Adopting SystemVerilog

Stuart Sutherland
Sutherland HDL, Inc.

Training engineers to be HDL wizards

www.sutherland-hdl.com

Alternate Title...

SUTHERLAND H D L
Training engineers to be HDL wizards

Are Your EDA Tools Giving You Headaches?

Objectives

SUTHERLAND HDL
Training engineers to be HDL wizards

- SystemVerilog has 584 pages of enhancements to Verilog!
  - The cover-to-cover page count of the current 3.1a LRM
- Should EDA companies implement every construct in SystemVerilog all at once?
  - Not feasible
  - Need to decide what enhancements to implement first
- How do you prioritize what enhancements should be implemented first?
  - This presentation will give some ideas...

A copy of this presentation is available at www.sutherland-hdl.com/papers/2004-DAC_SystemVerilog_adoption_plan.pdf

Caveat Number One

SUTHERLAND HDL
Training engineers to be HDL wizards

- The suggestions presented are the opinion of Stu Sutherland
  - Opinions not based on survey data, either scientific or informal
  - Opinions are based on ~~too many~~ lots of years of experience

  - Been involved with Verilog since 1988
  - Worked with or for several EDA companies
  - Independent Verilog consultant for 12 years
  - Involved with IEEE Verilog standard since inception
  - Involved with SystemVerilog standard since inception
  - Seen Verilog used (and abused) at 100+ companies

Is Interoperability Important?

SUTHERLAND HDL
Training engineers to be HDL wizards

- The IEEE Verilog-2001 standard has been out for 3+ years
  - The standard was done by early 2000
  - EDA companies knew what would be in the standard in 1998
  - Still cannot use many of the new features in Verilog-2001

Why can't I use Verilog-2001 after 3+ years?

- Vendor A implemented @*, and vendor B has not, **but...**
- Vendor B implemented constant functions, and Vendor A has not
  - Therefore, I can not use either @* or constant functions!
- Similar story for most of Verilog-2001

Interoperability Is Important!

SUTHERLAND HDL
Training engineers to be HDL wizards

A plea to EDA Companies...
Please don't make the same mistake with SystemVerilog!

- EDA companies should agree on what features to implement first
  - Customers won't buy new tools if every Vendor supports different SystemVerilog features

Categorizing SystemVerilog

SUTHERLAND HDL
Training engineers to be HDL wizards

- Productivity enhancements
  - Make it easier to model with Verilog
- RTL enhancements
  - Make RTL models safer
- Data encapsulation enhancements
  - Make it easier to manage lots of design data
- Assertions
  - Embed designer's intent for verification purposes
- Abstract modeling enhancements
  - Model more functionality with fewer lines of code
- Non object-oriented testbench enhancements
  - Make it easier to do directed testing
- Object-oriented testbench enhancements
  - Re-usable, coverage-driven, constrained random testing

What Comes Next...

- ❑ **Productivity enhancements**
- ❑ RTL modeling enhancements
- ❑ Data encapsulation enhancements
- ❑ Assertions
- ❑ Abstract modeling enhancements
- ❑ Non object-oriented testbench enhancements
- ❑ Object-oriented testbench enhancements
- ❑ Summary

What Are "Productivity Enhancements"

SUTHERLAND HDL
Training engineers to be HDL wizards

- Productivity enhancements:
  - Make it easier to model in Verilog
  - Eliminate common modeling errors
  - Do *not* add significant new functionality
- Caveat Number Two
  - The list in this presentation is not complete
    - Not enough time to cover everything
    - Intent is to give an idea of what types of enhancements fall into the "productivity" category
      - The same caveat is true for subsequent categories

Enhanced Specification of Time Units and Time Precision

SUTHERLAND HDL
Training engineers to be HDL wizards

- In Verilog, 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 (…);
  timeunit 1ns;
  timeprecision 10ps;
  ...
```

Enhanced Literal Value Assignments

SUTHERLAND HDL
Training engineers to be HDL wizards

- In Verilog, there is no simple way to fill a vector with all 1's

```
parameter N = 64;
reg [N-1:0] data_bus;
data_bus = 64'hFFFFFFFFFFFFFFFF; //set all bits of data_bus to 1
```

could also use operations, such as replicate or invert

vector width must be hard coded

- SystemVerilog enhances assignments of a literal value
  - All bits of a vector can be filled with a literal 1-bit value
    - **`x'0`** fills all bits on the left-hand side with 0
    - **`x'1`** fills all bits on the left-hand side with 1
    - **`x'z`** fills all bits on the left-hand side with z
    - **`x'x`** fills all bits on the left-hand side with x

