

## Modeling FIFO Communication Channels Using SystemVerilog Interfaces



**Stuart Sutherland**  
Sutherland HDL, Inc.  
Portland, Oregon, USA  
stuart@sutherland-hdl.com

1

## Objectives

- Introduce SystemVerilog
- Concepts of Communication Channels
- Tutorial on SystemVerilog Interfaces
- Modeling a FIFO Interface Channel Using Mailboxes
- Modeling a FIFO Interface Channel Using Queues
- Modeling a Synthesizable FIFO Interface Channel
- Conclusions





# Verilog, SystemVerilog and SystemC

- ◆ Each hardware design language has unique capabilities
  - ◆ This paper is *not* about what language is best
  - ◆ This paper *is* on how SystemVerilog enables modeling inter-module communication at a higher level of abstraction

- ◆ This chart shows...
  - ◆ SystemVerilog does not replace SystemC
  - ◆ SystemVerilog bridges a gap between Verilog and SystemC

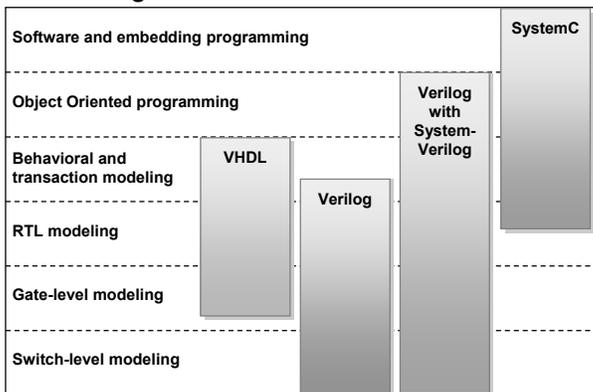


Chart reflects the author's perception of general language overlap

# What's Next

- ✓ Introduce SystemVerilog
- **Concepts of Communication Channels**
- Tutorial on SystemVerilog Interfaces
- Modeling a FIFO Interface Channel Using Mailboxes
- Modeling a FIFO Interface Channel Using Queues
- Modeling a Synthesizable FIFO Interface Channel
- Conclusions



## Inter-Module Communication

- ◆ Verilog connects models using detailed **module ports**
  - ◆ Each discrete signal must be declared as a port
- ◆ SystemC provides communication **channels**
  - ◆ Encapsulate how information is transferred between modules
  - ◆ Allow abstract, high-level communication
- ◆ SystemVerilog has communication **interfaces**
  - ◆ Encapsulate how information is transferred between modules
  - ◆ A new paradigm for inter-module communication

Can SystemVerilog interfaces provide the same abstract communication capabilities as SystemC channels?

Can SystemVerilog interfaces be synthesized?

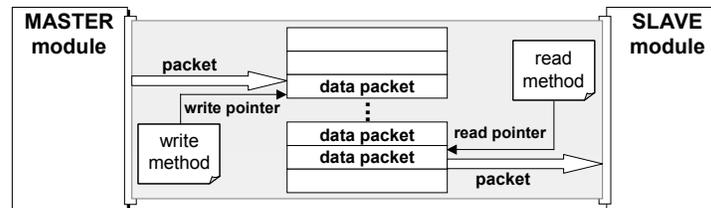


## SystemC Channels

- ◆ SystemC provides **channels** and **interfaces**
  - ◆ A “channel” encapsulates how information is transferred between blocks of a design
  - ◆ An “interface” defines a set of methods (functions) to send and receive data through the channel
- ◆ There are two types of channels
  - ◆ *Built-in channels* that are pre-defined in SystemC
    - ◆ Includes FIFO, Mutex, and Semaphore channels
    - ◆ Represent abstract communication
    - ◆ Generally not synthesizable
  - ◆ *User-defined channels*
    - ◆ Can be modeled as abstract or at a more detailed level
    - ◆ Can be synthesizable

## A SystemC FIFO Channel

- ◆ FIFOs are typically used to communicate between design blocks that operate on different, asynchronous clocks
- ◆ The SystemC built-in FIFO channel:
  - ◆ Is unsized — any amount of data can be in the channel
  - ◆ Is untyped — any data type can be sent through the channel
  - ◆ Provides write/read methods to send data through the channel
    - ◆ Methods are built-in — end user does not see how they work



