SystemVerilogCSP: Modeling Digital Asynchronous Circuits Using SystemVerilog Interfaces

Arash SAIFHASHEMI 1, Peter A. BEEREL 2
Ming Hsieh Department of Electrical Engineering, University of Southern California

Abstract. This paper describes how to model channel-based digital asynchronous circuits using SystemVerilog interfaces that implement CSP-like communication events. The interfaces enable explicit handshaking of channel wires as well as abstract CSP events. This enables abstract connections between modules that are described at different levels of abstraction facilitating both verification and design. We explain how to model one-to-one, one-to-many, one-to-any, any-to-one and synchronised channels. Moreover, we describe how to split communication actions into multiple parts to model more accurately less concurrent handshaking protocols that are commonly found in many asynchronous pipelines.

Keywords. CSP, SystemVerilog, asynchronous circuits.

Introduction

In a digital circuit, often modelled as a network of interconnected modules, it is important to know when each module can safely sample input data and when the data at the output of each module is ready to be sampled by the next module. Channel-based asynchronous circuits are a family of digital circuits in which adjacent modules communicate and synchronise with each other via handshaking. Each module synchronises with its predecessors for receiving input data and with its successors for sending output data. Compared to synchronous circuits, replacing the clock network with handshaking signals can reduce the power consumption and improve performance [1].

Asynchronous circuit designers often use a CSP-like [2] language to specify a circuit’s intended behaviour at a high-level of abstraction. Therefore, one can consider asynchronous circuits to be a hardware implementation of CSP programs. Data transfers between asynchronous modules are modeled by CSP-like communication events: modules Receive (or Input) data on input channels, and Send (or Output) data on output channels. We use Send and Receive in this paper to model CSP-like output (!) and input (?) operators [2].

Several implementations of CSP have been suggested for modeling asynchronous circuits, among them are [3-9]. Ideally, such a hardware description should be via a standardised and widely used language supported by commercially available CAD tools. In addition, this language should also support timing and delay control constructs such that hardware performance (throughput and latency) can be analysed via simulation. Lastly,

---

1 Corresponding Author: Arash Saifhashemi, EEB 346, University of Southern California, 3740 McClintock Ave, Los Angeles, CA, 90089, USA; E-mail: saifhash@usc.edu
2 Peter A. Beerel is also Chief Scientist at Fulcrum Microsystems, Calabasas, California, 91302.
allowing models at several levels of abstraction (especially, high-level and gate level) is of
great utility as it facilitates a top-down design approach. There are several popular
software-based implementations of CSP [10-12]. But these languages do not have ample
low-level hardware design constructs for delay and switch-level modeling and are rarely
supported by hardware CAD tools. CHP, Communicating Hardware Processes [13], is a
concurrent programming notation inspired by CSP and Dijkstra’s guarded commands [14].
It is capable of modeling both high-level communication actions and low level switching
activities, but it lacks delay and timing constructs. Tangram [12] is a CSP-based language
for designing asynchronous circuits but it is supported by a very limited number of CAD
tools. In contrast, Verilog and VHDL are the most commonly used hardware description
languages (HDLs) for designing hardware, as they are standardised by IEEE and supported
by most CAD tools. Using these two languages facilitates mixed mode simulation of
asynchronous designs together with legacy synchronous circuits. However, these two
languages do not have inherent constructs for modeling CSP-like communication actions.
SystemC has also been applied [7, 15]. While SystemC is standardised, Verilog and VHDL
are better suited for structural and switch-level designs [16].

There are numerous methods in literature that attempt to customise Verilog and VHDL
with high-level CSP-like communication actions. Several authors [3-5] have developed
packages for VHDL to model CSP communication actions. Verilog has also been used [8,
9] due to its other powerful feature called fined-grained concurrency (nested begin-end and
fork-join blocks), by which processes can create multiple nested threads of execution. This
is a highly desirable feature for modelling asynchronous circuits. The initial Verilog
implementation [8] is based on library C functions, call, where a set of C functions calls is
hidden behind Verilog macros. Interfacing from Verilog to C, however, makes the
simulation speed very slow. A later implementation [9] omits the need for C functions,
however, still uses Verilog macros and adds extra bits to data ports for handshaking signals,
which complicates debugging and monitoring the status of channels. Also, it does not
support highly used handshaking protocols such as two-phase and 1-of-2 dual rail
handshaking protocols.

