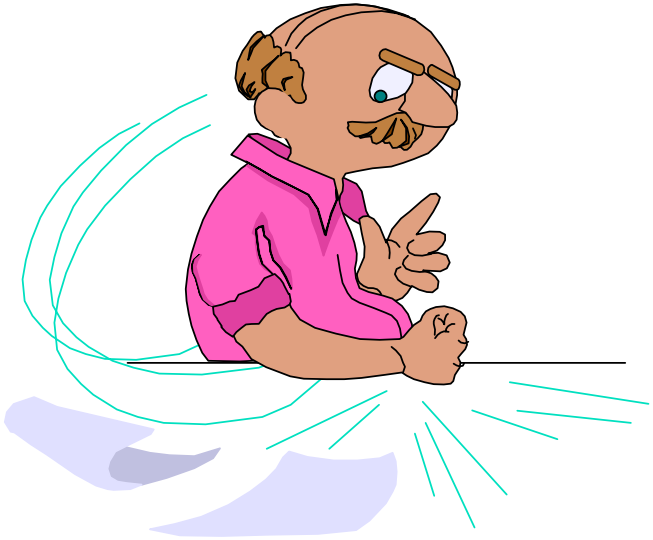
SNUG SAN JOSE 2006



SystemVerilog Assertions Are For Design Engineers, Too!

Don Mills
LCDM Engineering
Salt Lake City, Utah
mills@lcdm-eng.com

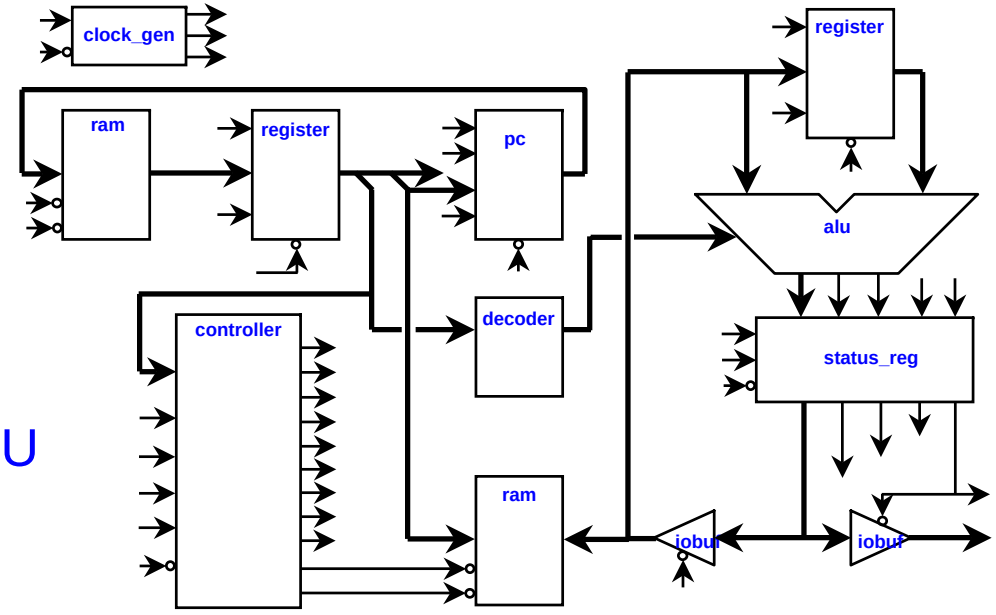
Stuart Sutherland
Sutherland HDL, Inc.
Portland, Oregon
stuart@sutherland-hdl.com

Presentation Overview

- Why engineers don't do assertions
  - Excuses, excuses, excuses
- SystemVerilog Assertions overview
  - Immediate assertions
  - Concurrent assertions
- Where assertions should be specified
  - Where to put the assertion code
  - Who should write the assertions
  - Developing an assertions test plan
- Assertions for Design Engineers
  - Verifying design assumptions
  - Examples, examples, examples
- Lessons Learned

