DV club
Austin TX
September 9, 2015

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

A copy of these presentation slides is available at
sutherland-hdl.com/papers.html

SystemVerilog Assertions and Assertion Planning

a tutorial on
Writing SystemVerilog Assertions
and Planning Where to Use Assertions

Stu Sutherland
Sutherland HDL, Inc.

Agenda

Part 1: A short tutorial on SystemVerilog Assertions

Part 2: Who should write assertions?

Part 3: Planning where to use assertions

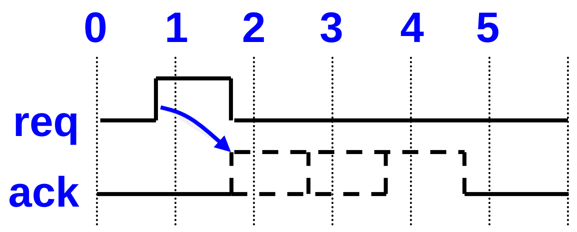
The goal of this tutorial is to encourage both verification engineers and design engineers to take advantage of SystemVerilog Assertions!

Part One:

A Short Tutorial On SystemVerilog Assertions

What Is An Assertion?

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



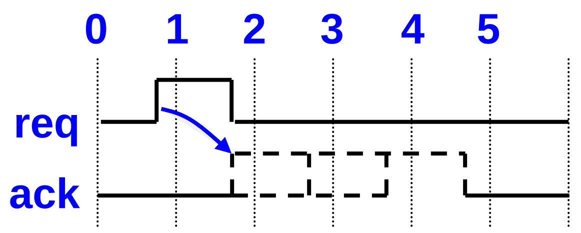
Design Specification:
After the request signal is asserted, the acknowledge signal must arrive 1 to 3 clocks later

- Assertions are used to:
  - Document design intent (e.g.: every request has an acknowledge)
  - Verify design meets the specification over simulation time
  - Verify design assumptions (e.g.: state value is one-hot)
  - Localize where failures occur in the design instead of at the output
  - Provide semantics for formal verification
  - Describe functional coverage points
  - And… requires clarifying ambiguities in spec

Which one of these is the most important to you in your projects?

Verification Without Assertions

- Before there were special assertion languages like SVA, verification checks had to be coded with programming statements



Design Specification:

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

```
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
```

To test for a sequence of events requires many lines of Verilog code (hard to write)

- With SVA, this check can be done with one line of code!

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

Embedded Verification Checking and Synthesis

- Checking code written in Verilog must be hidden from Synthesis
  - Synthesis cannot distinguish the hardware model from the embedded checker code

```
if (critical_condition)                          } RTL code
   // do true statements
else
//synthesis translate_off                        } checker code
if (critical_condition == 0)
//synthesis translate_on
   // do the not true statements                 } RTL code
//synthesis translate_off                        } checker code
else
   $display("critical_condition is bad");
//synthesis translate_on
```

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

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(critical_condition));
if (mode) ...   // do true statements
else      ...   // do not true statements
```

`assert` is ignored by synthesis and can be disabled during simulation

SystemVerilog Has Two Types of Assertions

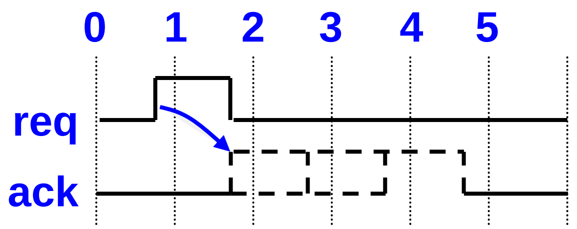
- Immediate assertions test for a condition at the current time

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

generate a fatal error if 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



multi-clock “sequences” can be defined with very concise code

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

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

concurrent assertions run as a background process in parallel with the RTL code

Concurrent assertions have an extensive set of operators to describe complex design conditions

Assertion Severity System Tasks

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

- The assertion messages can be printed with severity levels
  - `$info ( "message", message_arguments ) ;`
    - No severity; just print the message
  - `$warning ( "message", message_arguments ) ;`
    - A run-time warning; software continues execution
  - `$error ( "message", message_arguments ) ;`
    - A run-time error severity; software continues execution
  - `$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 tool controls as with $finish)

If a severity level is not specified, assertion messages default to an error level

- **The user-supplied message is appended to a tool-specific message containing the source file location and simulation time**
- **The message is optional; if not specified the tool-specific message will still be printed**

Assertions in a UVM testbench should use the UVM message functions, such as uvm_report_warning and uvm_report_error, so that the messages are tracked by UVM

Assertion Pass/Fail Actions

- Assertions can have both pass and fail “actions”