A number of researchers [17, 18] suggested using SystemVerilog [19] (a superset of
Verilog) interfaces to implement Send and Receive actions. A SystemVerilog interface is an
entity that can include a bundle of signals. The idea is to place all data and handshaking
signals inside an interface and define Send and Receive actions as the interface member
tasks. In this paper we will focus on developing such an interface. Compared to Verilog
implementations, using SystemVerilog interfaces provides modelling flexible CSP-like
communication actions based on multiple handshaking protocols. Also, it enables designers
to use mnemonic values for the status of channels [5], which facilitates debugging of the
circuits. Moreover, mixed-mode simulation of two communicating modules each at a
different level of abstraction [1, 5, 9] is possible without the need of instantiating extra
modules.

Compared to initial attempts to use SystemVerilog [17, 18], we will implement several
handshaking protocols so that modules described at a low level of abstraction (i.e. using
explicit handshaking signals) can communicate with modules described at high level of
abstraction (i.e. modules that use Send/Receive). Also, we implement shared channels, such
as one-to-many (broadcast), one-to-any, and any-to-one channels [10, 20]. We also improve
the modelling of split communication actions [1, 13] and extend it to support multiple
protocols. A split communication action is a form of modelling where one communication
event is split into multiple events. Using split communication, we show how to model
simultaneous and synchronised Receive actions on multiple inputs of a module (i.e. when a
module simultaneously receives from multiple ports but no Receive action shall start until
all senders are ready to Send). We further show how to accurately model modules that reshuffle handshaking events of Receive or Send on multiple ports.

Section 1 of this paper introduces SystemVerilog semantics and SystemVerilog interfaces. Section 2 presents the basic definitions of Send and Receive tasks. Section 3 describes the details and applications of split communication. Section 4 shows synchronised Receives on multiple input channels. Section 5 presents the implementation of shared channels. Section 6 includes performance evaluation, and Section 7 is summary and conclusions.

1. SystemVerilog

A SystemVerilog description, as explained in [19], consists of connected threads of execution or processes. Processes are objects that can be evaluated, that can have state, and that can respond to changes on their inputs to produce outputs. Processes are concurrently scheduled elements.

Every change in state of a net or variable in the system description being simulated is considered an update event. Processes are sensitive to update events. When an update event is executed, all the processes that are sensitive to that event are considered for evaluation in an arbitrary order. The evaluation of a process is also an event, known as an evaluation event.

SystemVerilog interfaces [19] encapsulate the implementation of communication actions and handshaking signals between modules. A circuit at the lowest level of abstraction can be described using a schematic diagram in which numerous wires connect adjacent modules. This diagram can equivalently be converted to a SystemVerilog netlist. Using SystemVerilog interfaces, one can describe a circuit while all wires between blocks are abstracted into interface connections. Moreover, one can define member tasks and functions for a SystemVerilog interface. In this paper, we will use SystemVerilog task construct to implement Send and Receive actions. Through a Send/Receive task pair we model CSP-like communication.

Consider the following CSP processes:

\[
\text{SENDER} = (\text{mid!}v \rightarrow \text{SENDER}) \\
\text{RECEIVER} = (\text{mid?}x \rightarrow \text{RECEIVER})
\]

Figure 1.a shows the graphic representation of a system consisting of an instance of each of these processes running concurrently: (s||r). Figure 1.b shows how a typical asynchronous circuit designer would implement such a system in hardware by using explicit handshaking signals for synchronisation of two modules. Using a handshaking protocol, the req signal ensures that the RECEIVER module waits for the SENDER to send data. The ack signal ensures that the SENDER does not progress until the RECEIVER receives the data. Figure 1.c shows a block diagram representation of these two modules based on SystemVerilog interfaces. In this representation, explicit handshaking signals are encapsulated in the SystemVerilog interface.

