Modeling FIFO Communication Channels Using SystemVerilog Interfaces

Stuart Sutherland
Sutherland HDL, Inc.
stuart@sutherland-hdl.com

ABSTRACT

The Verilog Hardware Description Language (Verilog HDL) does not have a direct equivalent to SystemC channels. These channels are often used for modeling abstract, high-level communications between modules. This paper shows how the SystemVerilog extensions to Verilog can be used to model high-level communication channels.

SystemVerilog extends the Verilog HDL with a powerful ***interface*** construct. This construct provides a way to encapsulate the communication between the major blocks of a design. Interfaces can be used for more than just data encapsulation, however. An interface can also contain procedural code, tasks and functions. This paper provides a brief tutorial on SystemVerilog interfaces, and then delves into one way in which interfaces can be used for high-level modeling. A FIFO channel is used as the basis for examples in this paper, but the concepts presented can be applied to other types of communication channels, such as a mutex channel. The FIFO behavior is modeled using other powerful SystemVerilog constructs: mailboxes and queues. The interface FIFO channel is modeled to be reconfigurable; it can be configured to pass data of any data type, including integers, reals, vectors of any size, and user-defined types. The paper also discusses synthesizing SystemVerilog interfaces and the modeling constructs used in the examples shown.

Table of Contents

1.0 SystemC channels versus SystemVerilog interfaces 3
2.0 SystemVerilog interface tutorial 4
  2.1 Referencing signals defined within an interface 5
  2.2 Interface module port declarations (modports) 5
  2.3 Selecting modports 5
  2.4 Interface methods 7
  2.5 Interface processes 8
  2.6 Parameterized SystemVerilog interfaces 8
3.0 Modeling FIFO channels using SystemVerilog mailboxes 9
  3.1 Overview of SystemVerilog mailboxes 9
  3.2 A FIFO channel using mailboxes 11
  3.3 Connecting and using the interface channel 13
  3.4 Synthesis considerations with mailboxes 14
4.0 Modeling FIFO channels using SystemVerilog queues 15
  4.1 Overview of SystemVerilog queues 15
  4.2 An abstract FIFO channel using queues 16
  4.3 Connecting and using the interface channel 17
  4.4 Synthesis considerations with queues 17
5.0 A synthesizable FIFO channel using queues 18
  5.1 Connecting and using the interface channel 21
  5.2 Synthesis considerations 21
6.0 Conclusions 22
7.0 Future work 23
8.0 References 24
9.0 About the author 24

List of Figures and Examples

Figure 1: Inter-module communication using a FIFO 4
Figure 2: Verilog Ports vs. Interface Ports 5
Example 1: SystemVerilog Interface with modports 6
Example 2: Interface modport selection as part of the module instance 7
Example 3: Interface modport selection as part of the module definition 7
Example 4: Interface methods imported into modules 8
Example 5: Calling imported interface methods 9
Example 6: Interface with procedural code 9
Example 7: Parameterized interface FIFO size 10
Example 8: Parameterized interface data types 10
Figure 3: UNI and NNI ATM switch data packets 13
Example 9: SystemVerilog definitions for UNI and NNI ATM switch data packets 13
Example 10: Complete SystemVerilog FIFO interface channel using SystemVerilog mailboxes 14
Example 11: Netlist connecting two modules using the FIFO interface channel 15
Example 12: Abstract SystemVerilog FIFO interface channel using SystemVerilog queues 18
Example 13: Synthesizable SystemVerilog FIFO interface channel using queues 20

1.0 SystemC channels versus SystemVerilog interfaces

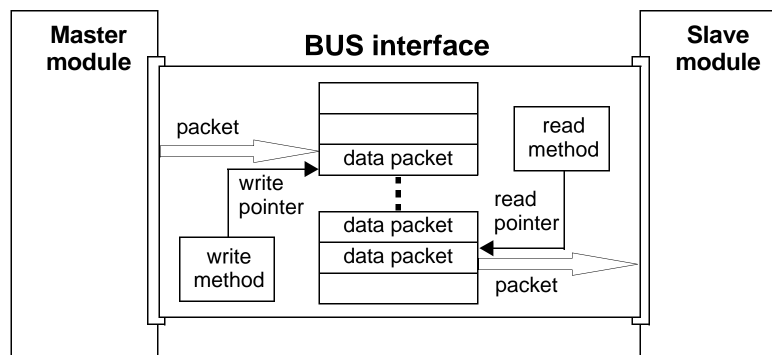
SystemC provides *channels* and *interfaces* for communication between modules. A SystemC *channel* encapsulates how information is transferred between modules. A SystemC *interface* defines a set of methods (functions) for a channel. These methods are used to send and receive information through a channel, and to probe and control a channel. Channels can be user defined, and can be built from other SystemC constructs, such as ports, module instances, other channels, and processes. The current SystemC standard, 2.0.1[1] also includes a number of pre-defined channels, such as `sc_signal`, `sc_fifo`, `sc_mutex`, and `sc_sempahore`. In general, these pre-defined SystemC channels are not considered to be synthesizable[2]. They do not contain the level of logic detail required for synthesis tools to realize the intended hardware implementation.

SystemVerilog adds an *interface* construct to the Verilog language. A SystemVerilog interface is essentially the same as a SystemC channel. An interface encapsulates the communication information between Verilog modules. This encapsulation can include the module port definitions, tasks, functions, always blocks, continuous assignments, assertions, and other modeling constructs. Interfaces can also include instances of other interfaces, allowing more complex, hierarchical interfaces to be modeled.

Unlike SystemC, SystemVerilog does not provide a set of pre-defined communication channels. SystemVerilog provides the general purpose interface construct, enabling designers to model any type of communication functionality. SystemVerilog also provides a number of powerful extensions to Verilog that make it easy to model complex communications between modules at a higher level of abstraction than is possible with Verilog. These extensions include *semaphores*, *mailboxes* and *queues*.

This paper shows how the functionality of one of the more complex SystemC built-in channels, a FIFO (First-In, First-Out), can be easily modeled using SystemVerilog interfaces. FIFOs are typically used to communicate data between design blocks that are operating with different, asynchronous clocks. Both high-level abstract versions of a communication channel are presented, as well as synthesizable versions. The concepts shown in this paper can be readily adapted to a variety of other communication channel types.

