Communicating Process Architectures 2005
Jan Broenink, Herman Roebbers, Johan Sunter, Peter Welch, and David Wood (Eds.)
IOS Press, 2005

High Level Modeling of Channel-Based Asynchronous Circuits Using Verilog

Arash Saifhashemi [a,1] and Peter A. Beerel [b]

[a] *PhD Candidate, University of Southern California, EE Department, Systems Division*
[b] *Associate Professor, University of Southern California, EE Department, Systems Division*

Abstract. In this paper we describe a method for modeling channel-based asynchronous circuits using Verilog HDL. We suggest a method to model CSP-like channels in Verilog HDL. This method also describes nonlinear pipelines and high-level channel timing properties, such as forward and backward latencies, minimum cycle time, and slack. Using Verilog enables us to describe the circuit at many levels of abstraction and to use the commercially available CAD tools.

Keywords. CSP, Verilog, Asynchronous Circuits, Nonlinear Pipelines

Introduction

A digital circuit is an implementation of a concurrent algorithm [2]. Digital circuits consist of a set of modules connected via ports for exchanging data. A port is an electrical net whose logical value is read and/or updated. A complex module may consist of a collection of simpler modules working in parallel, whose ports are connected by wires. At a higher level of abstraction, however, complex modules can often be modeled as a process, which communicate with other complex modules through communication channels [1] that are implemented with a set of ports and wires and a handshaking protocol for communication.

This paper focuses on the modeling and simulation of a large class of asynchronous circuits which use CSP (Communicating Sequential Processes [1,2]) channels for communication. In particular, any digital circuit that does not use a central clock for synchronization is called asynchronous. In channel-based asynchronous circuits, both synchronization and data communication among modules are implemented via channel communication. In fact, communication actions on channels are synchronous, i.e. the read action in a receiving module is synchronized with the write action of the sending module. This synchronization removes the need of a global clock and is the foundation of a number of demonstrated benefits in low-power and high-performance [9,10]. Unfortunately, asynchronous circuits will not gain a large foothold in industry until asynchronous design is fully supported by a commercial-quality CAD flow. In this paper, we present a method to enhance Verilog with CSP constructs in order to use commercially available CAD tools for developing channel-based asynchronous circuits.

To model the high-level behavior of channel-based asynchronous designs, designers typically use some form of CSP language that has two essential features: channel-based communication and fined grained concurrency. The former makes data exchange between

[1] Corresponding Author: Arash Saifhashemi, Department of Electrical Engineering – Systems, EEB 218, Hughes Aircraft Electrical Engineering Building, 3740 McClintock Ave, Los Angeles, CA, 90089, USA; E-mail: saifhash@usc.edu

module models abstract actions. The latter allows one to define nested sequential and concurrent threads in a model. Thus, a practical Hardware Description Language (HDL) for high-level asynchronous design should implement the above two constructs. Furthermore, similar to many standard HDLs, the following features are highly desired:

- Support for various levels of abstraction: There should be constructs that describe the module at both high and low levels (e.g., transistor-level) of abstraction. This feature enables modeling designs at mixed-levels of abstraction, which provides incremental verification, as units are decomposed into lower-levels and enables arrayed units (e.g., memory banks) to be modeled at high-levels of abstraction to decrease simulation run-time. Also, this enables mitered co-simulation of two-levels of abstraction, in which the lower-level implementation can be verified against the higher-level, golden specification.
- Support for synchronous circuits: A VLSI chip might consist of both synchronous and asynchronous circuits [13]. The design flow is considerably less complex if a single language can describe both, so that the entire design can be simulated using a single tool. Consequently, modeling clocked units should be straightforward.
- Support for timing: Modeling timing and delays is important at both high and low-levels of the design. Early performance verification and analysis of the high-level architecture using estimated delays is critical to avoid costly re-design later in the design cycle. Later, it is essential to verify the performance of the more detailed model of the implementation using accurate back-annotated delays.
- Using available supporting CAD tools: In addition to the availability of powerful simulation engines, hooks to debugging platforms (e.g., GUI-based waveform viewers), synthesis tools, and timing-analyzers should also be available. There are many powerful CAD tools available in these areas, but in most cases they only support standard hardware design languages such as VHDL and Verilog.
- A standard syntax: The circuit description should be easily exchangeable among a comprehensive set of CAD tools. Using a non-standard syntax causes simulation of the circuit to become tool dependant.

