



March 11 - 12, 2002

Verilog: The Next Generation
Accellera's **SystemVerilog** Standard

by

Stuart Sutherland

Verilog HDL and PLI Expert

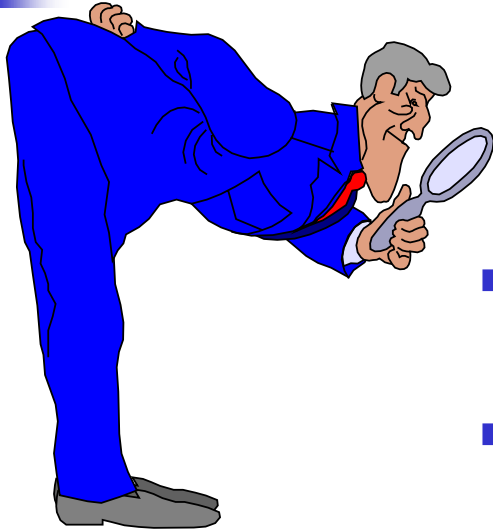
Sutherland HDL, Inc.



Training engineers
to be HDL wizards

HDL

Overview



- Define what is "SystemVerilog"
- Justify the need for SystemVerilog
- A whirlwind tour of the major features in SystemVerilog
- The status of the SystemVerilog standard



What is SystemVerilog

- **SystemVerilog** is a proposed set of extensions of the IEEE 1364 Verilog-2001 standard
 - Adds C language constructs to Verilog
 - Adds assertions to Verilog
 - Adds interfaces to Verilog
- Gives Verilog a new level of modeling abstraction
- Gives Verilog a new level of design verification
- Will “ship” in June 2002

Why Enhance Verilog?

- Hardware size and complexity is increasing
 - Engineers must design more gates than ever before
 - Larger designs require working at a more abstract level
 - Larger designs require new verification techniques
- Hardware is still built with silicon
 - Proven engineering methods are still used
 - Verilog works, and works well
 - Tools such as simulation, synthesis, timing analysis work well
- The right solution is to extend what works

SystemVerilog is an Evolution

- SystemVerilog evolves Verilog, rather than replacing it
 - Gives engineers the best of Verilog and C

C language abstractions

- Structures
- Globals
- ++ operator
- Enumerated types

Standard Verilog HDL

- Familiar to H/W engineers
- Concurrency
- Proven to work

```
typedef struct {string s; int left} node;
int visited = 0; //global data
function int treeFind(string str, parent);
    if (parent == null)
        return null;
    visited++;
    ...
endfunction

state {S0, S1, S2} cstate; //enumeration types

always @(posedge clk)
    begin
        case (cstate)
            ...
        end
```



Who is Accellera?

- Accellera is an HDL/HVL standards organization
 - Formed by the merger of OVI and VI in 1999
 - Made up of volunteers from:
 - EDA companies
 - Hardware design companies
 - Consultants
 - Promotes the use and evolution of Verilog and VHDL
 - Several subcommittees explore solutions to current and future needs in hardware design
 - The Accellera “HDL+” committee is exploring the future needs for the Verilog language
 - **SystemVerilog** is a result of that exploration

Why is Accellera Specifying Changes to an IEEE Standard?

- The IEEE 1364 standards group is the official governing body of the Verilog language
 - Defined the 1364-1995 "Verilog-1995" standard
 - Defined the 1364-2001 "Verilog-2001" standard
- Our design needs are changing much faster than the IEEE can react
 - It took the IEEE 4 years to define and ratify Verilog-2001
 - "Slower than molasses flowing uphill in the winter"
 - Accellera can react much faster than the IEEE
 - A de facto standard can be created in months instead of years
 - Accellera expects the IEEE to adopt the enhancements in SystemVerilog as part of the next IEEE Verilog standard



SystemVerilog's Roots

- Most enhancements in SystemVerilog are from two sources:
 - A subset of the SUPERLOG language
 - Co-design Automation donated the “synthesizable” portion of its SUPERLOG language to Accellera
 - The Accellera HDL+ committee modified the subset to be an extension to Verilog-2001
 - A superset of the OVL assertions library
 - Verplex and other companies donated assertions libraries to Accellera
 - The Accellera Assertions committee is defining assertions syntax and semantics for both Verilog and VHDL
 - The Accellera HDL+ committee is adding the Verilog assertions extensions to SystemVerilog

