SUTHERLAND HDL
Training engineers to be SystemVerilog wizards
www.sutherland-hdl.com

snug 25
Synopsys Users Group

Who Put Assertions In My RTL Code? And Why?

How RTL Design Engineers Can Benefit from the Use of SystemVerilog Assertions

Stuart Sutherland
Sutherland HDL, Inc.

March 23, 2015
SNUG San Jose, California

Agenda

The goal of this paper is to encourage RTL design engineers to take advantage the many ways in which SystemVerilog Assertions can help them!

Part 1: A brief overview of SystemVerilog Assertions

Part 2: Assertions that Design Engineers should write

Part 3: SystemVerilog constructs with built-in checks

Part 4: Simulation and Synthesis support for SVA

"Immediate and concurrent assertions are heavily used at my uP company by both design and verification engineers…for assertions that are just testing RTL behavior, the assertions are embedded directly in the RTL code itself." (an engineer at a large Micro Processor company)

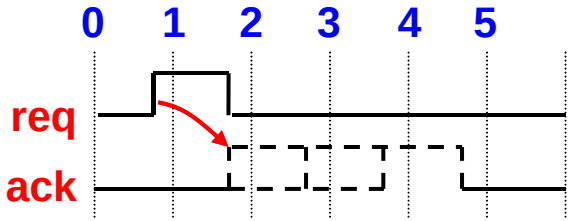
Synopsys Users Group

Part One

A Short Tutorial On SystemVerilog Assertions

What Is An Assertion?

- An assertion is a statement that a certain property must be true



Design Specification:
After the request signal is asserted, the acknowledge signal must arrive 1 to 3 clocks later

- Assertions are used to:
  - Verify design meets the specification over simulation time
  - Describe functional coverage points
  - Provide semantics for formal verification

Are these benefits important in your projects?

- Designers benefit from adding assertions in the RTL code by:
  - Documenting design intent (e.g. state machine uses one-hot)
  - Clarifying ambiguities in spec (e.g. can ack be tied high?)
  - Validate design assumptions (e.g. critical inputs are connected)
  - Localize where failures occur in the design instead of at the output

Embedded Verification Checking and Synthesis

- Without assertions, embedded checks must be hidden from Synthesis using conditional compilation or pragmas
  - Embedded checking can make RTL code look ugly!

How many design engineers do you know that will add this much extra code to an if…else RTL statement?

```
if (mode)
   // do true statements
else
//synthesis translate_off
if (mode == 0)
//synthesis translate_on
   // do the not true statements
//synthesis translate_off
else
   $display("mode is either an X or Z");
//synthesis translate_on
```

RTL code

checker code

RTL code

checker code

This checking code is hidden from synthesis, but it is always active in simulation (not easy to disable for reset or for low-power mode)

- SystemVerilog Assertions are easier, and synthesis ignores SVA

```
assert (!$isunknown(mode)) else $error("mode is either an X or Z");
if (mode) ...  // do true statements
else      ...  // do not true statements
```

`assert` is ignored by synthesis
`assert` can be disabled in simulation

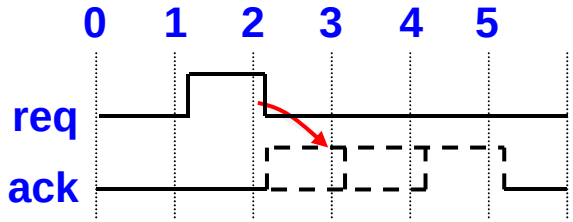
SystemVerilog Has Two Types of Assertions

- **Immediate assertions** test for a condition at the current time
  - Similar to an `if`…`else` statement, but with assertion controls

```
always_comb begin
  assert ($onehot(state)) else $error;
  case (state) ...  // RTL code
```

reports an error if the `state` variable is not a one-hot value

- **Concurrent assertions** test for a sequence of events spread over multiple clock cycles
  - Execute as a background process in parallel with the RTL code




```
a_reqack: assert property (@(posedge data_clk) req |-> ##[1:3] ack)
          else $error;

always_ff @(posedge clock)  // RTL code
  if (data_ready) req <= 1;
  ...
```

reports an error if `ack` is not high within 1 to 3 clock cycles after `req`

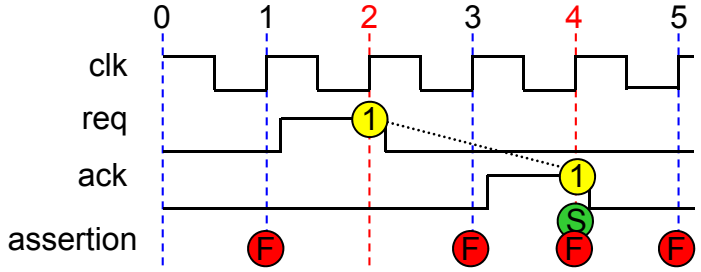
`##` represents a "*cycle delay*" – a "cycle" in this example is from one posedge of `data_clk` to the next positive edge