Several languages have been used for designing asynchronous circuits in the literature. They can be divided in the following categories:

1. A new language. A new language with syntax similar to CSP is created, for which a simulator is developed. Two examples of this method are LARD and Tangram [3,14]. Simulation of the language is dependant on the academic tool and tool support/maintenance is often quite limited. Also, the new language usually does not support modeling the circuit at the lower levels of abstraction such as at the transistor and logic levels.
2. Using software programming languages like C++ and Java. For example, Java has been enhanced with a new library, JCSP [4,12], in order to support CSP constructs in Java. This approach does not support timing and mixed level simulation. Furthermore, integration with commercially-available CAD tools is challenging.
3. Using standard hardware design languages such as Verilog and VHDL. Because of the popularity of VHDL and Verilog among hardware designers, and also the wide availability of commercial CAD tool support, several approaches have been made to enhance these languages to model channel-based asynchronous circuits. Previous works by Frankild, et al. [5], Renaudin, et al. [6], and Myers [7] employ VHDL to design asynchronous circuits. In VHDL, however, implementing fine grained concurrency is cumbersome, because modeling the synchronization

between VHDL processes requires extra signals. Moreover, in some cases [6], the native design language must be translated into VHDL. This makes the debugging phase more cumbersome, because the code that is debugged in the debugger is different from the original code. Signals may have been added, and port names may have been changed, forcing the designer to know the details of the conversion. T. Bjerregaard, et al. [18] propose using SystemC to model asynchronous circuits and have created a library to support CSP channels. Similar to VHDL, implementing fine-grained concurrency in SystemC is cumbersome. Also, modeling timing is not addressed in their approach. In [8], Verilog together with its PLI (Programming Language Interface) has been proposed. Using Verilog, modeling the fine-grain concurrency is easily available by using the built-in fork/join constructs of Verilog. The PLI has been used for interfacing Verilog and pre-compiled C-routines at the simulation time. Using the PLI, however, has two disadvantages: first, the PLI interface significantly slows down the simulation speed, and secondly, the C code must be recompiled for all system environments, making compatibility across different system environments a challenge. Lastly, in the Verilog-PLI approach, handshaking variables are shared among all channels of a module. Unfortunately, this scheme breaks down for systems such as non-linear pipelined circuits in which multiple channels of a module are simultaneously active.

This paper addresses the problems of the Verilog-PLI method [8] and makes CSP constructs available in Verilog, without the above limitations. Besides the basic channel implementation, we propose to model performance of asynchronous pipelines by modeling the forward/backward latency and minimum cycle time of channels as timing parameters to our high-level abstract model. It is worthwhile mentioning that using Verilog also enables one to migrate to SystemVerilog [19], which commercial CAD tools are beginning to support. Since SystemVerilog is a superset of Verilog, our method will be directly applicable to future CAD tools that support SystemVerilog.

The remainder of this paper is organized as follows. In Section 1, relevant background on CSP and non-linear pipelines is presented. Section 2 explains the details of implementing SEND/RECEIVE macros in Verilog. Section 3 describes the modeling of asynchronous pipelines using these macros. Section 4 describes further improvements to the method such as monitoring the channels' status, implementing channels that reshuffle the handshaking protocol, and supporting mixed mode simulation. Section 5 presents a summary and conclusions.

1. Background

In this section we briefly describe relevant background on CSP communication actions and asynchronous nonlinear pipelines [9].

1.1 Communicating Sequential Processes

Circuits are described using concurrent processes. A process is a sequence of atomic or composite actions. In CSP, a process P that is composed of atomic actions $s_1, s_2, \ldots, s_n$, repeated forever, is shown as follows:

```
P = *[s1 ; s2 ; … ; Sn]
```