Case Study: Assertions for a Small DSP

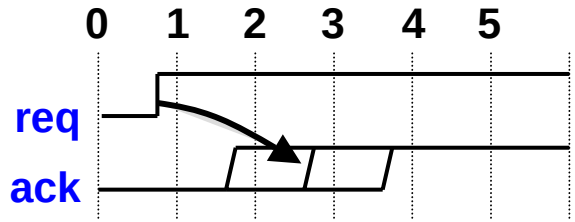
- A small Digital Signal Processor (DSP) design is used in this presentation to illustrate how to use SystemVerilog Assertions
- The DSP contains…
  - A clock generator and reset synchronizer
  - A state machine
  - Several registers
  - A program counter
  - Combinatorial decoder and ALU
  - Program and data memories
  - A tri-state data bus



- The DSP is used as a training lab in Sutherland HDL courses
  - Synthesis students get to model the DSP as a final project
  - Assertion students get to add assertions to the DSP and testbench

Assertion-like Checks Can Be Written in Plain Verilog

- An assertion is a design condition that must be true
  - Assertions can be written in Verilog, but…It's a lot of extra code!



Each request must be followed by an acknowledge within 1 to 3 clock cycles

To test for a sequence of events requires several lines of Verilog code

- Difficult to write, read and maintain
- Cannot easily be turned off during reset or other don't care times

```
always @(posedge req) begin
  @(posedge clk) ; // synch to clock
  fork: watch_for_ack
    parameter N = 3;
    begin: cycle_counter
      repeat (N) @(posedge clk);
      $display("Assertion Failure", $time);
      disable check_ack;
    end // cycle_counter
    begin: check_ack
      @(posedge ack)
      $display("Assertion Success", $time);
      disable cycle_counter;
    end // check_ack
  join: watch_for_ack
end
```

Verilog-based Checker's Must be Hidden from Synthesis

- A checking function written in Verilog looks like RTL code
  - Synthesis compilers cannot distinguish the hardware model from the embedded checker code
  - To hide Verilog checker code from synthesis compilers, extra synthesis pragma's must be added to the code

```
if (if_condition)
   // do true statements
else
//synthesis translate_off
if (!if_condition)
//synthesis translate_on
   // do the not true statements
//synthesis translate_off
else
   $display("if condition tested either an X or Z");
//synthesis translate_on
```

RTL code

checker code

RTL code

checker code

How many engineer's will go to this much extra effort to add embedded checking to an if…else RTL statement?

Obstacles to Writing Self-Checking RTL Code

LCDM Engineering & SUTHERLAND HDL
training engineers to be Verilog and SystemVerilog wizards

- There are a number of reasons design engineers are reluctant to embed self-checking code within synthesizable RTL models…
  - ☹ Not enough time in the design schedule
  - ☹ Makes the RTL code hard to read
  - ☹ Not synthesizable
  - ☹ Might inadvertently change design behavior
  - ☹ Will slow down simulation
  - ☹ It's the verification team's job

These Are Valid Concerns!

SystemVerilog Assertions overcome all of these obstacles!

Advantages of SystemVerilog Assertions

- SystemVerilog Assertions have several advantages over coding assertion checks in Verilog…
  - Concise syntax!
    - Dozens of lines of Verilog code can be represented in one line of SVA code
  - Ignored by Synthesis!
    - Don't have to hide Verilog checker code within convoluted translate_off / translate_on synthesis pragmas
  - Can be disabled!
    - SystemVerilog assertions can be turned off during reset, or until simulation reaches a specific simulation time or logic state
  - Can have severity levels!
    - SystemVerilog assertion failures can be non-fatal or fatal errors
    - Simulators can enable/disable failure messages based on severity

SystemVerilog Has Two Types of Assertions

- Immediate assertions test for a condition at the current time

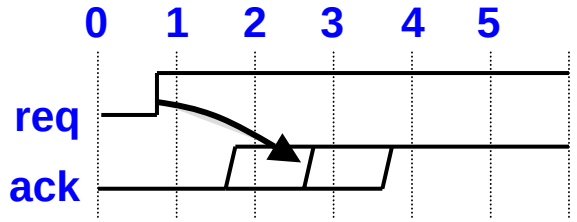
```
always @(state)
  assert ($onehot(state)) else $fatal;
```

generate a fatal error state variable is not a one-hot value

An immediate assertion is the same as an `if...else` statement, but with assertion controls

- Concurrent assertions test for a sequence of events spread over multiple clock cycles

a complex sequence can be defined in very concise code




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

One line of SVA code replaces all the Verilog code in the example three slides back!