```
reg [N-1:0] data_bus;
data_bus = '1; //set all bits of data_bus to 1
```


The SystemVerilog logic Data Type

SUTHERLAND HDL
Training engineers to be HDL wizards

- In Verilog, the term "**`reg`**" confuses new users
  - The name would seem to infer a hardware *register*
  - In reality, **`reg`** is a general purpose variable that can represent either combinational logic or sequential logic
- SystemVerilog's 4-state **`logic`** type is identical to a **`reg`**
  - **`logic`** is a more intuitive name for new Verilog users
    - Historically, **`logic`** comes from the Superlog language, where it had different semantic rules than reg
    - SystemVerilog made the rules equivalent for backward compatibility with existing Verilog models
    - Verilog also has equivalent keywords, (e.g. **`wire`** and **`tri`**) that historically once had different semantics

SystemVerilog Relaxes Verilog Variable Semantic Rules

SUTHERLAND HDL
Training engineers to be HDL wizards

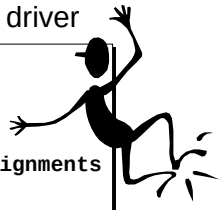
- Verilog has strict rules on when to use a variable (eg. **`reg`**) and when to use a net (e.g. **`wire`**)
  - Context dependent
  - A variable cannot be "driven" by a continuous assignment or an output port

- SystemVerilog allows variables to be used in the same places a net can be used:
  - Limited to a single driver type (procedural, continuous or output of a module/primitive instance)
    - 1-driver limit prevents inadvertent shared variable behavior where wired-logic resolution is needed

An Example of Using Variables With SystemVerilog's Relaxed Rules

SUTHERLAND HDL
Training engineers to be HDL wizards

- The same data type can be used for the entire model
  - With the restriction that signals only have a single driver



```
module compare (output logic lt, eq, gt,
                input  logic [63:0] a, b);

  always @(a or b)
    if (a < b) lt = 1'b1;    //procedural assignments
    else       lt = 1'b0;

  assign gt = (a > b);       //continuous assignments

  comparator u1 (eq, a, b);  //module instance

endmodule
```

Most designs mostly only have a single driver for each signal — the "single driver" restriction on variables can can catch design errors at compile time!

Module Port Connections

SUTHERLAND HDL
Training engineers to be HDL wizards

- 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 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

Module Instance Port Connection Shortcuts

SUTHERLAND HDL
Training engineers to be HDL wizards

- 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]), .qb());
```

For Loop Variables

SUTHERLAND HDL
Training engineers to be HDL wizards

- In Verilog, the variable used to control a **`for`** loop must be declared prior to the loop

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

i must be declared outside the loop

- SystemVerilog allows the declaration of the **`for`** loop variable within the for loop itself

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

i can be declared within the loop

  - Makes the loop variable local to the loop
  - Prevents the possibility of a conflict between multiple **`for`** loops

Enhanced Task/Function Formal Arguments

SUTHERLAND HDL
Training engineers to be HDL wizards

- In Verilog:
  - The directions of all arguments must be declared

```
function add (input a, b);
  ...
endfunction
```

- In SystemVerilog:
  - There is a default direction of input

```
function add (a, b);
  ...
endfunction
```

Task/Function Arguments: Passing By Name

SUTHERLAND HDL
Training engineers to be HDL wizards

- 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 integer subtractor(integer a, b);
  return(a - b);
endfunction
```

Uses same syntax as named module port connections

.name and .* connections can also be used

Function Return Values

SUTHERLAND HDL
Training engineers to be HDL wizards

- In Verilog:
  - Functions must have a return value
  - The return is assigned to the name of the function

```
function [31:0] mult (input [31:0] a, b);
  mult = a * b;
endfunction
```

- In SystemVerilog:
  - Functions can return a value using a **`return`** statement, like C

```
function int mult (int a, b);
  return(a * b);
endfunction
```

More consistent with C style functions

Multiple Statements In Tasks and Functions

SUTHERLAND HDL
Training engineers to be HDL wizards

- In Verilog:
  - Multiple statements within a task or function must be grouped using **begin**...**end**
- In SystemVerilog:
  - Statement grouping is not required
    - Multiple statements execute as if in a **begin**...**end** block

```
function int check_errors (input packet sent, received);
  int error_count;
  $display("data sent was %0h", sent.data);
  if (sent !== received)
    ...
  return(error_count);
endfunction
```

a begin...end sequence is implied

There are several more enhancements involving tasks and functions that are not covered in this presentation!

Block Names and Statement Labels

SUTHERLAND HDL
Training engineers to be HDL wizards

- Verilog 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

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

  - Specific statements can be given a “label”
    - Identifies a single statement

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

Named End Statements

SUTHERLAND HDL
Training engineers to be HDL wizards

- SystemVerilog also extends to ability to specify an ending name with **`endmodule`**, **`endinterface`**, **`endprimitive`**, **`endprogram`**, **`endtask`**, **`endfunction`**, **`endclass`**, **`endproperty`** and **`endsequence`**
  - The ending name must match the name used with the corresponding beginning of the code block