Figure 1. (a) CSP channel (b) schematic with wires (c) block diagram with interfaces.
The block diagram representation of module interconnects is similar to CSP’s channel representation in which the details of communication implementation are not revealed. Figure 2 shows the description of the SENDER module, RECEIVER module, and the TOP module connecting them together via a Channel interface in SystemVerilog. The always keyword is used to represent a general-purpose procedural and repetitive behaviour. The SystemVerilog $random system call is used in the SENDER module to generate random data.

```
module SENDER (interface R);
    parameter WIDTH = 8;
    parameter CT = 10;
    logic [WIDTH-1:0] v;
    always begin
        v=$random()%2**(WIDTH-1);
        R.Send(v);
        #CT;  //Delay
    end
endmodule

module RECEIVER (interface L);
    parameter WIDTH = 8;
    parameter CT = 10;
    logic [WIDTH-1:0] x;
    always begin
        L.Receive(x);
        #CT; //Delay
    end
endmodule

module TOP;
    Channel mid ();  //Interface definition
    SENDER   s (mid);
    RECEIVER   r (mid);
endmodule
```

Figure 2. (Left) SENDER module, (Right) RECEIVER module, (Bottom) TOP module.

From the SENDER’s point of view, R.Send is a task call on interface R; similarly, from RECEIVER’s point of view, L.RECEIVE is a task call on interface L. Both these modules do not need to know the type L or R interfaces. Module TOP, however, has to specify the type of the interface connecting SENDER and RECEIVER together. In this case, the interface type is called Channel, which we will define later.

### 2. Implementation of Communication Actions with Channel Interface

One can use one of many handshaking protocols to implement Send and Receive actions. In this section we present the implementation of Send and Receive using commonly used handshaking protocols in designing asynchronous circuits. First, we define two new data types: ChannelStatus and ChannelProtocol. ChannelStatus holds the status of the channel for debugging purposes [5]. Initially, we define a channel to have three possible status values. If a channel is idle, it means there is no activity on the channel, r_pend means a receiving process has called Receive and is waiting for the sending process. s_pend means a sending process has called Send and is waiting for the receiving process. ChannelProtocol will be used as a parameter for the channel, which specifies what handshaking protocol should be used for communication actions.

Figure 3 shows the definition of these two new user types together with a simplified definition of the Channel interface. The simplified interface has only two one-bit req and ack handshaking signals that can be used for both two and four-phase bundled data protocols [1]. Parameters WIDTH and hsProtocol specify the width of the data in the channel and the handshaking protocol respectively. The Channel interface is a shared resource: both Sender and Receive modules have access to all members of the interface.
Having handshaking protocols defined, we define Send and Receive tasks. Note that these tasks are member tasks of the interface and are defined inside the interface definition block. Figure 4 shows the basic definition for Send and Receive tasks using four-phase or two-phase handshaking protocols.

```verilog
task Send (input logic[WIDTH-1:0] d);  
if (hsProtocol == P4PhaseBD)  
begin  
data = d;  
req = 1;  
status = s_pend;  
wait (ack == 1 );  
req = 0;  
wait (ack == 0 );  
status = idle;  
end  
else if (hsProtocol == P2PhaseBD)  
begin  
data = d;  
status = hsPhase;  
wait (ack == hsPhase );  
status = idle;  
hsPhase = ~hsPhase;  
end  
endtask  

task Receive(output logic[WIDTH-1:0] d);  
if (hsProtocol==P4PhaseBD)  
begin  
status = r_pend;  
wait (req == 1 );  
d = data;  
ack = 1;  
wait (req == 0 );  
ack = 0;  
status = idle;  
end  
else if (hsProtocol == P2PhaseBD)  
begin  
status = r_pend;  
wait (req == hsPhase );  
d = data;  
ack = hsPhase;  
status = idle;  
end  
endtask
```

Handshaking signals are modified by one task and accessed by the other. The status variable, however, is modified by both tasks. Initially, the value of status is idle. Notice that there are no timing delays in these tasks. This implies the assignments in each task will be executed in sequence and atomically. If Send (Receive) task is called in one process after a Receive (Send) task has already been called in a corresponding process, the communication action takes place at zero time, i.e., both tasks will finish in zero time. However, if Send (Receive) is called in one process, but the corresponding Receive (Send) task has not been called yet, the status of the interface will be equal to the mnemonic value of s_pend (r_pend) and the Send (Receive) task will be blocked on the first wait statement. When both tasks finish, status will be set back to idle. This makes debugging easier as the designer can track the status of all channels in the design as illustrated in Figure 5.
Next, an implementation of *Probe* and *Peek* [13, 21] is presented. A process *probes* its port to see if some other process connected to that port has initiated a communication action on that port without committing to any communication action. Therefore, it can be defined as a Boolean function. Calling *Probe* is non-blocking. *Peek* is a blocking action when - just like *Receive* - a process blocks until the corresponding process initiates a *Send* action. Upon seeing the initiation of *Send*, the receiving process only samples the data without actually committing to *Receive*.