Usually, processes do not share variables, but they communicate via *ports* which are connected by *channels*. Each port is either an input or an output port. A communication action

consists of either sending a variable to a port or receiving a variable from a port. Suppose we have a process *S* that has an output port *out* and a process *R* that has an input port *in*, and suppose *S.out* is connected to *R.in* via channel *C*. The *send* action is defined to be an event in *S* that outputs a variable to the *out* port and suspends *S* until *R* executes a *receive* action. Likewise, a *receive* action in *R* is defined to be an event that suspends *R* until a new value is put on channel *C*. At this point, *R* resumes and reads the value. The completion of *send* in *S* is said to *coincide* with the completion of *receive* in *R*. In CSP notation, sending the value of the variable *v* on the port *out* is denoted as follows:

```
(out!v)
```

Receiving the value *v* from the port *in* is denoted as:

```
(in?v)
```

Another construct, called a *probe,* has also been defined in which a process p_1 can determine if another process p_2 is suspended on the shared channel *C* for a communication action to happen in p_1 [2]. Using the *probe*, a process can avoid deadlock by not waiting on receiving from a channel on which no other process has a pending write. *Probe* also enables the modeling of arbitration [2].

For two processes *P* and *Q*, the notion *P*||*Q* is used to denote that processes *P* and *Q* are running concurrently. On the other hand, the notion *P;Q* denotes that *Q* is executed after *P*. We can also use a combination of these operators, for example, in the following:

```
*[(p1 || (p2;p3) || p4) ; p5]
```

process p_1 will be executed in parallel with p_4. At the same time $p_2;p_3$ will be executed. Finally, once all p_1, p_2, p_3, and p_4 finish, p_5 will be executed. This nested serial/concurrent processes at deeper levels enable modeling fine grained concurrency.

1.2 Asynchronous pipelines

A channel in an asynchronous circuit is physically implemented by a bundle of wires between a sender and a receiver and a handshaking protocol to implement *send* and *receive* actions and the synchronization. Various protocols and pipeline stage designs have been developed that trade-off robustness, area, power, and performance. Channels are point-to-point from an output port of one process to an input port of another process. Linear pipelines consist of a set of neighboring stages with one input and one output port. We can describe a stage of a simple linear pipeline that receives value *x* from its left port and sends *f*(*x*) on its right port as follows:

```
Buffer = *[in?x ; y=f(x) ; out!y]
```

For this stage we define the following performance metrics [9] that are defined with the assumption that data is always ready at the *in* port, and a receiver is ready to receive data from the *out* port.

1. Forward latency: The minimum time between the consecutive *receive* at the *in* port and *send* at the *out* port
2. Backward latency: The minimum time between the consecutive *send* at the *out* port and the *receive* at the *in* port
3. Minimum Cycle time: The minimum time between two consecutive *receive* actions (or between two consecutive *send* actions). In the above example, the minimum cycle time is equal to the minimum value of the sum of execution times of *receive*, f(x) calculation, and *send*

A pipeline is said to be non-linear if a pipeline stage has multiple input and/or output channels. A pipeline stage is said to be a *fork* if it can send to multiple stages. A pipeline stage

is said to be a *join* if it has input channels from multiple predecessor stages. Furthermore, complex non-linear pipelines support conditional communication on all channels, i.e., depending on the value read from a certain control input channel, the module either reads from or writes to different channels.

Asynchronous circuits are often implemented using fine grained non-linear pipelines to increase parallelism. In this paper, we show how to model the performance properties of such a pipeline at a high level of abstraction. In particular, in high-level performance models, it is necessary to estimate the amount of internal pipelining within a process. This pipelining is characterized as *slack* and is associated with ports of pipeline stages as follows:

1. Input port slack: The maximum number of *receive* actions that can be performed at the input port, without performing any *send* action at the output port(s) of the pipeline stage.
2. Output port slack: The maximum number of *send* actions that can be performed at the output port, without performing any *receive* action at the input port(s) of the pipeline stage.

We adopt the modeling philosophy that the performance of the pipeline stage can be adequately modeled by specifying the forward, backward, and the minimum cycle time of the associated slack at the input and output ports. In Section 3, we will describe how to capture and model the *slack* in our Verilog models.

2. Communication Actions in Verilog

Our approach to modeling communication actions in Verilog is to create two macros SEND and RECEIVE that model a hidden concrete implementation of the handshaking protocol [2] for synchronization. The challenge we faced is associated with the limited syntax and semantics of Verilog macros: Verilog macros only supports textual substitution with parameters, but do not support creating new variables via parameter concatenation as is available in software languages like C.