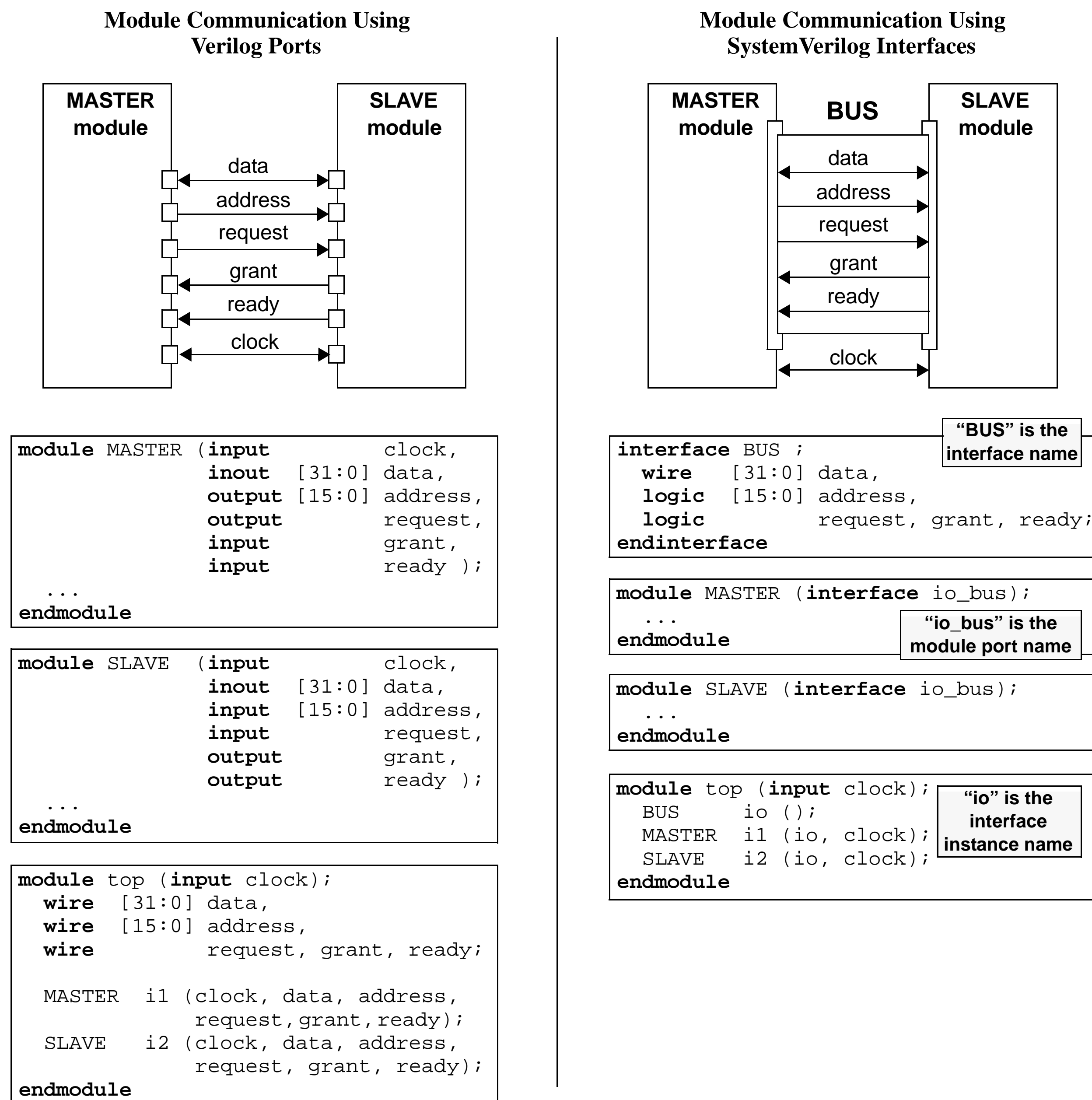
Figure 4. Inter-module communication using a FIFO



2.0 SystemVerilog interface tutorial

The basic building block for a SystemVerilog interface is the keyword pair **`interface`**... **`endinterface`**. This keyword pair is used to define a separate structural block, similar to a Verilog module. In its most basic form, an interface simply encapsulates the signals that are used to communicate between Verilog modules. The interface block is then used as a module port, replacing multiple discrete ports for each communication net. Figure 5 contrasts module interconnections using Verilog module ports with a basic SystemVerilog interface. Observe the usage of the **`interface`**...**`endinterface`** keyword pair, and how the `MASTER` and `SLAVE` modules use the interface as a module port.

Figure 5. Verilog Ports vs. Interface Ports



2.1 Referencing signals defined within an interface

Modules that are connected to an interface can reference the interface signals by using a relative hierarchical path name. The path name is formed by prepending the name of the interface port to the signal name. For example:

```
module MASTER (interface io_bus);   // "io_bus" is the port name
  always @(io_bus.request)          // "request" is inside the interface
    ...
```

2.2 Interface module port declarations (modports)

Each module connected to an interface may need to see a different view of the signals within the interface. In the example above, the `request` signal is an output from the `MASTER` module, and is an input to the `SLAVE` module. SystemVerilog interfaces provide a means to define different views of an interface. The definition is made within the interface, using the **`modport`** keyword, which is an abbreviation for *module port*. An interface can have any number of modport definitions, each describing a different view of the signals within the interface. Examples of two modport declarations are:

Example 14. SystemVerilog Interface with modports

```
interface BUS ;
  wire   [31:0] data;
  logic  [15:0] address;
  logic         request, grant, ready;

  modport master_ports (inout  data,
                        output address,
                        output request,
                        input  grant,
                        input  ready );

  modport slave_ports  (inout  data,
                        input  address,
                        input  request,
                        output grant,
                        output ready );

endinterface
```

2.3 Selecting modports

SystemVerilog provides two ways to specify which modport a module interface port should use:

- As part of the interface connection to a module instance
- As part of the module port declaration in the module definition

Selecting the modport at the module instance

When a module is instantiated and an instance of an interface is connected to a module instance port, the specific modport of the interface can be specified, as illustrated in Example 15.

