

# Push-Button Engineering and SystemVerilog

Stuart Sutherland

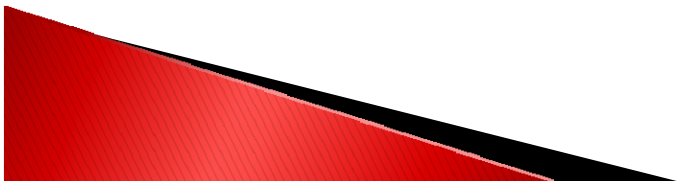
President

Sutherland HDL, Inc.



*Training engineers  
to be HDL wizards*

[www.sutherland-hdl.com](http://www.sutherland-hdl.com)



# About the Presenter...

## ▶ Stuart Sutherland

- SystemVerilog design and verification consultant
  - Involved in hardware design since 1982
  - Have been using Verilog since 1988
  - Specializes in providing Verilog/SystemVerilog training
  - Bachelors degree in Computer Science with an emphasis in Electronic Engineering
  - Pursuing a Master's degree in education
- Member of the IEEE Verilog and SystemVerilog standards groups since 1993
  - Editor of IEEE 1364 Verilog and IEEE 1800 SystemVerilog Language Reference Manuals (LRMs)
  - Author of multiple books on Verilog and SystemVerilog

# Push Button Engineering



- ▶ “Electronic Design Automation” means:
  - I think about what I want a new design to do
  - I sketch the design idea on a napkin
  - I scan in the napkin and *push a button*
  - A software program transforms the ideas into a fully working, verified netlist ready to implement in silicon!
- ▶ Do we have push button engineering today?
- ▶ Do we even want push push button engineering?

# Yes!



- ▶ Yes, we do want push button engineering
  - Design automation makes engineering easier
  - Design automation enables engineers to do bigger designs
  - Design automation does not eliminate engineering jobs
    - It eliminates the mundane work so we can be creative!
- ▶ Yes, we really do have push button engineering today
  - How did we design in the 1960s, 1970s, and 1980s?
  - Drafting was “automated” with schematic capture tools
  - Gate-level design was automated with RTL synthesis
- ▶ *SystemVerilog is a new level of push button engineering!*



# SystemVerilog is the Next Evolution of Verilog



## SystemVerilog-2005

verification design	}	assertions	mailboxes	classes	dynamic arrays
		test program blocks	semaphores	inheritance	associative arrays
		clocking domains	constrained random values	strings	references
		process control	direct C function calls		
		interfaces	packages	int	globals
		nested hierarchy	2-state modeling	shortint	enum
		unrestricted ports	packed arrays	longint	typedef
		automatic port connect	array assignments	byte	structures
		enhanced literals	queues	shortreal	unions
		time values and units	unique/priority case/if	void	casting
		specialized procedures	compilation unit space	alias	const
					break
					continue
					return
					do-while
					++ -- += -= *= /=
					>> << >>= <<=
					&=  = ^= %=

## Verilog-2001 / Verilog-2005

ANSI C style ports	standard file I/O	(* attributes *)	multi dimensional arrays
generate	\$value\$plusargs	configurations	signed types
localparam	`ifndef `elsif `line	memory part selects	automatic
constant functions	@*	variable part select	** (power operator)

## Verilog-1984 (IEEE Verilog-1995)

modules	\$finish \$fopen \$fclose	initial	wire reg	begin-end	+ = * /
parameters	\$display \$write	disable	integer real	while	%
function/tasks	\$monitor	events	time	for forever	>> <<
always @	`define `ifdef `else	wait # @	packed arrays	if-else	
assign	`include `timescale	fork-join	2D memory	repeat	

# Top-10 Reasons Engineers Love SystemVerilog

## ► Abstract modeling enhancements

- 1) Abstract data types
- 2) Non-ambiguous procedures
- 3) New programming statements and operators
- 4) Enhanced tasks and functions
- 5) Simplified netlists and interfaces



**More functionality  
in fewer lines of code!**

## ► Powerful verification enhancements

- 1) Assertions
- 2) Race-free testbench/design communication
- 3) Object-oriented types and techniques
- 4) Constrained random tests and coverage
- 5) Direct programming interface (DPI)