Among different protocols, the bundled data handshaking protocol [10] has the lowest simulation overhead: for a bundle of signals, we must have an extra output signal called *req* in the sender and an extra input called *ack* in the receiver. When the sender wants to send data, it asserts the value of this extra bit, *req*, to assert that the new data is valid. Then, it waits for the receiver to receive the data. Once the data is received, the receiver informs the sender by asserting the *ack* signal. Finally, both *req* and *ack* will be reset to zero. The behavior of this protocol in Communicating Hardware Processes (CHP) notation [2], a hardware variant of CSP, is as follows:

```
Sender:
*[req=1 || d7…d0=produced data;[ack];req=0;[~ack]]

Receiver:
*[ [req] ; buffer= d7…d0 ; ack=1 ; [~req] ; ack=0]
```

Here, `[x]` means wait until the value of so*x* becomes true.

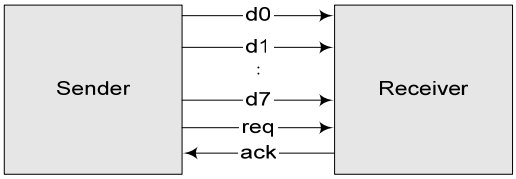

Figure 1. Bundled data Protocol

Our goal is to use Verilog macros to hide the handshaking details and make the actions abstract. First, we hide the extra handshaking signals, i.e., *req, ack.* This can be achieved by having two extra bits on each port: bit 0 is used for the *req* signal, and bit 1 is used for the *ack* signal. A naive Verilog implementation of the bundled data protocol, using those bits is shown in Figure 2. Suppose that the *out* port of the *Sender* module is connected to the *in* port of the *Receiver* module.

```
module Sender(out);
  output [7+2:0]out;
  reg [7:0]d;
  always
  begin
    //Produce d
    out[9:2]=d;
    out[0]=1;
    wait(out[1]==1);
    out[0]=0;
    wait (out[1]==0);
  end
endmodule
```

```
module Receiver(in);
  input [7+2:0] in;
  reg [7:0] d;
  always
  begin
    wait (in[0]==1);
    d=in[9:2];
    in[1]=1'b1;
    wait (in[0]==0);
    in[1]=0; //error
    //Consume d
  end
endmodule
```

Figure 2. Verilog Implementation of Sender and Receiver modules (A naive version)

The above code, however, does not work, because in the receiver module we are writing to an input port which is illegal in Verilog. Changing the port type to *inout* does not solve the problem, because writing to an *inout* port is also illegal in the sequential blocks of Verilog, i.e., the *always* block. Our solution is to use the *force* keyword that allows us to change the value of any net type, and in particular the *reg* type, which is a variable type in Verilog that is used in sequential blocks.

Our goal is to hide the handshaking protocol using macros such as *`SEND(port, value)* and *`RECEIVE(port, value)*. In the above code, one issue is that the width of *in* and *out* ports must be available to the macros, so that the macros can assign the eight significant bits of *in* to *d* (`d=in[9:2]`). Rather than passing this width as an extra parameter to the macros, we used a dummy signal as shown in Figure 3:

```
module Sender(out);
  output [2+7:0] out;
  reg [7:0]d;
  always
  begin
    //Produce d
    force out={d,out[1],1]};
    wait(out[1]==1);
    force out[0]=0;
    wait (out[1]==0);

  end
endmodule
```

```
module Receiver(in);
  input [2+7:0] in;
  reg [7:0]d;
  reg[1:0] dummy;
  always
  begin

    wait (in[0]==1);
    {d,dummy}=in;
    force in[1]=1;
    wait (in[0]==0);
    force in[1]=0;

    //Consume d
  end
endmodule
```

