



SUTHERLAND
& HDL
LCDM
ENGINEERING

HDVL += (HDL & HVL)

***SystemVerilog 3.1: The Hardware
Description AND Verification Language***

Stuart Sutherland
Sutherland HDL, Inc.
www.sutherland-hdl.com

Don Mills
LCDM Engineering
www.lcdm-eng.com

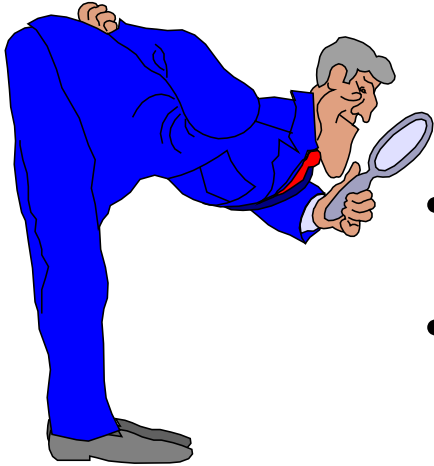


*Training engineers
to be HDL wizards*

Art The Muscian's Fan Club



This presentation will...



- Define what is “SystemVerilog”
- Provide an overview of the major features in SystemVerilog
- Show the current status of the SystemVerilog standard
- Discuss Synopsys support for SystemVerilog
- **The primary goal is to introduce you to the many exciting features in SystemVerilog — It’s a lot!**

What is SystemVerilog?

- SystemVerilog extends the IEEE 1364 Verilog-2001 standard
 - Adds abstract, system-level modeling constructs to Verilog
 - Adds extended test bench features to Verilog
- SystemVerilog is being released in two primary stages
 - SystemVerilog 3.0 (released June 2002)
 - Extends the hardware modeling aspects of Verilog
 - SystemVerilog 3.1 (to be released June 2003)
 - Extends the verification aspects of Verilog
- SystemVerilog is being defined by Accellera
 - Accellera is a consortium of EDA and engineering companies
 - Expected to be adopted by the IEEE in the next revision to the 1364 Verilog standard

System Verilog

3.1	<ul style="list-style-type: none"> assertions test program blocks clocking domains process control 	<ul style="list-style-type: none"> mailboxes semaphores constrained random values direct C function calls 	<p style="color: green; margin: 0;">from C / C++</p> <ul style="list-style-type: none"> classes inheritance strings dynamic arrays associative arrays references
3.0	<ul style="list-style-type: none"> interfaces nested hierarchy unrestricted ports automatic port connect enhanced literals time values and units specialized procedures 	<ul style="list-style-type: none"> dynamic processes 2-state modeling packed arrays array assignments enhanced event control unique/priority case/if 	<ul style="list-style-type: none"> int shortint longint byte char shortreal void globals enum typedef structures unions casting const break continue return do-while ++ -- += -= *= /= >> << >>= <<= &= = ^= %=

Verilog-2001

<ul style="list-style-type: none"> ANSI C style ports generate localparam constant functions 	<ul style="list-style-type: none"> standard file I/O \$value\$plusargs `ifndef `elsif `line @* 	<ul style="list-style-type: none"> (* attributes *) configurations memory part selects variable part select 	<ul style="list-style-type: none"> multi dimensional arrays signed types automatic ** (power operator)
--	--	---	--

Verilog-1995

<ul style="list-style-type: none"> modules parameters function/tasks always @ assign 	<ul style="list-style-type: none"> \$finish \$fopen \$fclose \$display \$write \$monitor `define `ifdef `else `include `timescale 	<ul style="list-style-type: none"> initial disable events wait # @ fork-join 	<ul style="list-style-type: none"> wire reg integer real time packed arrays 2D memory 	<ul style="list-style-type: none"> begin-end while for forever if-else repeat 	<ul style="list-style-type: none"> + = * / % >> <<
---	--	---	--	--	---

SystemVerilog 3.0 Features

The presentation will only highlight some of these exciting features
The full paper has more details on all of these features

- SystemVerilog 3.0 enhances Verilog **modeling** constructs

- Interfaces between modules
- Global declarations
- Global tasks and functions
- Global statements
- Time unit and precision enhancements
- C language data types
- 2-state data types
- User defined types
- Enumerated types
- Structures and unions
- Type casting
- Literal value enhancements