A Mile High View of SystemVerilog

SystemVerilog

User Defined Ports
State Machines
Interfaces
Packed Arrays
Structures
Unions

Block Labeling
Dynamic Processes
Assertions
2/4 State Variables
Timeunits
unique/priority fork/case/if

C
int
shortint
longint
shortreal
double
char void
globals

enum
typedef
struct union
casting
const

break continue
return goto
++ -- += -= *=
/= >>= <<=
&= |= ^= %=

ANSI C style ports
generate
localparam
const func

standard file I/O
\$value\$plusargs
`ifndef `elsif `line
@*

(* attributes *)
configurations
memory part selects
variable part select

multi dimensional arrays
signed unsigned
automatic
** (power operator)

modules
parameters
function/tasks
always @
assign

\$finish \$fopen \$fclose
\$display \$write
\$monitor
`define `ifdef `else
`include `timescale

initial
disable
events
wait # @
fork/join

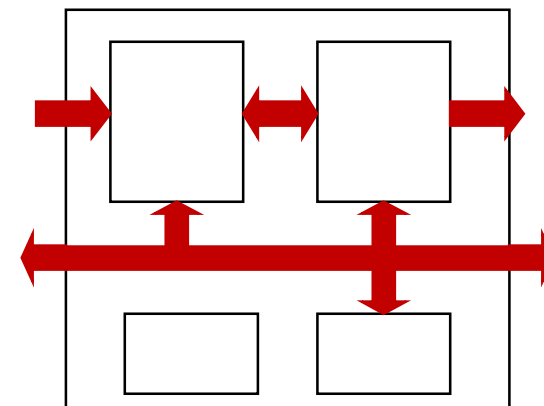
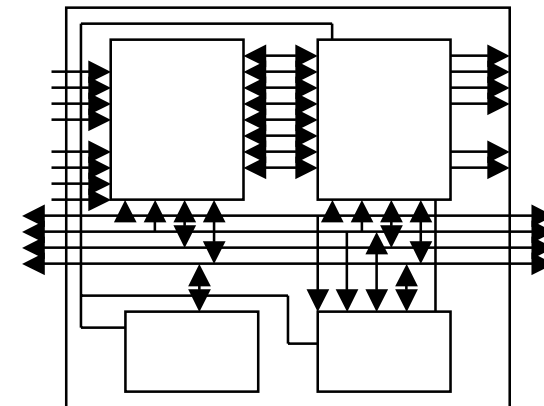
wire reg
integer real
time
packed arrays
2D memory

begin end + = * /
while %
for forever >> <<
if else
repeat

Verilog 2001
Verilog 1995

Interfaces

- Verilog-1995/2001 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 interfaces
 - Connections between models are bundled together
 - Connection definitions are independent from all modules





Interface Example

- 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 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

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

Interfaces Can Contain Logic

- 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 tasks and functions
 - Operations shared by all 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

Attend the tutorial this afternoon for an in-depth look at SystemVerilog interfaces!



Global Declarations and Global Statements

- Verilog-1995/2001:
 - Does not have a true global name space
 - Only module names and primitive names are global
 - Cannot declare common variables, functions or blocks
 - Verilog-1995/2001 allows multiple top-levels of hierarchy
- SystemVerilog adds a global space
 - A **\$root** space is the implied top-level of the hierarchy
 - Any declaration not in a module or interface is in \$root
 - Variables, functions, tasks, ...
 - All modules can reference objects declared in the global \$root space

Global Declaration and Functionality Example

- Declarations and statements outside of any module or interface are in the global **\$root** space

```
reg error_flag; //global variable

function compare (...); //global function

always @(error_flag) //global statement
    ...

module chip1 (...);
    FSM u2 (...);
    always @(data)
        error_flag = compare(data, expected);
endmodule

module FSM (...);
    ...
    always @(state)
        error_flag = compare(state, expected);
endmodule
```

global declarations
and statements

Time Unit and Precision

- In *Verilog-1995/2001*, time units are a module property
 - Declared with the ``timescale` compiler directive

```
forever #5 clock = ~clock;
```

5 what?

