of the rest of the world and the PLI interface. Hence, we do not need to have four different variables in the PLI interface. Instead, we define two variables. In addition, although it is said in [2] that channels do not have buffers in them, we allocate a buffer for the data that has to be exchanged. This is not a violation because it doesn't have anything to do with the synthesis and is just for modeling purpose.
Consequently, we should define a structure for each channel in the PLI interface as follows:

```
typedef struct t_channel{
        BOOL        bIsBufferFull;
        BOOL        bReadDone;
        char        *buffer;
        handle hReadPrt; //Stores the read port
        handle hWritePrt;//Stores the write port
}
SChannel;
```

As can be seen, each two req and ack signals are tied together and formed a single variable: bIsBufferFull and bIsReadRequest. Additionally, we have two fields that store handles to the writer's and reader's ports. This is necessary because once a PLI communication routine is called, it cannot find the ports of the calling module are connected to which ports of another module. However, having both ports handles, it can find whether they are on a channel or not.
To make the description easy, consider an example. Suppose we have two producer and consumer modules. Each module has a sequential block. One of them continuously produces data, while the other consumes that data. They communicate via a channel.

At the first glance, it seems that the algorithm shown in figure 4 should be implemented.
Unfortunately, however, this code does not work. As mentioned before, while one PLI routine is executed the simulator is blocked and cannot simulate anything else. Therefore, when one of the processes is blocked in a while loop within the PLI system call, the whole simulation will be blocked, and as a result, the simulation stops.

	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=value;
	bIsBufferFull=1;
	while(!bReadDone);
	bIsBufferFull=0;
	while(bReadDone);
	}
		Read_Calltf(){
		while(!bISBufferFull);
		readVal=buffer;
		bReadDone=TRUE;
		while(bIsbufferFull);
		bReadDone=FALSE;
		}

Figure 4. Preliminary implementation of Producer

Generally, it is not possible to do wait actions in a PLI routine because of the sequential property of the PLI. However, we need at least two wait actions in order to do the communication action.
One possible solution to this problem is to push wait actions back to the Verilog code and leave the rest of the job to the PLI. In this way, the PLI routines do not block the simulation anymore. In other words we let PLI just do those tasks that will not be blocked while Verilog can do wait actions. Thus, if a process is blocked, other processes can still go on.

To do that, we have to do some alterations on our channel structure. The new one is:

```
typedef struct t_channel{
        BOOL        bIsBufferFull;
        BOOL        bIsReadRequest;
        char        *prtBuffer;
        handle      hReadDone;
        handle      hReadClear;
        handle      hReadPort;
        handle      hWritePort;
}
SChannel;
```

Note that in this structure we changed the type of bReadDone signal to handle (hReadDone). This was because we want to do a wait action on this signal which should be in Verilog code. Therefore, as described, we will define this variable in our Verilog module, and we will store its handle in the PLI interface shared memory. Also, we added another handle called hReadClear which will be explained soon.

Now we can describe the main algorithm. The algorithm for write action can be 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
```

Note that the Write function here, is different from the previous one. In below a pseudo code of each function is given.

```
Write(out,data){
buffer = data;
        bIsBufferFull = TRUE;
        if (bIsReadRequest)
      change the value of bReadClear(using
      hReadClear) to TRUE;
}
RegisterReadDoneFlag(out,bReadDone){
        hReadDone = handle(bReadDone)
}

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

Obviously none of these functions can stop the simulation because we do not have a wait action in them.
Before describing the whole algorithm, let's look at the consumer process:

```
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
```

There are three new functions which are described below using pseudo code.

```
RegisterReaderFlag(in,bReadClear){
        if(bIsBufferFull==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;
}
```