Example 15. Interface modport selection as part of the module instance

```
interface BUS ;
  ...
  modport master_ports (...);
  modport slave_ports  (...);
endinterface
```

```
module MASTER (interface io_bus); // generic interface port
  ...
endmodule
```

```
module top (input clock);
  ...
  BUS    io ();                        // instance of an interface

  MASTER i1 (io.master_ports, clock);  // connect I/F using master modport
  ...
endmodule
```

Selecting the modport at the module definition

An alternate style of selecting the modport view of an interface is to specify which modport of an interface is to be used directly as part of the module port declaration. This style requires that the module port be declared using the interface name, rather than using the `interface` keyword.

Example 16. Interface modport selection as part of the module definition

```
interface BUS ;
  ...
  modport master_ports (...);
  modport slave_ports  (...);
endinterface
```

```
module SLAVE (BUS.slave_ports io_bus, // interface port using slave modport
              input clock);           // other ports
  ...
endmodule
```

```
module top (input clock, resetN);
  ...
  BUS   io ();            // instance of an interface

  SLAVE i2 (io, clock);    // connect to interface
  ...
endmodule
```

There is no particular advantage to either of the ways of specifying which modport should be used. SystemVerilog simply provides two different modeling styles.

2.4 Interface methods

An interface can be used not only to encapsulate the data connecting modules, but also the communication protocols between the modules. SystemVerilog allows Verilog tasks and functions to be declared within an interface. These tasks and functions are referred to as *interface methods*. Interface methods can operate on any of the signals within the interface. Values can be passed into interface methods from outside the interface as input arguments. Values can be written out from interface methods as output arguments or function returns.

Interface methods offer several advantages for modeling large designs. The code for communicating between modules does not need to be distributed across multiple modules. Instead, the communication code is encapsulated in the interface, and shared by each module connected using the interface. The modules connected to the interface simply call the interface methods, instead of having to implement the communication protocol functionality within the modules.

In order for a module to call an interface method, the method name should be imported into the module. This is done by using the **`import`** keyword as part of a modport definition. Modports specify interface information from the perspective of the module. Hence, an import declaration within a modport indicates that the module is importing the task or function from the interface.

Example 17. Interface methods imported into modules

```
interface BUS (input clock, resetN);
  wire   [31:0] data;
  logic  [15:0] address;
  logic         request, grant, ready;

  task Read (...);
    ...
  endtask

  task Write (...);
    ...
  endtask

  function bit ParityGen (...);
    ...
  endfunction

  modport master_ports (import Read, Write,
                        input  ... );

  modport slave_ports  (import Read, Write, ParityGen,
                        input  ... );

endinterface
```

An imported task or function is called by prepending the interface port name to the task or function name.

Example 18. Calling imported interface methods

```
module SLAVE (BUS.slave_ports io_bus); // interface port
  ...
  always @(posedge io_bus.clock) begin
    if (io_bus.request)
      io_bus.Read(...);  // call Read method in interface
    ...
  end
  ...
endmodule
```

2.5 Interface processes

In addition to methods (tasks and functions), interfaces can contain communication functionality that can be described using `always` procedural blocks and `assign` continuous assignment statements. An interface can also contain verification code using `program` blocks, `initial` blocks, and `final` blocks.

Example 19. Interface with procedural code

```
interface fifo_channel ;
  ...
  // calculate fifo_empty flag
  always_ff @(posedge read_clock, negedge read_resetN) begin
    if (!read_resetN) fifo_empty <= 0;
    else              fifo_empty <= (rd_ptr_next == rd_ptr_synced);
  end
  ...
endinterface
```

2.6 Parameterized SystemVerilog interfaces

SystemVerilog interfaces can be parameterized, in the same way as Verilog modules. This allows model details such as a FIFO size to be based on parameter constants. Each time the interface model is instantiated, the values of parameters can be redefined. The following example illustrates a parameterized interface, and an instance of the interface where the size of the FIFO and FIFO pointers are redefined. This example utilizes the Verilog-2001 style of parameter declarations and parameter redefinition. The older Verilog-1995 style can also be used with interfaces.

Example 20. Parameterized interface FIFO size

```
interface fifo_channel #(parameter FifoSize = 8, PtrSize = 4);
  ...
  bit [PtrSize:0] write_pointer, read_pointer;
  ...
endinterface
```

```
module top_level (input clock, resetN);
  ...
  fifo_channel #(.FifoSize(16), .PtrSize(5)) FIFO (); // interface instance
  ...
endmodule
```

SystemVerilog extends the capabilities of Verilog parameters to also allow data types to be parameterized. This is done by declaring the parameter constant with the keyword pair **parameter type**. Parameterized types substantially enhances the ability to write reconfigurable Verilog models. Each instance of a module or interface can be reconfigured to work with completely different types of data. An example of an interface declaration with both a parameterized size and parameterized types is shown below. Instances of the interface are also shown, that reconfigure the interface declaration. The FIFO interface examples presented later in this paper utilize parameterized, reconfigurable interfaces (see Examples 23, 25 and 26).

Example 21. Parameterized interface data types

```
interface fifo_channel #(parameter FifoSize = 8, PtrSize = 4,
                         parameter type DataType = int );
  ...
  DataType write_data;  // packet coming from sending module
  DataType read_data;   // packet going to receiving module
  ...
endinterface
```

```
module top_level (input clock, resetN);
  ...
  fifo_channel #(.DataType(real)) FIFO (); //  interface instance
  ...
endmodule
```

3.0 Modeling FIFO channels using SystemVerilog mailboxes

3.1 Overview of SystemVerilog mailboxes

SystemVerilog adds Object-Oriented programming to Verilog, along with object class definitions, object methods, object inheritance, and other object-oriented capabilities.

SystemVerilog includes a predefined mailbox class object to the Verilog language. A mailbox is a communication mechanism whereby data can be transferred from one process to another process.

Mailboxes behave somewhat like real-life mailboxes. A "*message*" (a value in the form of a Verilog or SystemVerilog data type) can be delivered to a mailbox. That message can then be retrieved from the mailbox by the same or a different process. Messages are retrieved in the order in which they were delivered, similar to a FIFO.

A SystemVerilog mailbox is a predefined class object. Some of the features of mailboxes are:

- Mailboxes are created using the object-oriented `new()` method.
- Mailboxes can be defined to allow messages of any data type to be delivered and retrieved.
- Mailboxes can be defined to only allow messages of a specific data type to be delivered and retrieved.
- Mailbox capacity can be unconstrained; that is, any number of messages can be delivered without retrieving messages.
- Mailbox capacity can be constrained to have a maximum capacity.
- Pre-defined methods are provided to deliver messages, retrieve messages, and to examine the mailbox contents.