Concurrent Assertions Run in the Background Throughout Simulation

- Concurrent assertions start a new check every clock cycle

```
assert property (
  @(posedge clk)
  req ##2 ack )
else $error;
```

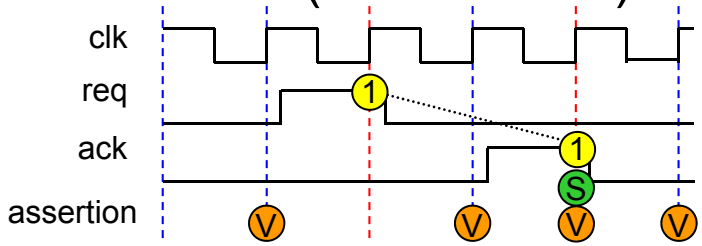
GOTCHA – this assertion will fail every clock cycle in which there is no `req`



- Assertions can be qualified with implication operators (**|->**, **|=>**)
  - If a condition is true, the sequence is evaluated
  - If a condition is false, the sequence is not evaluated (a don't care)

```
assert property (
  @(posedge clk)
  req |-> ##2 ack )
else $error;
```

don't do anything when there is no `req`



  - ***Antecedent*** — the expression before the implication operator
    - The evaluation only continues if the antecedent is true
  - ***Consequent*** — the expression after the implication operator
  - ***Vacuous success*** — when the antecedent is false, the check is not of interest, so evaluation is aborted without considering it a failure

Concurrent Assertions Only Sample Values on Clock Edges

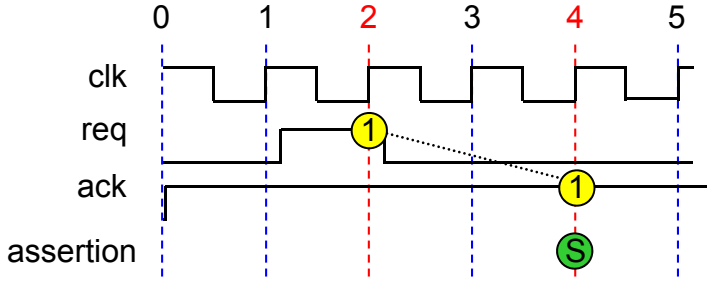
- Concurrent assertions can sample logic levels on each clock cycle

```
assert property (
  @(posedge clk)
  req |-> ##2 ack)
else $error;
```

GOTCHA – the assertion passes even though ack did not change

Is it OK for ack to be pulled high?

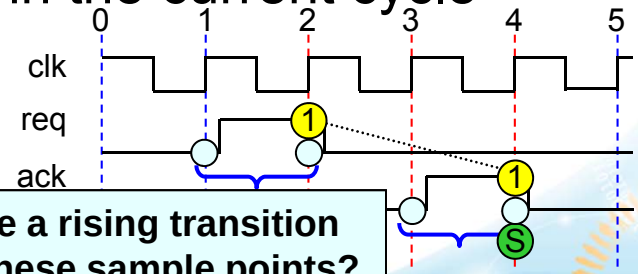
Writing assertions encourages engineers to read/clarify the spec!



- Concurrent assertions can look for a change between the last sampled value and the current sampled value
  - **$rose** – returns true if there was a rising change in the current cycle
  - **$fell** – returns true if there was a falling change in the current cycle
  - **$changed** – returns true if there any change in the current cycle
  - **$stable** – returns true if there no change in the current cycle

```
assert property (
  @(posedge clk)
  $rose(req) |-> ##2 $rose(ack) ;
else $error;
```

req and ack must transition



Was there a rising transition between these sample points?

Immediate and Concurrent Assertion Pros and Cons

You're the engineer – which of these pros and cons are most important in your project?