```
assert property (p_req_gnt)
  $info("Handshaking passed in %m");
else
  $error("Handshaking failed in %m");
```

User-defined pass/fail statements can do anything you want

(Tip: use %m to display the hierarchy path to the statement containing message)

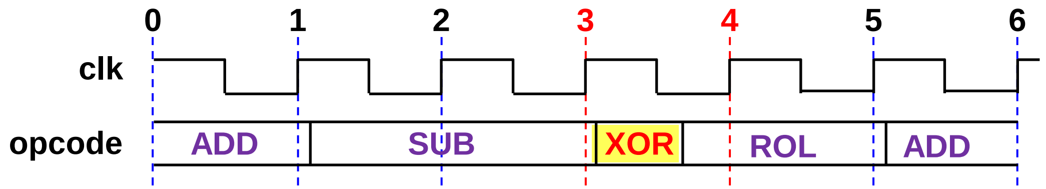
- The pass and fail statements can be any procedural statement
  - Multiple statements must be grouped with **begin**…**end**
- The pass action is optional
  - If left off, then no action is taken when the assertion passes
- The fail action is also optional
  - If left off, then a default, tool-generated message is printed with an error message severity level

```
always @(negedge reset)
  a_fsm_reset: assert (state == LOAD);
```

No action if pass, default message if fail

Detecting Glitches with 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.

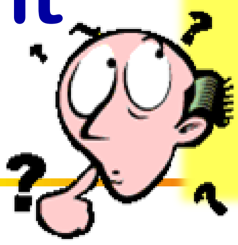
This glitch within a clock cycle will never be stored in my registers – I can ignore it.

You need an ***immediate assertion!***

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**

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 use binding (later slide)
  - 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 later slide)
  - Works with simulation and formal verification
- Cons:
  - More difficult to define (and debug)
  - Can be far from code being checked
  - Cannot detect glitches

Concurrent Assertions are Cycle Based

- 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
    - The first expression is evaluated immediately
    - Subsequent expressions are evaluated at later clock cycles

```
property p_request_grant;
  @(posedge clock)
  request ##1 grant ##[1:3] !request && !grant;
endproperty

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

"`@(posedge clock)`" is not a delay, it specifies what a cycle is for this property

Delay 1 cycle

Delay for a range of cycles

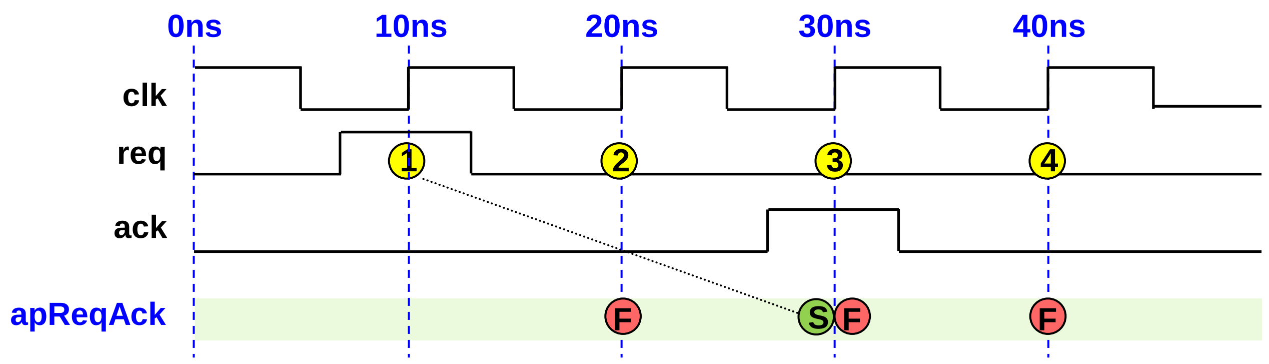
- `request` **must be followed one clock cycle later by** `grant`
- `grant` **must followed one to three clock cycles later by** `!request` **and** `!grant`

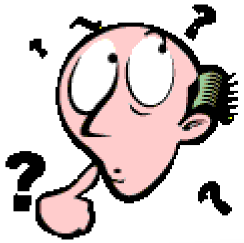
SUTHERLAND HDL
training engineers to be SystemVerilog wizards
sutherland-hdl.com

A Concurrent Assertion Gotcha

- Concurrent assertions start a new thread every clock cycle

```
property pReqAck;
  @(posedge clk)
  req ##2 ack;
endproperty: p_req_ack
apReqAck: assert property (pReqAck);
```





The design is working... So why do I get 3 assertion failures?

