February 22-24, 2006

Cadence Incisive Verification Platform

SystemVerilog Workshop

Tim Pylant
Cadence Design Systems, Inc.

Agenda

- What is SystemVerilog?
- Increase Designer Productivity
  - Design Constructs
  - SystemVerilog Assertions
- Enhanced Testbench Capability
  - Testbench Constructs
  - Randomization and Constraints
- Coverage
  - SystemVerilog Coverage
- Summary

Issues Facing RTL Designers and Verification Engineers Today

- Designs are getting bigger
  - Need to code for reuse and higher abstraction
  - Need more efficient coding constructs
- Testbenches are growing exponentially
  - Generate more tests faster
  - More advanced techniques needed over traditional simulation
  - Need ways to measure verification progress
- What users want
  - An "incremental" change to their existing environment
  - Re-use of their existing code
  - Vendor-independent design and testbench language
  - Minimal learning curve

Productivity is critical!

3

What is SystemVerilog?

- ***SystemVerilog offers productivity!***
  - It is a concise, unified language for design and testbenches.
  - A single simulation tool can verify a design with advanced testbench and verification features included
- SystemVerilog adds extensions to the IEEE Verilog 2001 standard:
  - C/C++ type language constructs for efficient programming
  - Language enhancements for synthesis and downstream tools
  - Interfaces to encapsulate communication between design blocks
  - Assertions and coverage for new verification techniques
  - System-level testbench features to allow advanced verification methodologies
  - Lightweight interface to C/C++ programs

A single language for modeling complete digital systems!

4

SystemVerilog Misunderstandings

- Perception:
  - SystemVerilog is just Verilog - It's not a new language.
  - SystemVerilog will be easy to learn because it is "just Verilog"
- Reality:
  - SystemVerilog is a *significant* set of extensions to Verilog
  - SystemVerilog object-oriented features are just like C++ and SystemC and will take some time to learn

5

SystemVerilog Features

- SystemVerilog has a lot of new content
  - 97 new keywords
  - 31 different sections of the LRM
  - 584 pages in the SV3.1a LRM, all new content
- Major categories of SystemVerilog features
  - Convenience and synthesis features
  - Data Types
  - Interfaces
  - SystemVerilog Assertions (SVA)
  - Designer testbench
  - Object-oriented testbench
  - Coverage
  - Direct Programming Interface (DPI)

Designer Productivity Features

Testbench Enhancements

6

Design and Verification Productivity with SystemVerilog

- What is SystemVerilog?
- Increase Designer Productivity
  - Design Constructs
  - SystemVerilog Assertions
- Enhanced Testbench Capability
  - Testbench Constructs
  - Randomization and Constraints
- Coverage
  - SystemVerilog Coverage
- Summary

7

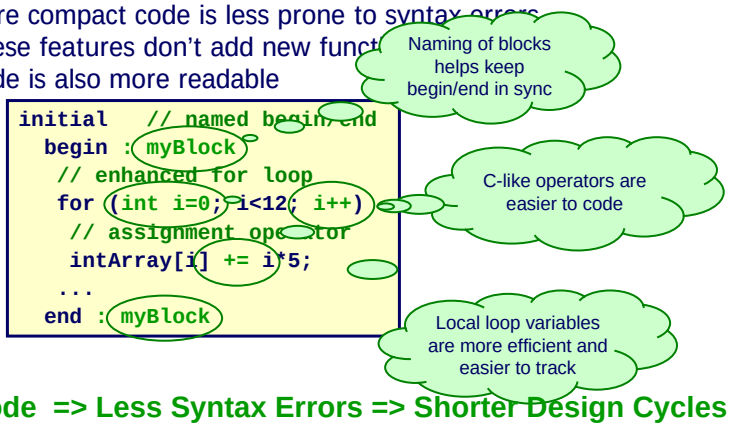
What you can do with SystemVerilog designer productivity features?

- Convenience features allow you to describe more functionality with less lines of code
- Synthesis features allow you to more clearly specify design intent
- New data types and packages added for improved readability and re-use
- Interfaces allow you to simplify design block communication
- Assertions allow you to specify and validate design behavior

8

SystemVerilog Convenience Features

- Allow you to describe more functionality with less code
  - More compact code is less prone to syntax errors
  - These features don't add new functionality
  - Code is also more readable



Less Code => Less Syntax Errors => Shorter Design Cycles!

Assignment and Increment/Decrement Operators

- SystemVerilog adds new **assignment** operators:

`+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, <<<=, >>>=`

  - They are semantically equivalent to a *blocking* assignment:

```
coord_X += 8;  // same as coord_X = coord_X + 8;
```

- SystemVerilog also adds **increment** and **decrement** operators:
  (*Pre-* `++var, --var` and *Post-* `var++, var--`)
  - These operators are also implemented as a *blocking* assignment
  - Use them in loops: `for ( i=0; i < 13; i++) ...`
  - Standalone statements:

```
--count;// count = count - 1;
size *= 2;// size = size*2;
```

- **Note:** In Incisive, **assignment** and **increment/decrement** operators are implemented as standalone statements only (not allowed in expressions)

Unsized Literals

- SystemVerilog adds the ability to specify unsized literal single-bit values with a preceding apostrophe (’), but without base specifier to left-fill the value with that bit.
- Legal values are: **’0, ’1, ’X, ’x, ’Z, ’z**

```
logic [7:0] tmpreg= '1; //8'hff
logic [11:0] dbus;

initial
begin
      dbus = '0;  // 12'h000
  #20 dbus = 'X;  // 12'hx
  #20 dbus = '1;  // 12'hFFF
  #20 dbus = 'z;  // 12'hz
end
```

```
module scale_mux (out, in_a,
                  in_b, sel_a);
parameter size = 1;
output [size-1:0] out;
input  [size-1:0] in_a, in_b;
input   sel_a;

assign out =  sel_a ? in_a :
             !sel_a ? in_b : 'x;
endmodule
```

11

Enhanced “for” Loop

- In SystemVerilog, loop control variables can be declared within the loop – creating a variable local to the loop.
  - SystemVerilog allows unnamed blocks to contain variable declarations
- Verilog 2001 requires:

```
integer i;  // index declared outside the for loop
for (i=0; i<10; i=i+1) ...   // unnamed block....
```

- SystemVerilog:

```
for( integer i=0; i<10; i++) ...
```

- SystemVerilog also allows multiple initialization and step statements in a for loop (unlike Verilog 2001)

```
for (integer i=0, integer j=0; i*j <= 250; i++, j++)
  $display("i:%d  j:%d  i*j:%d", i, j, i*j);
```