A mailbox is created by calling the mailbox class **`new()`** method. When `new()` is called, it can optionally be passed in a constant value that indicates the maximum capacity of the mailbox. If the maximum capacity is not specified, then the mailbox is unconstrained, and can hold any number of messages. Two examples of declaring a mailbox are:

```
mailbox DataChannel = new;        // unbounded mailbox

mailbox PacketChannel = new(5);   // bounded mailbox, max of 5 packets
```

By default, SystemVerilog mailboxes can store messages consisting of any Verilog or SystemVerilog data type. However, it is illegal to place a value into the mailbox as one data type, and try to retrieve the value as a different data type. Therefore, the code using the mailbox must keep track of what type of information is being passed through the mailbox. Mailboxes can also be typed, making it so that only values of a specific data type can be passed through the mailbox. This eliminates the burden of the code needing to keep track of the message types. Explicitly defining the mailbox type is done as part of the call to the **`new()`** function, as shown in the following examples:

```
mailbox #(int) DataChannel = new;       // unbounded mailbox of int types

typedef struct {int a, b, real y} packet_t;
mailbox #(packet_t) PacketChannel = new(5);  // bounded mailbox of
                                             // packet_t structure types
```

The mailbox class object includes several methods for passing messages through the mailbox. These are:

- **`put(`**`<message>`**`)`** — a task that places a message in a mailbox. If the mailbox is bounded, the process is suspended (blocked) until there is room in the mailbox to place the message.
- `status = `**`tryput(`**`<message>`**`)`** — a function similar to `put()`, except that the process does not suspend if the mailbox is full. If the mailbox is not full, the message is delivered and `tryput()` returns 1. If the mailbox is full, the message is not delivered, and `tryput()` returns 0.

- **get(**<variable>**)** — a task that retrieves a message from a mailbox. If the mailbox is empty, the process is suspended (blocked) until a message is placed in the mailbox. If there is a type mismatch between the variable specified as an argument to `get()` and the message in the mailbox, a runtime error is generated and the message is not retrieved.
- status = **tryget(**<variable>**)** — a function similar to `get()`, except that the process will not suspend if the mailbox is empty. `tryget()` returns 1 if the mailbox is not empty and the message type matches the type of the argument to `tryget()`. It returns 0 if the mailbox is empty. It returns -1 if the mailbox is not empty, but there is a type mismatch between the variable specified as an argument to `tryget()` and the message in the mailbox.
- **peek(**<variable>**)** — a task that copies a message from a mailbox without removing it from the mailbox. If the mailbox is empty, the process is suspended (blocked) until a message is placed in the mailbox. If there is a type mismatch between the variable specified as an argument to `peek()` and the message in the mailbox, the message is not copied, and a runtime error is generated.
- status = **trypeek(**<variable>**)** — a function similar to `peek()`, except that the process will not suspend if the mailbox is empty. `trypeek()` returns 1 if the mailbox is not empty and the message type matches the type of the argument to `trypeek()`. It returns 0 if the mailbox is empty. It returns -1 if the mailbox is not empty, but there is a type mismatch between the variable specified as an argument to `trypeek()` and the message in the mailbox.
- n = **num()** — a function that returns the number of messages currently in a mailbox.

3.2 A FIFO channel using mailboxes

SystemVerilog mailboxes provide a simple means to model FIFO communication behavior at a high level of abstraction. A SystemVerilog interface using mailboxes very closely approximates a SystemC FIFO built-in channel.

Example 23, which follows, uses a SystemVerilog interface to fully encapsulates a FIFO communication channel between two modules. All of the write and read logic is within the interface, so that the modules communicating through the interface do not need to be concerned with details such as keeping count of how many values are stored in the FIFO. These details are handled within the interface.

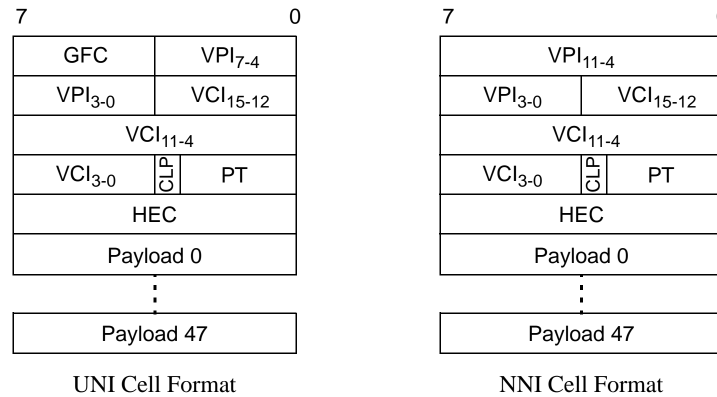
The mailbox that represents the FIFO is bounded to a specific FIFO size. The example has a default bound of 8 FIFO elements, but the FIFO size can be redefined each time the FIFO is instantiated.

SystemVerilog mailboxes have methods for both blocking and nonblocking writes and reads to/from the FIFO. This example uses the nonblocking write and read methods. It would be easy to add blocking methods, if desired.

The mailbox used in this example is explicitly typed. Each location of the FIFO will store the same value types, making it easy to ensure that values read back as the correct data type (this capability does not exist in the SystemC FIFO built-in channel). The data type of the FIFO is parameterized, allowing each instance of the interface to be customized to store a specific type of data. In this example, the data stored in the FIFO is an ATM cell data packet, represented as a

SystemVerilog structure. Two different types of ATM packets are defined, a UNI cell and a NNI cell. Figure 6 illustrates the two types of packets, and Example 22 shows the SystemVerilog definitions of these packets.