Figure 3. Correct Version of Sender and Receiver

The dummy signal is two bits, thus the variable *d* is always assigned to the actual data bits of *in,* i.e., bit 2 and higher. Therefore, the first two bits - the handshaking variables - are thrown away. Notice that the dummy signal is written, but never read. We make the above code more efficient by moving the resetting phase of the handshaking protocol to the

beginning of the communicating action, thereby, removing one wait statement. In this way, the *Sender* both resets the *ack* signal of the *Receiver* (bit 1) and sets its own *req* signal (bit 0). Similarly, the *Receiver* reads data, and then both resets the *req* signal of the *Sender* and sets its own *ack* signal.

```
module Sender(out);
  output [2+7:0] out;
  reg [7:0]d;
  always
  begin
    //Produce d

    force out={d,2'b01]};
    wait(out[1]==1);

 end
endmodule
```

```
module Receiver(in);
  input [2+7:0] in;
  reg [7:0]d;
  reg[1:0] dummy;
  always
  begin

    wait (in[0]==1);
    {d,dummy}=in;
    force in[1:0]=2'b10;

    //Consume d
  end
endmodule
```

Figure 4. Optimized Version of Sender and Receiver

The final definitions of the two macros for SEND and RECEIVE are as follows:

```
`define SEND(_port_,_value_)    begin\
            force _port_={_value_,2'b01};\
            wait (_port_[1]==1'b1);\
        end

`define RECEIVE(_port_,_value_)    begin\
            wait (_port_[0]==1'b1);\
            {_value_,dummy}=_port_;\
            force _port_[1:0]=2'b10;\
        end
```

We also need to hide the dummy signal definition and input/output port definitions:

```
`define USES_CHANNEL          reg [1:0] dummy;
`define OUTPORT(port,width)   output[width+1:0] port;
`define INPORT(port,width)    input[width+1:0]  port;
`define CHANNEL(c,width)      wire[width+1:0]   c;
```

The designer should use the `USES_CHANNEL macro in modules that incorporate the communication protocol. The INPORT/OUTPORT and CHANNEL macros add two more bits to each port for handshaking.

The final versions of *Sender* and *Receiver* together with a top module that instantiates them are shown in Figure 5.

```
module Sender(out);
  `OUTPORT(out,8);
  `USES_CHANNEL
   reg [7:0]d;
   always
    begin
     //Produce d
   `SEND(out,d)
   end
endmodule
```

```
module Receiver(in);
   `INPORT(in,8);
   `USES_CHANNEL
   reg [7:0]d;
   always
   begin
      `RECEIVE(in,d)
      //Consume d
   end
endmodule
```

```
module top;

   `CHANNEL (ch,8)
   Sender p(ch);
   Receiver c(ch);

endmodule
```

Figure 5. Final Version of Sender/Receiver

As shown in Figure 5, the SEND/RECEIVE macros are used in the same level of abstraction as they are used in CSP.

3. Modeling Performance

In this section we show how we can incorporate the pipeline performance properties such as forward/backward latency, minimum cycle time, and slack in our model.

3.1 Timing

The buffer described in Section 1.2 can be described in Verilog as shown in Figure 6. *FL* and *BL* are the forward and backward latencies as defined in Section 1.2. The slack of this buffer is 1 on both ports.

```
module buf (left, right);
    parameter width = 8;
    parameter FL = 5;
    parameter BL = 10;
    `USES_CHANNEL
    `INPORT(left,width)
    `OUTPORT(right,width)
    reg [width-1:0] buffer;
    begin
        `RECEIVE(left, buffer)
        #FL;
        `SEND(right, buffer)
        #BL;
    end
endmodule
```

Figure 6. Modeling a Simple Buffer

Now, consider the description of a simple two-input function, *func* with the following description in CHP notation:

```
func: *[A?a||B?b ; c=func(a,b) ; C!c]
```

Also, consider a pipelined implementation of the above function that has slack 3 on A, 2 on B, and 2 on C. We can model the behavior of the pipeline using the circuit shown in Figure 7.

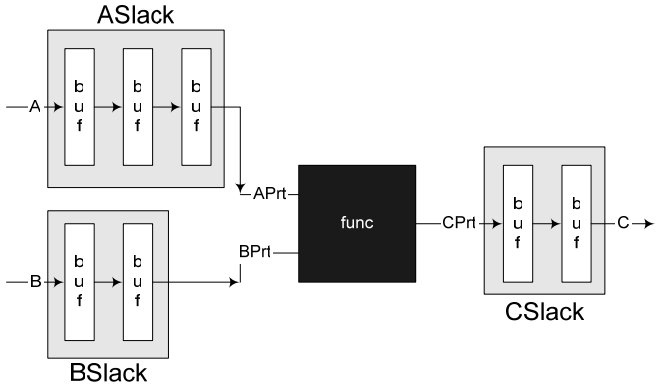

Figure 7. A pipelined two-input function with slacks 3 and 2 on inputs and 2 on the output