An evaluation thread starts at time 10

Request is true so the thread continues, and at time 30 this thread succeeds

New evaluation threads start at time 20, 30 and 40

Request is false so each of these threads fails at those times

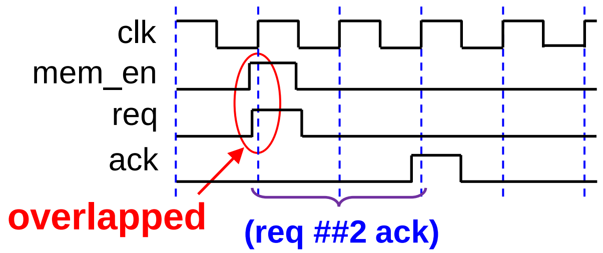
The fix for this gotcha is something called an implication operator (see next page)

Conditioning Sequences Using Implication Operators

- A sequence can be conditioned with an implication operator
  - If the condition is true, the sequence is evaluated
  - If the condition is false, the sequence is not evaluated (a don't care)

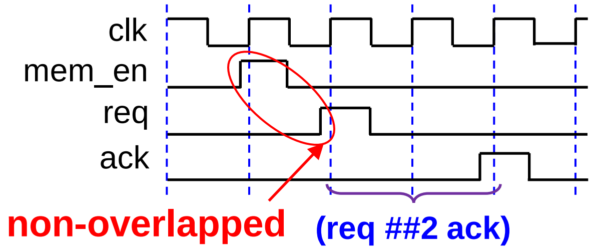
- |-> overlapped implication: sequence evaluation starts immediately

```
property p_req_ack;
  @(posedge clk)
  mem_en |-> req ##2 ack;
endproperty: p_req_ack
```



- |=> non-overlapped implication:
  sequence evaluation starts at the next clock

```
property p_req_ack;
  @(posedge clk)
  mem_en |=> req ##2 ack;
endproperty: p_req_ack
```



The |=> (non-overlapped implication) is the same as |-> ##1 (overlapped plus 1 cycle)

Implication Terminology

- **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** — if the antecedent is false, the property is considered “vacuously true”
  - The check is not of interest, so evaluation is aborted without considering it a failure

“antecedent” (or cause) — if FALSE, the property succeeds vacuously; if TRUE, then the sequence continues

```
property p_handshake;
  @(posedge clk)
   request  |=> acknowledge ##1 data_enable ##2 done;
endproperty: p_handshake
```

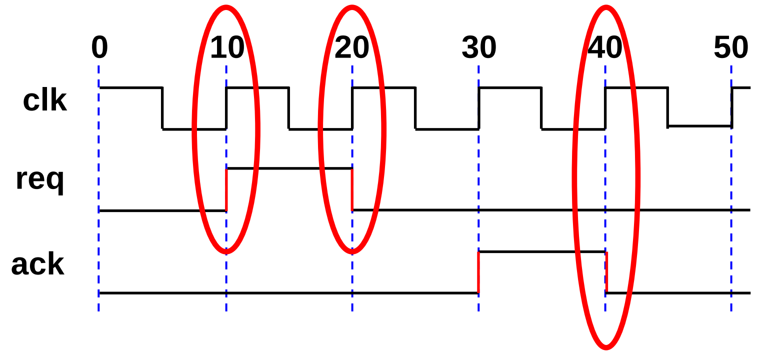
“implication operator”

“consequent” (or effect) — only evaluated if antecedent succeeds

Using Concurrent Assertions with Zero-delay RTL Models

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

- In RTL models, most signals change on a clock edge
  - RTL models are typically modeled with zero delay
  - Register outputs change immediately on the clock edge



```
property p_req_ack;
  @(posedge clk)
  req |-> ##2 ack ##1 !ack;
endproperty: p_req_ack

ap_req_ack: assert property (p_req_ack);
```

- ❑ Is req 0 or 1 at time 10? **0**
- ❑ What about at time 20? **1**
- ❑ Is ack 0 or 1 at time 40 **1**
- ❑ At what time does this assertion pass or fail? **50**

Concurrent assertions sample values in a *"Preponed event region"* – the assertion always sees the value that existed before the clock edge causes any changes

Quiz: Does This Assertion Match the Design Specification?