Assertion Severity Levels

- The severity of an assertion failure behavior can be specified
  - `$fatal [ ( finish_number, "message", message_arguments ) ] ;`
    - Terminates execution of the tool
    - **finish_number** is **0**, **1** or **2**, and controls the information printed by the tool upon exit (the same levels as with $finish)
  - `$error [ ( "message", message_arguments ) ] ;`
    - A run-time error severity; software continues execution
  - `$warning [ ( "message", message_arguments ) ] ;`
    - A run-time warning; software continues execution
  - `$info [ ( "message", message_arguments ) ] ;`
    - No severity; just print the message

Software tools may provide options to suppress errors or warnings or both

```
always @(negedge reset)
  assert (state == LOAD)
  else $fatal(0,"FSM %m behaved badly at %d", $time);
```

```
always @(negedge reset)
  assert (state == LOAD) else $warning;
```

Expression Sequences and the ## Cycle Delay

- A sequence is a series of true/false expressions spread over one or more clock cycles
- ## represents a ***"cycle delay"***
  - Specifies the number of *clock cycles* to wait until the next expression in the sequence is evaluated

```
property p_request_grant;
  @(posedge clock) request ##1 grant ##1 !request ##1 !grant;
endproperty

ap_request_grant : assert property (p_request_grant); else $fatal;
```

"@(posedge clock)" is not a delay, it specifies what ## represents

- request must be followed one clock cycle later by grant
- grant must followed one clock cycle later by !request
- !request must be followed one clock cycle later by !grant

SystemVerilog Design Constructs With Built-in Assertions

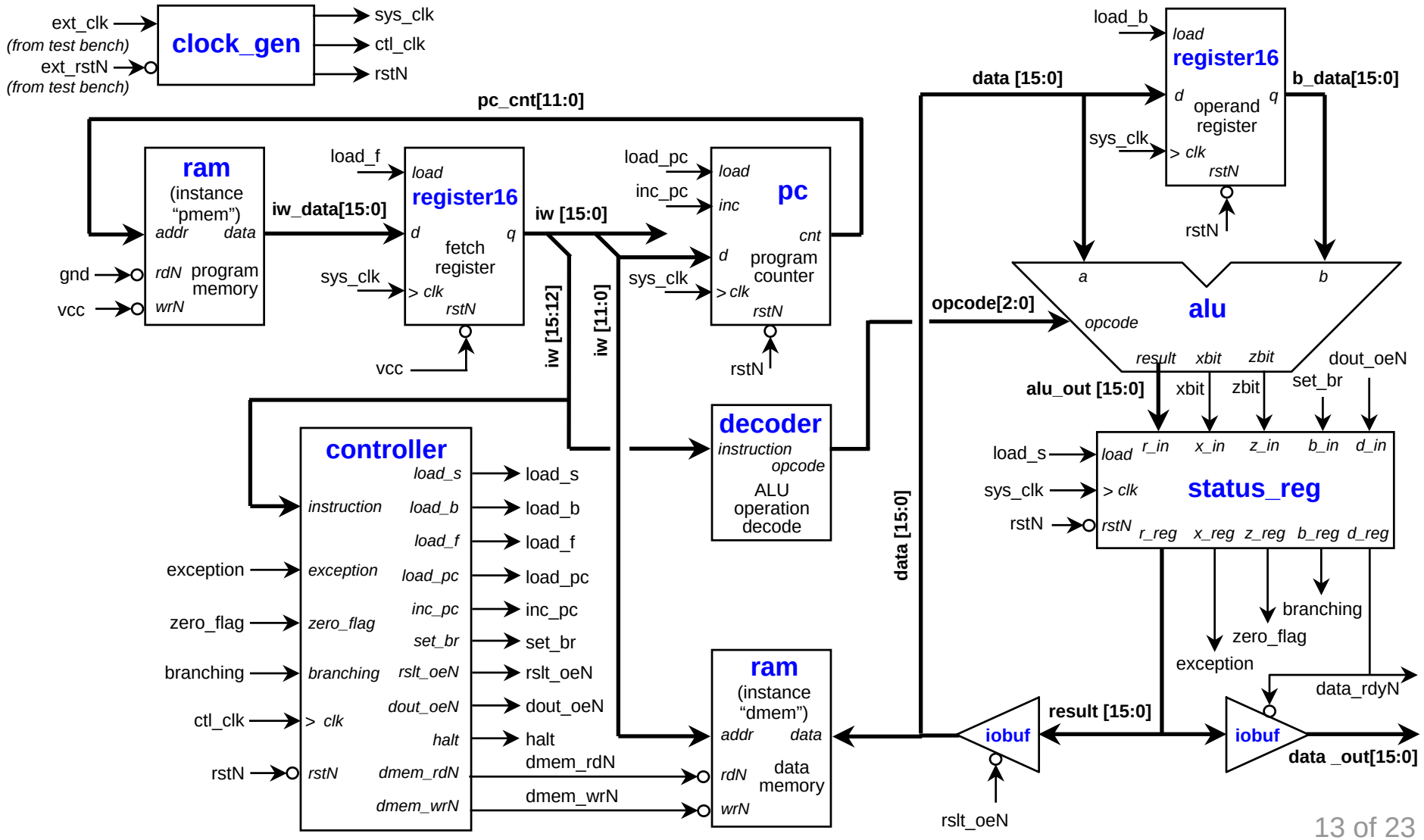
- Some SystemVerilog constructs have built-in assertion-like checking!
- **`always_comb`** / **`always_ff`**
  - Allows tools to check that procedural code matches intent
  - Check that procedural block variables are not written to elsewhere
- **`unique case`** / **`unique if…else`**
  - Check that decision statements are fully specified
  - Check that decision branches are mutually exclusive
- Enumerated variables
  - Check that assignments are within the legal set of values
  - Check that two or more labels (e.g. state names) do not have the same value