Implementing *Probe* and *Peek* turns out to be straightforward. In fact, *Probe* is just checking to see if the interface status is equal to *s_pend*. Figure 6 shows the *Peek* task definition.

```verbatim
task Peek (output logic[WIDTH-1:0] d);
    wait (status != s_pend );
    d = data;
endtask
```

Figure 6. Implementation of Peek.

It is worthwhile showing how using the *Channel* interface one can use the same testbench module for testing a buffer at different levels of abstraction. Figure 7 shows the structural detailed design of a micropipeline buffer [22] as well as a diagram of a testbench testing the buffer using *Channel* interface.

Figure 8 presents the description of the micropipeline buffer at three different levels of abstraction: *mp_fb_csp*, *mp_fb_dataflow*, and *mp_fb_struct*. Notice that except for *mp_fb_csp*, the other two types of the buffer directly access the interface signals. *mp_fb_struct* is the direct translation of Figure 7 (Left) into SystemVerilog. Figure 8 (Bottom) shows the testbench module. The *BUFFER* macro specifies which type of buffer should be used. Regardless of which type of buffer is chosen, the description of the *testbench* module remains the same. This facilitates an efficient top-down design and verification paradigm in which units are initially behaviourally described using CSP and then individually refined into their structural implementations. A similar approach has been used in the Proteus asynchronous synthesis framework [23], where instead of SystemVerilog, the circuit is designed in a proprietary CSP-like language [24]. In this design flow, the synthesised transistor-level netlist is co-simulated against the original high-level description. The same input stimuli are applied to both circuits and the output results of both circuits are expected to be the same. Using SystemVerilog, we are trying to enable the designer to use a standard hardware description language to achieve the same results.
module mp_fb_csp (interface L, interface R);
logic data;
always begin
    L.Receivedata();
    R.Send(data);
end
endmodule

module mp_fb_struct (interface L, interface R);
  celement ce (L.req, pd_bar, c);
  not   inv (pd_bar, pd);
  cap_pass cp (c, L.ack, R.ack, pd, L.data, R.data);
endmodule

module mp_fb_dataflow (interface L, interface R);
  CPState state = ST_PASS;
  logic phase = 1;
  always begin
    wait(L.req==phase && R.ack==!phase);
    L.ack = (phase);
    R.req = (phase);
    state = ST_CAPTURE;
    wait(R.ack==phase);
    state = ST_PASS;
    phase = ~phase;
    R.phase = ~ R.phase;
    always @(state, L.data)
      if (state== ST_PASS)
        R.data = L.data;
  end
endmodule

module testbench;
Channel  a(), b(); //Interface definition
Sender  s   (a);
`BUFFER  b0   (a,b);
Receiver  r   (b);
endmodule

Figure 8. Micropipeline buffer at (Left) CSP and structural levels, (Right) dataflow level, (Bottom) testbench module.

3. Split Communication

In asynchronous circuits it is very common to interleave the handshaking actions of two or more channels [1, 13]. For example, enclosed handshaking [1] is one form of communication action in which the handshaking on one channel is enclosed within the handshaking of another one. Figure 9 shows a Call module [13] that encloses the handshaking actions of port R into the handshaking actions of port L using two-phase handshaking protocol. This module first waits for a value change on Lreq. Then it flips the value of Rreq, without acknowledging L (i.e., without finishing the Receive). Only after finishing the communication action on R, does it flip the value of Lack to acknowledge the sender on port L. Notice that in Figure 9 two complete cycles of communication actions on ports L and R are shown.