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

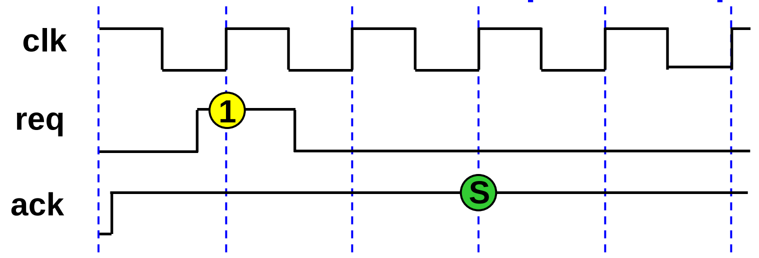
- Given the following assertion:

```
property p_req_ack;
 @(posedge clk) req |-> ##2 ack;
endproperty: p_req_ack

ap_req_ack: assert property (p_req_ack);
```

If there is an ack, then req must be true 2 clock cycles later

- Will this event sequence pass or fail?



The assertion will pass — it checks that ack is true on the 2nd clock after req; it does not check for *when* ack transitioned to true

- Is this a faulty design or a faulty assertion?
  - It depends on the design specification!

Spec 1: A req (request) should be followed two cycles later by ack (acknowledge). The ack line can be tied high so that all requests are automatically acknowledged.

Spec 2: A req (request) should be followed two cycles later by a rising edge of ack (acknowledge). The ack is only allowed to be high for one clock cycle.

Value Change Functions

- Special functions test for a value change between adjacent clock cycles:

```
$rose(expression, cycle_definition);
```
  - Returns true if the LSB of the expression changed to 1

```
$fell(expression, cycle_definition);
```
  - Returns true if the LSB of the expression changed to 0

```
$changed(expression, cycle_definition);
```
  - Returns true if the value of the expression changed

```
$stable(expression, cycle_definition);
```
  - Returns true if the value of the expression did not change

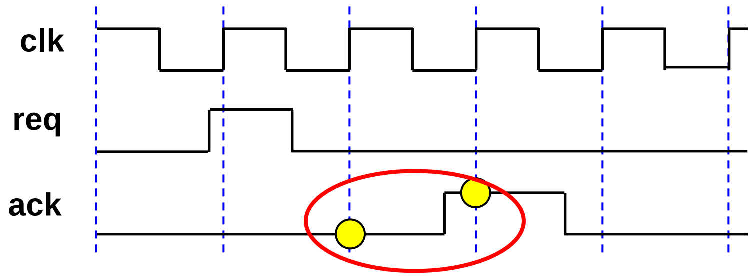
These functions evaluate sampled values – they compare the value that was sampled at the beginning of the previous clock cycle with the value that was sampled at the beginning of the current clock cycle

The *cycle_definition* is optional and seldom needed (see notes below); It specifies what clock to use for sampling the expression (e.g.: `$rose(ack, @posedge master_clk)`)

Design Spec: A req (request) should be followed two cycles later by a rising edge of ack (acknowledge). The ack is only allowed to be high for one clock cycle.

```
property p_req_ack;
  @(posedge clk)
  req |-> ##2 $rose(ack) ##1 !ack;
endproperty: p_req_ack

ap_req_ack: assert property (p_req_ack);
```



Assertions and Coverage

- Coverage is a metric for evaluating the progress of verification
  - Measures the completeness of the verification tests
- SystemVerilog provides two types of coverage
  - Functional coverage reports how many times specific values occur
    - Uses `covergroup` definitions
    - Checks current values at specific times during simulation
  - State coverage reports how many times assertions evaluate
    - Uses an assertion `cover` statement
    - Checks for sequences of conditions over multiple clock cycles

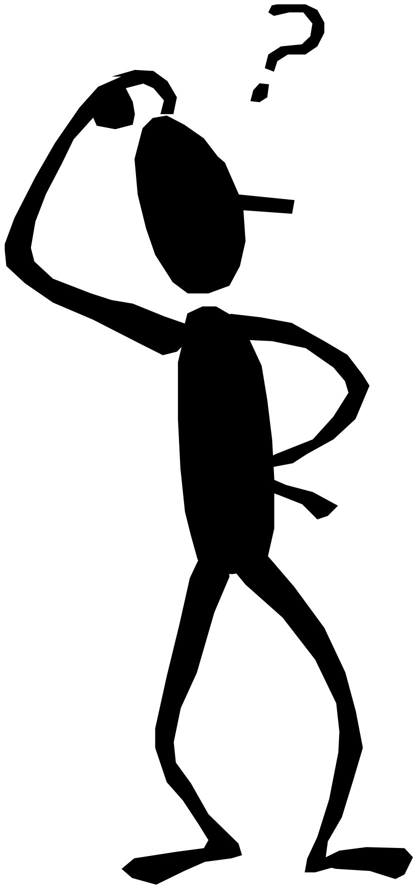
Did I test the FIFO operating at a nearly full condition?