12

iff Event Control

- Provides conditional qualification of an event control

```
always @(posedge clk iff (enable == 1))
          data_out <= data_in;
```

- The event expression only triggers if the expression after the **iff** is true, in this case when **enable** is equal to 1.
- Expression is evaluated when '**clk**' changes, and not when '**enable**' changes.

```
module count8bit (input clk, rst_n, enable,
                  output logic [7:0] count  );
always @(posedge clk iff (enable == 1) or negedge rst_n)
  if (!rst_n)
     count <= '0;
  else
     count <= count+1;  // only increments if enable=1
endmodule
```

13

Procedural Block Enhancements

- Verilog 2001 provides the **always** procedural block.
  - Sensitivity lists, pragmas and coding styles are used to specify implementation intent of **always** blocks

```
always @(a or b or c or d) // combinational logic
always @(posedge clk or rst_) // sequential logic
```

- SystemVerilog adds implementation-specific procedural blocks to Verilog: **always_comb**, **always_latch**, **always_ff**
  - They reduce ambiguity in design by clearly indicating the hardware intent for a procedural block.
  - Simulation, formal, lint, synthesis, ec and other downstream tools have a consistent specification.

14

Procedural Blocks: always_comb

- SystemVerilog provides an **always_comb** procedure for modeling combinational logic behavior.
- Verilog 2001:
```
always @(b or c)
  a = b & c;
```
- SystemVerilog:
```
always_comb // NO sensitivity list
  a = b & c;
```

```
always_comb     // implied sensitivity list
  begin : combBlock
    case (opcode)
     ADD: result <= dataA + dataB;
     AND: result <= dataA & dataB;
     CPT: result <= myfunc(dataA, dataB);
     CLR: result <= 0;
    endcase
  end : combBlock
```

15

always_comb

- The **always_comb** procedure is different than a normal **always** procedure:
  - It has an inferred sensitivity list which includes every variable read by the procedure and in any function called by the procedure.
  - The variables written on the left-hand side of assignments cannot be written to by any other process.
  - The procedure is automatically triggered once at time zero, after all **initial** and **always** blocks, so that the outputs of the procedure are consistent with the inputs.
  - Statements in an **always_comb** cannot include those that block, have blocking timing or event controls or **fork...join** statements.

16

always_comb vs. always @*

- **always_comb** is implemented differently than **always @***
  - **always_comb** executes once at time zero and **always @*** waits for a change in the inferred sensitivity list.
  - **always_comb** is sensitive to changes in any functions in the block and **always @*** is only sensitive to changes in the arguments of a function.
  - **always @*** can include timing and can have variables that are assigned in multiple always blocks.

17

always_latch

- Example

```
always_latch  // inferred sensitivity list
   if(en)
      q <= d;
```

- Specifications/Restrictions:
  - **always_latch** has an inferred sensitivity list that executes identically to the **always_comb** procedure.
  - The variables written on the left-hand side of assignments shall not be written to by another process.
  - Statements in an **always_latch** shall not include those that block, have blocking timing or event controls or **fork...join** statements

18

always_ff

- Example

```
always_ff @(posedge clock or negedge
  reset)
   if (!reset)
      r1 <= 0
   else r1 <= r2 + 1;
```

- Specification/Restrictions
  - Contains one and only one event control and no blocking timing controls.
  - Variables written in the **always_ff** may not be written in any other block

19

Ports: .name implicit port connection

- SystemVerilog implicit port connections allows a user to reduce typing when the net name and the port name are identical.
- Verilog 2001: ordered port connections – *risky*

```
counter c1 (clk, rst, ld, data, cnt);
```

- Verilog 2001: named port connections – *safe but very verbose*

```
counter c1 (.data(data), .clk(clk), .rst(rst), .ld(ld),
  .cnt(cnt);
```

- SystemVerilog: implicit .name ports – *safe and less verbose*

```
counter c1 (.data, .clk, .rst, .ld, .cnt);
```

- In SystemVerilog, when names don't match, use named ports

```
counter c1 (.data, .clk, .rst(reset), .ld(load), .cnt);
```

- NOTE: You cannot have implicit wires with this method

```
wire [7:0] data, cnt;
wire clk, rst, ld;
```

20

Ports: .* implicit port connection

- SystemVerilog provides an additional way to reduce verbosity in placing module instances.
- The **.*** syntax automatically connects any ports that match exactly for that module instance.
  - Any connections that can't be inferred must be matched up manually
  - Cannot have implicit wires with this method either.
- Example from previous slide:

```
counter c1 (.*, .rst(reset), .ld(load));
```

- Mixing of positional and dot star implicit port connections are not allowed.

```
counter c1 ( dbus, mclk, .*);  // Not allowed
```

21

Summary: Convenience and Synthesis Features

- SystemVerilog includes convenience features to allows you to describe more functionality with less code
  - More compact code is less prone to syntax errors
  - Code is also more readable
- SystemVerilog adds features to allow you to specify design intent for synthesis and downstream tools
  - Don't need to use as many pragmas and synthesis directives
  - Simulation, formal, lint, synthesis, equivalence checking and other downstream tools have a consistent specification.

22

New Data Types in SystemVerilog

- SystemVerilog adds new data types to Verilog 2001, including:
  - 4-state **logic** type – similar to **reg**
  - 2-state **int, bit, byte, longint, shortint** types that are initialized to zero at time zero

bit	user-defined vector
byte	8-bit signed integer
int	32-bit signed integer
shortint	16-bit signed integer
longint	64-bit signed integer

```
// initialized to 'x
logic [7:0] data;
// 2-state variables
// initialized to zero
bit [2:0] opcode;
int index;
byte char1, char2;
```

  - User-defined data types (**typedef**)
  - Enumeration data types (**enum**)
  - **void** as the return type for a function that returns no value
  - Packages to declare new types, common tasks and functions

23

SystemVerilog logic Data Type

- The **logic** type was added to SystemVerilog to reduce confusion when using the "**reg**" data type.
  - **reg** is defined as a general-purpose **variable** in Verilog 2001 that can represent either sequential logic or combinational logic
  - In hardware – **reg** (register) refers to sequential design elements.
- **logic** has equivalent functionality to **reg** and can be used anywhere that a **reg** is traditionally used.

```
module counter (output [3:0] dout, input clk, rst, cnt);
logic [3:0] dout;
always_ff @(posedge clk or posedge rst)
  if (rst)
     dout <= '0;
  else if (cnt)
     dout <= dout + 1;
```