By using these constructs, designer's get the advantages of self-checking, without having to write assertion statements!

What's Next

- ✓ Why engineers don't do assertions
  - Excuses, excuses, excuses
- ✓ SystemVerilog Assertions overview
  - Immediate assertions
  - Concurrent assertions
- ❑ Where assertions should be specified
  - Where to put the assertion code
  - Who should write the assertions
  - Developing an assertions test plan
- ❑ Assertions for Design Engineers
  - Verifying design assumptions
  - Examples, examples, examples

Assertions for the DSP Example



Where Assertions Can be Specified

Assertion Based Verification should take advantage of all of these capabilities

- SystemVerilog Assertions can be…
  - Embedded in the RTL code
    - Executes as a programming statement, in-line with the RTL procedural code
    - Will be ignored by synthesis
  - In the design model, as a separate, concurrent block of code
    - Executes in parallel with the design code throughout simulation
    - Will be ignored by synthesis
  - External to the design model, in a separate file
    - Can be bound to specific instances of design models
    - Executes in parallel with the design code throughout simulation
    - Allows verification engineers to add assertions to the design without actually modifying the design code
    - Synthesis never sees the assertion code

Guideline!

- Design engineers should write assertions to verify assumptions that affect the functionality of a design block
  - The assertion documents the designer's assumptions
  - Example: The ALU block assumes that the A, B and opcode inputs will never have a logic X or Z value
- Verification engineers should write assertions that verify design functionality meets the overall design specification
  - The assertion verifies that the designer correctly implemented the specification
  - Example: The zero flag output of the ALU block should always be set if the ALU result output is zero

Developing An Assertions Test Plan

- Before writing assertions, you need an ***"Assertions Test Plan"***
  - Specifies what functionality needs to be verified with assertions
  - What type of assertion is needed for each test
    - Immediate or concurrent?
  - Where the assertion should be placed
    - Embedded in the design?
    - At the system interconnect level?
    - Bound into the design?
  - Which team is responsible for writing each assertion
    - The verification team?
    - The design team?

The Assertions Test Plan should be developed *before* any design code is written!