It is not possible to accurately model the behaviour of this module in an abstract way only with Send/Receive tasks. Some researchers [1, 17] suggest using split communication actions in which a communication action is split into several events. A communication action is considered complete when all of these events happen. Figure 10 shows SplitReceive and SplitSend implemented using either two or four-phase handshaking protocols. The second argument of these tasks specifies which part of the handshaking action should be executed. Figure 10 also shows the high level description of the Call module.
### Figure 9. Call module using two-phase enclosed handshaking.

```
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
</table>
| task SplitReceive (output logic[WIDTH-1:0] d, input integer part);
case(hsProtocol)
P4PhaseBD: begin
case (part)
1: begin
  status = r_pend;
  wait (req == 1);
end
2: begin
  d = data;
  ack = 1;
end
3: begin
  wait (req == 0);
end
4: begin
  status = idle;
end
caseend
endcase
end //P4PhaseBD
P2PhaseBD: begin
   case (part)
   1: begin
     status = r_pend;
     wait (req == hsPhase);
     d = data;
   end
   2: begin
     ack = hsPhase;
     status = idle;
   end
caseend
endcase
endtask
```

```
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
</table>
| task SplitSend (input logic[WIDTH-1:0] d, input integer part);
case(hsProtocol)
P4of2: begin
  case (part)
  1: begin
    data = d;
    req = 1;
    status = s_pend;
  end
  2: begin
    wait (ack == 1);
  end
  3: begin
    req = 0;
  end
  4: begin
    wait (ack == 0);
    status = idle;
  end
caseend
endcase
end //P4PhaseBD
P2PhaseBD: begin
   case (part)
   1: begin
     data = d;
     req = hsPhase;
     status = s_pend;
   end
   2: begin
     wait (ack == hsPhase);
     status = idle;
     hsPhase = ~hsPhase;
   end
caseend
endcase
end //P2PhaseBD
dendtask
```

### Figure 10. (Top Left) SplitReceive, (Top Right) SplitSend, and (Bottom) Call Module.

```
module Call (interface left, interface right);
parameter WIDTH = 8;
logic [WIDTH-1:0] data;
always
begin
  left.SplitReceive (data, 1);
  right.Send (data);
  left.SplitReceive (data, 2);
end
endmodule
```
Here we present another application of split communication. A commonly used, fast, and stable type of pipeline stages in asynchronous fined grain pipelines is called the pre-charged half buffer, \textit{PCHB} [17]. A \textit{PCHB} pipeline stage is implemented using dual-rail four-phase handshaking [13]. This buffer is described as \(*[L?x;R!x]\) in \textit{CHP} [13] using CSP-like input and outputs: The buffer \textit{Receives} \(x\) from \(L\), and \textit{Sends} it to \(R\). A lower level description of this buffer in \textit{CHP}, where the handshaking protocol is explicitly described, can be defined as:

\[
*[
\begin{array}{l}
[l_0 \lor l_1]; [l_0 \rightarrow r.0 \uparrow \times [l_1 \rightarrow r.1 \uparrow \times \text{Lack}\uparrow]; \\
[R_{\text{ack}}]; r.0 \downarrow, r.1 \downarrow;[-l_0 \land -l_1]; L_{\text{ack}} \downarrow;[-R_{\text{ack}}]
\end{array}
\].
\]

In this notation, \(*[\ ]\) means repeat the statements inside the brackets for ever. For a binary variable \(v\), \([v]\) means wait (and block) until the value of \(v\) is one. The notation \([\neg v]\) means wait (and block) until the value of \(v\) is zero. The notation \(v \uparrow\) means set the value of \(v\) to one. Similarly, \(v \downarrow\) means set the value of \(v\) to zero. In the above notation, a dual-rail four-phase handshaking protocol is used. The signals \(l_0, l_1,\) and \(L_{\text{ack}}\) are handshake signals for channel \(L\). Similarly, \(r_0, r_1,\) and \(R_{\text{ack}}\) are handshake signals for channel \(R\). The symbol \(\lor\) represents logical OR, and the \(\land\) symbol represents logical AND.