The pipeline can have different forward/backward latencies on each port. For high-level modeling, it is desirable to make these parameters (forward/backward latency, and slack) abstract, and avoid the requirement of explicitly instantiating extra buffers on each port. We propose to enhance the INPORT/OUTPORT macros so that they include all these parameters, i.e., all slack buffers are instantiated through INPORT/OUTPORT macros in module *f* automatically. Suppose we have the following information about the ports given in Table 1:

Table 1. Information about ports of the pipeline

Port	*Width*	*Slack*	*Forward Latency*	*Backward Latency*
A	8	3	5	10
B	8	2	10	5
C	8	2	15	15

We can define a new macro, INPUT, as follows:

```
`INPUT(slackModName,portName,portAlias,width,slack,BL,FL)
```

In a similar way, the *OUTPUT* macro can be defined for output ports. The INPUT macro instantiates a module called `slackModule` and identifies the value of forward/backward latency and slack through parameter passing. It also connects `slackModule` to the *func* module. Figure 7 shows how we use this macro. The details of the INPUT and OUTPUT macros are given in Figure 8.

```
module adder (A, B, C);
  `USES_CHANNEL
 reg [width-1:0] abuf, bbuf, cbuf;
`INPUT(ASlack,A,aPort,8,3,5,10)
`INPUT(BSlack,B,bPort,8,2,10,5)
`OUTPUT(CSlack,C,cPort,8,3,15,15)
     always
    begin
       fork
       `RECEIVE(aPort, abuf)
       `RECEIVE(bPort, bbuf)
       join
       cbuf = func(abuf,bbuf);
       `SEND(cPort, cbuf)
    end
endmodule
```

Figure 7. A Pipelined Implementation Of func

```
`define INPUT(slackName,portName,portAlias,\
        width,slack,BL,FL)\
        input[width+1:0] prtName;\
        wire [width+1:0] prtAlias;\
        SlackModule #(width,slack,BL,FL)\
        slackName(prtName,prtAlias);

`define OUTPUT(slackName,prtName,prtAlias,\
        width,slack,BL,FL)\
        output[width+1:0] prtName;\
        wire [width+1:0] prtAlias;\
        SlackModule #(width,slack,BL,FL)\
        slackName(prtAlias,prtName);
```

Figure 8. INPUT/OUTPUT Macros for a Pipeline.

```
module slackModule (left, right);
 parameter width = 8;
 parameter SLACK = 5;
 parameter     FL = 0;
 parameter     BL = 0;
 `USES_CHANNEL
 `INPORT(left,width)
 `OUTPORT(right,width)
 wire [width+1:0] im [SLACK-1:0];
 genvar i;
 generate for (i=0; i<SLACK; i=i+1)
 begin:stage
  if (i==0)
   buffer #(width,FL, BL) buff(left, im[0]);
   else if (i==SLACK-1)
    buffer #(width,FL,BL)buff(im[i-1],right);
   else
    buffer #(width,FL,BL)buff(im[i-1],im[i]);
 endgenerate
endmodule
```

Figure 9. Description of slackModule

`slackModule` contains a chain of connected buffers as defined in Figure 6. In this module, the values of parameters specify how many buffers should be instantiated and what their latencies are. Although one can consider more efficient ways to implement this module to make the simulation faster, for simplicity here we use the generate loop construct of Verilog 2001 [15] in Figure 9.

4. Further Improvements

In this section, we first show how to monitor the status of channels and ports in a GUI debugging tool. Next, we explain how to model non-pipelined circuits that have a reshuffled handshaking protocol. Then, we explain how further improvements can be obtained using a converter program. Finally, we consider some extensions.