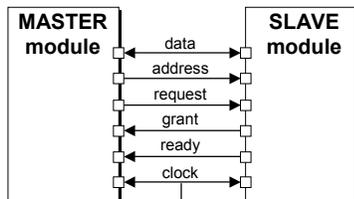
## What's Next

- ✓ Introduce SystemVerilog
- ✓ Concepts of Communication Channels
- **Tutorial on SystemVerilog Interfaces**
- Modeling a FIFO Interface Channel Using Mailboxes
- Modeling a FIFO Interface Channel Using Queues
- Modeling a Synthesizable FIFO Interface Channel
- Conclusions



# Inter-Module Communication: Verilog Style

- ◆ Verilog connects modules at the implementation level



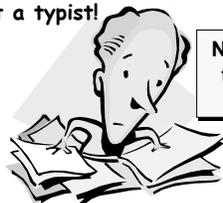
Connection details are in the module

```
module MASTER (input clock,
               inout [31:0] data,
               output [15:0] address,
               output request,
               input grant,
               input ready );
    ...
endmodule
```

Connection details are duplicated in other modules

```
module SLAVE (input clock,
              inout [31:0] data,
              input [15:0] address,
              input request,
              output grant,
              output ready );
    ...
endmodule
```

I want to be an engineer, not a typist!



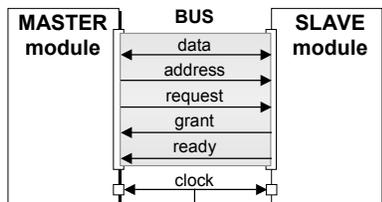
Netlists must duplicate the connection detail (yet again)

```
module top (input clock);
    wire [31:0] data;
    wire [15:0] address;
    wire request, grant, ready;

    MASTER i1 (clock, data, address,
              request, grant, ready);
    SLAVE i2 (clock, data, address,
            request, grant, ready);
endmodule
```

# Inter-Module Communication: SystemVerilog Style

- ◆ Interfaces encapsulate inter-module communication



Connection details are in the interface

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

```
module MASTER (interface io_bus);
    ...
endmodule
```

```
module SLAVE (interface io_bus);
    ...
endmodule
```

Modules and netlist do not duplicate the connection details



Now I can concentrate on designing instead of typing!

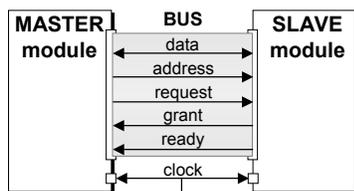
```
module top (input clock);
    BUS io ();
    MASTER i1 (io, clock);
    SLAVE i2 (io, clock);
endmodule
```

connect interface instance to module port

instantiate the interface (io is the instance name)

## Interface Modports

- ◆ The module's port direction definitions are moved to within the interface, using the `modport` keyword (for "module's ports")
  - ◆ Encapsulates port lists that used to be scattered in many modules



```
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
```

**A modport defines the port directions from the module's point of view**

## Selecting Which Modport Definition To Use

- ◆ SystemVerilog has two ways to select which modport to use
  - ◆ At the module instance — allows "generic module ports"

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

Each instance of the module can be connected to a different interface or a different modport

```
module MASTER (interface io_bus);
  ...
endmodule
```

```
module top (input clock);
  BUS io ();
  MASTER i1 (io.master_ports, clock);
  ...
endmodule
```

connect to interface using the master\_ports port list

- ◆ At the module instance — prevents incorrect corrections

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

Hard coded interface port means each instance of the module will be connected the same way

```
module SLAVE (BUS.slave_ports io_bus);
  ...
endmodule
```

```
module top (input clock);
  BUS io ();
  ...
  SLAVE i2 (io, clock);
endmodule
```

connect to interface using the slave\_ports port list

- ◆ Interfaces can contain **“methods”**
  - ◆ Defined using Verilog/SystemVerilog tasks and functions
  - ◆ Used to encapsulate inter-module communication functionality

```

interface BUS;
...
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
    
```