- Specialized always procedures
- Increment/decrement operators
- Unique decision statements
- Priority decision statements
- Bottom testing do-while loop
- Jump statements
- Statement labels
- Block name enhancements
- Task and function enhancements
- Continuous assignment enhancements
- Module port connection enhancements



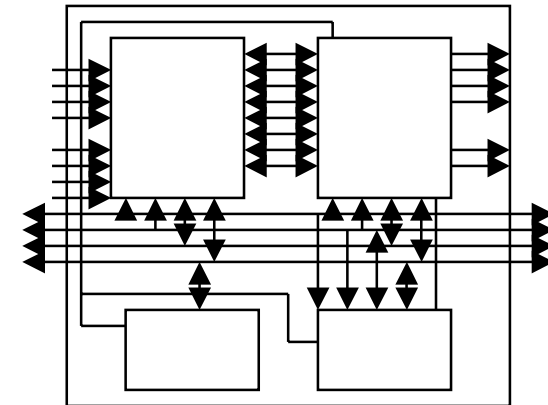
SystemVerilog 3.0's Roots

SystemVerilog is built on proven technologies!

- Most features in **SystemVerilog 3.0** are from three sources:
 - **A subset of the SUPERLOG language**
 - Co-design Automation donated the “synthesizable” portion of its SUPERLOG language to Accellera
 - Created by Peter Flake, Phil Moorby, and Simon Davidmann
 - **An implementation of the OVL assertions library**
 - Verplex donated their work on assertion libraries to Accellera
 - Real Intent and Co-design donated their assertion syntax and semantics to Accellera
 - **Proposals from the Accellera SystemVerilog committee**
 - The committee reviewed and refined the donations received
 - The committee defined additional enhancements to Verilog

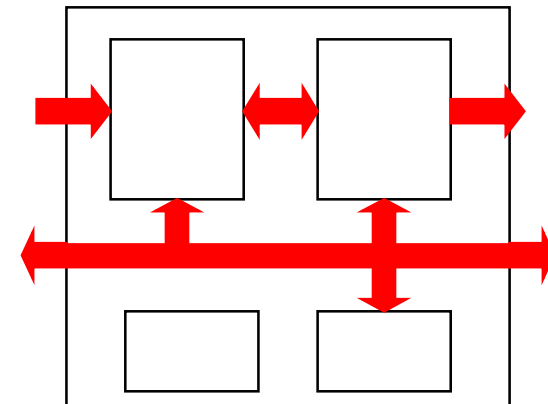
- Verilog connects models using module ports

- Requires detailed knowledge of connections to create module
- Difficult to change connections if design changes
- Port declarations must be duplicated in many modules



- SystemVerilog adds an **interface** block

- Connections between models are bundled together
- Connection definitions are independent from modules
- Interfaces can contain declarations, assertions and protocol checking



- Verilog has hardware-centric net data types
 - Intended to represent real connections in a chip or system
 - Models detailed hardware behavior using 4-state logic, strength levels and wired logic resolution
 - Can reduce simulation performance
 - Most hardware models only need abstract 2-state logic
- SystemVerilog adds abstract data types
 - 2-state types: `int`, `shortint`, `longint`, `char`, `byte`, `bit`
 - 4-state type: `logic`
 - Special types: `void`, `shortreal`
 - Allows modeling at a C-language level of abstraction
 - Efficient data types for simulation performance

- Verilog does not have enumerated types

- All signals must be declared
- All signals must be initialized to a value

(see Cliff Cumming's paper on modeling state machines with SystemVerilog this afternoon for more detailed examples)

- SystemVerilog adds enumerated types, using **enum**, as in C

```
enum {WAIT, LOAD, READY} states;
```

- Optionally, the data type of the enumerated types can be declared
 - The default data type is **int**
- Optionally, the values of enumerated names can be specified
 - The default initial value is 0
 - Subsequent names are incremented from the previous value

```
enum reg [1:0] {WAIT=2'b01, LOAD=2'b10, READY} states;
```

- Verilog does not have user-defined data types
- SystemVerilog adds user-defined types
 - Uses the **typedef** keyword, as in C

```
typedef int unsigned uint;  
uint a, b; //two unsigned integers
```

```
typedef enum {FALSE=1'b0, TRUE} boolean;  
boolean ready; //signal "ready" can be FALSE or TRUE
```