As can be seen, six PLI functions along with waits and initializations in the Verilog code have implemented the communication actions. We introduced two reg variables in the Verilog code, which are bReadDone and bReadClear.
The whole work is very easy: the producer module first 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. The producer then, waits on that 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 bReadClearance. Then, it stores its handle into the shared memory of the PLI interface. Again, the other side will set this flag later, using its handle. Next, it waits on that signal to become TRUE. Again, this does not block the simulation.
One can easily follow the algorithm and verify that whatever starts first, will wait for the other, and the communication action will finish at the same time of simulation. In other words, read and write *coincide* with each other.
Finally, we present a simple code for implementing probes:

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

It should be mentioned here that we have obviously violated the fact that communication actions should be abstract, while in our Verilog code the user has to write four lines of Verilog code for each communication action.
We use a simple converter program to change each READ and WRITE calls to these four lines. However, we have promised not to use a preprocessor tool as much as possible. Fortunately, here we can take advantage of the preprocessor of the Verilog language. It is very easy to define these four lines in a single macro. In this way the communication action will be hidden and becomes an abstract action from the designer's point of view, and at the same time, we have succeeded to keep our promise!

The final code for producer will be something like this:

```
`define WRITE(prt,d) begin\
   bReadDone=1'b0;\
   $Write(prt,d);\
   $RegisterReadDoneFlag(prt,bReadDone);\
   wait(bReadDone==1'b1);\
   $ResetWriteRequest(prt);\
   #0;\
   end
`define USES_CHANNEL        reg bReadDone;

module p(prt);
        ...
        `USES_CHANNEL
always

begin
   //Produce data

                `WRITE(out,data)
end
endmodule
```

One can do the same thing for the consumer module.

5.3 Generalizing the algorithm to work for more than one channel

Although we gave an example of only one port in the previous section, it is very easy to generalize the algorithm in order to enable it to manage more than one port. The approach that we used, is this: there is a list of channels in the PLI interface which are defined as channel structure type. Each time the PLI routine is called for a communication action, if the handle of the port is not in the list, the PLI routine adds it and allocates a new palce for the channel that is connected to this port. However, a problem is here that in the PLI interface we cannot directly find out which ports have formed a channel.
This can be solved through different ways. On way is to explicitly define each port and each channel in the Verilog code, using some PLI routines. However, the better way is to make the PLI interface find which ports are on the same channel. This can be done using Access and TF routines in the PLI. There are some routines that can find the net that a given signal is a member of. Therefore, using this approach, we do not have to explicitly define channels. The PLI interface allocates memory and stores the port handles whenever it reaches a new port which is not in its list. In addition, it can find out which two ports have made a channel.

5.4 Some extensions

In the previous sections we described the basis of implementing channels in Verilog HDL. We also claimed that in this way one can customize his modeling tool, and can extend the semantics of the CSP language, and at the same time stick to standards. In this section, we will show some of those extensions that are possible using the PLI. While one can take advantages of all facilities of PLI in the way that it

is used in synchronous designs, there are some cases that are beneficial only for asynchronous designers.
Here is a list of some possible extensions and usages of the PLI in asynchronous design.

1. Pure handshaking: In [2] a new communication action is introduced which is just used for synchronization between two processes. In this communication action, no data is exchanged. The two processes just do a handshake with each other for to be synchronized. While implementing this kind of communication is possible by using the same read and write action that we described above and a dummy data, one can change those routines and make a new one for this kind of action.
2. Type checking: One can easily extend these routines to do type checking tasks, such as data length matching in the communication actions. On the contrary, one may prefer to have a free framework and write whatever he wants and make the PLI routines to do some conversions when input and outputs do not match each other. For instance, in the example given in section one can change read, and write commands to detect overflow and underflow in a stack.
3. Statistical measurements: Considering the routines that implement communication actions, one can force them to store each communication action in a log file. Later, this can be used as a rough assessment of the circuits power consumption.
4. Time out: In asynchronous designs there are many cases that because of a mistake in the design the circuit goes to a deadlock state and the whole circuit stops working. This happens when a process waits on a signal, while no other process will change the value of that signal. In software word, this problem is solved using time outs. By means of the PLI, we can implement a piece of code to detect these conditions in communication actions. Whenever a process goes to a deadlock state, a time out occurs and the designer will be notified. This helps him to get rid of hours of debugging.
5. Two directional channels: The ports and channels that are described in [7] are one directional, i.e. they are input ports or output ports. However, one can write PLI routines and make a port become a two-way one. As an example of usage, consider a CPU that both reads from and write into a memory module.
6. Hyper channels: the channels that are described in [7] are limited to channels between two processes. However, some times there is a need for channels that are common between more than two processes [2]. It is possible to extend the PLI routines to support this kind of ports.
7. Even more extension of channels. By adding some simple routines, a channel communication can become a collection of many other tasks. For example, one can think of a channel as a TCP/IP connection between two processes. While two circuits are simulated on two separate machines, they can communicate with each other through a channel which is implemented on a TCP/IP or any other communication medium. This method can help to reduce the simulation time and is used for synchronous circuits as well.