Design Specification:

The FIFO `almost_full` flag is set when there are 2 locations remaining in the FIFO

```
caFifoNearMax: cover sequence
  (@(posedge clk)
  almost_full ##1 push ##1 push);
```

Assertion coverage is discussed in more detail later in the course

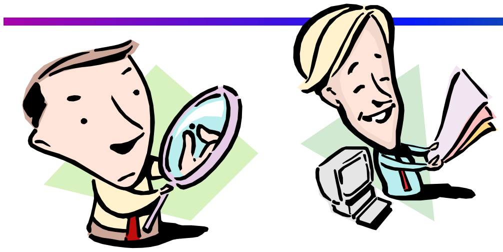


Part Two:

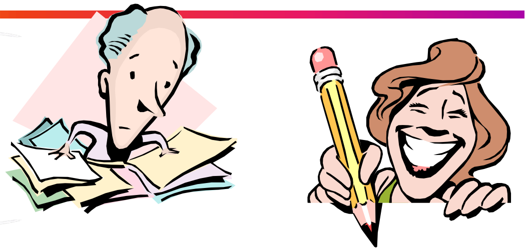
Who Should Write Assertions

Which Engineer Writes the Assertions for Your Projects?

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



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 training on SystemVerilog Assertions, and was sure that **Somebody** would write assertions to verify that the full design worked correctly. **Anybody** could have written them, but it ended up that **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.

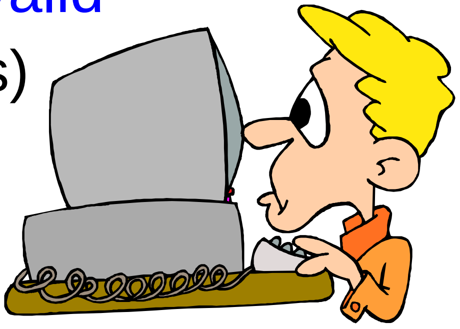
Guideline for Who Writes Assertions!

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

- **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 only be set if the ALU result output is zero
- **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

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 paper

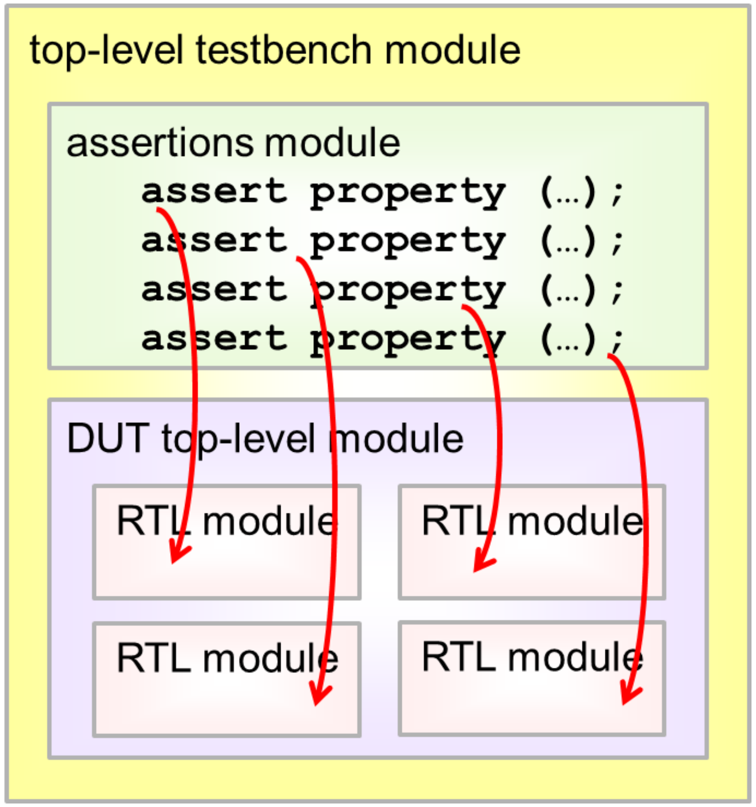
"Who Put Assertions In My RTL Code? And Why"

has examples of these types of assertions

(available to download from sutherland-hdl.com)

Verification Engineers Can Bind Assertions to Design Blocks

- 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

When To Embed Assertions, When To Bind In Assertions

Sutherland HDL 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!

Embedded versus Bound Assertions – Pros and Cons

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

SystemVerilog Design Constructs with Built-in Assertion Behavior

- 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
- `enum` Enumerated variables
  - Check that assignments are within the legal set of valu
  - Check that two or more labels (e.g. state names) do nc have the same value

By using these constructs, designer's get the advantages of self-checking code, without writing assertion statements!