**More testing  
in fewer lines of code!**

# Top 5 Modeling Enhancements

## 1) Abstract data types

- ✓ High-level data types
- ✓ Relaxed rules on data type usage
- ✓ Enumerated types
- ✓ User-defined types
- ✓ Structures



**More functionality  
in fewer lines of code!**

2) Non-ambiguous procedures

3) New programming statements and operators

4) Enhanced tasks and functions

5) Simplified netlists and interfaces

# SystemVerilog Relaxes Variable Restrictions

- ▶ Verilog has strict rules on when to use a variable (e.g. **reg**) and when to use a net (e.g. **wire**)
  - Context dependent
  - A variable cannot be “driven” by a continuous assignment or an output port
- ▶ SystemVerilog allows variables to be used in the same places a net can be used:
  - Simplifies writing models
  - Can evolve from algorithmic to behavioral to RTL to gate without changing data types
  - **Limited to a single driver**
    - Prevents unintentional shared variable behavior



# Enumerated Types

- ▶ Verilog does not have enumerated types
  - Regular variables must be used for state machines, etc.
    - Must specify actual logic values in models
    - Cannot limit values to a legal set
- ▶ SystemVerilog adds enumerated types, using **enum**
  - Allows modeling at a more abstract level
  - Strongly typed — prevents many common coding errors

```
enum {WAIT, LOAD, READY} state;  
always @(state or opcode)  
  case (state)  
    WAIT: next_state = LOAD;  
    ...
```



- Can specify hardware implementation details (e.g. 1-hot)

```
enum bit [3:0] {WAIT=3'b001, LOAD=3'b010, READY=3'b100} state;
```

# Structures

## ▶ SystemVerilog adds C-like structures to Verilog

A structure is a collection of elements that can be different types and sizes

- Can be used to bundle several variables into one object
- Can assign to individual signals within the structure
- Can assign to the structure as a whole
- Can pass structures through ports and to tasks or functions

```
struct {  
    int  i0, i1;  
    bit [7:0] opcode;  
} instruction_word;  
...  
instruction_word = {56,15,8'hF1};
```

The structure declaration  
syntax is similar to C

*Now I can work with whole  
blocks of data as one object!*



# Top 5 Modeling Enhancements

- 1) Abstract data types
- 2) Non-ambiguous procedures
  - ✓ Hardware specific procedures
  - ✓ Unique and priority decisions
- 3) New programming statements and operators
- 4) Enhanced tasks and functions
- 5) Simplified netlists and interfaces

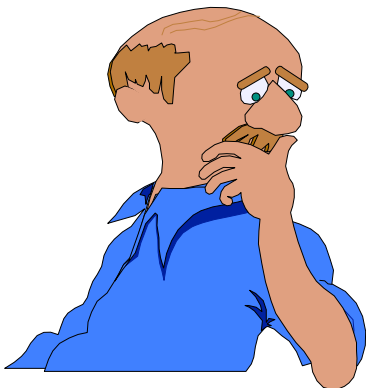


**More functionality  
in fewer lines of code!**

# Inferring Hardware

## From always Procedures

- ▶ Verilog's **always** procedure is a general purpose coding procedure
  - Used to model **combinational logic**, **latched logic**, **sequential logic**, **bus-functional logic**, and **testbench logic**
  - Software tools must “infer” (*guess*) what type of hardware an engineer intended based on context



```
always @(posedge clock)
begin
  if (mode) q1 = a + b
  else      q1 = a - b;
  q2 <= q1 | (q2 << 2);
  q2++;
end
```

Is **q1** combinational logic or sequential logic?

Does **q2** ever increment?

# Hardware Specific Procedures

- ▶ SystemVerilog adds hardware-oriented procedures
  - `always_ff` models sequential logic
  - `always_comb` models combinational logic
  - `always_latch` models latch-based logic
- ▶ These hardware-oriented procedures
  - Allow software tools to check that designer's intent has been correctly modeled
  - Require simulation, synthesis, and formal tools to use the same rules

```
always_comb  
  if (!mode)  
    y = a + b;  
  else  
    y = a - b;
```

no sensitivity list

contents must follow synthesis rules

Tools now know the designer's intent, and can verify that the code models combinational behavior