4.1 Debugging

As described before, since the *SEND* and *RECEIVE* actions are blocking, a circuit might deadlock when some module executes a *RECEIVE* action on a port for which no other module will execute a *SEND* action. Thus, for any language that implements the CSP communication actions, it is essential to make the debugging of channels straight-forward. One important issue is that the designer should see the status of each port and channel while simulating the circuit. This can be achieved by monitoring the handshaking signals, i.e., the extra two bits on each port. This will not work, however, for input ports since the *RECEIVE* action is passive, i.e., it does not change the value of the handshaking signals until it actually receives a value. To overcome this limitation, we used one more extra bit in the ports (i.e., a total of 3 extra bits per port). So, whenever the *RECEIVE* executes, it sets the third bit, and when it finishes, it resets the third bit. An example of monitoring the status of channels using GUI is shown in Figure 10, where we used mnemonic definitions sensitive to the last three bits of the channels. Another usage of this extra bit is that the designer can use it for implementing *probe*, which was defined in Section 1.1. Other researchers at Fulcrum Microsystems have independently identified a similar strategy.

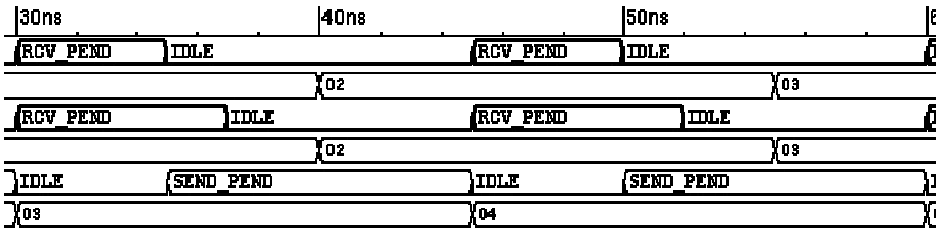

Figure 10. Debugging In GUI

4.2 Reshuffling the Handshaking Protocols

In the bundled data protocol that we described in Section 2, output ports are active (i.e. they initiate the communication), and input ports are passive [10]. It is possible to consider other handshaking protocols and/or to reshuffle the handshaking protocol of input and output ports. One example is a protocol that shuffles the handshaking of *receive* at the input and *send* at the output. In fact the CALL process [2], which essentially implements a slack-0 communication and is common in non-pipelined systems [14], can be implemented by the reshuffling of the

handshaking protocol. This module can be described as follows:

```
CALL:
*[ [l_req]; buffer= dn…d0; r_req=1 || dn…d0=buffer; [r_ack]; r_req=0; [~r_ack]; l_ack=1;
[~l_req]; ack=0 ]
```

The above buffer splits the handshaking of the left port in two parts, shown in bold face, and sends the value to the right port in the middle of these parts. Figure 11 shows the equivalent code in Verilog.

It is straight-forward to split our RECEIVE macro and create the new macros RECEIVE_PART1 and RECEIVE_PART2 and use it as in Figure 11. After reading data from the left port, the module connected to the left port will remain suspended, until the *send* on the right port is done.

```
module CALL (left, right);
    parameter width = 8;
    parameter FL = 5;
    parameter BL = 10;

    `USES_CHANNEL
    `INPORT(left,width)
    `OUTPORT(right,width)
    reg [width-1:0] buffer;
    begin
        `RECEIVE_PART1(left, buffer)
        `SEND(right, buffer)
        `RECEIVE_PART2(left, buffer)
    end
endmodule
```

Figure 11. Implementation of a Call Process

4.3 Improving the performance via conversion

As described in previous sections, there is overhead associated with these macros. Simulation speed can be further improved by converting these macros to the bundled data protocol in which the extra handshaking signals are explicitly declared as *regs* and manipulated without the use of *force* with an external converter program. The resulting Verilog code will be more cumbersome to debug, so we recommend using this converter on a unit-by-unit basis after each unit's correctness has been verified and only where the simulation speed is of great importance.

4.4 Extensions

Some of the extensions to the CSP channels that have been implemented in other methods can be considered here as well. For example, although not commonly used for asynchronous circuit designs, by using the PLI, communication actions can be extended to use TCP/IP ports and communicate through an entire network of computers, possibly for distributing the simulation load on a compute farm.