An Assertions Test Plan Example

- The assertions test plan defines…
  - What to verify
  - Which team is responsible for writing the assertion
- Example: Program Counter assertions test plan

Functionality to Verify	Assigned To
load and increment are mutually exclusive	design team
If increment, then d input never has any X or Z bits	design team
If !load and !increment, then on posedge of clock, pc does not change (must allow for clock-to-q delay)	verification team
If increment, then pc increments by 1 (must allow for clock-to-q delay)	verification team
If load, then pc == d input (must allow for clock-to-q delay)	verification team

Each design block has a similar assertions test plan

Assertion Plan Example 1: Assertions on ALU Inputs

- ALU ***design engineer*** assertions example

Functionality to Verify	Assigned To
After reset, the A, input never have any X or Z bits	design team
After reset, the B input never have any X or Z bits	design team
After reset, the opcode input never have any X or Z bits	design team
All instructions are decoded	design team
…	

```
always_comb begin
  // Check that inputs meet design assumptions (no X or Z bits)
  ai_a_never_x:   assert (^a !== 1'bx);
  ai_b_never_x:   assert (^b !== 1'bx);
  ai_opc_never_x: assert (^opcode !== 1'bx);

  unique case (opcode)  // "unique" verifies all opcodes are decoded
    ...  // decode and execute operations
  endcase
end
```

Design engineer assertions are simple to add, and can greatly reduce hard-to-find errors!

Assertion Plan Example 2: State Machine Assertions

- FSM ***design engineer*** assertions example

Functionality to Verify	Assigned To
State is always one-hot	design team
If !resetN (active low), state RESET	design team
If in DECODE state, prior state was RESET or STORE	design team

```
property p_fsm_onehot; // FSM state should always be one-hot
  @(posedge clk) disable iff (!rstN) $onehot(state);
endproperty
ap_fsm_onehot: assert property (p_fsm_onehot);

property p_fsm_reset; // verify asynchronous reset to RESET state
  @(posedge clk) !rstN |-> state == RESET;
endproperty
ap_fsm_reset: assert property (p_fsm_reset);

property p_fsm_decode_entry; // verify how DECODE state was entered
  @(posedge clk) disable iff (!rstN) state == DECODE |->
    $past(state) == RESET || $past(state) == STORE;
endproperty
ap_fsm_decode_entry: assert property (p_fsm_decode_entry);
```

Concurrent assertions can be used to verify coverage too!

Lessons Learned…

- Teaching engineers how to add assertions to the DSP training lab has shown that…
  - It takes time to write the assertions test plan
    - It is not a trivial task, but it is critical to successfully using SVA!
  - The assertion test plan helps identify similar assertions
    - Can write an assertion once, and use it in several places
  - Assertions should not just duplicate the RTL code
    - Engineers need to learn to think differently
  - Disabling assertions during reset is important
    - Hundreds of false assertion failures occur in the DSP during reset
  - The test plan needs to be flexible
    - Some times the responsibility for which team should write the assertion needs to change

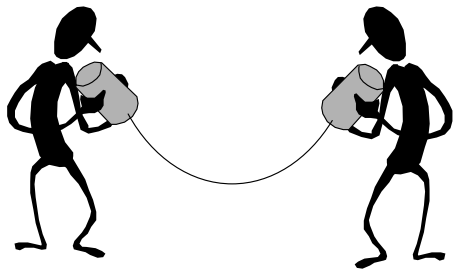
More Lessons Learned...

LCDM Engineering & SUTHERLAND HDL
training engineers to be Verilog and SystemVerilog wizards

- Assertions may require different design partitioning
  - The DSP ALU block is difficult to check with concurrent assertions because it is pure combinational logic (no clock)
  - Better design partitioning would put the ALU and its input and output registers into one design block
- Enumerated type definitions should be defined globally
  - The DSP state machine uses a local enumerated variable for the state names
  - Assertions written external to the state machine cannot access those enumerated names
- Enumerated types should have explicit values defined for each label
  - After synthesis, labels disappear and only logic values exist
  - Assertions become invalid if the label does not have an explicit value

Summary

- SystemVerilog Assertions enable true assertions based verification
  - Integrated into the Verilog/SystemVerilog language
    - Don't have to hide assertions in comments
    - Assertions have full visibility to all design code
    - Execution order is defined within simulation event scheduling
  - Easy to write (compared to other assertion solutions)
    - Immediate and concurrent assertions
    - A concise, powerful sequential description language
    - Sequence building blocks for creating complex sequences
  - Binding allows verification engineers to add assertions to a design without touching the design files
- SystemVerilog assertions are a team effort
  - Some assertions written by the design team
  - Some assertions written by the verification team



Questions & Answers...

LCDM Engineering & SUTHERLAND HDL
training engineers to be Verilog and SystemVerilog wizards

What will assertions reveal about my design?