Immediate Assertions

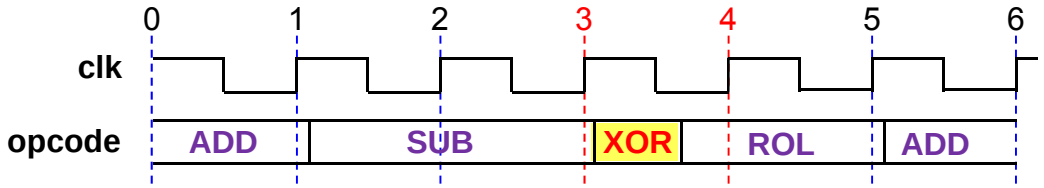
- Pros:
  - Easy to write – simple syntax
  - Close to code being checked
  - Can check asynchronous values between clock cycles
  - Self-documenting code
- Cons:
  - Cannot be bound (see slide 11)
  - Difficult to disable during reset or low-power
  - Must following good RTL practices to prevent race conditions (just like any programming statement)

Concurrent Assertions

- Pros:
  - Background task – define it and it just runs
  - Cycle based – no glitches between cycles
  - Can use binding (see slide 11)
  - Works with simulation and formal verification
- Cons:
  - More difficult to define (and debug)
  - Can be far from code being checked
  - Cannot detect glitches

When To Use Immediate Assertions, When To Use Concurrent Assertions

- There are many reasons signals might change more than once during a single clock cycle (a potential glitch)
  - Combinatorial decoding, clock domain crossing, async reset, …



This glitch within a clock cycle will affect my design functionality – I need to detect it.

You need an immediate assertion!

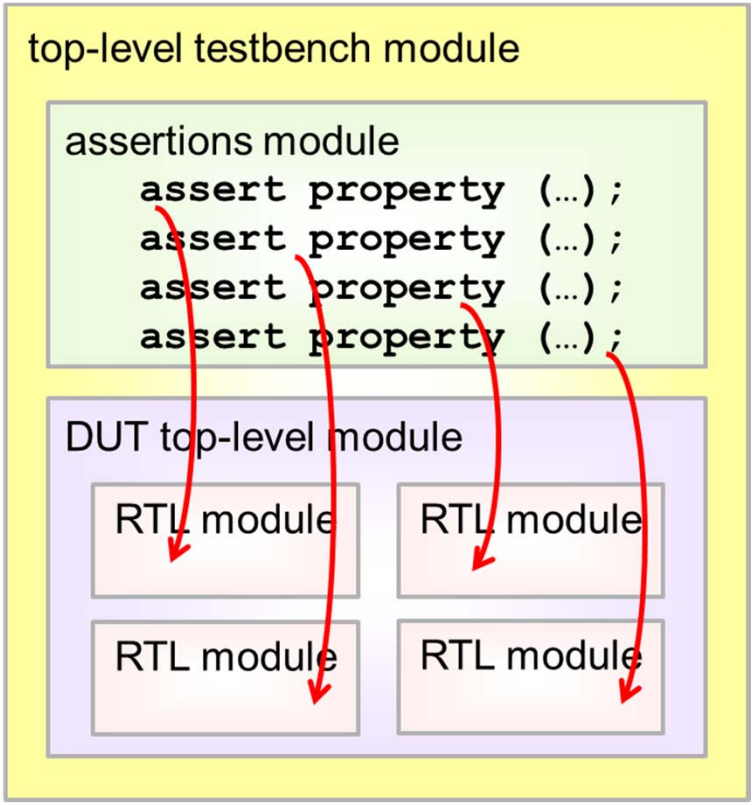
This glitch within a clock cycle will never be stored in my registers – I can ignore it.

You need a concurrent assertion!

- **Immediate assertions are programming statements that can evaluate values at any time**
- **Concurrent assertions are cycle based, and only evaluate values on a clock edge**

Assertion Binding

- SystemVerilog assertions can be defined in a separate file and:
  - Bound to all instances of a design module or interface
  - Bound to a specific instance of a design module or interface



- **Binding allows verification engineers to add assertions to a design without modifying the design files**
- **Binding allows updating assertions without affecting RTL code time-stamps (which could trigger unnecessary synthesis runs)**
- **Binding can also be used to bind in coverage and other functionality**

NOTE: Only concurrent assertions can be directly bound into other modules

Embedded versus Bound Assertions Pros and Cons

Which of these pros and cons are most important in your project?