Part Three:

Planning Where to Use Assertions

Developing An Assertions Test Plan

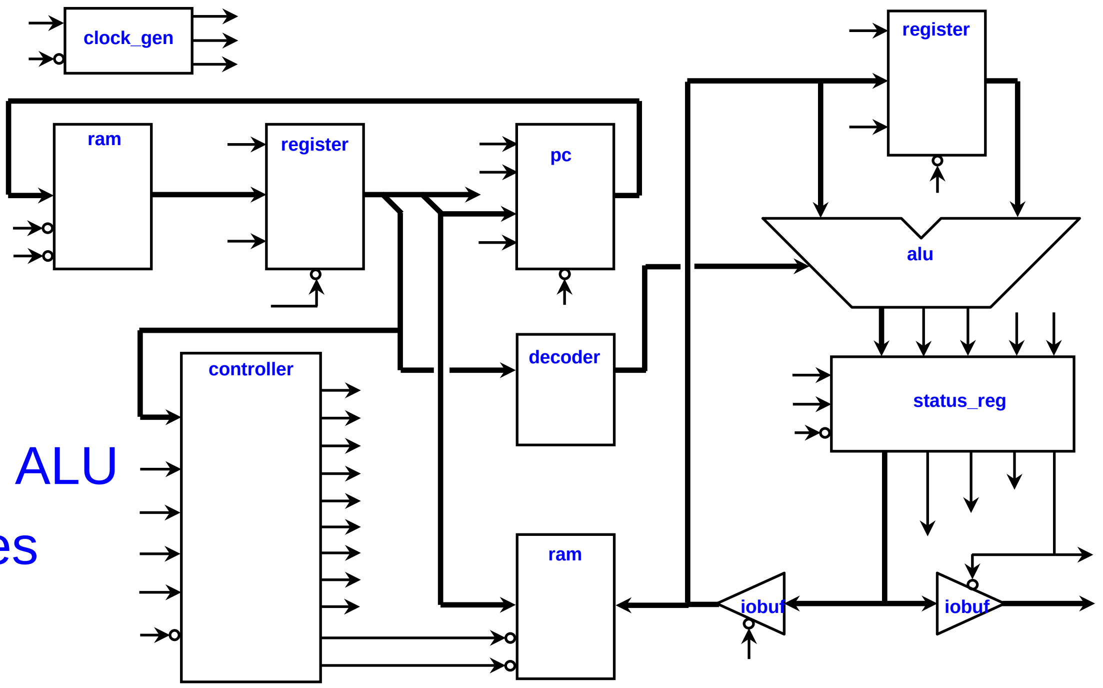
- Part of a Verification Test Plan is an ***"Assertions Test Plan"***
  - Specifies what functionality needs to be verified with assertions
  - What type of assertion is needed for each test
    - Immediate or concurrent?   Invariant, sequential or eventuality?
  - Where the assertion should be placed
    - 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?

Assertion Based Verification should take advantage of all of these capabilities

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

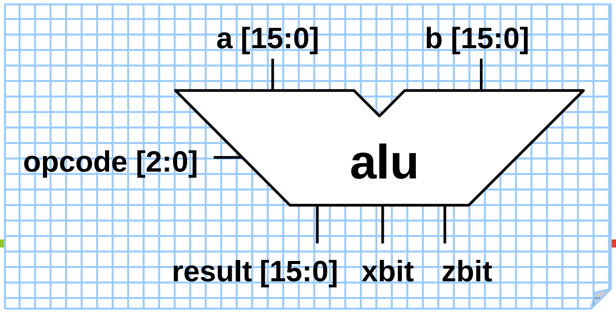
Case Study: Assertions for a Small DSP

- 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 has many places where assertions can be — and should be —used!

Assertions Test Plan: ALU



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

Functionality to Verify	Assertion Type	Assigned To
The **a**, **b** and **opcode** inputs never have any bits that are X or Z	immediate	design team
All **opcode** values are decoded	unique case	design team
The **zbit** output must be set whenever **result** output is 0	concurrent	verification team
... What other assertions are needed? (Answering this question is part of the final project…)		

```
module alu (...);
  always_comb begin
    ai_a_never_x: assert (!$isunknown(a));
    ...
    unique case (opcode)
      ...  // decode operations
    endcase
  end
```