**Methods encapsulate communication functionality into one place, instead of being fragmented across multiple modules**

**communication methods**

**modules can import interface methods through modports**

- ◆ Modules can reference signals and methods within the interface using a hierarchical path name

`<interface_port_name>.<signal_or_method_name>`

```

module MASTER (interface io_bus); // interface port
...
always @(posedge clock) begin
    if (io_bus.request)
        io_bus.Read(...);
...
end
...
endmodule
    
```

**interface port name**

**reference signal in interface**

**reference method in interface**

**Synthesis supports this special case of using hierarchical path names**

## Interface Processes

- ◆ An interface can contain procedural code and continuous assignments
  - ◆ initial, always, always\_ff, always\_comb, always\_latch, assign

Processes encapsulate communication functionality that might otherwise be spread across several modules

```
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
```

## Parameterized Interfaces

- ◆ Interfaces can be parameterized, the same as Verilog modules
  - ◆ Allows each instance of an interface to be reconfigured

```
interface fifo_channel #(parameter FifoSize = 8, PtrSize = 4,
                        parameter type DataType = int );
...
DataType write_data;
DataType read_data;
...
endinterface
```

SystemVerilog also allows data types to be parameterized

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

interface instance is reconfigured to use a different data type

## The FIFO Example Data Types

- ◆ The FIFO examples used in this paper transport packets for an ATM Switch

<b>NNI Cell</b>	<table border="1" style="border-collapse: collapse; width: 100%;"> <tr><td colspan="2" style="text-align: center;">VPI<sub>11-4</sub></td></tr> <tr><td style="text-align: center;">VPI<sub>3-0</sub></td><td style="text-align: center;">VCI<sub>15-12</sub></td></tr> <tr><td colspan="2" style="text-align: center;">VCI<sub>11-4</sub></td></tr> <tr><td style="text-align: center;">VCI<sub>3-0</sub></td><td style="text-align: center;">PT</td></tr> <tr><td colspan="2" style="text-align: center;">HEC</td></tr> <tr><td colspan="2" style="text-align: center;">Payload 0</td></tr> <tr><td colspan="2" style="text-align: center;">⋮</td></tr> <tr><td colspan="2" style="text-align: center;">Payload 47</td></tr> </table>	VPI <sub>11-4</sub>		VPI <sub>3-0</sub>	VCI <sub>15-12</sub>	VCI <sub>11-4</sub>		VCI <sub>3-0</sub>	PT	HEC		Payload 0		⋮		Payload 47		<pre>typedef struct { // UNI Cell     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;</pre>	<p><b>SystemVerilog adds structures and user-defined types to Verilog</b></p>
VPI <sub>11-4</sub>																			
VPI <sub>3-0</sub>	VCI <sub>15-12</sub>																		
VCI <sub>11-4</sub>																			
VCI <sub>3-0</sub>	PT																		
HEC																			
Payload 0																			
⋮																			
Payload 47																			
<b>UNI Cell</b>	<table border="1" style="border-collapse: collapse; width: 100%;"> <tr><td style="text-align: center;">GFC</td><td style="text-align: center;">VPI<sub>7-4</sub></td></tr> <tr><td style="text-align: center;">VPI<sub>3-0</sub></td><td style="text-align: center;">VCI<sub>15-12</sub></td></tr> <tr><td colspan="2" style="text-align: center;">VCI<sub>11-4</sub></td></tr> <tr><td style="text-align: center;">VCI<sub>3-0</sub></td><td style="text-align: center;">PT</td></tr> <tr><td colspan="2" style="text-align: center;">HEC</td></tr> <tr><td colspan="2" style="text-align: center;">Payload 0</td></tr> <tr><td colspan="2" style="text-align: center;">⋮</td></tr> <tr><td colspan="2" style="text-align: center;">Payload 47</td></tr> </table>	GFC	VPI <sub>7-4</sub>	VPI <sub>3-0</sub>	VCI <sub>15-12</sub>	VCI <sub>11-4</sub>		VCI <sub>3-0</sub>	PT	HEC		Payload 0		⋮		Payload 47		<pre>typedef struct { // NNI Cell     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;</pre>	
GFC	VPI <sub>7-4</sub>																		
VPI <sub>3-0</sub>	VCI <sub>15-12</sub>																		
VCI <sub>11-4</sub>																			
VCI <sub>3-0</sub>	PT																		
HEC																			
Payload 0																			
⋮																			
Payload 47																			