Figure 11 shows a full handshaking cycle on \(L\) and \(R\). Notice how handshaking phases of \(L\) are interleaved with \(R\) in the above \textit{CHP} program. Figure 11 also shows how each phase of handshaking can be considered one part of the split communication. On the input side when data is available, either \(l_0\) or \(l_1\) is high. If \(l_0\) is high, the input has Boolean value 0, if \(l_1\) is high, the input has Boolean value 1. After receiving a valid data, the input is acknowledged and the process waits until both \(l_0\) and \(l_1\) become low. At this point the acknowledge signal returns to zero.

\[
\begin{array}{|c|c|c|}
\hline
\text{Receive}_1 & \text{Send}_1 & \text{Receive}_2 \\
[l_0 \lor l_1] & r.0 \uparrow \text{ or } r.1 \uparrow & [R_{\text{ack}}] \\
L_{\text{ack}} \uparrow & \text{Send}_2 & r.0 \downarrow, r.1 \downarrow \\
\text{Receive}_3 & \text{Send}_3 & [-R_{\text{ack}}] \\
[-l_0 \land -l_1] & \text{Receive}_4 & [R_{\text{ack}}] \\
L_{\text{ack}} \downarrow & \text{Send}_4 & \\
\hline
\end{array}
\]

Figure 11. Split communication for the four-phase handshaking protocol used in \textit{PCHB}.

The SystemVerilog description of this buffer using split communication is presented in Figure 12. Compared to the original \textit{CHP} representation, *\([L?x;R!x]\)*, the description of Figure 12 can more accurately capture the interaction and synchronisations of the buffer with its environment, while still abstracting the implementation details. Notice that a top-level module such as a testbench can still communicate with the \textit{PCHB} buffer of Figure 12 using the non-split communication actions Send/Receive. Also, notice that the dual-rail implementation of split communication is not shown in Figure 10 to save space.
4. Synchronised Receives

Often a receiver module needs to Receive values from multiple input ports by calling multiple Receive tasks in parallel. Figure 13 (left) shows an Adder module that concurrently receives its inputs from two input ports. Using a SystemVerilog fork-join construct, both Receives can be executed concurrently. The simulator starts both Receives at the same time. The control then moves to the line after join when both Receives are done. However, each Receive can be executed independently, i.e., if one of the Receives is blocked, the other can still complete.

```verilog
module Adder1 (interface A, interface B, interface SUM);
parameter WIDTH = 8;
logic [WIDTH-1:0] a=0,b=0,sum=0;
always
begin
fork
A.Receive(a);
B.Receive(b);
join
sum = a + b ;
SUM.Send(sum);
end
endmodule
```

```verilog
module Adder2 (interface A, interface B, interface SUM);
parameter WIDTH = 8;
logic [WIDTH-1:0] a=0,b=0,sum=0;
always
begin
fork
A.Receive(a, 1);
B.Receive(b, 1);
join
fork
A.Receive(a, 2);
B.Receive(b, 2);
join
sum = a + b ;
SUM.Send(sum);
end
endmodule
```

Figure 13. (Left) Independent Receives (Right) Synchronised Receives with two-phase handshaking.

The hardware implementation of such concurrent Receives, however, is sometimes slightly different. Usually, a Muller C-Element [25] is used to synchronise the handshake of both ports. Therefore, if one Sender is late, both Receives get blocked until both senders commit to Send. A sample micropipeline [22] join stage is shown in Figure 14 [1].

There are two possible methods to implement such behaviour accurately. The first approach is to Probe both input channels and commit to Receive on either of the channels only when the status of both channels is not idle. Alternatively, we can use split communication as shown in Adder2 described in Figure 13 (Right).
Shared Channels

Shared channels are channels where each end of the channel is allowed to be connected to multiple processes [2, 20, 26]. The formal CSP semantics requires those processes to interleave their use of the shared channel-end and defines the interleaving operator [2]. In this section, we present modelling of one-to-many (Broadcast) and one-to-any channels in SystemVerilog that behave similar to the ones specified in [10, 20]. Any-to-one channels can be modelled in a similar manner that one-to-any channels are modelled by modifying the Send task. Therefore, we will not present the detailed description of this channel type to save space.

5.1 One-To-Many (Broadcast) Channels

It is common for a module to Send a value to multiple receivers. In asynchronous circuits this is often done using an explicit copy module which Receives a value from its input port and Sends that value to its multiple output ports. Figure 15 shows the description of an explicit copy module with two outputs. A fork-join construct is used to execute both Send commands in parallel.