# Unique and Priority Decisions

- ▶ Verilog defines that **case** decisions statements execute with priority encoding
  - In simulation, only the first matching branch is executed
  - “full\_case” and “parallel\_case” pragmas are used to tell synthesis to optimize
    - *Simulation does not check that these pragmas will work!*
- ▶ SystemVerilog adds **unique** and **priority** modifiers:
  - Works like the pragmas for synthesis
  - *Simulation will warn if decision statements do not match the specified behavior*

```
unique case(state)
  red:    if (sensor = 1) next_state = green;
  yellow: if (yellow_downcnt = 0) next_state = red;
  green:  if (green_downcnt = 0) next_state = yellow;
endcase
```

Automatically catches design errors



# Top 5 Modeling Enhancements

- 1) Abstract data types
- 2) Non-ambiguous procedures
- 3) **New programming statements and operators**
  - ✓ Increment and decrement operators
  - ✓ Assignment operators
  - ✓ Enhanced for-loops
  - ✓ New types of loops
- 4) Enhanced tasks and functions
- 5) Simplified netlists and interfaces



**More functionality  
in fewer lines of code!**

# Enhanced For-Loops

## ▶ In Verilog **for** loops

- The loop control variable be declared outside of the loop

```
integer i;  
always @(a or b)  
    for (i = 0; i <= 255; i = i + 1)  
        ...  
always @(posedge clock)  
    for (i = 0; i <= 255; i = i + 1)  
        ...
```

The declaration and usage can be separated by many lines of code

Concurrent loops can interfere with each other

## ▶ SystemVerilog enhances **for** loops

- The loop variable can be declared in the **for** loop
- Can have multiple initial and step assignments

```
always @(a or b)  
    for (int i = 0; i <= 255; i++)  
        ...  
always @(posedge clock)  
    for (int i = 0, j = 255; i <= 255; i++, j--)  
        ...
```

Loop variable is declared local to the loop

Concurrent loops cannot interfere

# New Types of Loops

- ▶ Verilog has **for**, **repeat** and **while** loops
  - The control is tested at the beginning of each loop pass
- ▶ SystemVerilog adds a **do—while** loop
  - The control is tested at the end of each pass of the loop

```
do
begin
    done = 1; OutOfBound = 1;
    if      (addr < 0)    done = 1;
    else if (addr > 255) OutOfBound = 1;
    else
        begin out = mem[addr]; addr -= 8; end
end
while (addr > -9 || addr <= 255)
```

the loop will execute at least once, setting **done** and **OutOfBound** flag

- ▶ SystemVerilog adds a **foreach** loop
  - Iterates through arrays of any size and number of dimensions

```
int array1 [0:7][0:255];
foreach ( array1 [i,j] )
    $display("i=%d  j=%d  array1[i][j]=%d", i, j, array1[i][j]);
```

# Top 5 Modeling Enhancements

- 1) Abstract data types
- 2) Non-ambiguous procedures
- 3) New programming statements and operators
- 4) Enhanced tasks and functions
  - ✓ C-like function returns
  - ✓ Default data types for formal arguments
  - ✓ Passing arrays and structures as arguments
  - ✓ Passing arguments by name
  - ✓ Passing arguments by reference
- 5) Simplified netlists and interfaces



**More functionality  
in fewer lines of code!**

# Task/Function Arguments: Passing By Name

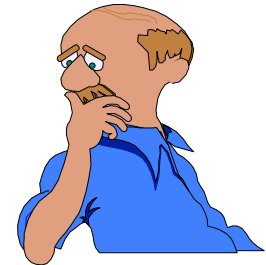
## ► In Verilog:

- Values are passed to tasks and functions by position

How can I know if `stack` and `data_bus` are in the right order?

```
always @(posedge clock)
    result <= subtractor( stack, data_bus );

function integer subtractor(input integer a, b);
    subtractor = a - b;
endfunction
```



## ► In SystemVerilog:

- Values can be passed using the formal argument name

```
always @(posedge clock)
    result <= subtractor( .b(stack), .a(data_bus) );

function int subtractor(int a, b);
    return(a - b);
endfunction
```

Uses same syntax as named  
module port connections

# Task/Function Arguments: Pass By Reference

## ▶ With Verilog tasks and functions:

- Input values are copied in — outputs are copied out

```
[always @(posedge clock)
  result = mult( data, stack[14] );
function integer mult(input [31:0] a, b);
  ...
```