```
module my_chip (...);
  ...
  task get_data (...)
    ...
  endtask: get_data
endmodule: my_chip
```

Specifying ending names helps to make large blocks of code more readable, but does not affect functionality in any way

What Comes Next...

- ✓ Productivity enhancements
- ❑ **RTL modeling enhancements**
- ❑ Data encapsulation enhancements
- ❑ Assertions
- ❑ Abstract modeling enhancements
- ❑ Non object-oriented testbench enhancements
- ❑ Object-oriented testbench enhancements
- ❑ Summary

RTL Enhancement Characteristics

SUTHERLAND HDL
Training engineers to be HDL wizards

- RTL enhancements
  - Make it easier to model for synthesis
  - Remove model ambiguity
  - Reduce pre- and post-simulation mismatches
  - Allow all tools to generate warnings if model does not match designer's intent

Just a few of the enhancements in this category will be shown

Hardware Specific Procedures

SUTHERLAND HDL
Training engineers to be HDL wizards

- The Verilog **`always`** procedure is a general purpose procedure
  - Used to model combinational, latched, and sequential logic
  - Software tools must "infer" (***guess***) what type of hardware an engineer intended based on procedure content and context
- SystemVerilog adds special hardware-oriented procedures: **`always_ff`**, **`always_comb`**, and **`always_latch`**
  - Simulation, synthesis and formal tools to use same rules
  - Tools can check that designer's intent has been modeled

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

no sensitivity list

contents must follow synthesis requirements for combinational logic

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

Unique and Priority Decisions

SUTHERLAND HDL
Training engineers to be HDL wizards

- Verilog 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
    - Parallel evaluation after synthesis *may* causes a mismatch in pre-synthesis and post-synthesis simulation results
- SystemVerilog adds **`unique`** and **`priority`** keywords:
  - Priority-encoded or parallel evaluation can be explicitly defined for both simulation and synthesis
  - Software tools can warn if **`case`** or **`if–else`** decisions do not match the behavior specified

New Operators

SUTHERLAND HDL
Training engineers to be HDL wizards

- 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++)
  ...
```

Enumerated Types

SUTHERLAND HDL
Training engineers to be HDL wizards

- With Verilog, constants must be used to give names to values

```
reg [2:0] traffic_light;

parameter red    = 0;
parameter green  = 1;
parameter yellow = 2;

always @(posedge clock)
  if (traffic_light == red)
    ...
```

The variable traffic_light could be assigned values other than red, green or yellow

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

```
enum {red, green, yellow} traffic_light;

always @(posedge clock)
  if (traffic_light == red)
    ...
```

Simplifies declaration of named values

Limits the legal values of a variable

Special Methods for Enumerated Types

SUTHERLAND HDL
Training engineers to be HDL wizards

- The implementation of enumerated types is not complete without their "*methods*"

<enum_variable>**.first** — returns the value of the first member in the enumerated list
<enum_variable>**.last** — returns the value of the last member in the enumerated list
<enum_variable>**.next**(<N>) — returns value of the Nth next value in the enumerated list
<enum_variable>**.prev**(<N>) — returns value of the Nth previous value in the enumerated list
<enum_variable>**.num** — returns the number of elements in the enumerated list
<enum_variable>**.name** — returns the name of the current value in the enumeration variable

```
enum {CNT0, CNT1, CNT2, CNT3, CNT4, CNT5, CNT6, CNT7} State, NextState;
...
always @(state) begin: confidence_check
  if (synched) next_state = State.next;  // increment by 1 state
  else (next_state = State.prev(2);      // decrement by 2 states
  $display("Next state will be %s (%b)", NextState.name, next_state);
end: confidence_check
```

The 2-state bit Data Type

SUTHERLAND HDL
Training engineers to be HDL wizards

- Verilog uses 4-state logic
  - Most hardware can be modeled with 2-state logic
  - Synthesis mostly only considers 2-state values

```
reg mode;
always @(a, b, mode)
  if (mode)
    y = a + b;
  else
    y = a - b;
```

Synthesis does not take into consideration that mode could have an X or Z value

- The SystemVerilog 2-state **`bit`** type
  - Models RTL logic the way synthesis sees the logic
  - Allows easy mixing of 2-state and 4-state in the same design
  - Uses the same rules with all SystemVerilog tools

Void Functions

SUTHERLAND HDL
Training engineers to be HDL wizards

- In Verilog:
  - Functions must have a return value
  - When a function is called, it is illegal to ignore the return

```
function [31:0] multiplier (input [31:0] a, b);
  ... // logic for secret multiplier algorithm
  mult = a * b;
endfunction

product = multiplier(u, v);
```

- In SystemVerilog:
  - Functions can be declared as type **`void`** (no return value)
  - Called like a task, but with the restrictions of functions

```
function void multiplier (int a, b, output longint y);
  ...
endfunction

multiplier(u, v, product);
```

- **Functions have restrictions (no delays, etc.)**
- **Helps ensure subroutines are synthesizable !**

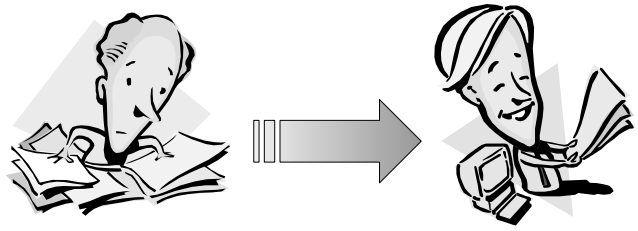
What Comes Next...

