



design & verification  
conference & exhibition

February 24 - 26, 2003

*SystemVerilog 3.1:*

*It's What The DAVEs In Your Company Asked For*

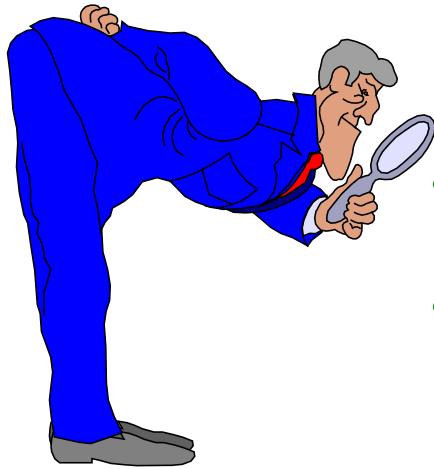
Stuart Sutherland  
Sutherland HDL, Inc.



*Training engineers  
to be HDL wizards*

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

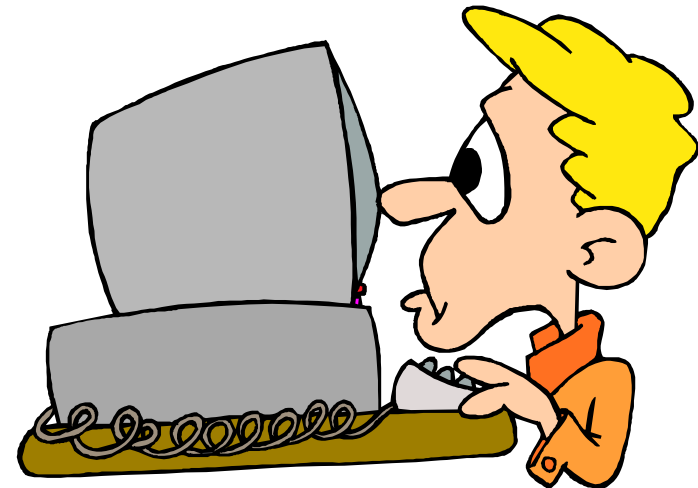
## This presentation will...



- Define what is “SystemVerilog”
- Provide an overview of the major features in SystemVerilog
- show the current status of the SystemVerilog standard
- The primary goal is to introduce you to the many exciting features that are in SystemVerilog — It’s a lot!

# Who Is DAVE?

**D**esign  
**A**nd  
**V**erification  
**E**ngineer



## Are You DAVE?

Some engineers specialize in design

Some engineers specialize in verification

***BUT...***

- All design engineers do some verification
- All verification engineers do some design

***Every engineer is a DAVE***  
***(Design And Verification Engineer)***



The early 1970's

ever increasing design size

DAVE

- Design sizes of a few hundred gates
- The same engineer did design and verification



The late 1970's

ever increasing design size

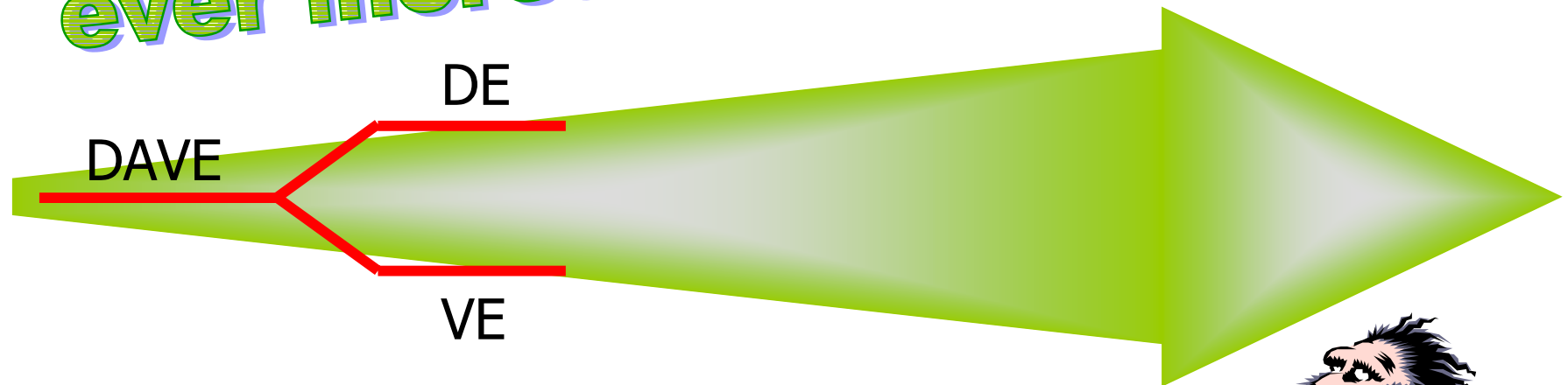


- Design sizes of a one or two thousand gates
- Some engineers did the design
- Other engineers “prototyped” the design
  - Verification was done using wire-wrapped breadboards