The use of an explicit copy module is tedious and makes debugging harder, since the designer has to instantiate a separate copy module for each case where there is one sender and multiple receivers. Moreover, often the implementation of the copy module behaves differently than the high level CSP description of it. In many implementations a Muller C-Element [25] is used to synchronise all communication actions. In the copy2 module of Figure 15, however, each Send (and hence the corresponding Receive) can execute independently. Using broadcast channels [10], the Send action of the sender and all Receive actions of all receivers are synchronised using barriers. Figure 16 shows a similar implementation of a broadcast channel in SystemVerilog. A counter shared by all receivers is used to keep track of the number of receivers executing the Receive task.
Each receiver first checks to see if the Sender has started by waiting on \textit{req} signal to become high. Then it saves the input data and checks the counter to see if it is the last receiver. If not, it increments the counter, set the status value to \texttt{s\_pend\_1toMany}, and then waits until the counter is reset back to zero. Otherwise, if it is the last receiver, it finishes the handshake with the sender and also resets the counter back to zero. The parameter \texttt{NUMBER\_OF\_RECEIVERS} is defined in the interface. It is set to the number of receivers upon the interface instantiation. The default value of this parameter is 1 for point-to-point channels. Notice that we can avoid using barriers [10] since SystemVerilog is an event driven language [19]. That is, the new value of \texttt{receiveCounter} (an \textit{update event}) will be seen by all other receivers \textit{before} they compare it to \texttt{NUMBER\_OF\_RECEIVERS} (\textit{evaluate event}). This is because in SystemVerilog \textit{update events} have higher priority than \textit{evaluate events}. If all receivers execute \texttt{Receive} at the same time, the simulator evaluates one of the receivers in an arbitrary order. Upon executing the increment of the counter, an update event will be scheduled that will update the value of the \texttt{receiveCounter} before all other receivers evaluate the if statement. The last receiver unblocks all other receivers as well as the sender. Figure 17 shows an example circuit followed by its SystemVerilog representation using a broadcast channel. The \texttt{data\_generator} module \texttt{dg} generates data and \texttt{Sends} it to its output port which is connected to three input ports belonging to \texttt{top}, \texttt{middle}, and \texttt{bottom} buffers. The output port of each buffer is connected to a \texttt{data\_bucket} module that \texttt{Receives} data from its input port and has no output port.
5.2 One-to-any Channels

A different type of channel, one-to-any, is discussed and implemented in [20]. This type of channel is between several receiving processes and only one sending process. Receiving processes compete with each other over using the channel. Only one receiver and the sender will be engaged in communication and actually use the channel at any one time.

Here we show how to model a similar behaviour in SystemVerilog. Figure 18 shows a modified version of basic Receive task, such that upon detecting the request from the sender, the value of req signal is changed to z (high-impedance value in SystemVerilog). This inhibits other Receiving processes from receiving, as they will be blocked on the wait statement. The wait statement compares req to hsPhase, where the latter has a binary value. Note that this was again possible due to prioritisation of update events to evaluate events in SystemVerilog. A new Boolean parameter called ONE2ANY is added to the interface definition that indicates whether the channel is a ONE2ANY channel or not. Also, notice that Figure 18 only shows the implementation using two-phase handshaking protocol.

```verilog
module OneToManyChannelExample;
    Channel #( .NUMBER_OF_RECEIVERS(3) ) left();
    Channel topRight(), midRight(), botRight();
    data_generator dg (left);
    buffer top (left, topRight);
    buffer middle (left, midRight);
    full_buffer bottom (left, botRight);
    data_bucket dbt (topRight);
    data_bucket dbb (botRight);
    data_bucket dbm (midRight);
endmodule
```

Figure 17. (Top) One-To-Many channel example, (Bottom) SystemVerilog implementation.

```verilog
task Receive(output logic[WIDTH-1:0] d);
    status = r_pend;
    wait (req == hsPhase);
    if (ONE2ANY)
        req = 'z; // Inhibits other receivers from receiving
    d = data;
    ack = hsPhase;
    status = idle;
endtask
```

Figure 18. Receive task for one-to-any channels.
The simple modification shown in Figure 18 changes the behaviour of Receive in the following way: If there is only one receiving process waiting on the wait statement for the req signal to change, that receiver participates in handshaking with the sender and they both engage in communication of data. If more than one receiver are waiting for the req signal to change, however, SystemVerilog semantics require that an arbitrary receiving process to be executed. As soon as this receiver is executed, it generates an update event for req signal indicating its new value to be z. This turns the comparison of the req signal in other processes with hsPhase to be false, and keeps those processes blocked on the wait statement. Therefore, similar to [20], the fairness [2] of Receive action between two competing receivers depends on the SystemVerilog simulator implementation and is not guaranteed.