- ✓ Productivity enhancements
- ✓ RTL modeling enhancements
- ❑ **Data encapsulation enhancements**
- ❑ Assertions
- ❑ Abstract modeling enhancements
- ❑ Non object-oriented testbench enhancements
- ❑ Object-oriented testbench enhancements
- ❑ Summary

Data Encapsulation Enhancement Characteristics

SUTHERLAND HDL
Training engineers to be HDL wizards

- Data encapsulation enhancements
  - Bundle several discrete signals or data together
  - Transfer whole groups of signals or data at once
  - Allow operations on a bundle of signals or data

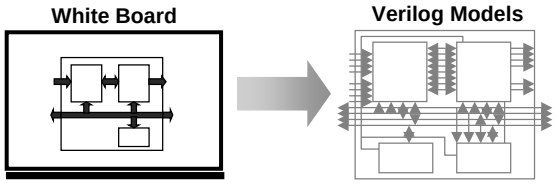


Just a few of the enhancements in this category will be shown

Interfaces

SUTHERLAND HDL
Training engineers to be HDL wizards

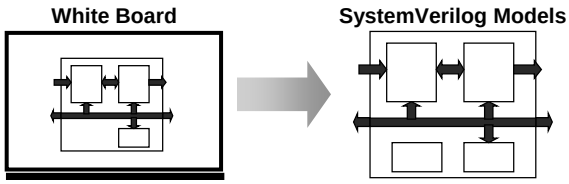
- Verilog connects models using detailed module ports



Connections must be modeled at implementation level

Interconnection details must be duplicated in every module

- SystemVerilog adds interfaces



Connections between modules are bundled together

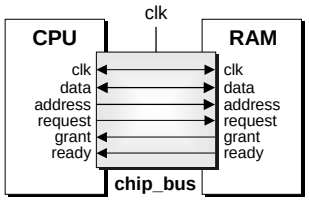
Modules use simple ports, an interface bundle

Interfaces Simplify Module Interconnections

SUTHERLAND HDL
Training engineers to be HDL wizards

```
interface chip_bus (input bit clk);
  bit        request, grant, ready;
  bit [47:0] address;
  bit [63:0] data;
endinterface
```

Connection details are in in the interface




```
module CPU (chip_bus io);
  ...
endmodule
```

Modules do not duplicate connection detail

```
module RAM(chip_bus pins);
  ...
endmodule
```

```
module top;
  bit clk = 0;
  chip_bus a(clk); //instantiate the interface
  RAM mem(a); //connect interface to module instance
  CPU cpu(a); //connect interface to module instance
endmodule
```

Netlists do not duplicate connection detail

Unpacked Arrays (Verilog Style)

SUTHERLAND HDL
Training engineers to be HDL wizards

- An "unpacked array" is an array of signals
  - The signal can be any data type
  - The array can have any number of dimensions

```
wire n [0:1023];
```
a 1-dimensional "unpacked array" of 1024 1-bit nets

```
real r [0:1023];
```
a 1-dimensional "unpacked array" of 1024 real variables

```
int a [0:7][0:7][0:7];
```
a 3-dimensional "unpacked array" of 32-bit int variables

Unpacked array dimensions come ***after*** the array name (as in Verilog)

- In Verilog:
  - Only one element within an array can be accessed at a time
- SystemVerilog adds:
  - The entire array can be referenced (e.g. `my_array = your_array`)
  - A "slice" of multiple elements can be referenced

Initializing and Assigning to Unpacked Arrays

SUTHERLAND HDL
Training engineers to be HDL wizards

- Unpacked arrays can be assigned using a list of values in **{ }** braces for each array dimension (similar to C)

```
int d [0:1][0:3] = { {7,3,0,5},{2,0,1,6} };
```

The { } braces are the C array initialize tokens, ***not*** the Verilog concatenate operator!

Must have a set of braces for each array dimension

```
d[0][0] is initialized to 7
d[0][1] is initialized to 3
d[0][2] is initialized to 0
d[0][3] is initialized to 5
d[1][0] is initialized to 2
d[1][1] is initialized to 0
d[1][2] is initialized to 1
d[1][3] is initialized to 6
```

- A replicate operator can be used to fill unpacked arrays

```
int d [0:1][0:3] = { 2{7,3,0,5} };
```

```
d[0][0] is initialized to 7
d[0][1] is initialized to 3
d[0][2] is initialized to 0
d[0][3] is initialized to 5
d[1][0] is initialized to 7
d[1][1] is initialized to 3
d[1][2] is initialized to 0
d[1][3] is initialized to 5
```