The early 1980's

ever increasing design size

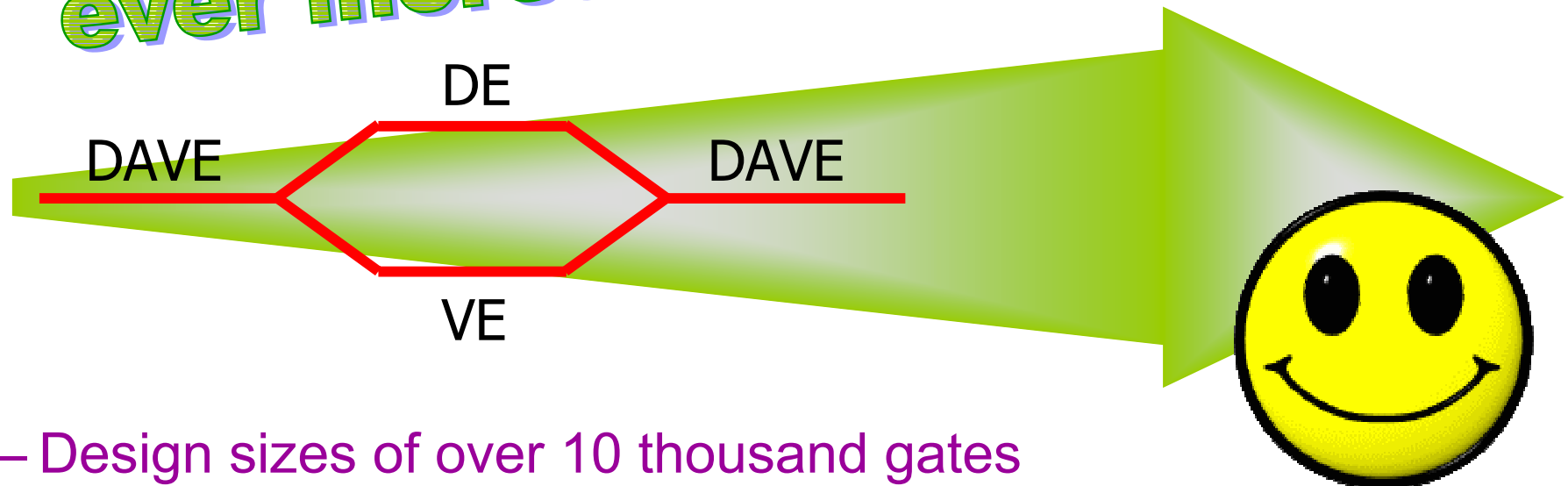


- Design sizes of a few thousand gates
- Some engineers did the design
- Other engineers did the verification
  - Each engineer used specialized languages and tools



The late 1980's

ever increasing design size

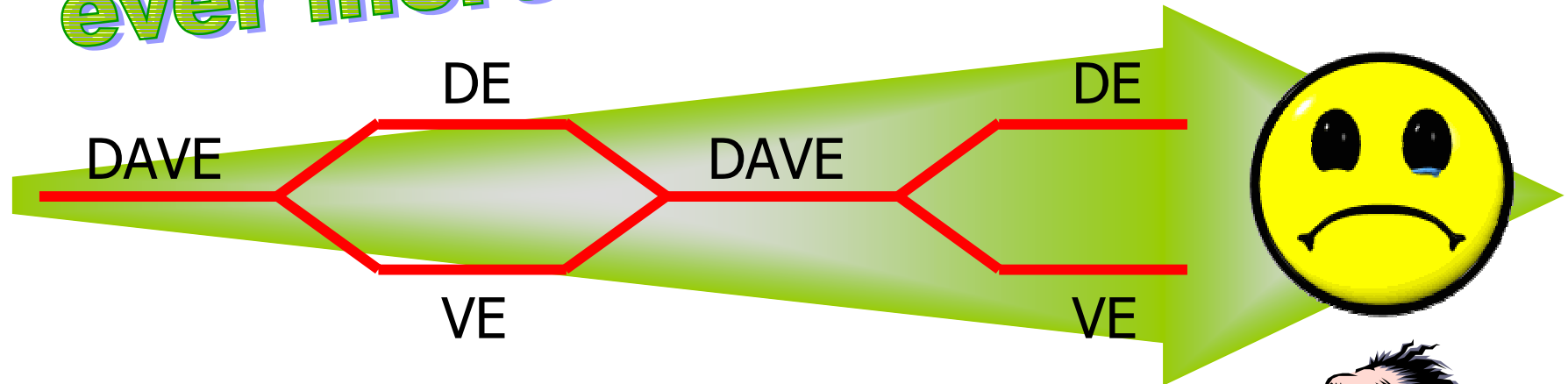


- Design sizes of over 10 thousand gates
- The “Verification Logic” (Verilog) HDL became popular
  - The same engineer could again do design and verification