24

User Defined Data Types

- The **typedef** keyword allows user-defined data types.
- Types must be declared before they are used
  - Outside of a module (global type) => visible for all lexically following design units
  - Inside a module or inside any declarative scope => visible only within that module or declarative scope
  - Inside a package and imported into the module

```
`ifdef WIDEBUS
 typedef logic [63:0] bus_t; // 64-bit wide bus type
`else
 typedef logic [31:0] bus_t; // 32-bit wide bus type
`endif
bus_t bus1, bus2;   // variables of type bus_t
```

26

Enumeration Data Types

- Enumerations are vectors with defined named constants (enumeration constants) of specific bit patterns:
  `enum <type> { list_of_enumerations } ;`
- Examples:

```
enum {IDLE, BEGIN_XFER, WAIT_DONE, END_XFER} myStates;
enum bit [1:0] { ZERO=0, ONE, TWO, THREE=3 } numbers;
enum logic { YES=1, NO=0, NOTSURE='x } answers;
// Enumerations using typedefs
typedef enum logic [1:0] { success, warning, error}
  statusT;
statusT retval; // a var retval of type statusT
retval = error; // assignment of error to retval
```

- Enumeration constants values:
  - Implicit values: first constant => 0 otherwise previous constant value +1 (error if previous value contains x or z)

27

SystemVerilog Packages

- A package is a new Verilog design unit, similar to the VHDL package
  - Used to share declarations among modules, interfaces, programs and other packages
  - Defines a single/global set of items that can be used by any design unit that imports that package

```
package global_types;
  typedef enum {FALSE, TRUE} boolean;  // global typedef
  bit timeout_error;  // global variable initialized to 0
  task simulation_timeout (input time runtime);  // global task
        #runtime timeout_error = 1;
         $display("TIMEOUT: %m: %t", $time);
         $finish;
  endtask : simulation_timeout
endpackage
```

28

Packages

- Two ways to reference data in a package
  - Use it's *package item reference full name*

```
package global_types;
  typedef enum {FALSE, TRUE} boolean;
  typedef enum logic {H='1, L='0, Z='z, X='x} logic_state
endpackage

module error_checks;
...
  global_types::boolean suppress_warnings = FALSE;
  global_types::logic_state initial_state = global_types::X;
...
endmodule
```

  - Use the **import** statement to provide visibility to identifiers in the package

```
import global_types::*;
module error_checks;
...
    boolean suppress_warnings = FALSE;
    logic_state initial_state = global_types::X;
...
endmodule
```

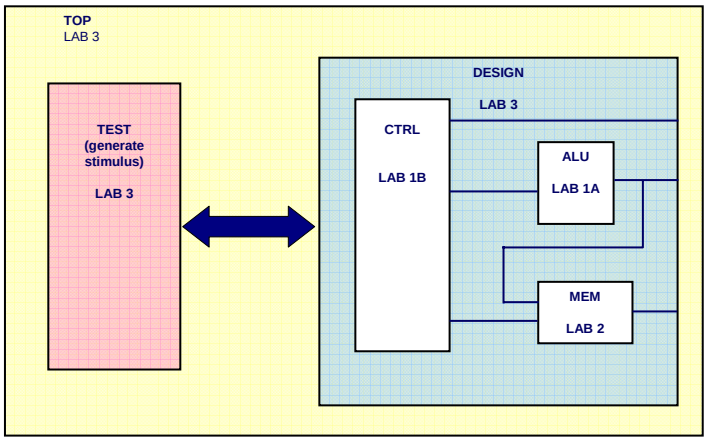
29

Summary:
What you can do with Data Types?

- 2-state data types (`bit, int, shortint, longint, byte`)
  - Remove “X” and “Z” states and are initialized to zero at time 0
  - Allows for variable types that are compatible with SysC and C/C++
- User-defined data types (`typedef`)
  - Allows users to define a type that is used throughout the design
  - Examples: modify bus widths without using parameters
- Enumerations (`enum`)
  - Give names to states in a FSM or op-codes in an instruction set
  - Can be used instead of parameter or ‘define
  - View, set breakpoints and debug enumerations using SimVision GUI
- Packages (`package .... endpackage`)
  - Define `typedefs`, functions/tasks and variables that can be re-used throughout the design
  - Good for common functions: printing messages, reading a bus, etc.

30

Design for Exercises
A Simple CPU



31

Today's Exercises

NOW

- 1A – Modeling an Arithmetic Logic Unit (ALU)
- 1B – Verifying a Simple Controller

LATER…

- 2A – Modeling a Memory Testbench
- 2B – SystemVerilog Interfaces
- 3 – Creating a Testbench with SystemVerilog Randomization
- 4 – SystemVerilog Coverage

32

Exercise 1A and 1B

- You will use SystemVerilog design constructs and data types to create an ALU module and to verify the controller design
- SystemVerilog Features to use:
  - Packages (**`package...endpackage`**)
  - enumerations and enum typedef (**`enum, typedef enum`**)
  - 4-state **`logic`** data type and new 2-state types (**`bit, int`**, …)
  - **`.name`** and **`.*`** port connections
  - **`always_ff`** keywords
  - unsized literal notation ('0, '1, 'x, 'z)
- Use: **`ncverilog +sv`** or **`ncvlog –sv`** command line options to compile modules containing SystemVerilog constructs
- SimVision also has enhancements for SystemVerilog

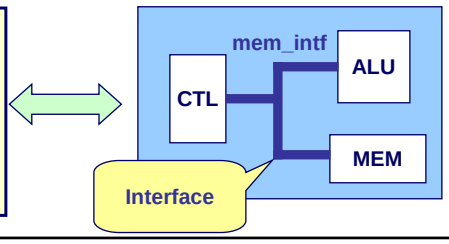
33

Review of Exercise 1A and 1B

SystemVerilog Interfaces

- Interfaces are a new feature added to SystemVerilog to:
  - Raise the level of abstraction and simplify design block communication by allowing a number of signals to be represented as a single port
  - Allow module port directional information and tasks/functions to be defined inside the interface.
  - Reduce the amount of code and promote reuse
  - Synthesizability allows usage in the design as well as the testbench

```
// Simple Interface
interface mem_intf;
  logic [15:0] data;
  logic  [4:0] address;
  logic        read;
  logic        write;
endinterface
```



Interface Definition

- A simple interface is a named bundle of nets/variables
- It is instantiated in a design and can be accessed
  - Through a port as a single item
  - Component nets/variables referenced where needed
- Interfaces allow you to define relationships between signals through module-like features:
  - Continuous assignments, tasks, functions, initial blocks, etc
- Additional features are unique to interfaces:
  - **`modports`** describe directional information for module ports and control the use of tasks/functions inside an interface