Assertion Binding

- Pros:
  - Do not need RTL file access permissions to add assertions
  - Adding assertions does not impact RTL file time-stamps
- Cons:
  - Assertions can be far from the code being checked
  - RTL engineers must edit multiple files to add assertions while the RTL modes is being developed
  - Cannot (easily) use immediate assertions

Assertions Embedded in RTL

- Pros:
  - Close to the code being verified
  - Can use both concurrent and immediate assertions
  - Document designer's assumptions and intentions
  - Assertion errors originate from same file as the failure
- Cons:
  - Adding/modifying an assertion could trigger automated regression or synthesis scripts

When To Embed Assertions, When To Bind In Assertions

This paper recommends[1]…

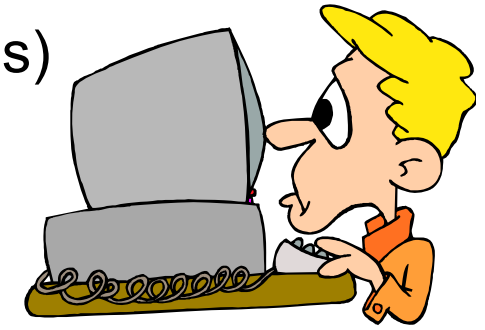
- Design engineers should embed assertions into the RTL code
  - Validate all assumptions (e.g. control inputs are connected)
  - Trap invalid data values where they first show up
  - Embedded assertions should be written at the same time the RTL code is being developed!
- Verification engineers should add bound-in assertions
  - Verify the design functionality matches the specification
  - Verify that corner cases work as expected (e.g.: FIFO full)
  - Verify coverage of critical data points
  - By using binding:
    - There is no need to check out and modify the RTL model files
    - Adding assertions not affect RTL file time stamps

[1]There can be exceptions to this guideline – you get paid the big money to figure out which way of specifying assertions is best for your projects!

Part Two

Assertions That Design Engineers Should Write

Design Engineers Should Add Assertions to RTL!



- RTL models ***assume*** inputs and other values are valid
  - Input ports are connected (no floating input values)
  - Control signals are never a logic X
  - State machine encoding is a legal value
  - Data values are within an expected range
  - Parameter redefinitions meet design requirements
- These assumptions ~~can be~~ *should be* verified using assertions
  - Most of these can be done with simple 1-line assertions
- The examples on the next few pages show how to:
  - Validate assumptions on reset values
  - Validate assumptions regarding value ranges
  - Validate assumptions on pulse widths
  - Validate parameter values after parameter redefinition
  - Eliminate problems with X-pessimism and X-optimism

Validating Assumptions On Critical Control Signals

An X or Z if…else control signal will take the "else" branch and propagate incorrect logic values that could:

- **Not be detected until much later in the design logic**
- **Not be detected until a much later clock cycle**
- **Go undetected**

VCS's Xprop can help by propagating an X instead of a known value, but must still trace back to the cause

- RTL models ***assume*** that control signals have known values
  - Reset is either 0 or 1, Enable is either 0 or 1, etc.
- A ***1-line immediate assertion*** or ***simple concurrent assertion***[1] can check this assumption!
  - Catch problems when and where they occur

```
module data_reg (input resetN,
                 ...          );
  always_ff @(posedge clk) begi
    assert ( !$isunknown(resetN) ) else $error("unknown value on resetN");
    if (!resetN) q <= 0;
    else         q <= d;
  end
```

Assumes `resetN` input is properly connected (an unconnected reset will set q <= d every clock cycle)

[1]Immediate and concurrent assertions handle glitches differently – use the type that best meets the needs of your project!

Something went wrong right here!