***DAVE was one again! DAVE was Happy!***

## The late 1990's

ever increasing design size



- Design sizes of hundreds of thousands gates
- Some engineers did the design
- Other engineers did the verification
  - The design engineer used different languages and tools than the verification engineer



# 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 that the IEEE add to the next 1364 Verilog standard

# Mile High View of SystemVerilog

## SystemVerilog

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

## 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 `ifndef `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	+ = * /	%	>> <<
---------	------------	----------------	----------	--------	---------------------------	-------------------	-----------	-----------------------	---------------------	---------	---------	--------	----------	-----------	----------	--------------	------	---------------	-----------	-----------	-------	-------------	---------	--------	---------	---	-------

This presentation will just highlight these features  
Refer to the paper for full details

# SystemVerilog 3.0 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

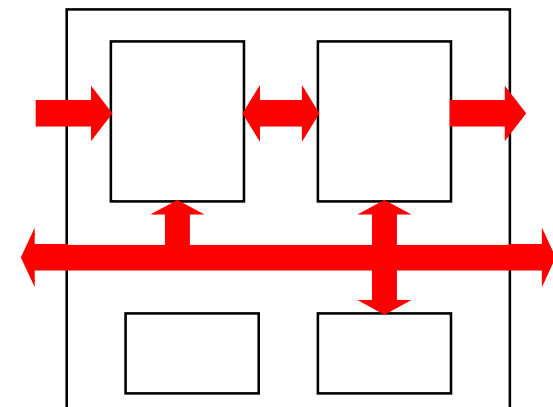
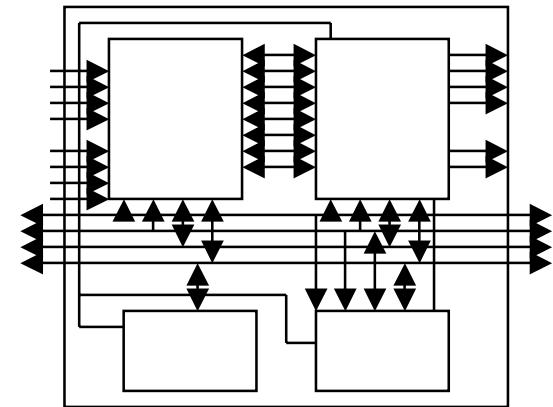
- Specialized always procedures
- Increment/decrement operators
- Unique decision statements
- Priority decision statements
- Bottom testing do-while loop
- Jump statements
- Statement labels
- Block name enhancements
- Task and function enhancements
- Continuous assignment enhancements
- Module port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

# SystemVerilog 3.0 Is Based On Proven Technology

- 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

# Interfaces

- **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 and protocol checking



# Abstract Data Types

- **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

# Enumerated Types

- **Verilog** does not have enumerated types
  - All signals must be declared
  - All signals must be initialized to a value
- **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;
```

# User-defined Types

- 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
```

# Structures

- **SystemVerilog** adds structures to Verilog
  - A collection of objects that can be different data types
    - 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

# New Operators

- 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++)  
...
```



# Reducing Model Ambiguity

- Verilog procedures are general purpose procedures
  - **always** is used to model **combinational**, **sequential** and **latched** logic
    - Software tools must “infer” (**guess**) what type of hardware an engineer intended based on context
  - **if-else-if** and **case** decisions execute with priority encoding
    - 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

This presentation will just highlight these features  
Refer to the paper for full details

## SystemVerilog 3.1 Features

- 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 3.1 Is Based on Proven Technology

- 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 “Sugar” assertion technology
    - 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

## Test Bench Blocks

- **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
```