36

Example Design Without an Interface

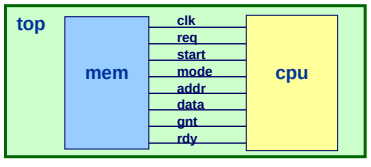
```
module memMod (
  input bit req, clk, start,
  logic [1:0] mode,
  logic [7:0] addr,
  inout wire [7:0] data,
  output bit gnt, rdy
  );
...
endmodule
```

```
module cpuMod (
  input bit clk, gnt, rdy,
  inout wire [7:0] data,
  output bit req, start,
  logic [7:0] addr,
  logic [1:0] mode
  );
...
endmodule
```

```
module top;
logic req, gnt, start, rdy;
logic clk = 0;
logic [1:0] mode;
logic [7:0] addr, data;

memMod mem (req, clk, start,
   mode, addr, data, gnt, rdy);

cpuMod cpu (clk, gnt, rdy, data,
   req, start, addr, mode);
...
endmodule : top
```



37

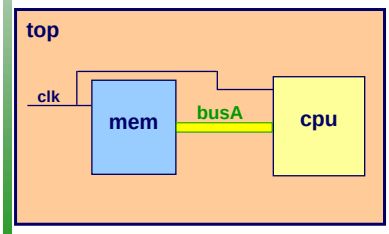
Example Using Interfaces

```
// Interface Definition
interface simple_bus;
  logic req, start, gnt, rdy;
  logic [1:0] mode,
  logic [7:0] addr,
  wire [7:0] data,
  ...
endinterface : simple_bus
```

all variables and nets are defined in one place

```
// Top-Level Testbench
module top;
logic clk = 0;
simple_bus busA ();
memMod mem (clk, busA);
cpuMod cpu (clk, busA);
...
endmodule
```

The testbench and other modules use the interface definition




```
module memMod (
  input clk,
  simple_bus bus
);
...
endmodule
```

```
module cpuMod (
  input clk,
  interface bus
  );
...
endmodule
```

Use of a generic interface

Interface References

- You are able to reference any object of an interface within any module that declared the interface in a port definition.
  - Interface variables are referenced relative to the interface name.
  - In this example – **bus** is used.

```
module memMod ( input clk, simple_bus bus );
  reg [31:0] mem [0:31];
  wire read, write;
  assign read  = (bus.gnt && (bus.mode == 0) );
  assign write = (bus.gnt && (bus.mode == 1) );
  always @(posedge clk)
    if (read)
       bus.data = mem[bus.addr];
    else if (write)
       mem [bus.addr] = bus.data;
endmodule
```

```
// Interface definition
interface simple_bus;
  logic req, start, gnt, rdy;
  logic [1:0] mode;
  logic [7:0] addr;
  logic [7:0] data;
endinterface : simple_bus
```

Interface Modports

- modports
  - Allow users to customize an interface for different modules
    - Provide direction information for module ports
    - Specifies which signals in the interface are accessible to a module

```
interface ms_bus;
  wire a, b, c, d;
  modport master (input  a, b, output c, d);
  modport slave  (output a, b, input  c, d);
     ...
endinterface
```

  - There are two ways to specify **`modports`** for a module definition:
    - Directly in the module header
    - In the module port connection when placing the instance

Using Modports (cont)

- Example of **modport** selection in the *instance declaration*
  - **interface ms_bus** defines a **master** and **slave modport.**
  - **mmod** specifies a **ms_bus interface** in the module definition for **Mbus**
  - **smod** specifies a **ms_bus interface** in the module definition for **Sbus**
  - The **testbench** creates an instance of **ms_bus** with the name **bus**.
  - The **mmod** instance connects **Mbus** with **bus** using the **master modport**.
  - The **smod** instance connects **Sbus** with **bus** using the **slave modport**.

```
interface ms_bus;
  wire a, b, c, d;
  modport master (input  a, b,
                  output c, d);
  modport slave  (output a, b,
                  input  c, d);
endinterface
```

```
module mmod (ms_bus bus);
endmodule
module smod (ms_bus bus);
endmodule
```

```
module testbench;
ms_bus bus ();
mmod mmod1 (.bus(bus.master));
smod smod1 (.bus(bus.slave));
endmodule
```

SystemVerilog Assertions (SVA)

- Specify and validate design behavior
  - Add to design blocks to specify expected behavior
  - Add to testbenches to verify communication between blocks and protocol sequences.
- Can also be used to provide functional coverage information on whether sequences of behaviors have

```
// concurrent assertions:
if_ab_then_cd : assert property
( @(posedge clk)
  a ##1 b |=> c ##1 d  );
```

```
// immediate assertion
always_comb
  if  (!sel)    mux_out = in0;
  else if (sel) mux_out = in1;
  else assert ('b1) $display
    ($time,,"Bad mux select");
```

- A SVA Workshop is offered to cover this topic in more detail

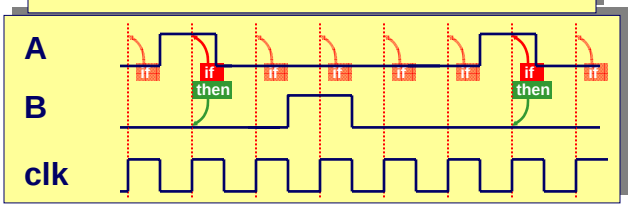
Conditional Assertions

- The implication operator:

```
|->
```

denotes IF – THEN

```
A_notB: assert property (
   @( posedge clk ) ( A ) |-> ( !B )
);
```

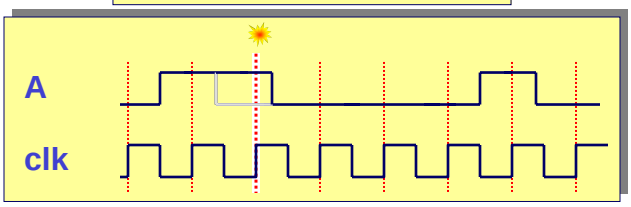


Conditional Assertions + Future Behavior

- Assertions can also watch for a sequence of events using:

`|=>` or `|->`

```
A_width: assert property (
  @( posedge clk ) ( A )
|=> ( !A ) );
```



44

Creating Sequences

- You can create a sequence of events using the *##<n>* operator.
  - This specifies the number of samples to wait before checking the next step in the sequence.