- SystemVerilog adds structures to Verilog
 - A structure is a collection of elements that can be different data types
 - Can be used to bundle several variables into one object
 - Can assign values to individual signals within the structure
 - Can assign values to the entire structure as one object
 - Can pass structures through ports and to tasks or functions

```

struct {
    real r0, r1;
    int  i0, i1;
    bit [15:0] opcode;
} instruction_word;
...
instruction_word.opcode = 16'hF01E;
  
```

The structure declaration is the same as in C

- Verilog does not have increment and decrement operators

```
for (i = 0; i <= 255; i = i + 1)
    ...
```

- SystemVerilog adds:

- ++ and -- increment and decrement operators
- +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=, <<<=, >>>= assignment operators

```
for (i = 0; i <= 255; i++)
    ...
```



- Verilog procedures are general purpose procedures
 - **always** procedures model **combinational**, **sequential** and **latched** logic
 - Software tools must “infer” (**guess**) what type of hardware an engineer intended based on context
 - **if-else-if** decisions and **case** statements execute with priority encoding (only the first matching branch is executed)
 - Synthesis will infer parallel execution based on context
- SystemVerilog adds specific procedures and decision modifiers

always_comb always_ff always_latch unique priority

**No ambiguities
on design intent!**

```

always_comb
begin
  next_state = state;
  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
end
  
```

Tools can verify the code models intended behavior



Verilog Hierarchy Enhancements

- SystemVerilog adds three major enhancements to representing design hierarchy
 - A global name space
 - Can contain declarations, tasks, functions and statements
 - Any module can reference global declarations
 - Avoids declaring the same information in multiple modules
 - Nested module declarations
 - Nested modules are only visible to their parent module
 - Protects hierarchy within Intellectual Property models
 - Automatic netlist connections
 - New `.name` and `.*` automatically connect nets and ports that have the same name

- SystemVerilog 3.1 enhances Verilog *verification* constructs
 - Still under development
 - Expected to be ratified in June 2003

- Test bench program blocks
- Assertions
- Clocking domains
- Constrained random values
- Mailbox process synchronization
- Semaphore process synchronization
- Event data type enhancements
- Dynamic process control
- References (safe pointers)

- Object Oriented classes
- String data type
- Dynamic arrays
- Associative arrays
- Enumerated type enhancements
- Tasks and function enhancements
- Enhanced for loops
- Enhanced fork—join
- Direct C language interface
- Assertion API

SystemVerilog

verification modeling	}	assertions test program blocks clocking domains process control	mailboxes semaphores constrained random values direct C function calls	classes inheritance strings	dynamic arrays associative arrays references
		interfaces nested hierarchy unrestricted ports automatic port connect enhanced literals time values and units specialized procedures	dynamic processes 2-state modeling packed arrays array assignments enhanced event control unique/priority case/if	int shortint longint byte char shortreal void	globals enum typedef structures unions casting const

Verilog-2001

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

Verilog-1995

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

SystemVerilog 3.1's Roots

SystemVerilog is built on proven technologies!

- Most features in **SystemVerilog 3.1** are from four sources:
 - The Synopsys **VERA-Lite** Hardware Verification Language
 - Powerful constructs for modeling test benches
 - The Synopsys **VCS DirectC** Application Programming Interface (API)
 - Allows Verilog code to directly call C functions (no PLI needed)
 - Allows C functions to directly call Verilog tasks and functions
 - The IBM **Property Specification Language (PSL)** assertion technology
 - Commonly referred to as **“Sugar”**
 - Allows design and verification engineers to add checks to models
 - The Synopsys **Assertion Application Programming Interface (API)**
 - Allows PLI applications to access and control assertions

The Accellera SystemVerilog committee is integrating these donations into SystemVerilog

- **Unifying the donations to avoid overlapping capabilities and keywords**
- **Modifying syntax to make everything look like Verilog**
- **Adding additional verification enhancements based on user and EDA vendor requests**