- A default assignment can be used

```
int d [0:1][0:3] = { default:'1 };
```

Packed Arrays

SUTHERLAND HDL
Training engineers to be HDL wizards

- In Verilog a *"range"* before the name declares a ***"vector"*** size

```
wire [0:63] n;
```

a 64-bit "vector" made up of 64 1-bit nets

A dimension ***before*** the signal name declares a vector of 1-bit signals

SystemVerilog calls the dimension ***before*** the name a "packed array"

- SystemVerilog adds "multidimensional packed arrays"

```
reg [3:0][7:0] a;
```

a 2-dimensional "packed array" of logic variables

	a[3]	a[2]	a[1]	a[0]
a	[7:0]	[7:0]	[7:0]	[7:0]

  - A "packed array" is an array of sub fields within a vector
    - Can only use be 1-bit data types: **`reg`**, **`logic`**, **`bit`**, **`wire`**, ...
    - A Verilog "vector" is a 1-dimensional packed array

Structures

SUTHERLAND HDL
Training engineers to be HDL wizards

- SystemVerilog adds C-like structures to Verilog

An array is a collection of elements that are all the same type and size;
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 {
  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

Unions

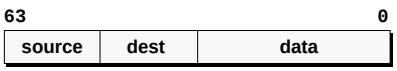
SUTHERLAND HDL
Training engineers to be HDL wizards

- A union is a single element that allows the storage of different data types in the same space
  - Allows same storage space to be referenced different ways

```
typedef struct packed{
  bit [15:0] source_address;
  bit [15:0] destination_address;
  bit [31:0] data;
} data_packet;

union {
  data_packet    data;
  bit [7:0][7:0] bytes;
} data_reg;

data_reg.data = data_in;   //assumes data_in is of type data_packet
dest_low_byte = data_reg.bytes[4];
```



two names for the same storage

User-defined Types

SUTHERLAND HDL
Training engineers to be HDL wizards

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

```
typedef enum {FALSE, TRUE} boolean;
boolean ready;  //variable "ready" can be FALSE or TRUE
```

```
typedef enum {WAIT, LOAD, READY} states_t;
states_t state, next_state;
```

The Compilation Unit Name Space ($unit)

SUTHERLAND HDL
Training engineers to be HDL wizards

- SystemVerilog provides a compilation unit name space
  - Allows declarations of variables, functions, etc. outside of modules
  - External declarations are visible to all code compiled at same time
  - External names can be directly referenced as **`$unit.<name>`**

```
bit reset;  //external reset
typedef struct {
  int  i0, i1;
  bit [15:0] opcode;
} instruction_t;
module top;  //instance of top-level module
  instruction_t out, in;
   ...
  register i1 (out, in, clock, reset);
endmodule
module register (output instruction_t q, ...)
   ...
  always @(posedge clk or posedge $unit.reset)
```

External variable

External user-defined type

When a name is referenced:
• First, follow Verilog search rules
• Second, look in $unit

Packages

SUTHERLAND HDL
Training engineers to be HDL wizards

- SystemVerilog adds packages to the Verilog HDL
  - Used to contain declarations that are used in many models
    - Modules can directly reference declarations in a package
    - Modules can import specific declarations from a package
    - Modules can import only what is needed from a package

```
package definitions;
  typedef struct {
    int  i0, i1;
    bit [15:0] opcode;
  } instruction_t;
endpackage
module register (output definitions::instruction_t q, ...)
   ...
module top;  //instance of top-level module
  import definitions::*;  // import whatever is needed from package
  instruction_t out, in;
   ...
```

Packages can contain net and variable declarations, task and function definitions, type definitions, class definitions, coverage definitions, operator overload definitions, ...

What Comes Next...

- ✓ Productivity enhancements
- ✓ RTL modeling enhancements
- ✓ Data encapsulation enhancements
- ❑ **Assertions**
- ❑ Abstract modeling enhancements
- ❑ Non object-oriented testbench enhancements
- ❑ Object-oriented testbench enhancements
- ❑ Summary

SystemVerilog Assertions

SUTHERLAND HDL
Training engineers to be HDL wizards

- SystemVerilog adds assertion syntax and semantics
  - Immediate assertions test for a condition at the current time

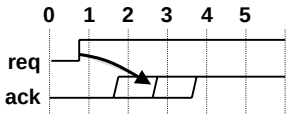
```
always @(state)
  assert (reset && (state != RST)) else $fatal;
```

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

```
sequence req_ack;
 @(posedge clk) req ##[1:3] $rose(ack);
endsequence

assert property (req_ack);
```



an event sequence is described using a declarative statement

SystemVerilog Assertion Sequences

SUTHERLAND HDL
Training engineers to be HDL wizards

- SystemVerilog can specify very complex event sequences using a simple and concise syntax
  - Unifies PSL and Verilog syntax to express sequences
  - Adds Verilog-like simulation timing and assertion control
  - Can specify:
    - Advancing one or more clock cycles, using ##
    - Boolean expressions, using special operators
      - **`and`**, **`intersect`**, **`or`**, **`first_match`**, **`throughout`**, **`within`**, **`$rose`**, **`$fell`**, **`$stable`**
    - Repetition of sequences
    - Implication of sequences

What Comes Next...