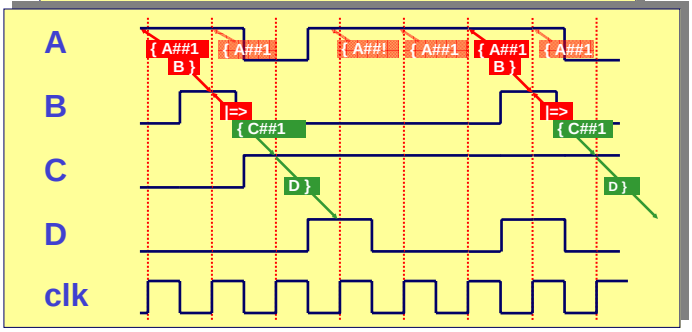
```
A_to_D_sequence: assert property
(
 @( posedge clk )
 ( A ##1 B ) |=> ( C ##1 D )
);
```

45

Working with Sequences

If A is followed by B, then next is C, and next is D.

```
A_to_D_sequence: assert property (
   @( posedge clk )  ( A ##1 B )
|=> ( C ##1 D )
);
```



Repeating Sequences

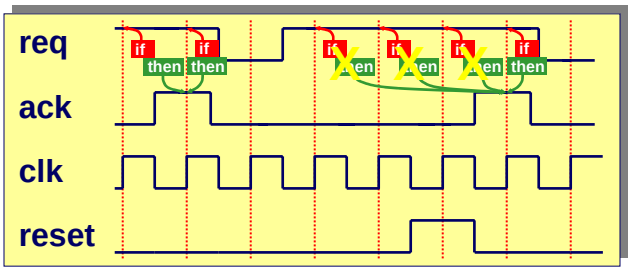
- To repeat a step in the sequence, follow the Boolean expression for the step with **[*<number>]**
- To repeat a step in the sequence for a range of numbers, follow the Boolean expression for the step with
  [*<min> : <max>]

```
ATM_CELL_ENABLE: assert property
(
  @( posedge clk )
    ( soc && clav ) |=> (
!en[*52] )
);
```

Terminating Assertions

- Assertions can be terminated using the ***disable iff*** construct.

```
req_ack: assert property (
   @( posedge clk ) disable iff
(reset )
    ( req ) |-> (##[0:$] ack )
);
```



48

Summary: SystemVerilog for Design Productivity

- Convenience features allow you to describe more functionality with less lines of code
- Synthesis features allow you to more clearly specify design intent
- Datatypes and packages added for improved readability and re-use
- Interfaces allow you to simplify design block communication
- Assertions allow you to specify and validate design behavior

49

Design and Verification Productivity with SystemVerilog

- What is SystemVerilog?
- Increase Designer Productivity
  - Design Constructs
  - SystemVerilog Assertions
- Enhanced Testbench Capability
  - Testbench Constructs
  - Randomization and Constraints
- Coverage
  - SystemVerilog Coverage
- Summary

50

What you can do with SystemVerilog testbench features?

- Enhancements to Verilog tasks and functions
- New datatypes, typedefs, interfaces and other convenience features
  - Raise level of abstraction of the testbench
  - Describe more functionality with less code
- Final blocks execute at end of simulation
  - Useful for error reporting, statistics collection and display

51

SystemVerilog Enhancements to Tasks and Functions

- SystemVerilog adds enhancements to tasks and functions
  - Multiple statements in a task/function don't require begin/end or fork/join block
  - Function **`output`** and **`inout`** ports
  - Functions can return a "**`void`**" type
  - Returning from a task/function before reaching the end (**`return`**)
  - Task/function arguments passed by name instead of order
  - Default task and function arguments values are allowed

52

SystemVerilog Functions: Output Arguments and Voids

- SystemVerilog allows function arguments to be declared with the same directional specifics as tasks (**`input, output, inout`**)
- The default direction for a function is **`input`** if it is not specified
- Functions can be declared without a return value

```
always @(a or b)
   add (a, b, sum);  // calling a void function

function void add ( input  integer a, b,
                    output integer sum  );
  sum = a + b;
endfunction : add  // endfunction name

// Void function with default value
function void print_err (integer error_cnt=0);
  $display("%d errors occurred", error_cnt);
endfunction
```

53

Jump Statement: Task/Function Return

return – exits from a task or function – in this example with a return value.

```
function integer mult (input integer num1, num2);
  begin
    if ((num1!=0) && (num2!=0))
      mult = num1*num2;    // V2001 function return method
    else
    begin
      $display("don't multiply by zero");
      return ('hx);
    end
  end
endfunction
```

return is only used inside a task/function

SystemVerilog Jump Statements (8.6)

SystemVerilog adds **break** and **continue** statements.

break – jumps out of a loop completely and continues with the next line of code after the loop:

```
for (reg[3:0] i=0; i<= 20; i++)
  begin
    if (i == 7)
      break;   // exits out of the loop after i=7
    $display("i: %d", i);  // prints 0,1,...6
  end
```

continue – jumps to the loop end and executes loop control.

```
for (reg[3:0] i=0; i<= 20; i++)
  begin
    if (i == 7)
      continue;   // skips next line when I=7
    $display("i: %d", i);  // prints all values except 7
  end
```

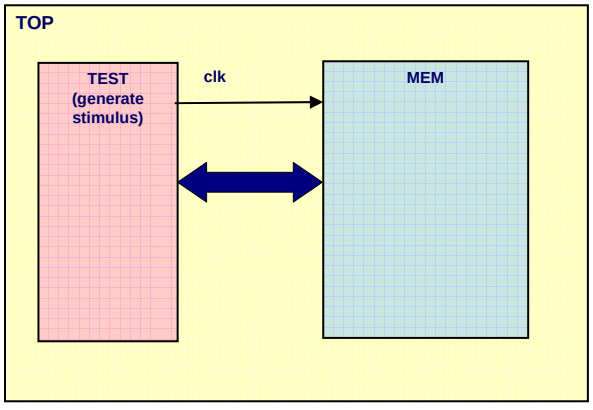
do ... while Loops

- Verilog2001 has **`for`**, **`while`**, **`repeat`** and **`forever`** loops
- SystemVerilog adds the "C" style **`do...while`** loop
  - Syntax: **`do <statement(s)> while (<condition>);`**
  - It always executes once and the ***`condition`*** is checked after statement(s) execute.

```
initial begin
  integer i = 3;
  do begin
    $write("I:%d", i);
    if (i < 5)  $display(" is a Low Number");
    else  $display(" is a High Number");
    i++;
  end
  while (i<= 10);  // condition is a boolean expression
end
```

56

Design for Exercise 2A and 2B A 32x8 Memory Design and Testbench



57

Today's Exercises

DONE!