**The program block helps ensure that test bench transitions do not have race conditions with the design**

# Assertions

- **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

**NOTE: At the time this paper was prepared, the exact keywords and expression syntax were still be defined**

# Object Oriented Classes

- 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
```

## Constrained Random Values

- 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 a cyclic random number
    - All possible values within a constrained will occur once and only once, and then a new random sequence will begin
  - Built-in class methods are used to constrain random values

# Mailboxes and Semaphores

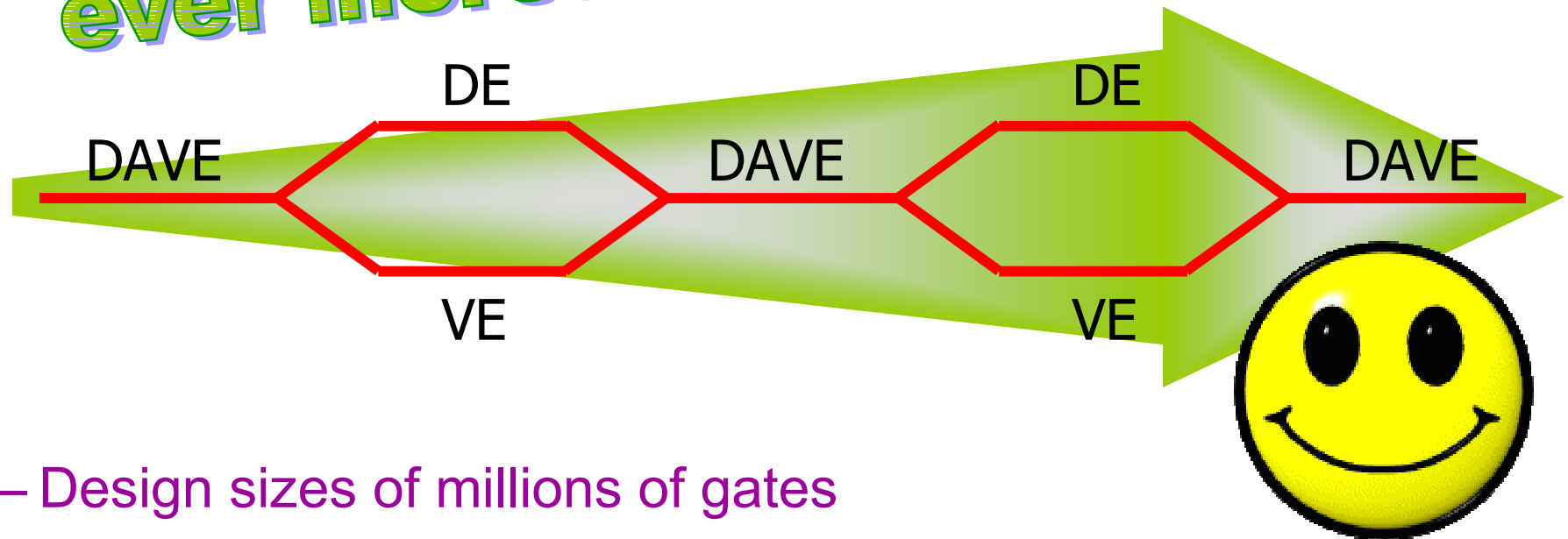
- **SystemVerilog** adds constructs for synchronizing processes
  - **Semaphores** are a built-in class that represents a bucket with a fixed number of keys
    - Class methods are used to check keys in and out
    - A process can check out one or more keys, and return them later
    - If not enough 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

# Direct C Language Interface

- **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, interfacing to C bus-functional models

# SystemVerilog Moves Design and Verification into the 21st Century

ever increasing design size



- Design sizes of millions of gates
- The “SystemVerilog” extensions to Verilog appeared
  - The same engineer (or team) can again do design and verification working with the same language

## Conclusion

- SystemVerilog combines an enhanced Hardware Description Language and an advanced Hardware Verification Language into **one unified language!**
  - SystemVerilog 3.0 enhances Verilog for modeling hardware
  - SystemVerilog 3.1 enhances Verilog for design verification
  - Presentation showed highlights — more details are in the paper

***DAVE is one again! DAVE is Happy!***



# Questions?

Copies of this presentation and full paper are available at :  
[www.sutherland-hdl.com/papers](http://www.sutherland-hdl.com/papers)

## SystemVerilog

3.1	test program blocks clocking domains mailboxes semaphores	persistent events process control constrained random values direct C function calls	classes inheritance strings	dynamic arrays associative arrays references	from C / C++
	3.0	assertions interfaces nested hierarchy unrestricted ports automatic port connect enhanced literals time values and units	dynamic processes 2-state modeling packed arrays array assignments specialized procedures enhanced event control unique/priority case/if	int shortint longint shortreal double char 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	+ = * / % >> <<
---	--	---	--	--	-----------------------