Figure 6. UNI and NNI ATM switch data packets



Example 22. SystemVerilog definitions for UNI and NNI ATM switch data packets

```
typedef struct packed {  // UNI Cell data packet
  bit         [ 3:0] GFC;
  bit         [ 7:0] VPI;
  bit         [15:0] VCI;
  bit                CLP;
  bit         [ 2:0] T;
  bit         [ 7:0] HEC;
  bit [0:47] [ 7:0] Payload;
} uniType;

typedef struct packed {  // NNI Cell data packet
  bit         [11:0] VPI;
  bit         [15:0] VCI;
  bit                CLP;
  bit         [ 2:0] PT;
  bit         [ 7:0] HEC;
  bit [0:47] [ 7:0] Payload;
} nniType;
```

Example 23. A complete SystemVerilog FIFO interface channel modeled using SystemVerilog mailboxes

```
interface fifo_channel_1 #(parameter      FifoSize = 8, PtrSize = 4,
                           parameter type DataType = uniType);

  DataType write_data;    // packet coming from sending module
  DataType read_data;     // packet going to receiving module

  bit fifo_empty, fifo_full;  // FIFO status flags

  /*********************************************************************
   * FIFO storage
   *********************************************************************/
  mailbox #(DataType) FIFO = new(FifoSize);  // FIFO is bounded mailbox

  /*********************************************************************
   * Methods to write to and read from the FIFO
   *********************************************************************/
  function automatic void Write (input DataType write_data);
    void'(FIFO.tryput(write_data));  // nonblocking write
    fifo_full = ~(FIFO.num < FifoSize);
  endfunction

  function automatic void Read (output DataType read_data);
    fifo_empty = (FIFO.num == 0);
    void'(FIFO.tryget(read_data) );  // nonblocking read
  endfunction

  /*********************************************************************
   * Module connections to the interface
   *********************************************************************/
  modport sender (input  write_data,   // sending module's connections
                  output fifo_full,
                  import Write);

  modport reader (output read_data,    // reading module's connections
                  output fifo_empty,
                  import Read);

endinterface: fifo_channel_1
```

3.3 Connecting and using the interface channel

The modules that are connected to the interface channel must:

1. Use the FIFO interface as a port of the module.
2. Call an appropriate write or read method within the interface to either place data into the FIFO or to retrieve data from the FIFO.

A top-level netlist connects the FIFO interface to the modules that communicate using the interface. The top-level module will:

1. Instantiate the interface.
2. Optionally set the interface FIFO size and pointer size (the default is 8 FIFO locations).

3. Optionally set the interface FIFO storage type (the default type is a UNI cell packet).
4. Instantiate and connect the interface to one or more modules.

Examples of two modules that utilize the FIFO interface and a top-level netlist are shown in Example 24, below. Observe that the module that stores values in the FIFO does not need complex pointers or any other logic to control the FIFO. The module simply calls the `Write` method in the interface, and the method takes care of storing the value into the next available location in the FIFO. Similarly, the module retrieving values from the FIFO simply calls a `Read` method in the interface, without the complexity of read pointers or any of the other details that would have been required in a Verilog model of a FIFO.

Example 24. Netlist connecting two modules using the FIFO interface channel

```
module top_level (input clock1, clock2, resetN);

  fifo_channel_1 #(.DataType(nniType)) FIFO ();  // instance of interface

  processor1 #(.DataType(nniType)) p1 (FIFO, clock1, resetN);

  processor2 #(.DataType(nniType)) p2 (FIFO, clock2, resetN);

endmodule: top_level
```

```
module processor1 #(parameter type DataType = uniType)
                   (fifo_channel_1.sender fifo,         // interface port
                    input clock, write_resetN);         // other ports

  DataType data_packet;    // packet data type is parameterized
  ...
  always_ff @(posedge clock) begin
    if (write_enable && !fifo.fifo_full)
      fifo.Write(data_packet);      // send packet to FIFO channel
  end
  ...
endmodule: processor1
```

```
module processor2 #(parameter type DataType = uniType)
                   (fifo_channel_1.reader fifo,      // interface port
                    input clock, resetN);            // other ports

  DataType data_packet;
  ...
  always_ff @(posedge clock) begin
    if (read_enable && !fifo.fifo_empty) begin
      fifo.Read(data_packet);       // get packet from FIFO channel
  end
  ...
endmodule: processor2
```

3.4 Synthesis considerations with mailboxes

The FIFO interface shown in Example 23 (in section 3.2) is ***not*** synthesizable. This example is

purposely modeled at a high level of abstraction, similar to SystemC FIFO channels. This high level of abstraction makes the model concise and simple to code, while still providing proper FIFO communication functionality. However, the abstract model does not contain the details that would be required for synthesis to properly realize the channel behavior in logic gates. In addition, current Verilog/SystemVerilog synthesis compilers do not support the SystemVerilog mailbox construct.

4.0 Modeling FIFO channels using SystemVerilog queues

4.1 Overview of SystemVerilog queues

A FIFO is essentially a queue of data. Verilog does not provide any specific constructs to represent queues of data. Engineers must model queues using lower-level RTL modeling constructs. SystemVerilog provides a convenient mechanism for modeling queues at a more abstract level than is possible with Verilog. A queue is represented as a variable-sized array. The array size increases each time an element is added to the array (a *push*), and the array size decreases each time an element is removed from the array (a *pop*).

Queues are declared using a similar syntax as Verilog arrays, but with the array size specified using a dollar sign (`$`). Optionally, the $ can be followed by `:<max_size>`, where max_size is a constant that specifies the maximum size to which the queue can grow. Some example queue declarations are:

```
int q1 [$];  // declaration of an empty queue, with unbounded max size
int q2 [$] = {1,2,3,5,8}; // unbounded queue initialized with 5 locations
int q3 [$:16];  // an empty queue, with a max size of 16 addresses
```

SystemVerilog provides a number of special methods to work with queues.

- **`size()`** returns the number of items in the queue. If the queue is empty, then it returns 0.
- **`insert(<index>,<value>)`** inserts the given value at the specified index position.
- **`delete(<index>)`** deletes the item at the specified index position.
- **`pop_front()`** removes and returns the first element of the queue.
- **`pop_back()`** removes and returns the last element of the queue.
- **`push_front(<value>)`** inserts the given value at the beginning of the queue.
- **`push_back(<value>)`** inserts the given value at the end of the queue.