- 1A – Modeling an Arithmetic Logic Unit (ALU)
- 1B – Verifying a Simple Controller

NOW

- 2A – Modeling a Memory Testbench
- 2B – SystemVerilog Interfaces

LATER…

- 3 – Creating a Testbench with SystemVerilog Randomization
- 4 – SystemVerilog Coverage

58

Exercise 2A and 2B

- In these Exercises you will be working with a Memory design.
  - You will be completing the memory testbench using SystemVerilog task/function enhancements
  - Then you will create a interface for the memory and its testbench.
- SystemVerilog Features to use:
  - Increment (`++`) and enhanced for loop (`for int i=0; i<32; i++)`
  - 4-state `logic` data type and 2-state types (`bit, int`, ...)
  - implicit `.name` and `.*` port connections
  - default task input arguments, default arguments with default value
  - task/function arg passing by name (`.name(name) or .name`)
  - `void` function (`function void <name> (...);`)
  - Simple interface definition and usage (`interface ... endinterface`)
  - Interface `modports` and tasks inside interfaces

59

Randomization of Scope Variables (std::randomize)

```
[std::] randomize ( [variable list])[with {constraint_block}];
```

- SystemVerilog introduces scope randomization, which allows you to assign unconstrained or constrained random values to variables in the current scope
  - Function args specify the variables to be assigned random values
  - randomize returns '**1**' if all random variables are valid, otherwise returns **'0'**

```
bit [7:0] addr, data;     //8-bit 2state variables
bit success;
initial begin
  for (int i=0;i<32;i++) begin
     success = randomize(addr, data);
     write_mem (addr, data);
  end
  for (int i=0;i<32;i++) read_mem (addr);
end
```


Scope Randomization with Constraints

- Use the **`with{...}`** clause to specify one or more constraint expressions.
- **`randomize`** returns a **1** if it succeeds or **0** if it is overconstrained.

```
bit [4:0] addr;     //5-bit 2state variable
byte data;          //8-bit 2state variables
bit success;
...
// Randomize addr, data: only between 32 and 126
success = randomize(addr, data) with {data>=32; data<=126;};
// Randomize addr: 25% between h00-h0f, 75% between h10-h1f
success = randomize(addr) with
       { addr dist { [5'h00:5'h0f]:=1,[5'h10:5'h1f]:=3};};
// Randomize data: between ranges of h41-h5a and h61-h7a
success = randomize(data) with
       { data inside { [8'h41:8'h5a],[8'h61:8'h7a] };};
```

Random Weighted Case (randcase)

- randcase
  - Case statement whose branches are randomly selected based on a branch weight
  - Probability of taking branch determined by weight/(sum of weights)

```
for (integer i=0; i<50;
  i++)
  begin
     randcase
       20 : gen_atm;
       30 : gen_ethernet;
       10 : gen_ipv4;
        5 : gen_crc_error;
     endcase
```

```
for (integer i=0; i<50; i++)
  begin
     randcase
       a     : gen_atm;
       a + b : gen_ethernet;
       a - b : gen_ipv4;
     endcase
```

62

Setting Seed for Randomization (process::self.srandom)

```
process::self.srandom( seed );
```

- The srandom() method allows manually seeding the Random Number Generator (RNG) of objects or threads. The RNG of a process can be seeded using the srandom() method of the process (see Section 9.9).
- The srandom() method initializes an object's random number generator using the value of the given seed.

```
initial
   //set a seed at the start
      process::self.srandom(100);
end
```

63

Final Blocks

- SystemVerilog adds a “final” procedural block that executes at the end of simulation.
  - Executes after explicit or implicit call to **`$finish`**
  - Similar an initial procedural block, final blocks only trigger once during a simulation (at the end)
  - Like a function, only zero-time statements are allowed
  - Typically used to display statistical information about the simulation

```
final begin  // executes at the end of simulation
 if (timeout_error)
   $display (“ERROR: %0t: Test Timed Out”, $time);
 else
   $display (“INFO: %0t: Test Complete”, $time);
 $display(“Error Count: %d”, error_count);
 $display(“Fifo Overflow Count: %d”, fifo_overflow);
end
```

64

Direct Programming Interface (DPI)

- DPI provides a means to:
  - Call ‘c’ functions (import) from SystemVerilog
  - Have a ‘c’ language function directly call a SystemVerilog task or function (export)
  - In Incisive Version 5.5 only import support is provided
    - Syntax to import a ‘c’ function

```
import {“DPI” | “DPI-C”} [context | pure] [c_identifier =] function
function_data_type function_identifier ([tf_port_list]);
```

    - Syntax to import a ‘c’ task

```
import {“DPI” | “DPI-C”} [context] [c_identifier =] task
task_identifier ([tf_port_list]);
```

65

Direct Programming Interface (DPI)

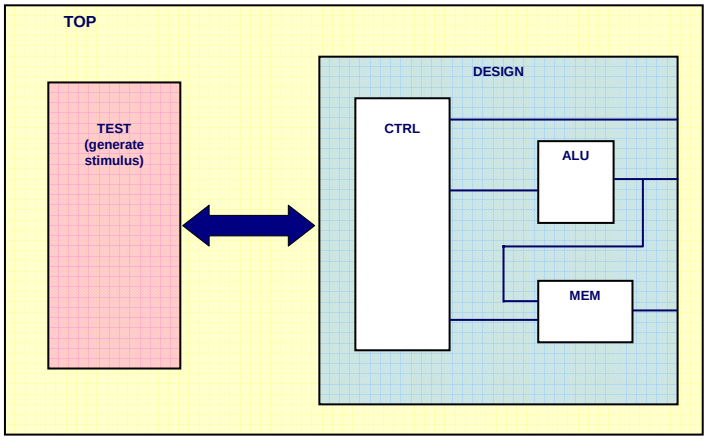
- Import call example:

```
import "DPI-C" pure calc_parity_func = function int parity_func
([input int a]);
```

  - **calc_parity_func** is the 'c' function name (c_identifier)
  - **int** is the datatype of the function return value (function_data_type)
  - **parity_func** is the function identifier as used in the SystemVerilog code
  - The function has one **input** a of type **int**
- Use model with IUS
  - The 'c' model must be compiled and linked into a shared object named libdpi.so (libdpi.sl on HP)
  - Include the shared object in the library path
  - Include the import clause in the Verilog source

66

Design for Exercises



67

Today's Exercises

DONE!

- 1A – Modeling an Arithmetic Logic Unit (ALU)
- 1B – Verifying a Simple Controller
- 2A – Modeling a Memory Testbench
- 2B – SystemVerilog Interfaces

NOW!