One can think of many other extensions.
At the time being, we do not know any other asynchronous simulator or modeling language that is capable of implementing these jobs. Obviously writing these routines is much easier than designing a new language, and then writing a simulator for that. These simple routines can empower every asynchronous designer to make the most of his standard Verilog simulator.

6. Fine-Grained Concurrency

As mentioned before, another most important motivation of to develop a new asynchronous simulator was that it was thought that standard languages, VHDL in most cases, are not capable of supporting fine-grained concurrency within sequential processes [7]. Here, we will show how we have used standard Verilog to implement concurrency within sequential processes.
Before that, however, we will give an example of a process following process:
*[A ; B || (C;D) ; E]
In the above code, A,B,C,D, and E are some tasks that their functionalities are not important. Suppose that they are just some communication actions.
First, task A should be executed. Then, B should be executed in parallel with (C;D). This means that while D is executed after C, they should be in parallel with B. Only after B,C, and D have finished can E be executed. One can consider more complex examples, however, this example is enough for us to describe our approach.
In [4] two verilog keywords are introduced for having concurrent blocks within sequential blocks. These keywords are fork and join. Consider this example:

```
always
   begin
        #10     a = 1;
      fork
        #10     b = 1;
        #2      c = 1;
        #100    d = 1;
      join
        #10     e = 1;
end
```

In this example, after 10 time units a is assigned. Then, the assignment of b, c, and d are executed concurrently, i.e. at time 20, b is equal to 1, at time 12, c is equal to 1, and at time 110 d is equal to 1. However, e cannot be assigned until all the statements in the fork-join block are executed. Therefore, its assignment will be executed at time 120.
Additionally, in [4] block statements are introduced as follows: “The block statements are a means of grouping two or more statements together so that they act syntactically like a single statement. There are two types of blocks in the Verilog HDL:

1. Sequential block, also called begin-end block
2. Parallel block, also called fork-join block”

Using this concept, since we can have statements in both parallel blocks and sequential blocks, we should also be able to nest one block in another block at many levels as we wish. Hence, one can implement fine-grained concurrency using these two blocks without any limitation. Now, consider our CSP example that we gave above. This is the equivalent code in Verilog HDL:

```
always
begin
      A;
fork
      B;
      begin
            C;
            D;
      end
      join
      E;
end
```

It is obvious that this is pure standard verilog and no other tool is needed to implement these kinds of actions.
Again, most asynchronous CAD-tools that we know do not have the flexibility of Verilog. In VHDL++ concurrency is just limited to communication actions, and in CHP_2VHDL one cannot use probes in concurrent parts. However, in the Verilog HDL the designer can do whatever he wants, as long as it is possible by Verilog language, in parallel or sequential blocks.