Validating Assumptions Regarding Value Ranges

- RTL code often ***assumes*** data values are within a valid range
  - Out of range values propagate as a functional bug
  - Can be difficult to detect
  - Might not be detected until downstream in both logic and clock cycles
- A ***1-line immediate assertion*** can check that values are within a required range!

```
module alu (input  logic [15:0] a, b,
            ...                        );
always_ff @(posedge clock)
  case (opcode)
    ADD_A_TO_B : result <= a + b;
    ... // other operations
    SHIFT_BY_B : begin
      assert (b inside {[1:3]}) else $error("b is out of range for shift");
      result <= a >> b;
    end
  endcase
```

Shift-right operation assumes the b input has a value of 1 to 3

Validating Assumptions On Pulse Widths

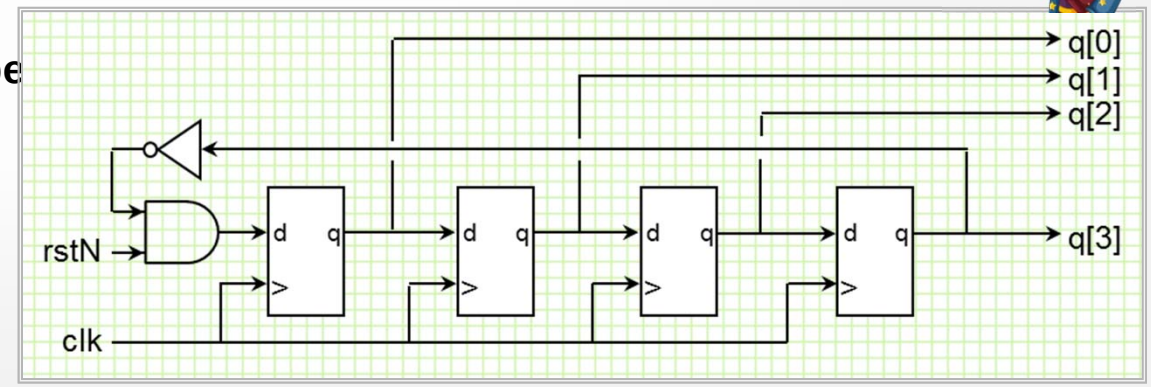
- RTL models sometimes ***assume*** certain signals remain true for some number of clock cycles
  - So that reset to propagate through multiple stages of logic
  - To allow devices to enter or leave low-power mode
- A simple ***concurrent assertion*** can check pulse widths!

Assumes `rstN` input meets the pulse width required by this model

```
module jcounter (input  logic clk, rstN,
                 output logic [3:0] q);

  assert property (@(posedge clk) $fell(rstN) |-> !rstN[*4])
  else $error("rstN did not remain low for at least 4 clock cycles");


  always_ff @(posedge clk) be
    q[0] <= ~q[3] & rstN;
    q[1] <= q[0];
    q[2] <= q[1];
    q[3] <= q[2];
  end
endmodule
```



Validating Parameter Values After Parameter Redefinition

- Parameterized models ***assume*** exterior code redefines the parameters to viable values
- An ***elaboration-time assertion*** can ensure redefined parameters have expected values!

```
module muxN  // 2:1 MUX (S == 1) or 4:1 MUX (S == 2)
#(parameter N=8, S=1)
(output logic [N-1:0] y,
 input  logic [N-1:0] a, b,
 input  logic [N-1:0] c=0, d=0, // c, d have default value if unconnected
 input  logic [S-1:0] sel
);

  generate
    if (S inside {[1:2]}); else $fatal(0,"S must be 1 or 2");
  endgenerate

  always_comb begin
    ...
  end
endmodule
```

Assumes `S` is only redefined to be `1` or `2`

Uh Oh, S was redefined to a value that won't work!

(`if...else` is used in generate blocks instead of `assert...else`)

Eliminating X-Pessimism and X-Optimism Gotchas

- RTL models are notorious for hiding problems involving X values
  - A non-X value is propagated instead of a logic X
    - Verification must determine the non-X value is incorrect functionality
    - Bugs must be traced back through logic and clock cycles to figure out where the problem first occurred
- A ***1-line immediate assertion***[1] can trap X values!
  - Do not need to detect and debug resulting functional bugs

```
always_comb begin

  assert final (!$isunknown(sel)) else $error("sel is X or Z");

  if (sel) y = a;
  else     y = b;
end
```

Caught you, you nasty X!