- 3 – Creating a Testbench with SystemVerilog Randomization

LATER…

- 4 – SystemVerilog Coverage

Exercise 3

- In this Exercise you will be working on the full design.
  - Create a top-level hierarchy for design and top
  - Modify the testbench to add random tests using SystemVerilog scope randomization
- SystemVerilog Features to use:
  - Enhanced `for` loop (`for int i=0; i<32; i++`)
  - 4-state `logic` data type and 2-state types (`bit`, `int`, …)
  - implicit `.name` and `.*` port connections
  - randomization wit constrains (`randomize`)
  - random case executions with weightings (`randcase`)

Design and Verification Productivity with SystemVerilog

- What is SystemVerilog?
- Increase Designer Productivity
  - Design Constructs
  - SystemVerilog Assertions
- Enhanced Testbench Capability
  - Testbench Constructs
  - Randomization and Constraints
- Coverage
  - SystemVerilog Coverage
- Summary

70

SystemVerilog Coverage

- Coverage allows the user to tell how well a design has been tested
  - Randomization requires coverage metrics to make sure constraints are correct and design functionality is being tested
- SystemVerilog has two types of functional coverage
  - SystemVerilog Assertions (SVA) provide control-oriented coverage
  - SystemVerilog **`covergroups`**, **`coverpoints`** and **`cross`** products for data-oriented functional coverage

SystemVerilog Assertion Coverage:

```
full_then_empty : assert
  property (@(posedge clk)
      fifo_full |=>
      ##[1:$] fifo_empty);
```

Functional Coverage Group:

```
covergroup cg1 @(posedge clk);
  Addr: coverpoint addr
  {  bins low   = { [0:'h0F], 19 };
     bins mid[] = { 16, 17, 18 };
     bins high  = { ['h14:'hFF] }; }
  AddrXvalid : cross Addr, valid;
endgroup : cg1
```

71

SystemVerilog Data-Oriented Coverage

- SystemVerilog provides language constructs for specification of functional coverage models.
- SystemVerilog coverage is achieved by doing the following:
  - Define a coverage model (**`covergroup`**)
  - Define coverage points for the model (**`coverpoint`**)
  - Defining cross-coverage points between coverage points (**`cross`**)
  - Optionally specify coverage point bins for tracking (**`bins`**)
  - Place instances of the coverage model in the design

72

Creating covergroup Instances

- The ***`covergroup`*** encapsulates the specification of a coverage model:
- Example of a ***`covergroup`*** definition:

```
covergroup cg1 @(posedge clk);
    < definition of covergroup >
endgroup
```

- The covergroup construct is a user-defined type.
- Multiple instances of that type can be created in different contexts.
  - covergroups can be placed inside a module or a named block
- Syntax for placing covergroup instances:

```
cg1 cg_inst = new( ) ;   // () is optional
```

73

Covergroup Example

define the covergroup (**cg1**)

covergroup instance (**cover_inst**)

define the coverpoints (**c1 and c2**) and cross points (**x1**)

```
module example;
logic clk;
logic [15:0] address;
logic [2:0]  opcode;
logic valid;

covergroup cg1 @(posedge clk);
   c1: coverpoint opcode;
   c2: coverpoint address;
   x1: cross c1, valid;
endgroup : cg1

cg1 cover_inst = new();

...

endmodule
```

74

Coverpoints (20.4)

- ***Coverpoints*** are the variables you are interested in tracking.
- From these variables you may be interested in tracking specific values or ranges of values
- During simulation, the values for variables defined as coverpoints are tracked and stored in a coverage database
- Syntax:

  [*coverpoint_id* **:**] **coverpoint** *variable_id* **;** |
  { bins_defn }
- Example:

```
c1 : coverpoint address;
c2 : coverpoint data;
```

75

Coverage Bins

- Bins can be created two ways – implicitly and explicitly
- While defining a **`coverpoint`**, if you do not specify any bins, Incisive will create implicit bins.
  - For an **`enum`** – it creates bins based on the data type. For example, an **`enum`** with a range of [3:0] will have 16 bins.
  - The maximum number of implicit/automatically created bins is 64 in Incisive (can be over-ridden using the **`auto_bin_max`** option)
- Use explicit binning when you know the values that you want to store
  - This is recommended

76

Bins – Vector and Scalar Bins

- There are two types of bins:
  - Scalar bin: for all values in the set of values only a single bin is created

```
coverpoint var1 {
  bins V = {1, 2, 5}; // bin V increments for 1,
  2 or 5
  }
```

creates a single bin

  - Vector bin: a unique bin is created for each value

```
coverpoint var1 {
  bins V[] = {1, 2, 5}; // bins V[1], V[2] and
  V[5]
  }
```

[ ] for multiple bins

77

Bins Example

```
module example_with_bins;
logic clk;
logic [15:0] address;
logic [2:0]  opcode;

covergroup cg1 @(posedge clk);
   c1: coverpoint opcode;    // implicit bins – 8 created
   c2: coverpoint address {
        bins low[] = { [0:'h0F] } ;    // 16 - low[0]...low[F]
        bins high  = { ['h1F:'hFF] } ; // 1 – high
        bins other = default ;         // 1 bin for [10:1E]
       }
endgroup : cg1

cg1 cover_inst = new();

...

endmodule
```

78

SystemVerilog Cross Coverage

- A *coverpoint* allows tracking of values received on a variable and perform binning on those values
- A *cross product* allows a user to keep track of simultaneous values received by more than one *coverpoint*

```
CrossAB: cross a, b;
```

  - Causes coverage engine to keep track of values of a and b together
- Crosses can be applied to:
  - Pre-defined coverpoints within the same covergroup
  - Variables which are visible in the scope
  - A combination of coverpoints and variables

79

Cross Product Example

- Example:

```
reg [1:0] a;
reg [3:0] b;
reg c;
covergroup cg @(posedge clk);
  CP_b: coverpoint b {
     bins b1   = { [9:12] };  //one bin b1
     bins b2[] = { [13:15] }; //3 bins:b2[13], b2[14], b2[15]
     bins restofb[] = default;//9 bins: ignored for cross
    }
  CP_c: coverpoint c; // two bins
  AxBxC: cross a, CP_b, CP_c; //32 bins: a(4) x CP_b(4) x CP_c(2)
endgroup : cg
```

Crosses created:

```
AxBxC.auto[0] = <a.auto[0], b.b1,     c.auto[0]>
AxBxC.auto[1] = <a.auto[0], b.b1,     c.auto[1]>
AxBxC.auto[2] = <a.auto[0], b.b2[13], c.auto[0]>
AxBxC.auto[3] = <a.auto[0], b.b2[13], c.auto[1]>
AxBxC.auto[4] = <a.auto[0], b.b2[14], c.auto[0]>
....
```

80

Running Coverage in Incisive

Recording functional coverage information with Incisive:

`ncverilog +tcl+<batchfile>`

tcl commands required (place in the batch file)