**data** and **stack** are copied into the function when it is called

## ▶ With SystemVerilog:

- Tasks and functions can “reference” the calling arguments
  - Uses the keyword **ref** instead of input, output or inout

```
always @(data)
  synch (data, clock);
task synch (int d, ref clk);
  @(posedge clk) //sync to clock
  ...
```

The task is sensitive to changes on **clock** in the calling scope

```
bit [31:0] stack [0:255];
always @(posedge clock)
  result = mult( data, stack );
function int mult(int a, ref b);
  ...
```

The function directly references **stack** in the calling scope

# Top 5 Modeling Enhancements

- 1) Abstract data types
- 2) Non-ambiguous procedures
- 3) New programming statements and operators
- 4) Enhanced tasks and functions
- 5) **Simplified netlists and interfaces**
  - ✓ Relaxed port connection rules
  - ✓ Automatic netlist connections
  - ✓ Interfaces



**More functionality  
in fewer lines of code!**

# Module Port Connections

- ▶ Verilog restricts the data types that can be connected to module ports
  - Only net types on the receiving side
  - Nets, regs or integers on the driving side
  - Choosing the correct type frustrates Verilog modelers
- ▶ SystemVerilog removes restrictions on port connections
  - Any data type on either side of the port
  - Real numbers (floating point) can pass through ports
  - Arrays can be passed through ports
  - Structures can be passed through ports



# Module Port Connection Shortcuts

- ▶ Verilog module instances can use port-name connections
  - Must name both the port and the net connected to it

```
module dff (output q, qb,  
           input clk, d, rst, pre);
```

```
...  
endmodule
```

```
module chip (output [3:0] q,  
            input [3:0] d, input clk, rst, pre);  
  dff dff1 (.clk(clk), .rst(rst), .pre(pre), .d(d[0]), .q(q[0]));
```

can be verbose and redundant

- ▶ SystemVerilog adds **.name** and **.\*** shortcuts

- **.name** connects a port to a net of the same name

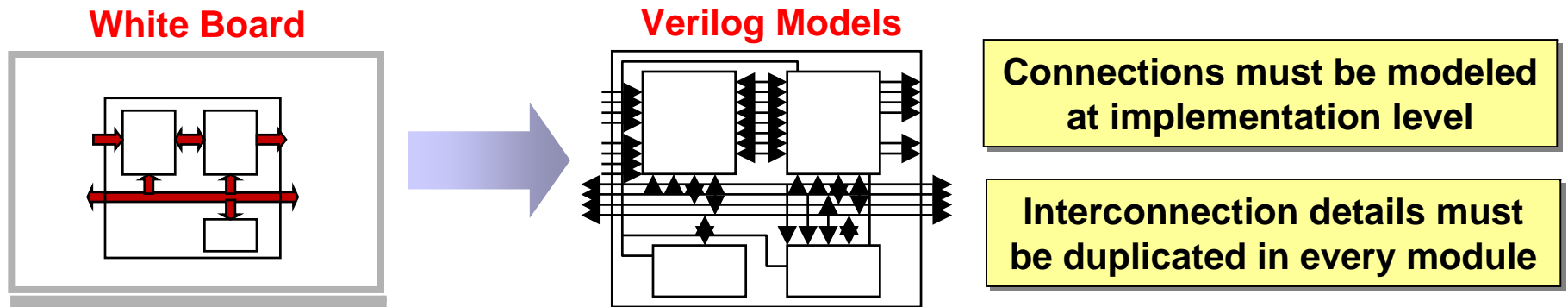
```
dff dff1 (.clk, .rst, .pre, .d(d[0]), .q(q[0]));
```