For mixed mode simulation, it is straightforward to use our method in a module that can communicate both at high and low levels of abstraction. For example, in figure 12, the *mixed_buf* module communicates at a high level on the left, but at a low level on the right, and the module uses explicit handshaking on its right side. Therefore, it can interface a circuit described at high-level to a circuit described at low-level, such as transistor level.

```
module mixed_buf (left, right_data right_req, right_ack,);
    parameter width = 8;
    `USES_CHANNEL
    `INPORT(left,width)
    output [width-1:0] right_data;
    output right_req;
    input  right_ack;
    reg [width-1:0] buffer;
    begin
        //Left side's high-level communication
        `RECEIVE(left, buffer) //Receive from left
       //Right side's explicit handshaking
       Right_data=buffer;
       right_req = 1;
       wait(right_ack==1);
       right_req = 0;
       wait(right_ack==0);
    end
endmodule
```

Figure 12. Modeling a Simple Buffer for Mixed Mode Simulation to Interface Modules Describe at Both High and Low Levels of Abstraction

5. Conclusions

This paper demonstrates that standard Verilog HDL can be used to model channel-based asynchronous circuits at a high level of abstraction, continuing to bridge the gap between the pervasiveness of commercially-standard tools with the advantages of asynchronous implementations. In particular, we described how to model CSP communication primitives using Verilog macros and its application to modeling asynchronous nonlinear pipelines and their typical performance characteristics, such as forward/backward latencies and slack. We then showed how to monitor the status of channels during debugging and also we provided an implementation of handshaking protocols for non-pipelined designs. Finally, we showed how to perform mixed mode simulations and to interface a module described at the high level to a module described at the low level. Compared to the state-of-the-art, this work is the first to support abstract non-linear pipelines in Verilog.

6. References

[1] C.A.R. Hoare., "Communicating Sequential Processes", Prentice Hall International, 1985
[2] A. J. Martin, "Synthesis of Asynchronous VLSI Circuits", Caltech-CS-TR-93-28, California Institute of Technology
[3] P. Endecott and S. Furber, "Modelling and Simulation of Asynchronous Systems using the LARD", http://www.cs.man.ac.uk/amulet/projects/lard/
[4] D. Nellans, V. Krishna Kadaru, and E. Brunvand, "ASIM - An Asynchronous Architectural Level Simulator", GLSVLSI'04
[5] S. Frankild and J. Sparso, "Channel Abstraction and Statement Level Concurrency in VHDL++", Danish Maritime Institute & Technical University of Denmark
[6] M. Renaudin, P. Vivet, F. Robin, "A Design Framework for Asynchronous/Synchronous Circuits Based on CHP to HDL Translation", Async99
[7] C. J. Myers, "Asynchronous Circuit Design", John Wiley and Sons, July 2001.
[8] A. Saifhashemi and H. Pedram, "Verilog HDL, powered by PLI: a suitable framework for describing and modeling asynchronous circuits at all levels of abstraction", DAC 40th
[9] A. M. Lines, "Pipelined Asynchronous Circuits", M.S. Thesis, Caltech, 1995.
[10] S. Hauck, "Asynchronous Design Methodologies: An Overview", Proceedings of the IEEE, Vol. 83, No. 1, pp. 69-3, January, 1995.
[11] K. van Berkel, R. Burgess, J. Kessels, M. Roncken, F. Schalij, and A. Peeters, "Asynchronous circuits for

low power: A DCC error corrector," In IEEE Design & Test of Computers, 11(2):22-32, summer 1994.
[12] P. D. Austin and P. H. Welch, "Java Communicating Sequential Process – JCSP", http://www.cs.ukc.ac.uk/projects/ofa/jcsp/
[13] P. A. Beerel, J. Cortadella, and A Kondratyev, "Bridging the Gap between Asynchronous Design and Designers", VLSID'04
[14] K. van Berkel, J. Kessels, M. Roncken, R. Saeijs, and F. Schali, "The VLSI-programming language Tangram and its translation into handshake circuits", EDAC'91
[15] IEEE Std. 1364-2001, IEEE Standard for Verilog Hardware Description Language, 2001
[16] Mentor Graphics, http://www.model.com/
[17] Cadence Design Systems, http://www.cadence.com/
[18] T. Bjerregaard, S. Mahadevan, and J. Sparsø, "A Channel Library for Asynchronous Circuit Design Supporting Mixed-Mode Modeling", PATMOS04
[19] http://www.systemverilog.org