## What's Next

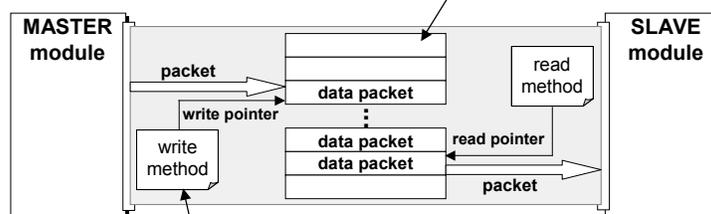
- ✓ Introduce SystemVerilog
- ✓ Concepts of Communication Channels
- ✓ Tutorial on SystemVerilog Interfaces
- **Modeling a FIFO Interface Channel Using Mailboxes**
- Modeling a FIFO Interface Channel Using Queues
- Modeling a Synthesizable FIFO Interface Channel
- Conclusions



## FIFO Channel Example 1: An Abstract Version

- ◆ The first FIFO interface example in this paper is modeled at a high level of abstraction
  - ◆ Closely approximates a SystemC FIFO built-in channel
  - ◆ Requires very few lines of code to model

The FIFO storage is modeled using SystemVerilog mailboxes



built-in mailbox methods will be used for writing and reading the FIFO

## Mailboxes

- ◆ A mailbox is a communication mechanism where data can be sent to a mailbox by one process and retrieved by another
- ◆ SystemVerilog mailboxes behave like real mailboxes
  - ◆ When a letter is delivered to a mailbox, the letter can be retrieved
  - ◆ If the post has not yet been delivered, a process can:
    - ◆ Choose to wait for post delivery, or
    - ◆ Come back later and check again
- ◆ Mailboxes can be either bounded or unbounded
  - ◆ A bounded mailbox is full when it contains a maximum number of messages
    - ◆ A process attempting to place a message into a full mailbox must wait until there is room available in the mailbox queue
  - ◆ Unbounded mailboxes never suspend send operations

## Overview of SystemVerilog Mailboxes

- ◆ SystemVerilog has a predefined object-oriented mailbox class

- ◆ A communication mechanism whereby
  - ◆ Messages can be placed in a mailbox by one process...



- ◆ ...and retrieved by another process



- ◆ If there are no messages in the mailbox, the retrieving process can:
  - ◆ Suspend and wait for a message (block)
  - ◆ Come back later and check again

## Mailbox Capacity

- ◆ A mailbox is a built-in class object
  - ◆ A new mailbox is “constructed” using a `new` function

`mailbox FIFO_1 = new;` ← `construct a mailbox called “FIFO_1”`

- ◆ The mailbox size can be unbounded or bounded
  - ◆ An unbounded mailbox is never full
  - ◆ A bounded mailbox has a max number of messages it can hold
    - ◆ A process attempting to place a message into a full mailbox must wait until there is room available in the mailbox
    - ◆ The size of the mailbox is passed as an argument to `new`

`mailbox FIFO_2 = new(8);` ← `construct a bounded mailbox`

SystemC built-in FIFOs are unbounded, but the example in this paper uses a bounded mailbox, which is more like hardware

## Mailbox Message Type

- ◆ A mailbox can be typeless or typed
  - ◆ A typeless mailbox can transfer any data type
    - ◆ The user must ensure that data is retrieved as the same data type in which it was stored
    - ◆ Run-time errors occur if try to retrieve the wrong type
  - ◆ A typed mailbox can only transfer data of a specific type
    - ◆ Sending or retrieving the wrong type is a compile-time error
    - ◆ The mailbox type is defined using parameter redefinition

```
mailbox #(int) FIFO_3 = new(8);
```

construct a typed mailbox

SystemC built-in FIFOs are typeless, but the example in this paper uses a typed mailbox, which is more like hardware