`coverage –setup <setup_options>`

`coverage –functional <functional_coverage_options>`

Setup Options: **`-testname <tname>`**
`-dut <instname>`
`-workdir <dirname>`

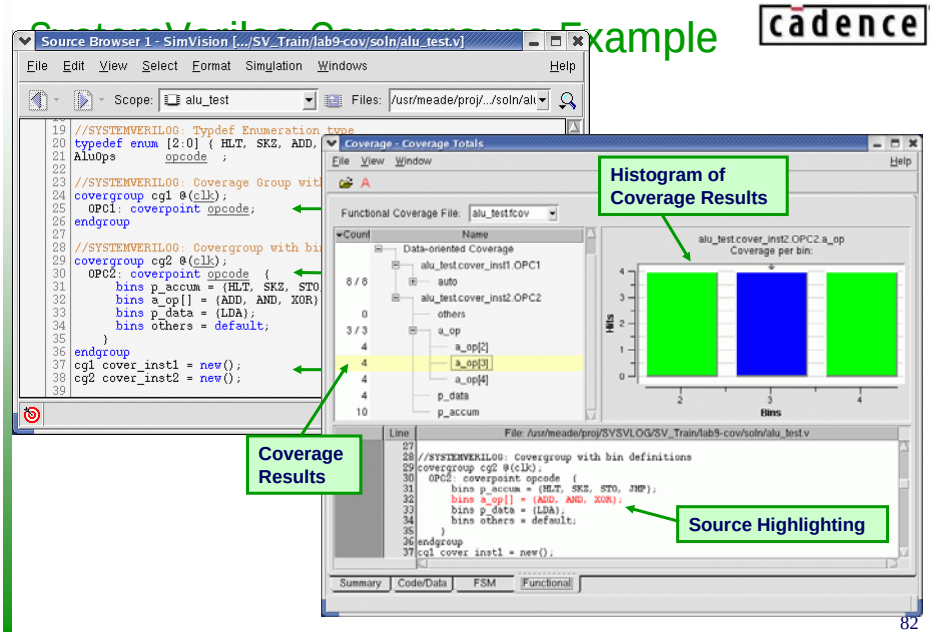
Coverage Options: **`-database –local_db <name>`** - for new dbase
`-database –aggregate_db <name>` - for existing dbase

Quick Setup (*batch.tcl*)

```
coverage –setup
coverage –functional
```

81

SystemVerilog Covergroup Example



82

Today's Exercises

DONE!

- 1A – Modeling an Arithmetic Logic Unit (ALU)
- 1B – Verifying a Simple Controller
- 2A – Modeling a Memory Testbench
- 2B – SystemVerilog Interfaces
- 3 – Creating a Testbench with SystemVerilog Randomization

FINALLY….

- 4 – SystemVerilog Coverage

83

Exercise 3 – SystemVerilog Coverage

- In this exercise, you will observe how covergroups work
  - Add covergroups to the alu testbench and measure coverage
  - Add covergroups to the design testbench and measure coverage

84

Design and Verification Productivity with SystemVerilog

- What is SystemVerilog?
- Increase Designer Productivity
  - Design Constructs
  - SystemVerilog Assertions
- Enhanced Testbench Capability
  - Testbench Constructs
  - Randomization and Constraints
- Coverage
  - SystemVerilog Coverage
- Summary

85

Cadence Is Committed to Standards

- ***Cadence commitment***
  - To ensure unified standards for advanced design and verification
  - Cadence has donated and opened up more than a dozen major proprietary languages and formats to the industry, including Verilog, GDSII and SDF.
- Cadence provides current and continuing support for the VHDL, Verilog, SystemVerilog, *e*, PSL, OVL, SystemC, Verilog-AMS, and VHDL-AMS standards
- Cadence is aggressively implementing SystemVerilog
  - Have already delivered SystemVerilog in Incisive, RTL Compiler, and Conformal LEC
  - Comprehensive roadmap in place for complete implementation across all Cadence product lines

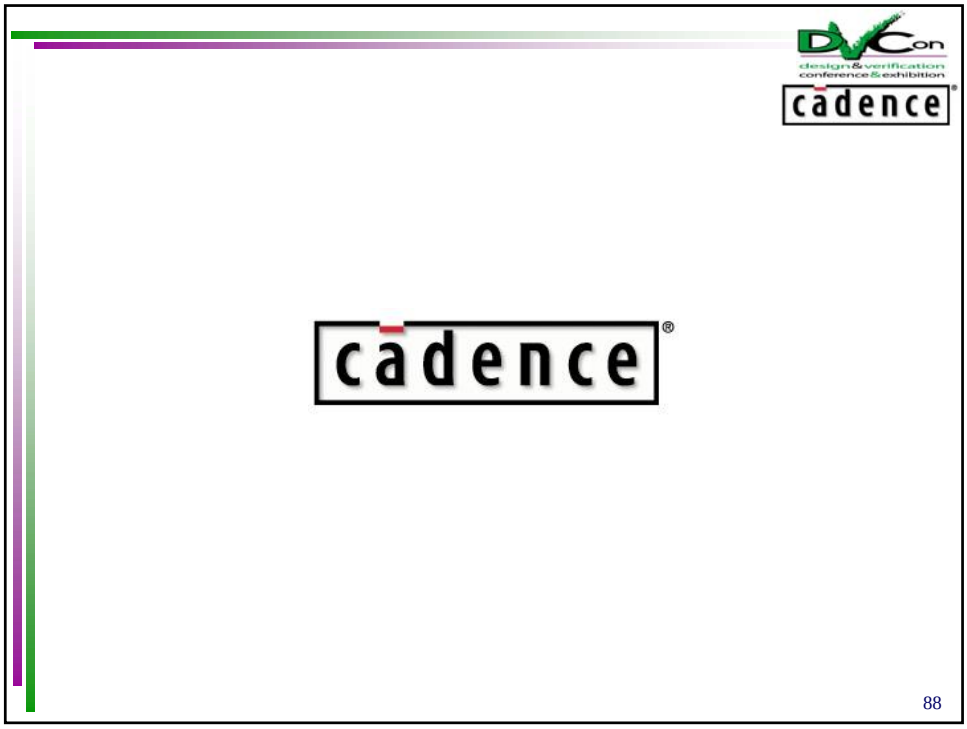
86

Summary

- Designers and Verification Engineers are facing major issues
  - Designs sizes are getting bigger
  - Testbenches are growing exponentially
- They want to improve their productivity without completely changing their design and verification methodology
- SystemVerilog offers features to improve productivity
- Cadence is integrating SystemVerilog into our complete verification product line to provide both performance and efficiency

Stayed tuned for more SystemVerilog news from Cadence!

87


DVCon
design&verification
conference&exhibition
cadence
cadence
88