These special queue methods are invoked by appending the method name to the queue name, with a period as a separator. For example:

```
int q1 [$];  // declaration of an empty queue that holds int values
int i;

q1.push_front(i);   // add i to new a location to beginning of the queue
i = q1.pop_back();  // get value at end of the queue and remove location
```

Queue locations can also be directly written to, or read from, by using an address to index into the queue. The first element in the queue is referenced using index 0. The last element in the queue is

referenced using **$**. Reading a value from the queue using a location address is a non-destructive copy; it does not modify the size or contents of the queue. Similarly, writing to a queue location using an address modifies the value in that location, but does not modify the size of the queue.

```
int q2 [$] = {1,2,3,5,8};  // a queue initialized with 5 locations
int i, j;

i = q2[0];      // read first element of the queue (which is storing 1)

j = q2[$];      // read last element of the queue (which is storing 8)
```

It is a run-time error to read from an empty queue. A model should test for an empty queue before reading from the queue. This test can be done using the `.size()` method or by comparing the queue to an empty set using { }. For example:

```
if ( q1 != {} )
  i = q1.pop_back();
```

It is also an error to add to a bounded queue that has reached its maximum size. A model should test that the queue size is not at the maximum size before adding to the queue.

```
if ( q1.size < MAX )
  q1.push_back(i);
```

Alternatively, the design logic could include some sort of mechanism to keep track of how many items are in the queue, such as a counter.

4.2 An abstract FIFO channel using queues

SystemVerilog queues provide a mechanism to model FIFO communication behavior at a high level of abstraction, similar to mailboxes. Unlike mailboxes, however, SystemVerilog queues cannot be typeless. The queue type is part of the queue declaration, and every location of queue must store the same data type.

The following example models a simple FIFO interface channel using SystemVerilog queues. As with the abstract example using SystemVerilog mailboxes (Example 23 in section 3.2), this example provides nonblocking `Write` and `Read` methods. The interface read and write methods check that the queue is not empty or full, to prevent run-time errors.

The capacity of the FIFO is parameterized, allowing each instance of the FIFO interface to be customized to a different size. The data type of the FIFO is also parameterized, allowing each instance of the interface to be customized to store a specific type of data. The default data type for the queue is a UNI cell type.

Example 25. An abstract SystemVerilog FIFO interface channel modeled using SystemVerilog queues

```
interface fifo_channel_2 #(parameter      FifoSize = 8, PtrSize = 4,
                           parameter type DataType = uniType);

  DataType write_data;  // packet coming from sending module
  DataType read_data;   // packet going to receiving module

  bit fifo_empty, fifo_full;  // FIFO status flags

  /**************************************************************************
   * FIFO storage
   *************************************************************************/
  DataType FIFO [$:FifoSize];   // FIFO is a dynamic queue

  /**************************************************************************
   * Methods to write to and read from the FIFO
   *************************************************************************/
  function automatic void Write (input DataType write_data);
    FIFO.push_back(write_data);   // push onto end of queue
    fifo_full = ~(FIFO.PtrSize < FifoSize);
  endfunction

  function automatic void Read (output DataType read_data);
    read_data = FIFO.pop_front;  // retrieve from beginning of queue
    fifo_empty = (FIFO.PtrSize == 0);
  endfunction

  /**************************************************************************
   * Module connections to the interface
   *************************************************************************/
  modport sender (input  write_data,   // sending module's connections
                  output fifo_full,
                  import Write);

  modport reader (output read_data,    // reading module's connections
                  output fifo_empty,
                  import Read);
endinterface: fifo_channel_2
```

4.3 Connecting and using the interface channel

The interface example shown above has the same modports and methods as mailbox FIFO version shown in Example 23. Therefore the interface can be configured and connected to modules in the same way as shown Example 24, in section 3.3. Both versions of the FIFO interface have the same modport names and modport lists. The names and arguments of the write and read methods are also the same. Therefore, the two interface examples are completely interchangeable.

4.4 Synthesis considerations with queues

SystemVerilog queues and the queue methods are considered synthesizable, provided that a maximum size for the queue is defined (though at the time this paper was written, neither

Synopsys VCS nor Synopsys HDL Compiler were supporting the queue construct and methods). However, Example 25, as shown above, may not synthesize well. Good FIFO logic needs to allow for corner cases such as a write to, and a read from, the FIFO occurring at the same time and at the same address. It is also important that the logic prevent glitches as different bits of the write or read pointers change value. The nuances of proper multi-clock asynchronous FIFO design are covered in two excellent SNUG 2002 papers, by Cummings[7] and by Cummings and Alfke[8].

5.0 A synthesizable FIFO channel using queues

To synthesize a reliable FIFO design, the model must reflect proper synchronization of the logic that writes values into the FIFO and the logic that reads values back from the FIFO. Since the write logic and read logic are typically in different clock domains running a different frequencies, proper clock synchronization is essential.

Interfaces can contain procedural code as well as methods. This allows FIFO read/write logic to be modeled using a more conventional RTL style. The SystemVerilog FIFO interface shown in the following example takes advantage of SystemVerilog queues and queue methods for a higher level of modeling abstraction. However, the FIFO full and empty logic is modeled at a more detailed RT level of abstraction. Specifically:

- The FIFO full and empty status is determined using write and read pointers, instead of using abstract queue methods.
- The write and read pointers are fully synchronized to the asynchronous write and read clocks, using 2-stage flip-flop synchronization.

The methodologies used for the write/read pointer logic and pointer synchronization are based on those presented in the SNUG 2002 paper on FIFO design presented by Clifford Cummings[7]. The pointers are derived from dual n-bit wide Gray code counters. Refer to the Cummings paper for a full description of the logic.

The use of SystemVerilog queues greatly simplifies addressing into the FIFO. Since the queue is dynamically sized, a write using the `push_back()` method always adds a new location to the end of the queue. A read using the `push_back()` method always pulls from the first address of the queue, and then removes that location. Hence, there is no need to maintain write and read address registers, as would be required in pure RT level FIFO models. This combination of using SystemVerilog queues and RTL logic provides the implementation detail needed for synthesis, while still benefitting from a more abstract modeling style.

As with the more abstract FIFO interfaces in Examples 23 and 25, the example shown in this section parameterizes the FIFO capacity and data type, allowing each instance of the FIFO interface to be customized.