## Blocking Mailbox Methods

- ◆ SystemVerilog mailboxes have several built-in methods
  - ◆ put (message) — places a message in a mailbox
    - ◆ If the mailbox is full, the process is suspended (blocked) until there is room in the mailbox to place the message
  - ◆ get (variable) — retrieves a message from a mailbox
    - ◆ If the mailbox is empty, the process is suspended (blocked) until a message is placed in the mailbox

```
mailbox #(int) FIFO = new(8);
int data_in;
always @(posedge write_clock) begin
  if (data_ready) FIFO.put(data_in);
end
```

```
int data_out;
always @(posedge read_clock) begin
  FIFO.get(data_out);
end
```



A run-time error occurs if the data type passed to get() does not match the data type in the mailbox

## Nonblocking Mailbox Methods

- ◆ Mailboxes can be used without blocking process execution
  - ◆ `status = tryput (message)` — similar to `put ()`, except that the process does not suspend if the mailbox is full
    - ◆ Returns 0 if the mailbox is full; the message is not delivered
    - ◆ Returns 1 if the mailbox is not full; the message is delivered
  - ◆ `status = tryget (variable)` — similar to `get ()`, except that the process will not suspend if the mailbox is empty
    - ◆ Returns 0 if the mailbox is empty
    - ◆ Returns 1 if the mailbox is not empty and the message type matches the argument type
    - ◆ Returns -1 if not empty, but the message type does not match between the argument type



There are additional mailbox methods that are not covered in this presentation

## A FIFO Channel Model Using Interfaces and Mailboxes

```

interface fifo_channel_1 #(parameter      FifoSize = 8,
                           parameter type DataType = uniType);
    DataType write_data;
    DataType read_data;
    bit fifo_empty, fifo_full;

    mailbox #(DataType) FIFO = new(FifoSize);

    function automatic void Write (input DataType write_data);
        void' (FIFO.tryput(write_data));
        fifo_full = ~(FIFO.num < FifoSize);
    endfunction

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

    modport sender (input  write_data,
                   output fifo_full,
                   import Write);

    modport reader (output read_data,
                   output fifo_empty,
                   import Read);

endinterface: fifo_channel_1
    
```

parameterized size and type

storage is a bounded, typed mailbox

write/read methods are encapsulated in the interface

module port lists are encapsulated in the interface

## Connecting and Using the FIFO Interface Channel

“processor1” places data into the FIFO channel

```
module processor1 #(parameter type DataType = uniType)
    (fifo_channel_1.sender fifo,
     input clock, write_resetN);
    DataType data_packet;
    always_ff @(posedge clock)
        if (write_enable && !fifo.fifo_full)
            fifo.Write(data_packet);
    ...
endmodule
```

interface port

call to interface method

“processor2” retrieves data from the FIFO channel

```
module processor2 #(parameter type DataType = uniType)
    (fifo_channel_1.reader fifo,
     input clock, resetN);
    DataType data_packet;
    always_ff @(posedge clock)
        if (read_enable && !fifo.fifo_empty)
            fifo.Read(data_packet);
    ...
endmodule
```

interface port

call to interface method

a top-level module connects the interface to the modules

```
module top_level (input clock1, clock2, resetN);
    fifo_channel_1 #(.DataType(nniType)) FIFO ();
    processor1 #(.DataType(nniType)) p1 (FIFO, clock1, resetN);
    processor2 #(.DataType(nniType)) p2 (FIFO, clock2, resetN);
endmodule: top_level
```

instance of interface

connect the interface

## Synthesis Considerations of the FIFO Mailbox Model

- ◆ SystemVerilog mailboxes are *not* synthesizable
  - ◆ HDL Compiler (front-end to DC) does not currently support SystemVerilog’s object-oriented programming
  - ◆ Mailboxes do not have the level of detail needed to implement clock domain crossing FIFO logic



But don’t despair — this abstract model is still valuable!