- **.\*** automatically connects all ports and nets with the same name

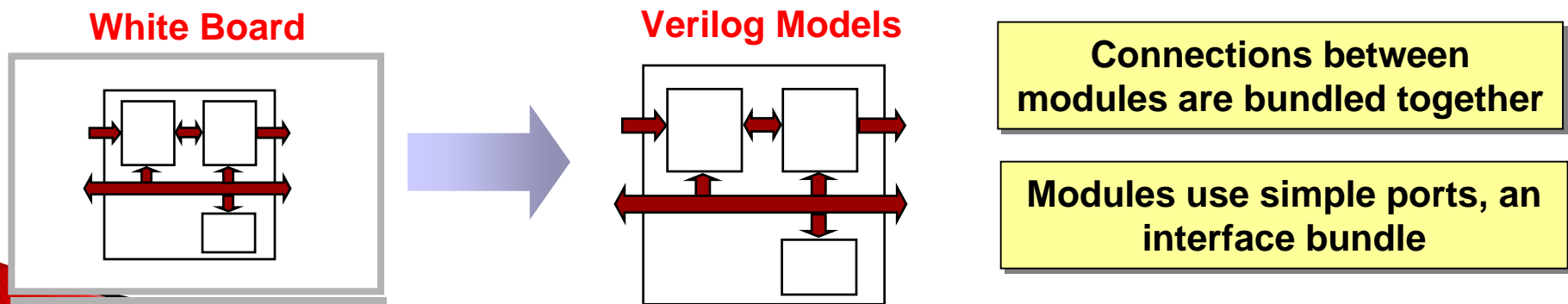
```
dff dff1 (.*, .q(q[0]), .d(d[0]));
```

# Interfaces: Communication Channels

- ▶ Verilog connects models using detailed module ports



- ▶ SystemVerilog adds compound “interface” ports



# Interfaces

## Can Contain Logic

- ▶ SystemVerilog interfaces are more than a bundle of wires
  - Interfaces can contain declarations
    - Variables, parameters and other data that is shared by all users of an interface can be declared in one location
  - Interfaces can contain tasks and functions (“methods”)
    - Operations shared by multiple connections to the interface can be coded in one place
  - Interfaces can contain procedures
    - Protocol checking and other verification can be built into the interface

# Top 5 Verification Enhancements

## 1) Assertions

- ✓ Clocked sequential assertions
- ✓ Unified assertions with PSL
- ✓ Assertions for design engineers
- ✓ Assertions for verification engineers



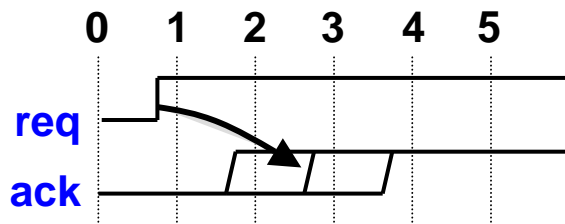
**More testing  
in fewer lines of code!**

- 2) Race free testbench / design communications
- 3) Object Oriented test programming
- 4) Constrained random tests and functional coverage
- 5) Direct Programming Interface

# Assertions

▶ An assertion is a design condition that must be true

- Assertions can be written in Verilog, but...It's a lot of code!



Each request must be followed by an acknowledge within 1 to 3 clock cycles



To test for a sequence of events requires several lines of Verilog code

- Difficult to write, read and maintain
- Cannot easily be turned off during reset or other don't care times

```
always @(posedge req) begin
  @(posedge clk) ; // synch to clock
  fork: watch_for_ack
    parameter N = 3;
    begin: cycle_counter
      repeat (N) @(posedge clk);
      $display("Assertion Failure", $time);
      disable check_ack;
    end // cycle_counter
    begin: check_ack
      @(posedge ack)
      $display("Assertion Success", $time);
      disable cycle_counter;
    end // check_ack
  join: watch_for_ack
end
```

# SystemVerilog Assertions

## ▶ SystemVerilog adds assertion syntax and semantics

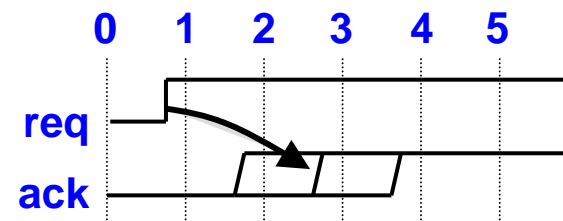
- *Immediate assertions* test for a condition at the current time

```
always @(state)
  assert (reset && (state != RST)) else $fatal);
```

generate a fatal error  
if reset is true  
and not in the reset state

- *Concurrent assertions* test for a sequence of events spread over time

a complex sequence can be  
defined in very concise code



```
a_reqack: assert property (@(posedge clk) req ##[1:3] ack;) else $error;
```

One line of SVA code replaces all the Verilog code in the previous example!