Check that inputs meet design assumptions

“unique” verifies that all opcode values that occur are decoded

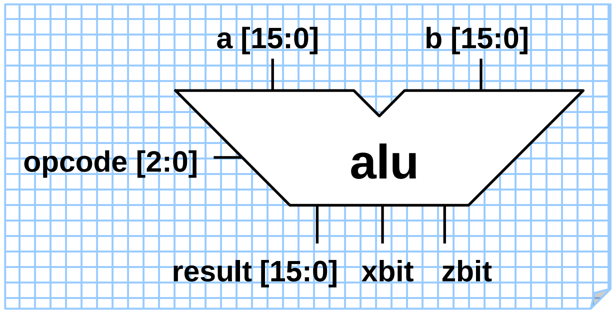
Be careful with using unique case – it also affects synthesis!

```
module assertions (...);  // to be bound to dsp top-level netlist
  property p_zbit_set_on_zero (clk);
    @(posedge clk) (alu_out == 0) |-> zbit;
  endproperty
  p_zbit_set_on_zero: assert property (p_zbit_set_on_zero(sys_clk));
```

On any cycle that alu_out is zero, zbit must be set

Exercise: Assertions Test Plan for the ALU

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

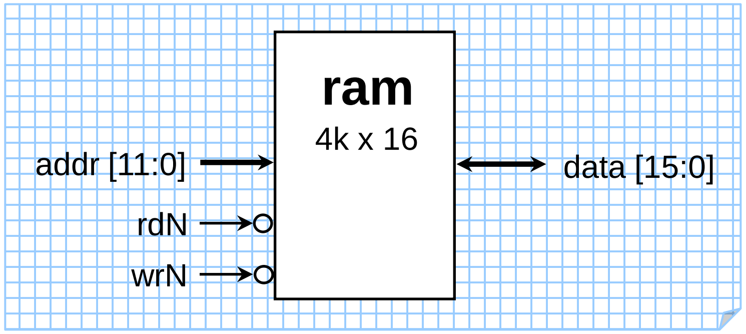


ASSIGNMENT:
Plan some assertions for the DSP blocks on <u>this page and the next 3 pages</u>. We will then discuss the test plan ideas as a class

- A combinational logic Arithmetic Logic Unit, that can perform 8 different types of operations (see notes section, below)

Functionality to Verify	Assertion Type	Assigned To
The **a**, **b** and **opcode** inputs never have any bits that are X or Z	immediate	design team
All **opcode** values are decoded	unique case	design team
After reset, **result** should never have an X or Z value	concurrent	verification team
zbit must be set if **result** is 0; and cleared if **result** is not 0	concurrent	verification team
xbit is set for Add operations if **a** + **b** is greater than (2**16)-1	concurrent	verification team
xbit is set for Multiply operation if **a** * **b** is greater than (2**16)-1	concurrent	verification team
xbit is set for Subtract operations if **a** is less than **b**	concurrent	verification team
xbit is never set for non arithmetic operations	concurrent	verification team

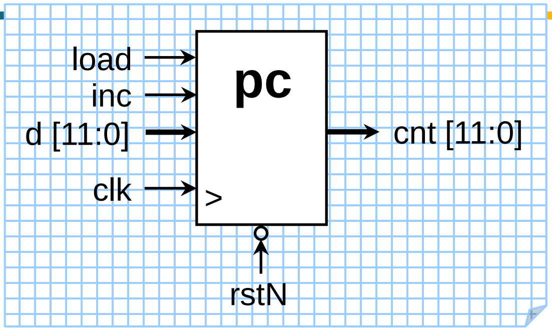
Exercise: Assertions Test Plan for the RAM



- A 4K by 16-bit asynchronous RAM with a bi-directional data port
  - Reading from the RAM occurs whenever **`rdN`** is low
  - Writing into the RAM occurs whenever **`wrN`** is low
  - Behavior is undefined if **`rdN`** and **`wrN`** are low at the same time

Functionality to Verify	Assertion Type	Assigned To
`!rdN` (read) and **`!wrN`** (write) are mutually exclusive	immediate	design team
`addr` never has X or Z bits when **`wrN`** is low (writing into RAM)	immediate	design team
`data` never has X or Z bits when **`wrN`** is low (writing into RAM)	immediate	design team
`addr` never has X or Z bits when **`rdN`** is low (reading from RAM)	immediate	design team
`data` never has X or Z bits when **`rdN`** is low (reading from RAM)	concurrent	verification team