- *SystemVerilog* adds:
 - Time units can be specified as part of the time value

```
forever #5ns clock = ~clock;
```

- Module time units and precision can be specified with keywords

```
module my_chip (...);  
    timeunits 1ns;  
    timeprecision 10ps;  
    ...  
endmodule
```

Abstract Data Type

- Verilog-1995/2001 has hardware-centric **net** and **reg** types
 - Intended to represent real connections in a chip or system
 - 4-state logic, strength levels, wired logic resolution
 - Costly to simulation performance; Most hardware is 2-state
 - Models real hardware, but costly to simulation performance
- 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

The 2-state **bit** Data Type

- Verilog-1995/2001 uses 4-state logic
 - Costly to simulation performance
 - Most hardware is 2-state logic
- Some simulator's "fake" 2-state with compilers
 - Not portable to other tools (not a standard)
 - Can have side affects if part of the design (or verification) needs 3-state or 4-state
- The SystemVerilog 2-state **bit** type
 - Allows mixing 2-state and 4-state in the same design
 - Will work the same on all SystemVerilog simulators

The 4-state **logic** data type

- One of the biggest headaches with Verilog-1995/2001 is determining when to use **reg** and when to use **wire**
 - Context dependent
 - As model changes, the data type required can change
- SystemVerilog's 4-state **logic** type:
 - Can be used in place of a **reg**
 - Can be used in place of a wire, but limited to 1 driver
 - Allows evolving a design from algorithmic to behavioral to RTL to gate without changing data types
 - 1-driver limit allows more efficient simulation performance

Signed and Unsigned Modifiers

- Verilog-1995 had one signed type
 - The **integer** type is a 32-bit signed variable
- Verilog-2001 adds signed nets and regs
 - Any vector size can be explicitly declared as signed
- SystemVerilog adds new signed types
 - The **int**, **shortint**, **longint**, **byte** and **char** are signed
- SystemVerilog adds unsigned declarations
 - Any signed type can be explicitly declared as **unsigned**

```
reg signed [63:0] data_bus;
```

```
byte unsigned ubyte;
```

User-defined Types

- Verilog-1995/2001 does not have user-defined data types
- SystemVerilog adds user-defined types
 - Uses the **typedef** keyword, as in C

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

- User-defined types can be referenced before being defined
 - Requires an empty typedef declaration, as in C

```
typedef int48; //full typedef definition is elsewhere  
int48 c;
```

Enumerated Types

- Verilog-1995/2001 does not have enumerated types
 - All signals must be declared
 - All signals must be given a value to be useful
- SystemVerilog adds enumerated types, using **enum**, as in C

```
enum {red,green,blue} RGB;
```

- The value of the first enumerated type can be specified
 - Subsequent types are incremented from the initial value
 - The vector size of the type is the size of the initial value
 - The default initial value is an integer of 0

```
enum {WAIT=2'b01, LOAD, READY} states;
```

- Enumerated types can be given a name, as in C

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

Structures and Unions

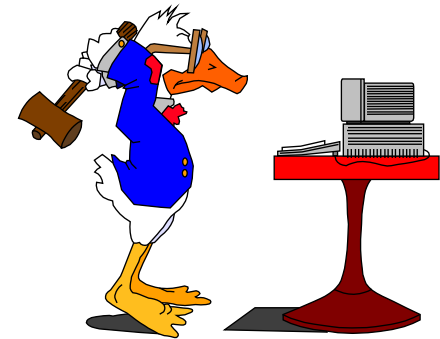
- Verilog-1995/2001 does not have structures or unions
- SystemVerilog adds structures and unions
 - Uses a similar syntax as C
 - Can be given a name, using **typedef**
 - Can be assigned a value as a whole

```
struct {  
    reg [15:0] opcode;  
    reg [23:0] addr;  
} IR;  
  
union {  
    int i;  
    shortreal f;  
} N;
```

```
typedef struct {  
    bit [7:0] opcode;  
    bit [23:0] addr;  
} instruction; //named structure type  
  
instruction IR; //allocate a structure  
  
IR = {5, 200}; //fill the structure
```

Module Port Connections

- **Verilog-1995/2001** 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
- **SystemVerilog** removes all 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



Redefinable Data Types

- Verilog-1995/2001 allows “parameterized” vector declarations

```
module register #(parameter size = 16)
    (output reg [size-1:0] q,
     input wire [size-1:0] d,
     input wire clock, reset);
```