In Figure 19 a special receiver module that receives with 50% probability is shown. Using SystemVerilog’s $random system call, if the channel status is not idle, the process commits to Receive with 50% probability. For two competing receivers, this makes the probability of one receiver starving the other for a prolonged period of time increasingly small. Note that based on SystemVerilog semantics, an explicit #0 delay, used in Figure 19, suspends the process and allows other suspended processes to evaluate and progress. This way, if $random call in one process dictates not to receive, the control is passed to other receiving processes (in an arbitrary order). Those processes each will decide whether to receive based on the result of their own $random system call.

6. Performance Evaluation

In this section, we present the experimental results for evaluating the performance of this method in terms of simulation time of a test circuit. We compared the simulation time of this method with that of VerilogCSP [9]. A simple linear pipeline consisting of one sender (Figure 2), ten buffers (Figure 8), and one receiver (Figure 2), was designed using both methods. The output of the sender is connected to the first buffer. The output of the (i)th buffer is connected to the input of (i+1)th buffer, and the output of the 10th buffer is connected to the receiver. Delays have been added to all modules such that each module has a local cycle time [1] of 10 time units. We simulated each circuit for different numbers of data items sent through the pipeline, as shown in Table 1. We used the ModelSim SE 6.6b simulator running on a Sun UltraSPARC based mainframe with the Sun Solaris 10 (10/08) operating system. The results show that SystemVerilog implementation is 12% to 20% faster than VerilogCSP. Although this data shows a slight increase in efficiency as the number of data items grow, experiments with larger number of data items do not show

---

```verilog
module RECEIVER (interface L);
    parameter WIDTH = 8;
    logic [WIDTH-1:0] x;
    integer randValue;
    always
       begin : main
          wait (L.status != idle);
          randValue = {$random()} % 3 ;
          if (randValue ==1)
             L.Receive(x);
          else
             begin
                #0;
                disable main;
             end
        end
endmodule

Figure 19. A receiver that receives from a one-to-any channel with 50% probability
```
increased efficiency and instead confirmed this range. The simulation run-time gains may be because of the somewhat unorthodox use of the Verilog `force` construct in [9] which may limit internal optimisations compared to our more natural use of SystemVerilog.

<table>
<thead>
<tr>
<th>Number of data items</th>
<th>100K</th>
<th>200K</th>
<th>300K</th>
<th>400K</th>
<th>500K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulation time in seconds (VerilogCSP)</td>
<td>45.14</td>
<td>76.38</td>
<td>107.60</td>
<td>139.57</td>
<td>170.62</td>
</tr>
<tr>
<td>Simulation time in seconds (SystemVerilogCSP)</td>
<td>40.12</td>
<td>65.00</td>
<td>89.70</td>
<td>115.52</td>
<td>141.99</td>
</tr>
<tr>
<td>Ratio</td>
<td>1.12</td>
<td>1.17</td>
<td>1.19</td>
<td>1.20</td>
<td>1.20</td>
</tr>
</tbody>
</table>

### 7. Summary and Conclusions

In this paper, we presented the implementation of high-level CSP-like communication actions in SystemVerilog. Compared to previous Verilog implementations, this method is more flexible, facilitates easier debugging, and supports a wide range of handshaking protocols. We presented the implementation of a split `Send` and `Receive` and the applications of split communication to more accurately describe commonly used asynchronous modules at a higher level of abstraction. We also showed how to implement shared channels and synchronised `Receives` on multiple input ports of a module.

We have compared the simulation time of this implementation to that of [9] and found that it decreases simulation time by 12% to 20%.

The SystemVerilog code introduced in this paper is called SystemVerilogCSP. This package can be downloaded for researches from our website: [http://jungfrau.usc.edu/](http://jungfrau.usc.edu/), and it is currently being used to teach the course EE-552 Asynchronous VLSI at the University of Southern California.

### References