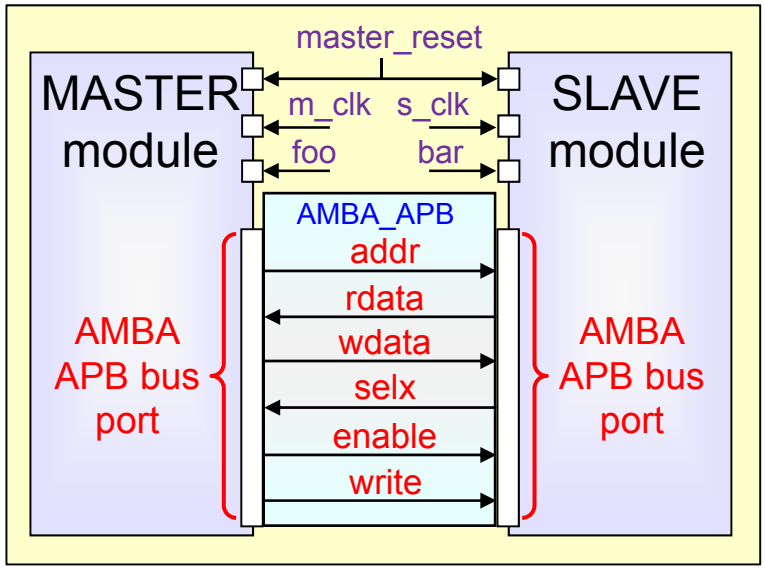
An unknown `sel` will propagate the value of `b`

VCS's Xprop can help by propagating an X instead b, but must still trace back to the cause

[1]Could also be written as a concurrent assertions – use the assertion type that best meets the needs of your project!

Self-Checking Interfaces

- An RTL interface port can be used to model bus protocols
  - Encapsulates the bus-related signals into a single port
- **Embedded assertions** in an interface can ***automatically detect protocol errors***
  - Protocol violations are detected at the moment they occur




```
interface AMBA_APB;
  logic [31:0] addr;
  logic [ 7:0] rdata, wdata;
  logic        selx, enable, write;

  property p_sel_enable;
    @(posedge clk)
    $rose(selx) |-> ##1 $rose(enable);
  endproperty: p_sel_enable

  assert property (p_sel_enable);

  ... // additional protocol checls
endinterface
```

Part Three

SystemVerilog Constructs With Built-in Assertion-like Checking

SystemVerilog Adds Better RTL Constructs to Verilog

- Traditional Verilog will allow writing code with functional errors
  - Allows engineers to model faulty behavior in order to prove a design will not work correctly
  - Puts a burden on Design Engineers to avoid dysfunctional code
  - Puts a burden on Verification Engineer to find dysfunctional code
- SystemVerilog adds constructs with built-in error checking!
  - Self-checking RTL modeling blocks
  - Self-checking decision statements
  - Self-checking assignment statements
- *Using these constructs is like getting free assertions!*
  - Can detect and prevent many types of functional bugs before synthesis

Self-Checking RTL Modeling Blocks

- Verilog always procedures model all types of design logic
  - Synthesis must "*infer*" (guess) whether an engineer intended to have combinational, latched or sequential functionality

```
always @(mode)
  if (!mode)
    o1 = a + b;
  else
    o2 = a - b;
```

where did all these latches come from?

- SystemVerilog has hardware-specific always procedures: **`always_comb`**, **`always_latch`**, **`always_ff`**
  - Documents designer intent
  - Enforces several synthesis RTL rules
  - Synthesis can check against designer intent

Free, built-in code checking – I like this!

```
always_comb
  if (!mode)
    o1 = a + b;
  else
    o2 = a - b;
```

```
Warning: test.sv:5:
Netlist for always_comb
block contains a latch
```

Self-Checking Decision Statements

- Verilog only defines simulation semantics for decision statements
  - Evaluate sequentially; only the first matching branch is executed
- Specifying synthesis **`parallel_case`** and **`full_case`** pragmas causes gate-level optimizations
  - Evaluate decisions in parallel, do Karnaugh mapping, etc.

***WARNING*: These optimizations are NOT verified in simulation!**

- SystemVerilog adds **`unique`**, **`unique0`** and **`priority`** decisions
  - Enable synthesis parallel_case and/or full_case pragmas
  - Enable run-time simulation checking for when the decision might not work as expected if synthesized with the pragma

```
always_comb
  unique case (state)
    ...
  endcase
```

- **Will get simulation warnings if `state` matches multiple branches (not a valid parallel_case)**
- **Will get simulation warnings if `state` doesn't match any branch (not a valid full_case)**