- Verilog uses hardware modeling constructs to model the verification test bench
 - No special semantics avoid race condition with the design
- SystemVerilog adds a special “program block” for testing
 - Events in a program block execute in a “verification phase”
 - Synchronized to hardware simulation events to avoid races

```

program test (input clk, input [15:0] addr, inout [7:0] data);

    @(negedge clk) data = 8'hC4;
                    address = 16'h0004;
    @(posedge clk) verify_results;

    task verify_results;
        ...
    endtask
endprogram
  
```

- Verilog requires the test bench modeler add special test clocks and delays to avoid race conditions with the design clock
- SystemVerilog adds a special “*clocking domain*” construct
 - Specifies a test “input skew” and “output skew”
 - Ensures verification events are offset from the design clock
 - Allows the test bench to use the same clock(s) as the design, without having race conditions with the design

```

clocking bus @(posedge clk);
  default input #2ns output #1ns; //default I/O skew
  input    enable, full;
  inout   data;
  output  empty;
  output  #6ns reset; //reset skew is different than default
endclocking

```

- Verilog does not provide an assertion construct
 - Model checking must be done by hard-coded logic or the Verilog PLI
- SystemVerilog adds assertion syntax and semantics
 - Assertion information is built into the language
 - Combines the assertion capabilities of Sugar (PSL), OVA, VERA, Verplex and other assertion languages
 - Uses Verilog-like syntax regular expressions

```
always @(State)
  if (State == FETCH)
    assert (request;[0:3];grant) @(posedge clk);
```

request must be true in the
FETCH state, and **grant** must
be true between
1 and **4** clocks later

- SystemVerilog adds “*classes*” to the Verilog language
 - Allows Object Oriented programming techniques
 - Can be used in the test bench
 - Can be used in hardware models
 - Classes can contain
 - Data declarations, referred to as the object’s “*properties*”
 - Tasks and functions, referred to as the object’s “*methods*”
 - Classes can have “*inheritance*”
 - Similar to C++

```

class Packet ;
  bit [3:0] command;
  bit [39:0] address;
  bit [4:0] master_id;
  integer time_requested;
  integer time_issued;
  integer status;

  task clean();
    command = 4'h0;
    address = 40'h0;
    master_id = 5'b0;
  endtask

  task issue_request( int delay );
    ... // send request to bus
  endtask
endclass

```

- The Verilog `$random` random number generator returns a 32-bit signed random number
 - No way to constrain the range of random values
- SystemVerilog adds:
 - `$urandom` — generates unsigned 32-bit random numbers
 - `$urandom_range` — generates unsigned 32-bit random numbers within a specified range
 - `rand` built-in class for creating distributed random number generators
 - A random value can occur more than once before all possible values in a constrained range have occurred
 - `randc` built-in class for creating cyclic random number generators
 - All possible values within a constrained range will occur once and only once, and then a new random sequence will begin
 - Built-in class methods are used to constrain random values

- SystemVerilog adds constructs for synchronizing processes
 - Semaphores are a built-in class that represents a bucket with a fixed number of keys
 - Class methods are used to check keys in and out
 - A process can check out one or more keys, and return them later
 - If not enough keys are available, the process execution stops and waits for keys before continuing
 - Mailboxes are a built-in class that allows messages to be exchanged between processes
 - Methods allow adding a message or retrieving a message
 - If no message is available to retrieve, the process can either wait until a message is added, or continue and check again later

- Verilog's **fork-join** blocks execution of the next statement until all the forked parallel threads have completed execution

```

initial
begin
  fork
    send_packet_task(1,255,0);
    send_packet_task(7,128,5);
  join
  verify_transaction;
end
  
```

Both tasks run in parallel

Won't continue past join until both tasks finish

- SystemVerilog 3.1 adds
 - **fork-join_none**
 - Statements that follow the **join_none** are *not* blocked from execution while the parallel threads are executing
 - **fork-join_any**
 - Statements that follow the **join_any** are blocked from execution until any of the parallel threads has completed execution

- **Verilog** uses the Programming Language Interface (PLI) to allow Verilog code to call C language code
 - Powerful capabilities such as traversing hierarchy, controlling simulation, modifying delays and synchronizing to simulation time
 - Difficult to learn
 - Too complex of an interface for many types of applications
- **SystemVerilog** adds the ability for:
 - **Verilog code to directly call C functions**
 - **C functions to directly call Verilog tasks and functions**
 - No PLI is needed for these direct function calls
 - Cannot do everything the PLI can do
 - Can do many things more easily than the PLI
 - Ideal for accessing C libraries and interfacing to C bus-functional models