- SystemVerilog allows data types to be “parameterized”
 - Data types to be changed using parameter definition

```
module foo #(parameter type VAR_TYPE = shortint;)
    (input logic [7:0] i, output logic [7:0] o);
    VAR_TYPE j = 0; /* j is of type shortint unless redefined */
    ...
endmodule

module bar;
    logic [3:0] i,o;
    foo #(.VAR_TYPE(int)) u1 (i, o); //redefines VAR_TYPE to an int
endmodule
```

Assigning Literal Values

■ Verilog-1995/2001

- Cannot fill all bits with 1 without specifying a vector size
- Strings are stored differently than in C
- Cannot assign to multiple array addresses in one statement

■ SystemVerilog enhances assignments of a literal value

- All bits of a vector can be filled with a literal 1-bit value
 - ``0`, ``1`, ``z`, ``x` fill all bits with 0, 1, z or x, respectively

```
data_bus = `1; //set all bits of data_bus to 1
```

- A string assigned to an array of char types will fill like C
- All or slices of arrays can be assigned values

Type Casting

- Verilog-1995/2001 does not allow a value to be cast from one data type to another
- SystemVerilog adds three casting operations
 - `<type>'(<value>)` — cast a value to any data type, including user-defined types

```
int'(2.0 * 3.0) //cast operation result to int
```

- `<size>'(<value>)` — cast a value to any vector size

```
17'(n - 2) //cast operation result to 17 bits wide
```

- `<sign>'(<value>)` — cast a value to signed or unsigned

```
signed'(y) //cast value to a signed value
```

New Operators

- Verilog-1995/2001 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++)  
...
```

Unique and Priority Decisions

- Verilog-1995/2001 defines that **if-else-if** decisions and **case** statements execute with priority encoding
 - In simulation, only the first matching branch is executed
 - Synthesis will infer parallel execution based on context
 - Can override the inference with “pragmas” such as “parallel_case”
 - Parallel evaluation after synthesis often causes a mismatch in pre-synthesis and post-synthesis simulation results
- SystemVerilog adds **unique** and **priority** keywords:
 - Priority-encoded or parallel evaluation to be explicitly defined for both simulation and synthesis
 - Software tools can warn if **case** or **if-else** decisions do not match the behavior specified

Bottom Testing Loops

- Verilog-1995/2001 has for, repeat and while loops
 - Each loop tests its control value at the beginning of each pass through the loop
 - Can require additional code outside the loop to execute statements once if the test value is false at the first pass of the loop
- SystemVerilog adds a **do-while** loop
 - The control is tested at the end of each pass of the loop
 - Prevents having to write redundant code outside of the loop

```
do
  begin
    ... //several lines of code
  end
while (count <= max_limit)
```



Jump Statements

- Verilog-1995/2001 has the odd-ball disable statement
 - Causes a named group of statement to jump to the end of the group
 - Can be used as a form of the C break and continue statements
 - Can be used to exit a task or function prematurely
- SystemVerilog adds the C language jump statements:
 - **break** — works like the C break
 - **continue** — works like the C continue
 - **return(<value>)** — return from a non-void function
 - **return** — return from a task or void function

Block Names and Statement Labels

- Verilog-1995/2001 allows a statement group to have a name
 - Identifies all statements within the block
 - Creates a new level of model hierarchy

```
begin: block1 ... end
```

- SystemVerilog adds:
 - A name can be specified after the end keyword
 - Documents which statement group is being ended
 - Specific statements can be given a "label"
 - Identifies a single statement
 - Does not create a new level of hierarchy

```
begin: block2 ... end: block2
```

```
shifter: for (i=15; i>0; i--)
```

Qualified Sensitivity Lists

- Verilog-1995/2001 uses “sensitivity lists” to control when statements in a procedure are evaluated
 - Statements within the procedure must be evaluated every time the sensitivity list triggers, even if no logic changes

```
always @(data or enable)  
    if (enable == 1) out <= enable;
```

- SystemVerilog adds an **iff** qualifier to sensitivity lists
 - Statements within the procedure are only executed if the sensitivity list triggers and the qualifier is true
 - Can improve simulation efficiency

```
always @(data or enable iff enable == 1)  
    out <= enable;
```

Expressions in Sensitivity Lists

- Verilog-1995/2001 does not define when a sensitivity list should trigger when there is an expression in the list
 - Trigger whenever an operand changes?
 - Only trigger when the result changes?