- ✓ Productivity enhancements
- ✓ RTL modeling enhancements
- ✓ Data encapsulation enhancements
- ✓ Assertions
- ❑ **Abstract modeling enhancements**
- ❑ Non object-oriented testbench enhancements
- ❑ Object-oriented testbench enhancements
- ❑ Summary

Abstract Modeling Enhancement Characteristics

SUTHERLAND HDL
Training engineers to be HDL wizards

- Abstract modeling enhancements
  - Model more functionality with fewer lines of code
  - More like programming than register transfer level code
  - Might not be synthesizable (today)
  - Important for Behavioral Modeling and Bus Functional Modeling

"Low-level" RTL modeling is too primitive for bus-functional models

Just a few of the enhancements in this category will be shown

Abstract Data Types

SUTHERLAND HDL
Training engineers to be HDL wizards

- Verilog has hardware-centric data types
  - Intended to represent real connections in a chip or system
  - At the system and RTL level, models only need 2-state logic
    - Tri-state busses are the only place 4-state logic is needed
- SystemVerilog adds several new data types
  - C-like data types create a bridge between C and Verilog
    - **`byte`** — an 8-bit 2-state integer
    - **`shortint`** — a 16-bit 2-state integer, similar to a C **`short`**
    - **`int`** — a 32-bit 2-state integer, similar to a C **`int`**
    - **`longint`** — a 64-bit 2-state integer, similar to a C **`longlong`**
    - **`shortreal`** — a 32-bit single-precision floating point, the same as a C **`float`**
    - **`void`** — no value (used for function returns)

Jump Statements

SUTHERLAND HDL
Training engineers to be HDL wizards

- Verilog uses the **`disable`** statement as a go-to
  - Causes a named group of statements to jump to the end
- 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

```
task send_packet(...);
  if (^data == 1'bx) begin
    $display("Error...");
    return; //abort task
  end
  ...
endtask
```

```
function real absolute(input real a);
  if (a >= 0.0) return(a);
  else          return(-a);
endfunction
```

The `return` statement allows terminating a task or function before reaching the end

New Bottom Testing Loop

SUTHERLAND HDL
Training engineers to be HDL wizards

- Verilog has **`for`**, **`repeat`** and **`while`** loops
  - The control value is tested at the beginning of each pass through the loop
    - Can require extra 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
    - Can eliminate the need, in some cases, to have the loop code both inside the loop and outside the loop (to execute once)

must execute loop at least once to set done or OutOfBound flag

```
do
  begin
    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)
```

Task/Function Arguments: Passing By Reference

SUTHERLAND HDL
Training engineers to be HDL wizards

- In Verilog:
  - Inputs are copied into tasks and functions
  - Outputs are copied out of tasks

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

- In SystemVerilog:
  - Task/function arguments can "reference" the calling arguments
    - Uses the keyword **`ref`** instead of input, output or inout
    - Using **`const ref`** makes the reference read-only

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

The function receives a pointer to "stack" in the calling scope (note: the C "&" is not used)

Type Casting

SUTHERLAND HDL
Training engineers to be HDL wizards

- SystemVerilog adds casting operations to the Verilog language
  - **`<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
```

Redefinable Data Types

SUTHERLAND HDL
Training engineers to be HDL wizards

- 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 also allows data types to be "parameterized"
  - Data types can be changed using parameter definition

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

```
module bar;
  ...
  foo #(.VAR_TYPE(int)) u1 (...); //redefines VAR_TYPE to be an int
endmodule
```


Direct Programming Language Interface

SUTHERLAND HDL
Training engineers to be HDL wizards

- 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 a "Direct Programming Interface" (DPI)
  - Verilog code can directly call C functions
  - C functions can directly call Verilog 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 interfacing to bus-functional models written in C

What Comes Next...

- ✓ Productivity enhancements
- ✓ RTL modeling enhancements
- ✓ Data encapsulation enhancements
- ✓ Assertions
- ✓ Abstract modeling enhancements
- ❑ **Non object-oriented testbench enhancements**
- ❑ Object-oriented testbench enhancements
- ❑ Summary

Special Test Bench "Program Blocks"

SUTHERLAND HDL
Training engineers to be HDL wizards

- Verilog uses modules to model the test bench
  - Modules are intended to model hardware
  - No special semantics to avoid race conditions with the design
- SystemVerilog adds a special "program block" for testing
  - Events are executed in a "reactive phase"
    - 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;
    ...
  endtask
endprogram
```

- **No race conditions between program block and design blocks**
- **In a module, this example could have race conditions with the design, if the design used the same posedge of clock.**