Example 26. A synthesizable SystemVerilog FIFO interface channel modeled using SystemVerilog queues

```
interface fifo_channel_3 #(parameter      FifoSize = 8, PtrSize = 4,
                           parameter type DataType = uniType);

  DataType write_data;  // packet coming from sending module
  DataType read_data;   // packet going to receiving module

  bit             fifo_empty, fifo_full;  // FIFO status flags

  bit             write_clock, write_enable, write_resetN;
  bit [PtrSize:0] wr_ptr, wr_ptr_synced, wr_ptr_next;
  bit [PtrSize:0] wr_cntr, wr_cntr_next;
  bit             wr_ptr_2nd_msb, wr_ptr_synced_2nd_msb;

  bit             read_clock, read_enable, read_resetN;
  bit [PtrSize:0] rd_ptr, rd_ptr_synced, rd_ptr_next;
  bit [PtrSize:0] rd_cntr, rd_cntr_next;

  /*************************************************************************
   * FIFO storage
   ***********************************************************************/
  DataType FIFO [$:FifoSize];  // FIFO is a dynamic queue

  /*************************************************************************
   * Methods to write to and read from the FIFO
   ***********************************************************************/
  function automatic void Write (input  DataType write_data);
    FIFO.push_back(write_data);   // push onto end of queue
  endfunction

  function automatic void Read (output DataType read_data);
    read_data = FIFO.pop_front;  // retrieve from beginning of queue
  endfunction

  /*************************************************************************
   * Module connections to the interface
   ***********************************************************************/
  modport sender (input  write_data,   // sending module's connections
                  output fifo_full,
                  output write_clock, write_enable, write_resetN,
                  import Write,
                  // internal signals used within the interface
                  inout wr_ptr, wr_ptr_synced, wr_ptr_next,
                  inout wr_cntr, wr_cntr_next,
                  inout wr_ptr_2nd_msb, wr_ptr_synced_2nd_msb
                 );

  modport reader (output read_data,    // reading module's connections
                  output fifo_empty,
                  output read_clock, read_enable, read_resetN,
                  import Read,
                  // internal signals used in the interface
                  inout rd_ptr, rd_ptr_synced, rd_ptr_next,
                  inout rd_cntr, rd_cntr_next
                 );
```

(continued on next page)

```
/**********************************************************************
 * Clock Synchronization Logic
 *********************************************************************/
// Read-domain to write-domain synchronizer
always_ff @(posedge write_clock, negedge write_resetN)
  if (!write_resetN) {wr_ptr_synced,wr_ptr} <= 0;
  else               {wr_ptr_synced,wr_ptr} <= {wr_ptr,rd_ptr};

// Write-domain to read-domain synchronizer
always_ff @(posedge read_clock, negedge read_resetN)
  if (!read_resetN) {rd_ptr_synced,rd_ptr} <= 0;
  else              {rd_ptr_synced,rd_ptr} <= {rd_ptr,wr_ptr};

/**********************************************************************
 * Write Control Logic
 *********************************************************************/
// generate a Gray code pointer
always_ff @(posedge write_clock, negedge write_resetN)
  if (!write_resetN) wr_ptr <= 0;
  else               wr_ptr <= wr_ptr_next;

// generate next Gray code value
always_comb begin
  for (int i=0; i<=PtrSize; i++)
    wr_cntr[i] = ^(wr_ptr>>i);
  if (!fifo_full && write_enable)
    wr_cntr_next = wr_cntr + 1;
  else
    wr_cntr_next = wr_cntr;
  wr_ptr_next = (wr_cntr_next>>1)^wr_cntr_next;
end

// generate fifo_full flag
assign wr_ptr_2nd_msb = wr_ptr_next[PtrSize]^wr_ptr_next[PtrSize-1];
assign wr_ptr_synced_2nd_msb =
    wr_ptr_synced[PtrSize]^wr_ptr_synced[PtrSize-1];
always_ff @(posedge write_clock, negedge write_resetN) begin
  if (!write_resetN)
    fifo_full <= 0;
  else
    fifo_full <= ((wr_ptr_next[PtrSize]     != wr_ptr_synced[PtrSize]) &&
                  (wr_ptr_2nd_msb           == wr_ptr_synced_2nd_msb ) &&
                  (wr_ptr_next[PtrSize-2:0] == wr_ptr_synced[PtrSize-2:0])
                 );
end
```

(continued on next page)

```
/*********************************************************************
 * Read Control Logic
 *********************************************************************/
// generate a Gray code pointer
always_ff @(posedge read_clock, negedge read_resetN)
  if (!read_resetN) rd_ptr <= 0;
  else              rd_ptr <= rd_ptr_next;

// generate next Gray code value
always_comb begin
  for (int i=0; i<=PtrSize; i++)
    rd_cntr[i] = ^(rd_ptr>>i);
  if (!fifo_empty && read_enable)
    rd_cntr_next = rd_cntr + 1;
  else
    rd_cntr_next = rd_cntr;
  rd_ptr_next = (rd_cntr_next>>1)^rd_cntr_next;
end

// generate fifo_empty flag
always_ff @(posedge read_clock, negedge read_resetN) begin
  if (!read_resetN)
    fifo_empty <= 0;
  else
    fifo_empty <= (rd_ptr_next == rd_ptr_synced);
end

endinterface: fifo_channel_3
```

5.1 Connecting and using the interface channel

The interface example shown above is not interchangeable with the previous two interface examples (Examples 23 and 25). The modports in this last example are a slightly different. This is because the lower-level RTL code in this example requires clock and reset signals that the more abstract models did not need. Other than this difference in the modport lists, however, the three examples are very similar. They have the same modport names and method names. Switching a design from using one of the abstract FIFO interface examples to the RT-level FIFO interface example is simple to do.

5.2 Synthesis considerations

Example 26, above, is synthesizable. However, the Synopsys HDL Compiler product imposes some restrictions on the SystemVerilog constructs used, that are important to note.

Interface declaration order. Interfaces must be declared before they can be referenced as ports of a module. This is not a restriction of SystemVerilog, or of the Synopsys VCS simulator. Only HDL Compiler has this limitation.

Methods must be automatic. Tasks and functions in an interface must be declared as automatic in order to be synthesized. HDL Compiler duplicates the functionality of a task or function in each module that calls the task or function. In the source code, there is a single copy of the task or

function, but after synthesis there can be many copies of the same logic. Automatic tasks and functions create new storage for each call, making each call appear as if it were a unique copy of the task or function. Thus, simulation of automatic tasks and functions will be have the same behavior as the synthesized copies of the task/function functionality.

Internal interface signals must be included in modports. All interface signals used by a module, even indirectly, must be listed in the modport declaration used by that module. In Example 26 in Section 5, the interface contains several procedural blocks that model the functionality for the write and read pointers, and the FIFO full and empty flags. The modules connected to the interface only need access to the full and empty flags and the Write and Read methods. The modules do not need to access the signals used internally within the interface. However, HDL Compiler requires that any internal signals that affect, or are affected by, the signals or methods used by a module must also be included in the modport list for that module. For example, the `fifo_full` signal in the modport list is derived from several internal signals within the interface. These internal signals must be included in the modport along with `fifo_full`.