# Advantages of SystemVerilog Assertions

- ▶ SystemVerilog Assertions have several advantages
  - Concise syntax!
    - Many lines of Verilog code can be represented in one line
  - Ignored by Synthesis!
    - Don't have to hide Verilog checker code within convoluted `translate_off` / `translate_on` synthesis pragmas
  - Can be disabled!
    - SystemVerilog assertions can be turned off during reset, or until simulation reaches a specific simulation time
  - Can have severity levels!
    - SystemVerilog assertion failures can be non-fatal or fatal
    - Simulators can suppress messages based on severity

# Assertions Are for Design Engineers, Too



- ▶ Design engineers should write assertions to verify assumptions that affect the functionality of a design block
  - The assertion documents the designer's assumptions
    - Example: An ALU design assumes that the A, B and opcode inputs will never have a logic X or Z value
- ▶ Verification engineers should write assertions that verify design functionality meets the overall design specification
  - The assertion verifies that the designer correctly implemented the specification
    - Example: The zero flag output of the ALU block should only be set if the ALU result output is zero

# Top 5 Verification Enhancements

- 1) Assertions
- 2) Race free testbench / design communications
  - ✓ Enhanced scheduling
  - ✓ Program block
  - ✓ Clocking domain
- 3) Object Oriented test programming
- 4) Constrained random tests and functional coverage
- 5) Direct Programming Interface



**More testing  
in fewer lines of code!**

# Special Test Bench “Program Blocks”

- ▶ Verilog uses hardware modules to model the test bench
  - No special semantics to avoid race conditions with the design
- ▶ SystemVerilog adds a special “program block” for testing
  - Identifies verification code
  - Program events are automatically synchronized to hardware events to avoid races

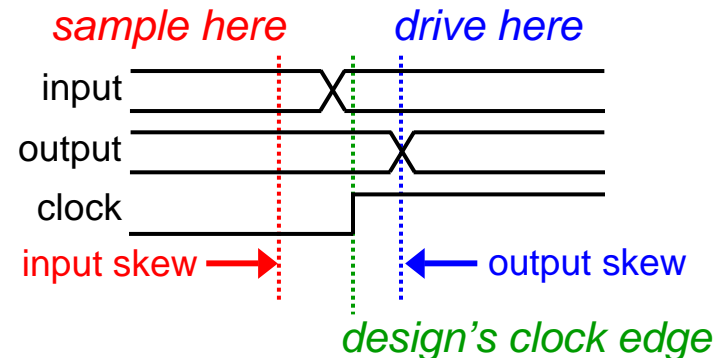
```
program test (input clk, input [15:0] addr, inout [7:0] data);  
  initial begin  
    @(negedge clk) data = 8'hC4;  
                    address = 16'h0004;  
    @(posedge clk) verify_results;  
  end  
  task verify_results;  
    ...  
endprogram
```

The testbench is called a “program block” because test code is more like software than hardware

# Cycle-Based Verification Timing

- ▶ Verilog has no special clock semantics for verification
  - Must write extra code to avoid race conditions with design
- ▶ SystemVerilog adds a special *“clocking domain”* construct
  - Specifies a test “input skew” and “output skew”
    - Offsets verification events are from the design clock

```
clocking bus @(posedge clock);
default input #2ns output #1ns;
input    enable, full;
inout    data;
output   empty;
output   #6ns reset;
endclocking
```



**Input skew is how long before a clock edge a value should be sampled**

- The default is 1step (one simulation time unit)

**Output skew is how long after the clock edge before a value should be driven**

- The default is 0

# Top 5 Verification Enhancements

- 1) Assertions
- 2) Race free testbench / design communications
- 3) Object Oriented test programming
  - ✓ Classes
  - ✓ Inheritance
  - ✓ Polymorphism
  - ✓ Data hiding and protection
  - ✓ Mailboxes and semaphores
- 4) Constrained random tests and functional coverage
- 5) Direct Programming Interface



**More testing  
in fewer lines of code!**

# Object Oriented Classes

## ▶ SystemVerilog adds “classes” to the Verilog language

- Allows Object Oriented Programming techniques

- Class definitions can contain

- Data declarations (“*properties*”)
- Tasks and functions (“*methods*”)

- Similar to C++ and VERA:

- Can have **inheritance** and **polymorphism**
- Can have **public**, **local** or **private** data encapsulation
- Automatic **garbage collection**

```
class Packet ;
    rand bit[3:0] cmd;
    local int      status;
    myStruct      header;
    constraint c1 { cmd[0] == 1'b0; }
    function int  get_status();
        ...
    endfunction
    task set_cmd(input bit a);
        ...
    endtask
endclass
```

```
class udpPacket extends Packet ;
    ...
endclass
```

# Enhanced Synchronization: Mailboxes and Semaphores

## ▶ SystemVerilog has built-in classes to synchronize tests

### ◦ Semaphores

- Represents a bucket with a fixed number of keys
- Class methods are used to check keys in and out
- If keys are available, a process suspends execution and waits for keys before continuing



### ◦ Mailboxes

- Represents a FIFO to exchange messages between processes
- Class methods allow adding a message or retrieving a message
- If no message is available, a process can either wait until a message is added, or continue and check again later



# Top 5 Verification Enhancements

- 1) Assertions
- 2) Race free testbench / design communications
- 3) Object Oriented test programming
- 4) Constrained random tests and functional coverage
  - ✓ Distributed random values
  - ✓ Cyclic random values
  - ✓ Constrained random values
  - ✓ Separate RNG for each object
  - ✓ Functional coverage
- 5) Direct Programming Interface



**More testing  
in fewer lines of code!**

# Enhanced Random Values

- ▶ Verilog `$random` returns a 32-bit signed random number
  - Cannot constrain the random values returned
- ▶ SystemVerilog adds constrained random values
  - `rand` — distributed random numbers
  - `randc` — cyclic random numbers

```
class Bus;  
  rand bit [15:0] addr;  
  rand bit [31:0] data;  
  randc bit [3:0] mode;  
  constraint word_align {addr[1:0] == 2'b0;}  
endclass
```

The variables to receive random values, and any constraints on the random values, are specified using class objects and class methods

```
Bus bus = new;  
repeat (50)  
  begin  
    assert (bus.randomize());  
    ...
```

Generate 50 random values for the variables in the “bus” object

Calling `randomize()` selects values for all random variables in a class object, using the constraints specified

# Functional Coverage

- ▶ SystemVerilog functional coverage provides:
  - User specified **cover groups** (encapsulates coverage models)
    - **Cover points**
      - Variables and expressions to be tracked
    - **Cover bins**
      - Values or value ranges to be tracked
    - **Cross coverage**
      - Value sets received by multiple cover points

```
enum {WAIT, LOAD, READY} state, next_state;  
logic [7:0] data;  
  
covergroup cg_state_coverage @(posedge clk);  
  states: coverpoint state;  
  values: coverpoint data {  
    bins low = { [0:127] };  
    bins high = { [128:255] };  
  }  
endgroup
```

# Top 5 Verification Enhancements

- 1) Assertions
- 2) Race free testbench / design communications
- 3) Object Oriented test programming
- 4) Constrained random tests and functional coverage
- 5) **Direct Programming Interface**
  - ✓ A simplified complement to the Verilog PLI
  - ✓ Verilog can directly call C functions
  - ✓ C functions can directly call Verilog functions



**More testing  
in fewer lines of code!**

# Direct Programming Interface

- ▶ Verilog uses the Programming Language Interface (PLI) to allow Verilog code to call C language code
  - Powerful capabilities — traversing hierarchy, controlling simulation, modifying delays and time synchronization
  - Difficult to learn
  - Too complex for many types of applications
- ▶ SystemVerilog adds a **Direct Programming Interface (DPI)**
  - Verilog code can directly call C functions
  - C functions can directly call Verilog functions
  - Can do many things more easily than the PLI
    - Cannot do everything the PLI can do
  - Ideal for interfacing to bus-functional models written in C

# SystemVerilog Adds a New Level of Push Button Engineering

- ▶ Modeling enhancements enable engineers to model larger designs and avoid common design errors
  - Abstract data types
  - Non-ambiguous procedures
  - New programming statements and operators
  - Enhanced tasks and functions
  - Simplified netlists and interfaces
- ▶ Powerful verification enhancements enable verifying complex designs more efficiently and effectively
  - Assertions
  - Race-free testbench/DUT communication
  - Object-oriented types and techniques
  - Constrained random test and coverage
  - Direct Programming Interface (DPI)