- Easy and fast to model
- Can be used to verify inter-module communications
- Can be used to verify FIFO size at various clock speeds

The next few pages show another way to model a FIFO interface channel

## What's Next

- ✓ Introduce SystemVerilog
- ✓ Concepts of Communication Channels
- ✓ Tutorial on SystemVerilog Interfaces
- ✓ Modeling a FIFO Interface Channel Using Mailboxes
- **Modeling a FIFO Interface Channel Using Queues**
- Modeling a Synthesizable FIFO Interface Channel
- Conclusions



## FIFO Channel Example 2: Using SystemVerilog Queues

- ◆ SystemVerilog adds dynamic queues to Verilog
  - ◆ A dynamic array — can grow and shrink in size during simulation
  - ◆ Can represent FIFO, LIFO or other types of queues
- ◆ A queue is declared like an array, but using \$ for the range
  - ◆ Optionally, a maximum size for the queue can be specified

`int q1 [$];` ← an empty queue, with an unbounded size

`int q2 [$] = {1,2,3,5,8};` ← unbounded queue, initialized with 5 locations

`typedef struct {int a, b; bit flag} packet_t;`  
`packet_t q3 [$:16];` ← a bounded queue, with a maximum size of 16

A queue can only hold one data type, which is specified when the queue is declared

- ◆ SystemVerilog queues have several built-in methods
  - ◆ `push_front(<value>)` — adds a new location at the beginning of the queue with the value
  - ◆ `push_back(<value>)` — adds a new location at the end of the queue with the value
  - ◆ `variable = pop_front()` — removes the first element of the queue and returns its value
  - ◆ `variable = pop_back()` — removes the last element of the queue and returns its value
  - ◆ `insert(<index>, <value>)` — changes the value of a queue location without changing the queue size
  - ◆ `variable = <queue_name> [<index>]` — retrieves the value of a queue location without changing the queue size
  - ◆ `variable = size()` — returns the current number of elements in the queue

It is a run-time error to write to a full queue or to read from an empty queue

```

interface fifo_channel_2 #(parameter    FifoSize = 8,
                          parameter type DataType = uniType);
    DataType write_data;
    DataType read_data;
    bit fifo_empty, fifo_full;
    DataType FIFO [FifoSize];
    function automatic void Write (input DataType write_data);
        FIFO.push_back(write_data);
        fifo_full = ~(FIFO.PtrSize < FifoSize);
    endfunction
    function automatic void Read (output DataType read_data);
        read_data = FIFO.pop_front();
        fifo_empty = (FIFO.PtrSize == 0);
    endfunction
    modport sender (input write_data,
                   output fifo_full,
                   import Write);
    modport reader (output read_data,
                   output fifo_empty,
                   import Read);
endinterface: fifo_channel_2
  
```

parameterized size and type

storage is a bounded queue

write/read method names are the same as previous example

modport definitions are the same as previous example

## Connecting and Using the FIFO Interface Example 2

- ◆ The second FIFO example can be used in the same way as the first example
  - ◆ The modport definitions are exactly the same
  - ◆ The method names are exactly the same
  - ◆ The method formal arguments are the same

the FIFO mailbox example and the FIFO-queue example are completely interchangeable

```

module top_level (input clock1, clock2, resetN);
  fifo_channel_1 #(.DataType(nniType)) FIFO ();
  fifo_channel_2 #(.DataType(nniType)) FIFO (); ← instance of interface
  processor1 #(.DataType(nniType)) p1 (FIFO, clock1, resetN);
  processor2 #(.DataType(nniType)) p2 (FIFO, clock2, resetN);
endmodule: top_level

```

instance of interface

## Queues versus Mailboxes



Is there an advantage to modeling the FIFO with a SystemVerilog queue instead of a mailbox?