7. Example of a Real Circuit

In this subsection we will show an example of a circuit described at CSP level in Verilog language, using PLI routines.
In [2] there is an example of a lazy stack. A process called stack element is defined as follows:

```
stack_element≡ process(in?int(8),
out!int(8),get?int(8),put!int(8))
       x:int(8)
       E
end
E =*[  [in -> in?x     out -> get?x
       ];
       [out-> out!x    in-> put!x
       ]].
```

Using our method the following code would be the equivalent program in Verilog HDL.

```
`include “channels.v”
module stack_element(in, out, get, put);
`USES_CHANNEL
input          [7:0]    in, get;
output         [7:0]    out, put;
reg            [7:0]    x;
always
begin
   while($Probe(in)!=1&&$Probe(out)!=1)
   begin
      #`step_delay;
   end
   if ($ProbePort(in)) `READ(in,x)
   else if ($ProbePort(out)) `READ(get,x)
   while ($Probe(in)!=1 && $Probe(out)!=1)
   begin
      #`step_delay;
   end
   if ($ProbePort(out)) `WRITE(out,x)
   else if ($ProbePort(in))`WRITE(put,x)
   #`step_delay;
end
endmodule
```

Using stack_element, one can define a lazy stack as follows:

```
module lazy_stack(in, out);
input [7:0 in;
output [7:0] out;
wire [1:3] get, put;
stack_element m1
(in,out,get[1],put[1]);
stack_element m2
(put[1],get[1],get[2],put[2]);
stack_element m3
(put[2],get[2],get[3],put[3]);
endmodule
```

A testbench for this code can be like this:

```
module lazy_stack_tb;
`CHANNEL_INIT
  wire in, out;
  reg [7:0] readVal;
  initial
  begin
     `WRITE(in, 8'd5)
     `WRITE(in, 8'd6)
     `READ(out, readVal)
     `READ(out, readVal)
      end
endmodule
```

We belive that one can underestand the same thing from both codes without any extra interpretation.

8. Concluding Remarks

As another, not important motivation for a new language in some articles it is said that unlike the CSP language the syntaxes of VHDL and Verilog HDL are verbose. However, although this is a true fact and also everybody is used to write his CSP program in a syntax different from Verilog, it seems that changing the syntax of this language is inevitable. The reason is that some of its symbols do not exist in ASCII table. As a result, instead of the original ones, everybody uses his own symbols when they want to write their code in a plain ASCII text editor. Therefore, we believe that when some syntaxes should be changed it is not important that other ones change too. Furthermore, if it really matters to stick to the syntax, in most cases one can use compiler’s preprocessor to do the conversion. Consider these two examples, does two pieces of code really different?

1. CSP: in?data
   Verilog: `WRITE(in,data)
2. Production Rule: a -> b+
   Verilog: assign b= (a==1) ? 1 : b;

8.1 What is left?

We have described two important features of the CSP language, and we suggested a method to implement them in the Verilog language. However, the CSP language has some other features, although not very important, Verilog (1364-95) cannot support them. Among them we will mention some more important ones.

1. Records: in [2] a record is defined as a collection of fields. It has the same meaning as in PASCAL language. Although Verilog lacks this data type, it is very easy to use this concept by means of bit select.
2. Replication construct: in [2] a replication block is defined which functions like GENERATE blocks in VHDL. While in Verilog 95 the only way to implement this construct was vectorized instantiation, which is limited in this case, in Verilog 2001 *genvar* and *generate* keywords have been added to support this concept.

In table 1 we show some advantages and drawbacks of our method. We showed that Verilog HDL is a suitable HDL for modeling asynchronous circuits at all levels of abstraction. We described how it is possible to use PLI to implement two important features of the CSP language: channels and fine-grained concurrency.

Table 1. Comparison of presented method with others

Feature	Verilog +	LARD	VHDL	CHP2V
Portability	√	×	×	×
Os Independency	√	×	√	√
Supported by a wide range of providers	√	×	×	×
Description of the circuit at all levels of abstraction	√	×	√	×
Easily customizable	√	×	×	×
Public access	√	√	√	×
A new language	×	√	√	√
Limitations in implementing CSP	×	×	√	√

References

[1] S. Hauck "Asynchronous Design Methodologies: An Overview," *Proc. Of the IEEE*, vol. 83, no. 1, 1995.