Queues not yet implemented. At the time this paper was written, neither the Synopsys VCS simulator nor the Synopsys HDL Compiler synthesis tool has implemented the SystemVerilog queue construct and its methods.

For more information on what SystemVerilog constructs are supported by Synopsys HDL Compiler, refer the Synopsys HDL Compiler SystemVerilog Users Guide[6].

6.0 Conclusions

Abstract hardware modeling languages must provide a means to represent complex functionality with concise code. SystemC is one such abstract modeling language. For inter-module communications, SystemC provides communication channels for modeling complex communications at an abstract level. A FIFO channel, for example, allows data to be transferred between modules using simple, built-in modeling constructs. In Verilog, representing this same FIFO communication would require much more complicated code.

One of the promises of the SystemVerilog extensions to Verilog is the ability to model at a higher level of abstraction than is possible with just the Verilog HDL. Using SystemVerilog, complex functionality can be modeled with concise, readable code. This paper has examined this promise in the context of a complex FIFO communication channel between two modules, similar to SystemC FIFO channels.

The SystemVerilog interface construct was used to encapsulate the communications. An interface allows designers to create complex, user-defined module port types that contain both the interconnecting signals between modules, and the methods used to transfer information across the interface. Three FIFO interface examples were shown in the paper.

First, the SystemVerilog built-in mailbox object and object methods were used to model FIFO behavior with just a few lines of code. Using a mailbox, a complex communication channel was simple and easy to model. The mailbox model very closely resembles a SystemC FIFO channel. However, the object-oriented SystemVerilog mailbox and methods are abstract constructs that do

not provide the implementation details required for synthesis to realize complex hardware, such as proper synchronization when multiple clock domains are involved. The SystemVerilog mailbox construct is ideal for high-level abstract modeling, but it is not currently considered a synthesizable a construct.

Second, the FIFO communication was modeled using SystemVerilog queues, which are a special form of dynamic arrays. SystemVerilog queues also have built-in methods to make it easy to move data through the FIFO. SystemVerilog queues are synthesizable (with some restrictions). However, the built-in methods are abstract constructs that do not provide the implementation details required for effective synthesis.

The third representation of a FIFO interface channel also used SystemVerilog queues. In this example, however, the asynchronous multiple clock domain synchronization was modeled using traditional synthesizable RTL code. This approach required writing considerably more code than the two abstract approaches. However, the use of SystemVerilog queues still simplified the model a great deal in comparison to a complete FIFO modeled entirely at the RTL level.

The Synopsys HDL synthesis compiler already supports a significant number of the SystemVerilog extensions to Verilog that are intended for the use of modeling hardware behavior. A complex FIFO communication channel can be modeled at a synthesizable level, and still benefit from the encapsulation of signals and functionality offered by the use of interfaces.

This paper has shown that SystemVerilog interfaces can model complex inter-module communication at both a high level of abstraction and at a synthesizable level.

7.0 Future work

There are two major areas where the FIFO channel examples presented in this paper can be improved:

- Make it so that the abstract interfaces and the synthesizable interface can be interchanged with no modifications to the modules that utilize the FIFO channel
- Add assertions to the interface channels

In the examples listed in this paper, the modport lists for the abstract FIFO channels are not the same as the modport lists for the RTL FIFO channel. The primary reason for the difference is because the RTL interface requires additional communication with the modules, such as for clock and reset. These signals are not needed when the FIFO is modeled at a more abstract, object-oriented level. This means that the modules that connect to the interface must be hard coded to use a specific version of the FIFO interface. It would be convenient to make the abstract and RTL FIFO interfaces fully interchangeable.

Assertions can be used to add checking and error reporting to SystemVerilog interfaces. Assertion checks that could be added to a FIFO interface channel include: checking for FIFO underflow (attempt to read when no data is in the FIFO), checking for FIFO overflow (attempt to write to a full FIFO), and simultaneous write and read to the same FIFO location.

8.0 References

[1] *SystemC 2.0.1 Language Reference Manual Revision 1.0*, Open SystemC Initiative, San Jose, California, 2003. www.systemc.org.

[2] *SystemC Methodologies and Applications*, edited by W. Mueller, W. Rosenstiel and J. Ruf., pp. 221-222. Kluwer Academic Publishers, Boston, MA, 2004, ISBN 0-4020-7479-4.

[3] *SystemVerilog 3.1a: Accellera's Extensions to Verilog*, Accellera, Napa, California, 2004. www.accellera.org.

[4] *IEEE Std. 1364-2001 standard for the Verilog Hardware Description Language*, IEEE, Pascataway, New Jersey, 2001. ISBN 0-7381-2827-9.

[5] *SystemVerilog for Design: A Guide to Using SystemVerilog for Hardware Design and Modeling,* Stuart Sutherland, Simon Davidmann and Peter Flake. Kluwer Academic Publishers, Boston, MA, 2004, ISBN 0-4020-7530-8.

[6] *SystemVerilog Synthesis User Guide, Version V-2004.06*, June 2004, Synopsys, Inc.

[7] *Simulation and Synthesis Techniques for Asynchronous FIFO Design*, SNUG San Jose 2002 paper, Cliff Cummings. www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO1_rev1_1.pdf

[8] *Simulation and Synthesis Techniques for Asynchronous FIFO Design with Asynchronous Pointer Comparisons*, SNUG San Jose 2002 best paper, Cliff Cummings and Peter Alfke, www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO2_rev1_1.pdf.

9.0 About the author

Stuart Sutherland has been involved with SystemVerilog since its inception in 2001. He is a member of the IEEE P1800 SystemVerilog standards group, and the Accellera SystemVerilog committee that created SystemVerilog (where he served as the editor of the SystemVerilog 3.0, 3.1 and 3.1a reference manuals). Mr. Sutherland has also been an active member of the IEEE 1364 Verilog standards group since 1984, where he serves as co-chair of the PLI task force. Mr. Sutherland is an independent Verilog consultant, who specializes in providing expert training on Verilog, SystemVerilog and the Verilog PLI.

Mr. Sutherland welcomes your comments and feedback regarding this paper. You can contact him by e-mail at stuart@sutherland-hdl.com.

This paper and other SNUG papers by Stuart Sutherland are available for download at www.sutherland-hdl.com/papers