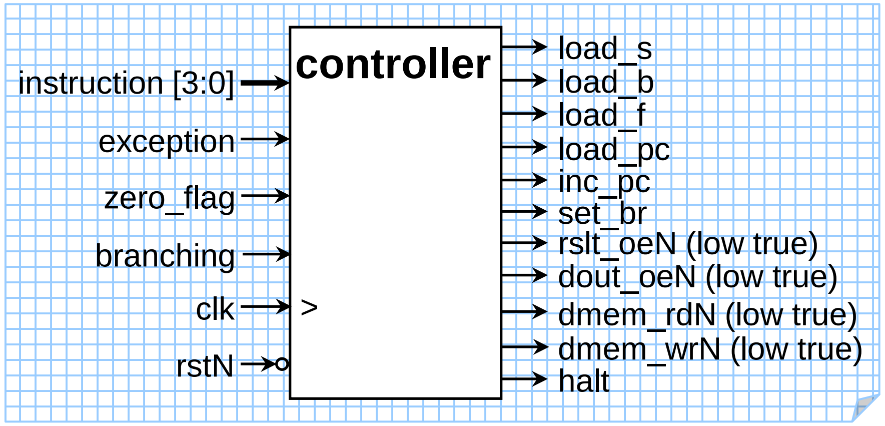
Exercise: Assertions Test Plan for the Program Counter



- The Program Counter is a sequential logic 12-bit counter
  - `rstN` is asynchronous — the counter is reset to zero
  - Counter loads with the value of the `d` input on the positive edge of `clk`, when `load` is high
  - Counter increments on the positive edge of `clk`, when `inc` is high

Functionality to Verify	Assertion Type	Assigned To
`load` and `inc` (increment) are mutually exclusive	immediate	design team
If `load` is high, `d` never has X or Z bits	immediate	design team
If `load`, then, after `clk` to `cnt` delay, `cnt` has the value of `d`	concurrent	verification team
If `inc`, then, after `clk` to `cnt` delay, `cnt` has incremented by 1	concurrent	verification team
If `!load` and `!inc`, then `cnt` does not change	concurrent	verification team

Exercise: Assertions Test Plan for the Controller



- The controller is a 1-hot finite state machine that sets the control lines for the various DSP blocks
  - **`rstN`** is asynchronous — the controller resets to the **RESET** state

Functionality to Verify	Assertion Type	Assigned To
The **`instruction`** input never has X or Z bits	immediate	design team
At a positive edge of **`clk`**, **`state`** is always 1-hot	concurrent	verification team
If in **`DECODE`** state, then the prior state was **`RESET`** or **`STORE`**	concurrent	verification team
If in **`LOAD`** state, then the prior state was **`DECODE`**	concurrent	verification team
If in **`STORE`** state, then the prior state was **`LOAD`**	concurrent	verification team

Assertion Test Plan Considerations

- Some things to think about when developing an Assertions Test Plan include…
- 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
- Which assertions should be disabled for reset or lower-power mode?
  - False assertion failures can occur if they are not disabled
- The test plan needs to be flexible
  - Some times the responsibility for which team should write the assertion needs to change

More Assertion Test Plan Considerations

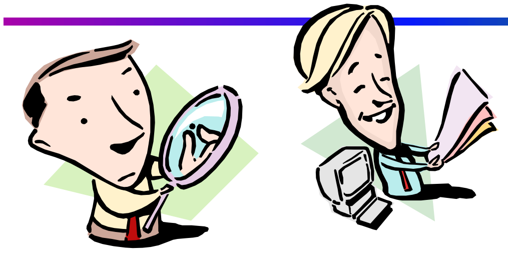
- Assertions may require different design partitioning
  - Example: The DSP ALU block is difficult to check with concurrent assertions because it is pure combinational logic (no clock)
    - Better design partitioning would put the ALU and its input and output registers into one design block
- Enumerated type definitions should be defined globally
  - Example: If the DSP state machine uses a local enumerated variable for the state names, then 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

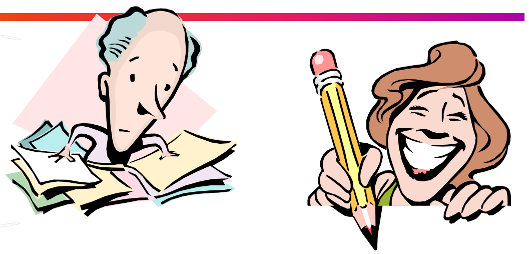
- SystemVerilog Assertions really do work!
  - An effective way to verify many aspects of design functionality
  - Find errors that functional verification might miss
- Verification Engineers should bind in assertions that validate the RTL code matches the design specification
- RTL Design Engineers should embed assertions that validate assumptions directly into RTL code as the code is being written
- There are big advantages to RTL designers specifying assertions
  - Validate design requirements work as specified
  - Validate assumptions on which the RTL model depends
  - Localize where functional problem occurred
  - Clarify specification ambiguities

Question? Comments?

A copy of these presentation slides is available at **sutherland-hdl.com/papers.html**



Once upon a time…



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

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 it ended up that **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.