[2] C. A. R. Hoare "Communicating Sequential Processes" *Communications of ACM* 21, 8, pp. 666-677, 1978.

[3] A. J. Martin *Synthesis of Asynchronous VLSI Circuits*, Technical Report, California Institote of Technology, Caltech CSTR: 1991.cs- tr- q3-28.

[4] IEEE Std p1364-2001, IEEE Standard Verilog® Hardware Description Language, IEEE Computer Society, 2001.

[5] S. Mittra *Principles of Verilog PLI*, Kluwer Academic Publishers, 1999.

[6] S. Frankild, J. Spars; "Channel Abstraction and Statement Level Concurrency in VHDL++," *Proc. Of 4th ACiD-WG workshop, Grenoble*, France, 2000.

[7] P. Endecott S. Furber "Modeling and Simulation of Asynchronous Systems using the LARD," *Proc. of the 12th European Simulation Multiconference*, Manchester, 1998.

[8] P. Viviet and M. Renaudin, "CHP2VHDL, a CHP to VHDL Translator Towards Asynchronous-design Simulation," *Handouts from the ACiD-WG Workshop on Specification models and languages and technology effects of asynchronous design,* Dipartemento di Elettronica, Polytecnico de Torino, Italy, 1998.

[9] A. Seifhashemi, H. Pedram, "Verilog HDL, a Replacement for CSP," *3rd ACiD-WG Workshop FP5, FORTH*, Crete, Greece, 2003.

[10] A. Seifhashemi, M. Naderi, K. Saleh, M. Salehi, H. Pedram, "PERSIA: An Asynchronous Synthesis Tool Based on Alain Martin's Method," CAD Tutorial, *9th IEEE Int. Symp. on Asynchronous Systems & Circuits, Vancouver*, Canada, 2003.

[11] A. Seifhashemi, H. Pedram, "Verilog HDL, Powered by PLI: a Suitable Framework for Describing and Modeling Asynchronous Circuits at All Levels of Abstraction," *40th Design Automation Conference, Anneheim*, CA, 2003.

Arash Saifhashemi Received his BS and MS degree in Computer Engineering from Amirkabir University of Technology (Tehran Polytechnic) in 2000 and 2003 respectively. He is currently a PhD student in USC. His research interests include asynchronous circuit synthesis. Email: saifhash@usc.edu

Mohsen Naderi Received his BS and MS degree in Computer Engineering from Amirkabir University of Technology (Tehran Polytechnic) in 1998 and 2001 respectively. He is currently a PhD student in Amirkabir University, of Technology. His research interests include low-power and high-performance circuit design and also asynchronous circuit design and synthesis.
Email: naderi@ce.aut.ac.ir

Hossein Pedram Received his BS degree in Electrical Engineering from Sharif University in 1979. He received his MS and PhD degrees from Ohio State University and Washington State University respectively in Computer Engineering. His work experience include 5 years in Iran Telecommunications Research Center as Senior Engineer from 1982 to 1987. Dr Pedram Has served as a faculty member in the Computer Engineering Department at Amirkabir University of Technology since 1992. He teaches courses in Computer architecture and distributed systems. His research interests include innovations in computer architecture such as asynchronous circuits, management of computer networks, distributed systems, and robotics.
Email: pedram@ce.aut.ac.ir

Arash Farhoodfar received his BS degree is Computer Engineering from Tehran Polytechnic University in 1996. He received his MS degree in Computer Engineering from Tehran University in 1999. For the past 4 years, Arash has been working in Applied Micro Circuit Co. (AMCC) as a senior principal engineer architecting high speed SONET and Forward Error Correction (FEC) circuits. Arash was one of the founding members of Polytechnic Asynchronous design group. His research interests includes high speed/low power hardware design such as Asynchronous hardware design and computational complexity.
Email: afarhood@amcc.com