Enhancements to Tasks & Functions

SUTHERLAND HDL
Training engineers to be HDL wizards

- SystemVerilog adds:
  - Support for arrays as task/function arguments
  - Passing argument by reference instead of value
    - Operate on arrays without copying array
    - Tasks can sense changes on arguments
  - Functions can have the same formals as tasks
    - Verilog only allows functions to have inputs

```
task mytask4(input [3:0][7:0] a, b[3:0],
             output [3:0][7:0] y[1:0]);
  ...
endtask
```

```
function int crc(ref byte packet[1000:1]);
   for( int j= 1; j <= 1000; j++ ) begin
      crc ^= packet[j];
   end
endfunction
```

```
input  // copy value in at beginning
output // copy value out at end
inout  // copy in at beginning, out at end
ref    // pass reference
```


Enhanced Fork-Join Processes

SUTHERLAND HDL
Training engineers to be HDL wizards

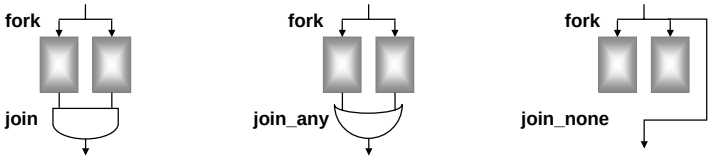
- Verilog supports parallel processes using **`fork`–`join`**
  - All parallel processes must finish execution before continuing

```
always @(posedge request)
  begin
    fork
      send_packet_task(1,255,0);
      send_packet_task(7,128,5);
    join
    start_results_checker;
  end
```

Both tasks run in parallel

When should the results checker start?

- SystemVerilog adds dynamic parallel processes using **`fork`–`join_any`** and **`fork`–`join_none`**



Final Blocks

SUTHERLAND HDL
Training engineers to be HDL wizards

- Verilog has two types of procedural blocks
  - **`initial`** blocks
    - Execute statements once, beginning at time 0
  - **`always`** blocks
    - Execute statements repeatedly, beginning at time 0

```
initial
 begin
  a = 0;
  #10 a = 1;
  ...
 end
```

```
always @(a or b)
 begin
  sum  = a + b;
  diff = a - b;
 end
```

- SystemVerilog adds **`final`** blocks
  - Executes statements once, *beginning when simulation finishes*
  - Restricted to statements that execute in zero time

```
initial
  ...
  #10000 $finish;

final
  $display("total errors detected = %d", err_count);
```

To duplicate final block behavior in Verilog requires using the Verilog PLI

What Comes Next...

- ✓ Productivity enhancements
- ✓ RTL modeling enhancements
- ✓ Data encapsulation enhancements
- ✓ Assertions
- ✓ Abstract modeling enhancements
- ✓ Non object-oriented testbench enhancements
- ❑ **Object-oriented testbench enhancements**
- ❑ Summary

Object Oriented Classes

SUTHERLAND HDL
Training engineers to be HDL wizards

- SystemVerilog adds C++ like "classes" to the Verilog language
  - Allows Object Oriented Programming techniques
    - Primary intent is for use in verification
    - Can be used in abstract h/w models
  - Can contain
    - "properties" (data declarations)
    - "methods" (tasks and functions)
  - Similar to C++
    - Inheritance
    - Public, local or private encapsulation
    - New objects created and initialized using **new**
    - Polymorphism

```
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);
    ...
  endtask
endclass
```


Constrained Random Values

SUTHERLAND HDL
Training engineers to be HDL wizards

- Verilog **$random** returns a 32-bit signed random number
  - No way to constrain the random values returned
- SystemVerilog adds:
  - **rand** built-in class for creating distributed random numbers
  - **randc** built-in class for creating cyclic random numbers
  - Random values can be constrained

```
class Bus;
  randc bit[15:0] addr;
  rand  bit[31:0] data;

  // constrain addr to be word aligned
  constraint word_align {addr[1:0] == 2'b0;}
endclass
```

Dynamic Arrays and Associative Arrays

SUTHERLAND HDL
Training engineers to be HDL wizards

- Verilog has static arrays
  - The size of the array is fixed at compile time and cannot change

```
reg [31:0] mem [0:1024];
integer table [0:255];
```

- SystemVerilog adds:
  - Dynamic arrays
    - The size of the array is left open-ended
    - Built-in class methods are used to change the array size during simulation

```
logic [31:0] mem [];
int table [];
```

  - Associative arrays
    - The index into the array can be non-sequential values
    - Built-in class methods are used to access the array

```
typedef enum {A, B, C, D} state;
int table [state];
data = table[A];
```


Enhanced Synchronization: Mailboxes and Semaphores

SUTHERLAND HDL
Training engineers to be HDL wizards

- SystemVerilog includes built-in class definitions to synchronize verification activity
  - Semaphores
    - Represents a bucket with a fixed number of keys
    - Built-in class methods used to check keys in and out
    - 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 (gives mutually exclusive control)
  - Mailboxes
    - Represents a FIFO to exchange messages between processes
    - Built-in methods allow adding a message or retrieving a message
    - If no message is available, the process can either wait until a message is added, or continue and check again later

Coverage

SUTHERLAND HDL
Training engineers to be HDL wizards

- SystemVerilog adds:
  - A coverage class

```
enum {red,green,blue} color;
covergroup g1 @(posedge clk);
  c: coverpoint color;
endgroup
```

  - Cross coverage capability

```
enum {red,green,blue} color;
bit [3:0] pixel_addr;
covergroup g2 @(posedge clk);
  Hue:    coverpoint pixel_hue;
  Offset: coverpoint pixel_offset;
  AxC:    cross color, pixel_addr;
endgroup
```

  - Creation of meaningful coverage bins

```
int v_a;
coverpoint v_a {
  bins a = { [0:63],65 };
  bins b[] = { [127:150],[148:191]};
  bins others[] = default;
}
```


What Comes Next...