Self-Checking Assignment Statements

Traditional Verilog

```
parameter [2:0]
  WAIT = 3'b001,
  LOAD = 3'b010,
  DONE = 3'b001;
parameter [1:0]
  READY = 3'b101,
  SET   = 3'b010,
  GO    = 3'b110;

reg [2:0] state, next_state;
reg [2:0] mode_control;

always @(posedge clk or negedge rstN)
  if (!resetN) state <= 0;
  else         state <= next_state;

always @(state)  // next state decoder
  case (state)
    WAIT : next_state = state + 1;
    LOAD : next_state = state + 1;
    DONE : next_state = state + 1;
  endcase

always @(state)  // output decoder
  case (state)
    WAIT : mode_control = READY;
    LOAD : mode_control = SET;
    DONE : mode_control = DONE;
  endcase
```

6 functional bugs (must detect, debug and fix)

SystemVerilog adds enumerated types

```
enum logic [2:0]
  {WAIT = 3'b001,
   LOAD = 3'b010,
   DONE = 3'b001}
  state, next_state;
enum logic [1:0]
  {READY = 3'b101,
   SET   = 3'b010,
   GO    = 3'b110}
  mode_control;

always_ff @(posedge clk or negedge rstN)
  if (!resetN) state <= 0;
  else         state <= next_state;

always_comb  // next state decoder
  case (state)
    WAIT : next_state = state + 1;
    LOAD : next_state = state + 1;
    DONE : next_state = state + 1;
  endcase

always_comb  // output decoder
  case (state)
    WAIT : mode_control = READY;
    LOAD : mode_control = SET;
    DONE : mode_control = DONE;
  endcase
```

7 syntax errors (compiler finds all the bugs)

Part Four

Simulation and Synthesis Support for SystemVerilog Assertions

Simulation and Synthesis Support for Assertions

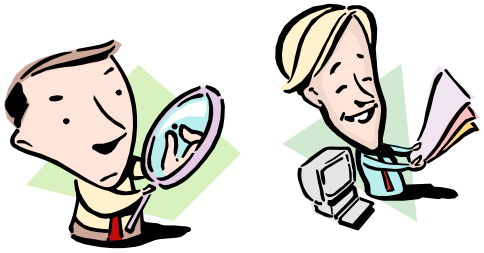
- Simulation should execute assertions; Synthesis should ignore

Assertion Construct	VCS v. 2014.12	Design Compiler v. 2014.09-SP5	Synplify-Pro v. 2014.09-SP2
Embedded Immediate Assertions	✓	✓	✓
Embedded Concurrent Assertions	✓	✓	✓
Property Blocks	✓	✓	✓
Sequence Blocks	✓	✓	✓
Disable Assertion During Reset	✓	✓	✓
Deferred Immediate Assertions	✓	✗	✗
Let Statements	✓	✓	✗
Checker Statement	✓	✗	✗
Validate Reset Example	✓	✓	✓
Validate Value Range Example	✓	✓	✓
Validate Pulse Width Example	✓	✓	✓
Validate Parameters	✗	✗	✗
always_comb with Latch Logic	✓	✓	✓
Enumerated Types with Faulty Logic	✓	✓	✓

Summary

- SystemVerilog Assertions really do work!
  - An effective way to verify many aspects of design functionality
  - Find errors that functional verification might miss
- RTL Design Engineers should embed assertions that validate assumptions directly into RTL code as the code is being written
  - Embed relatively simple immediate and concurrent assertions
  - Use RTL modeling constructs with built-in assertion-like checking
  - Synthesis compilers properly ignore embedded assertions
- There are big advantages to RTL designers specifying assertions
  - Validate assumptions on which the RTL model depends
  - Localize where functional problem occurred
  - Clarify specification ambiguities
  - Help to avoid RTL modeling gotchas

Question? Comments?



Once upon a time...

Four engineers worked on an important design. Their names were: Tom Somebody, Dick Everybody, Harry Anybody, and Sally Nobody.

Each engineer was responsible to design and verify a sub block of the design. Everybody had attended Sutherland HDL's SystemVerilog Assertions training, and was sure that Somebody would write assertions to verify that the full design worked correctly. Anybody could have written them, but Nobody did it.

When the design was implemented in silicon, it did not work according to the specification. Everybody blamed Somebody because Nobody did what Anybody could have done.

SUTHERLAND HDL
Training engineers to be SystemVerilog wizards
www.sutherland-hdl.com

snug 25
Synopsys Users Group

Thank You