**Yes! SystemVerilog queues are arrays, instead of abstract objects, and therefore are synthesizable.**

- ◆ Mailbox advantages for representing a FIFO
  - ◆ Very easy to model
  - ◆ Closely approximates the methods in SystemC FIFO channels
  - ◆ Intuitive to use at a system level or testbench level of design
- ◆ Queue advantages for representing a FIFO
  - ◆ Easy to model at an abstract level
  - ◆ More closely approximates a hardware FIFO
  - ◆ Can be synthesizable (more about this on the next page)

## Synthesis Considerations of the FIFO Queue Model

- ◆ The SystemVerilog queue construct is synthesizable, but...
  - ◆ HDL Compiler does not yet support queues (as of July 2004)
  - ◆ The write and read methods are too abstract
    - ◆ Do not contain the information needed to properly synthesize cross domain clock synchronization



- ◆ FIFO synchronization is covered in two SNUG papers from 2002
  - ◆ “Simulation and Synthesis Techniques for Asynchronous FIFO Design”, by Cliff Cummings
  - ◆ “Simulation and Synthesis Techniques for Asynchronous FIFO Design with Asynchronous Pointer Comparisons”, by Cliff Cummings and Peter Alfke

These papers can be downloaded from [www.sunburst-design.com/papers](http://www.sunburst-design.com/papers)

## What's Next

- ✓ Introduce SystemVerilog
- ✓ Concepts of Communication Channels
- ✓ Tutorial on SystemVerilog Interfaces
- ✓ Modeling a FIFO Interface Channel Using Mailboxes
- ✓ Modeling a FIFO Interface Channel Using Queues
- **Modeling a Synthesizable FIFO Interface Channel**
- Conclusions



- ◆ The Synthesis process:
  - ◆ Expands an interface port to separate ports (using the modport port list)
  - ◆ Makes local copies of the imported methods (tasks and functions)
    - ◆ Requires that the methods be declared as automatic

```
interface BUS ;
  bit request, grant;
  task automatic Read(...);
  ...
endtask
modport master_ports
  (output request,
   input grant,
   import Read );
endinterface
```

**RTL Model**

```
module MASTER (BUS.master_ports a,
               input clock );
  ...
endinterface
```

**Synthesized Model**

```
module MASTER (output request,
               input grant,
               input clock );
  ...
endinterface
```

synthesized copy of the Read task

- ◆ To be synthesizable, a FIFO model must have:
  - ◆ RTL level clock synchronizers
    - ◆ FIFOs are typically used to transfer data between domains running on different, asynchronous clocks
  - ◆ Add RTL level full and empty flag generation
    - ◆ Full and empty flags can be affected by different clocks, and therefore require extra hardware
- ◆ The advantages of abstract queues can be maintained
  - ◆ The interface still encapsulates the communication details
  - ◆ Modules still communicate by calling write and read methods
  - ◆ Queue methods can be used instead of RTL write pointers and read pointers (maybe)

It remains to be proven as to how well synthesis will implement queue methods...

# Synthesizable FIFO Using Queues and RTL Code

```

interface fifo_channel_3 #(parameter      FifoSize = 8,
                          parameter type DataType = uniType);
    DataType write_data;
    DataType read_data;
    bit fifo_empty, fifo_full;

    DataType FIFO [$:FifoSize];
    function automatic void Write (input DataType write_data);
        FIFO.push_back(write_data);
    endfunction
    function automatic void Read (output DataType read_data);
        read_data = FIFO.pop_front();
    endfunction

    modport sender (input  write_data,
                   output fifo_full, write_clock, write_enable, write_resetN,
                   import Write,
                   // internal signals used within the interface
                   inout wr_ptr, wr_ptr_synced, wr_ptr_next, wr_cntr,
                      wr_cntr_next, wr_ptr_2nd_msb, wr_ptr_synced_2nd_msb
                   );
    modport reader (...);

```

**parameterized size and type**

**storage is a bounded queue, same as previous example**

**write/read methods only push/pop data (full and empty flags are set using RTL code, on next page)**

**modport definitions are more detailed than previous example to support RTL code**

**(example continued on next slide)**

# Synthesizable FIFO (continued)

```

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};
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};
always_ff @(posedge write_clock, negedge write_resetN)
    if (!write_resetN) wr_ptr <= 0;
    else wr_ptr <= wr_ptr_next;
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
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)
    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])
    );