- ✓ Productivity enhancements
- ✓ RTL modeling enhancements
- ✓ Data encapsulation enhancements
- ✓ Assertions
- ✓ Abstract modeling enhancements
- ✓ Non object-oriented testbench enhancements
- ✓ Object-oriented testbench enhancements
- ❑ **Summary**

But Wait...There's More!

SUTHERLAND HDL
Training engineers to be HDL wizards

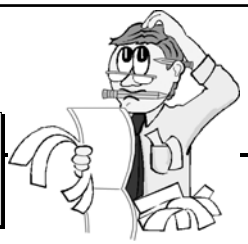
- There are many, many more enhancements to Verilog that were not covered in this presentation
  - tagged unions
  - static and automatic variables
  - const constants and const ref arguments
  - Nested modules
  - ref module ports (connect by reference)
  - Cycle-based testbench timing (clocking blocks)
  - Additional task and function formal argument enhancements
  - Assertion API, coverage API, extensions to Verilog VPI
  - And many more enhancements...
- These enhancements also fit into categories in this presentation

What Was Discussed

SUTHERLAND HDL
Training engineers to be HDL wizards

- SystemVerilog provides a large number of enhancements to the Verilog language
- It is not feasible to implement all the enhancements all at once
  - Need to prioritize what features to add first
- User's need the same features to work in all their tools
  - Cannot adopt SystemVerilog until EDA companies coordinate efforts
  - Don't make the same mistake as with implementing Verilog-2001!
    - Much of Verilog-2001 still can't be used due to lack of common support

So What Is The Plan?



SUTHERLAND HDL
Training engineers to be HDL wizards

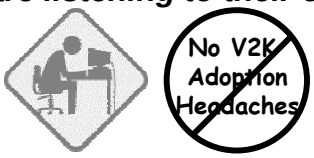
Caveat Number 3:
- **Stu Sutherland's opinion**
- **An "implementation standard" should define the implementation order**

- ❑ Start with a ***"Common Synthesis Subset"***
  - ❶ Productivity enhancements
  - ❷ RTL modeling synthesis enhancements
  - ❸ Data encapsulation enhancements
- ❑ Implement assertions
  - ❹ All of it — should be able to run everything in the SVA library
- ❑ Implement high-level language features
  - ❺ Abstract modeling enhancements (for BFMs and such)
- ❑ Implement OO verification capabilities
  - ❻ Non object-oriented testbench enhancements
  - ❼ Object-oriented testbench enhancements

Enabling Rapid Adoption of SystemVerilog

SUTHERLAND HDL
Training engineers to be HDL wizards

- ◆ ***EDA Companies are listening to their customers!***

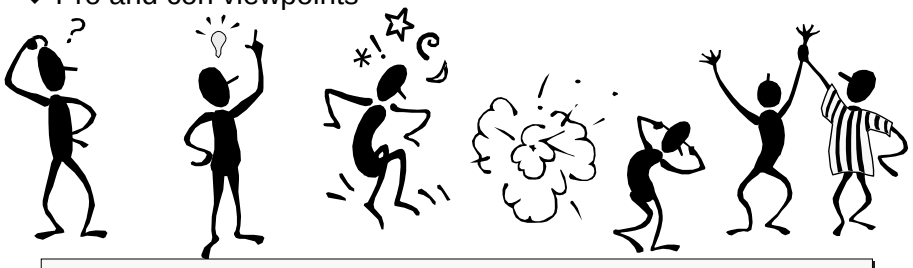


- ◆ A consortium of EDA companies and companies that design with Verilog is being created
  - ◆ The **"SystemVerilog Implementation Working Group"**
    - ◆ Coordinate the implementation of SystemVerilog
    - ◆ Increase confidence in implementation with test suites
    - ◆ Give Verilog users a way to provide implementation priorities
    - ◆ Enable engineers to adopt SystemVerilog more quickly!
- ◆ All EDA companies and user companies are invited to join!

What Do Others Think About A "Phased Implementation" Roadmap?

SUTHERLAND HDL
Training engineers to be HDL wizards

- You've heard Stu Sutherland's opinion...
- The panel that follows will provide:
  - What some EDA vendors think
  - What some Verilog designers think
  - Pro and con viewpoints



A copy of this presentation is available at www.sutherland-hdl.com/papers/2004-DAC_SystemVerilog_adoption_plan.pdf