Synopsys' Plans for SystemVerilog

- Synopsys is leading the industry in support for SystemVerilog
 - First company to announce intent for support
 - Acquired Co-design and SUPERLOG (the basis of SystemVerilog 3.0)
 - Donated VERA-lite, Direct-C and other key technologies for SystemVerilog 3.1
 - Has several engineers involved in SystemVerilog standards committees
- Synopsys tool status:
 - VCS version 7.1 (currently being tested) supports SystemVerilog 3.0
 - Design Compiler is expected to support SystemVerilog 3.0 in mid 2003
 - VCS will be extended to support for SystemVerilog 3.1 in Q4 2003
 - Assuming 3.1 is approved by Accellera on schedule
 - Other Synopsys tools will add support for appropriate portions of SystemVerilog

You can begin using SystemVerilog this year!

*SystemVerilog combines an enhanced **Hardware Description Language** and an advanced **Hardware Verification Language** into a unified*

Hardware Description AND Verification Language

- **SystemVerilog 3.0** (released June 2002) enhances Verilog for modeling hardware
 - Allows modeling much larger designs at more abstract level (less code)
 - Synopsys will be supporting these enhancements in VCS 7.1
 - Synthesis support in DC is expected mid 2003
- **SystemVerilog 3.1** (planned for June 2003) enhances Verilog for design verification
 - Allows developing efficient test benches for very large designs
 - Synopsys plans to support in VCS in fall 2003, if released on schedule

Ask Us About SystemVerilog!

SystemVerilog

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

Verilog-2001

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-1995

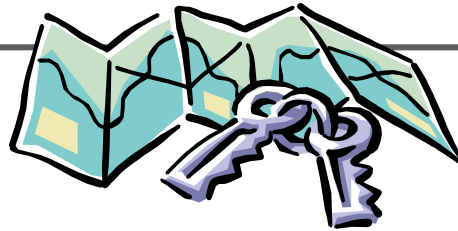
modules	\$finish \$fopen \$fclose	initial	wire reg	begin-end	+ = * /
parameters	\$display \$write	disable	integer real	while	%
function/tasks	\$monitor	events	time	for forever	>> <<
always @	`define `ifdef `else	wait # @	packed arrays	if-else	
assign	`include `timescale	fork-join	2D memory	repeat	



Supplemental Slides

What is Accellera?

- Accellera is a consortium of EDA companies and Engineering companies
 - It is the merger of Open Verilog International (OVI) and VHDL International (VI)
 - Primary goal is to encourage the development and use of EDA standards
- Renown industry experts are defining SystemVerilog
 - Simulation algorithm experts
 - Including Phil Moorby (the creator of Verilog) and Peter Flake (one of the creators of HILO and SUPERLOG)
 - Synthesis compiler experts
 - Verification experts
 - Verilog design engineers and consultants
 - IEEE 1364 Verilog standard working group members



- **SystemVerilog 3.0** (released June 2002)
 - Extends the **hardware modeling** capabilities of Verilog
 - Dozens of important enhancements
 - Allows modeling much larger designs more efficiently
 - Several EDA vendors are already implementing these features
- **SystemVerilog 3.1** (to be released June 2003)
 - Extends the **verification** capabilities of Verilog
 - Adds many important test bench constructs
 - Allows efficient verification of very large designs
 - Clarifications to 3.0 based on feedback from EDA vendors

- SystemVerilog interfaces are more than just 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 views
 - Each module connected to the interface can see specific signal names, signal directions, and methods
 - Interfaces can contain tasks and functions
 - 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

- An interface is defined separately from any module
 - New keywords: **interface**, **endinterface**

```

interface chip_bus; // Define the interface
    wire read_request, read_grant;
    wire [7:0] address, data;
endinterface: chip_bus

module CPU (chip_bus io, input clk);
    //io.read_request references a signal in the interface
endmodule

module RAM(chip_bus pins, input clk);
    ...
endmodule

module top;
    reg clk = 0;
    chip_bus a; //instantiate the interface

    RAM mem(a, clk); //connect interface to module instance
    CPU cpu(a, clk); //connect interface to module instance
endmodule

```