```

**clock synchronization**

**write control**

**fifo\_full logic**

**(example continued on next slide)**

## Synthesizable FIFO (continued)

```

always_ff @(posedge read_clock, negedge read_resetN)
  if (!read_resetN) rd_ptr <= 0;
  else
    rd_ptr <= rd_ptr_next;
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

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

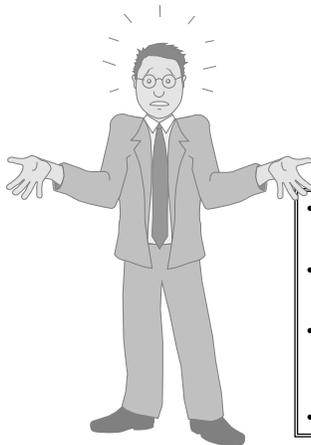
```

read control

fifo\_empty logic

## What's The Point?

If writing a synthesizable RTL FIFO Interface takes so much code, why not just leave the RTL logic in the modules?



- Interfaces represent inter-module communication
  - a FIFO in a module is another design hierarchy block
- Interfaces simplify module connections to the FIFO
  - One interface port instead of dozens of ports
- Interfaces encapsulate communication methods
  - Inter-connecting modules call interface methods instead of using discrete signals
- Abstract and RTL interfaces are easily interchangeable

## Connecting and Using the RTL/Queue FIFO Interface

- ◆ The RTL/queue FIFO interface is *almost* interchangeable with the abstract FIFO interfaces
  - ◆ The method names are the same
  - ◆ The method formal arguments are the same
  - ◆ But..., the modport definitions are not the same, because the RTL needs additional signals, such as clock and reset

```

module processor1 #(parameter type DataType = uniType)
    (fifo_channel_3.sender fifo,
     input_clock, write_resetN);
    DataType data_packet;
    always_ff @(posedge fifo.clock)
        if (write_enable && !fifo.fifo_full)
            fifo.Write(data_packet);
    ...
    
```

Clock and reset are part of the interface, instead of separate module ports

The methods and other FIFO signals are the same as with the abstract FIFO

Is there a way to make the models completely interchangeable?  
 Yes, clock and reset could also have been part of the abstract FIFO models (even if not used inside the interface)



## Synthesis Considerations

- ◆ Synthesis places some restrictions on interfaces and queues that should be noted
  - ◆ Interface declaration order
    - ◆ Synthesis must compile the interface and the module using the interface together, with the interface read in first
  - ◆ Methods must be automatic
    - ◆ Tasks and functions in an interface that are called by modules must be automatic, so that each module sees a local copy
  - ◆ Internal interface signals must be included in modports
    - ◆ Signals directly *and indirectly* referenced by a module must be listed in the module's modport list
  - ◆ Queues not yet implemented
    - ◆ At the time this paper was written, neither VCS nor HDL Compiler were supporting SystemVerilog queues
      - ◆ Examples were tested using static arrays as a work around)

These restrictions are **NOT** part of the SystemVerilog standard!

## What's Next

- ✓ Introduce SystemVerilog
- ✓ Concepts of Communication Channels
- ✓ Tutorial on SystemVerilog Interfaces
- ✓ Modeling a FIFO Interface Channel Using Mailboxes
- ✓ Modeling a FIFO Interface Channel Using Queues
- ✓ Modeling a Synthesizable FIFO Interface Channel
- **Conclusions**



## Conclusions

- ◆ Verilog inter-module communication is at a low-level
- ◆ SystemVerilog adds interfaces to Verilog
  - ◆ High-level inter-module communication
  - ◆ Encapsulates communication details in one place
  - ◆ Eliminates port declaration redundancy
- ◆ SystemC provides abstract communications channels, such as FIFOs, mutex, and semaphores
  - ◆ The built-in SystemC channels are not synthesizable
- ◆ SystemVerilog interfaces can effectively model the behavior of SystemC channels
  - ◆ Can be simple and abstract, but not synthesizable
  - ◆ Can be more detailed, to be synthesizable



## Future Work



- ◆ Make the three interface examples fully interchangeable
  - ◆ The method names and arguments are already the same
  - ◆ The modports need to be the same
- ◆ Add assertions to the interface
  - ◆ Assert that no writes are made to a full FIFO
  - ◆ Assert that no reads are made to an empty FIFO
- ◆ Prove that the mixed RTL/queue example is synthesizable
  - ◆ Waiting for HDL Compiler to implement the queue construct and queue methods

## Questions?