```
always @( ( a * b ) ) ...
```

```
always @( memory[address] ) ...
```

- SystemVerilog adds a **changed** keyword
 - Only trigger if the result of the expression changes

```
always @(changed ( a * b ) ) ...
```

```
always @(changed memory[address] ) ...
```

Specialized Procedural Blocks

- Verilog-1995/2001 infers combinational, sequential and latched logic from an always block based on context
 - The designer cannot specify the intended behavior
 - Different products may infer different behavior
- SystemVerilog adds three new procedures:
 - always_comb** **always_ff** **always_latch**
 - Specifies the intended behavior of the procedure
 - Tools can issue warnings if the inferred behavior is different than the intended behavior

Dynamic Processes

- Verilog-1995/2001 only supports static statement groups
 - All statements in the flow must have executed before reaching the end or join

```
fork
  send_packet_task(1,255,0);
  send_packet_task(7,128,5);
join
```

Both tasks run in parallel

Won't "join" until both tasks finish

- SystemVerilog adds dynamic process statements
 - Each statement is a separate thread
 - Statements do not need to complete before the end or join is reached

```
begin
  process send_packet_task(1,255,0);
  process send_packet_task(7,128,5);
end
```

Both tasks run in parallel

Will "end" without waiting for tasks to finish

Static and Automatic Task/Function Enhancements

- **Verilog-1995** only supports static tasks and functions
 - All local storage is shared by all calls
 - Re-entrant tasks and recursive functions do not work
- **Verilog-2001** adds automatic tasks and functions
 - All local storage is stacked
 - Supports re-entrant tasks and recursive functions
- **SystemVerilog** adds:
 - Static storage in an automatic task or function
 - Automatic storage in a static task or function
 - Uses **static** and **automatic** declarations, as in C



Multiple Statements in Tasks and Functions

- Verilog-1995/2001 only allows a single statement or a single statement group within a task or function
 - Groups are formed with **begin–end** or **fork–join**
- SystemVerilog allows any number of statements within a task or function
 - **begin–end** or **fork–join** no longer required
 - Ungrouped statements execute in sequence, as if enclosed in a **begin–end** block

Continuous Assignment Enhancements

- **Verilog-1995/2001** only allows net data types to be used on the left-hand side of continuous assignments statements
- **SystemVerilog** allows most data types to be used
 - Variable types can only be driven by a single continuous assignments
 - The **reg** variable type cannot be used



State Machine Modeling

- **Verilog-1995/2001** does not have any special constructs to model state machines
 - State machines are inferred from the RTL coding style
- **SystemVerilog** adds special constructs for modeling state machines at a more abstract level
 - Special state variable type
 - Enumerated types
 - Special transition statement and transition operator
 - Defines changes from one state to another
 - Eliminates need for intermediate "next_state" variables



Assertions

- Verilog-1995/2001 does not provide an assertion construct
 - Model checking must be done by hard-coded logic
 - Assertion libraries, such as the OVL library, add assertion checks in the form of module instances
 - Some tools add assertion checks using the Verilog PLI
- SystemVerilog adds assertion syntax and semantics
 - Assertion information is built into the language
 - No special modules or PLI calls are needed



Status of SystemVerilog

- SystemVerilog will “ship” in June 2002
- The Accellera HDL+ committee has completed 90+% of the SystemVerilog “Language Reference Manual”
- Still pending (as of March 2002):
 - Fine tune new state machine modeling constructs
 - Fine time assertions specifications
 - Discuss deprecating certain Verilog constructs (e.g. defparam)
 - Review for accuracy

Plans for SystemVerilog

- **June 2002** — The first release of the **SystemVerilog** standard
 - Co-design already supports SystemVerilog for simulation
 - Get-2-Chip already supports much of SystemVerilog for synthesis
- Accellera will turn over **SystemVerilog** to the IEEE 1364 Verilog standards group
 - Accellera hopes the IEEE will use SystemVerilog as a basis for the next version of the Verilog
- Accellera will continue to review design and verification needs, and define new versions of **SystemVerilog** as required.



Conclusion

- **SystemVerilog** adds a large number of major enhancements to the Verilog-2001 standard
- Accellera's HDL+ technical committee is developing the standard
 - No red tape, as in the IEEE
 - A working standard can be done in less than 1 year instead of 4 or 5 years
- **SystemVerilog** should become the next generation of the IEEE 1364 Verilog standard

Any Questions?



A copy of this presentation will be available at
www.sutherland-hdl.com