Getting Started with SystemVerilog Assertions

DesignCon-2006 Tutorial

by Sutherland HDL, Inc., Portland, Oregon

Getting Started With

SystemVerilog Assertions

presented by **Stuart Sutherland** of

SUTHERLAND HDL

training Engineers to be SystemVerilog Wizards!

www.sutherland-hdl.com

3

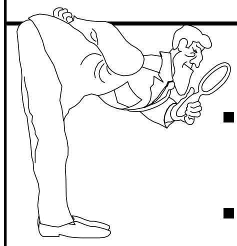
About the Presenter...

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

- **Stuart Sutherland, a SystemVerilog wizard**
  - Independent Verilog/SystemVerilog consultant and trainer
    - Hardware design engineer with a Computer Science degree
    - Heavily involved with Verilog since 1988
    - Specializing in Verilog and SystemVerilog training
  - Member of the IEEE 1800 SystemVerilog standards group
    - Involved with the definition of SystemVerilog since its inception
    - Technical editor of SystemVerilog Reference Manual
  - Member of IEEE 1364 Verilog standards group since 1993
    - Past chair of Verilog PLI task force
    - Technical editor of IEEE 1364-1995, 1364-2001 and 1364-2005 Verilog Language Reference Manual

www.sutherland-hdl.com

This presentation will...



- Provide an overview of some of the major features in SystemVerilog Assertions
- Show how to write basic SystemVerilog Assertions

✓ **The goal is to provide enough detail to get started with SystemVerilog Assertions!**

- But, there are lot of SVA features that we cannot cover in this 3-hour tutorial
- Sutherland HDL's complete training course on SystemVerilog Assertions is a 3-day workshop

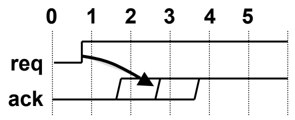
visit www.sutherland-hdl.com for details on our comprehensive SystemVerilog workshops

What This Tutorial Will Cover

- ❑ Why assertions are important
- ❑ SystemVerilog Assertions overview
  - Immediate assertions
  - Concurrent assertions
- ❑ Where assertions should be specified
  - Who should specify assertions
  - Developing an assertions test plan
- ❑ Assertions for Design Engineers
  - Verifying design assumptions
- ❑ Assertions for Verification Engineers
  - Verifying functionality against the specification
  - Specifying complex event sequences
- ❑ Special SystemVerilog Assertion features
  - Assertion system tasks and functions
  - Assertion binding
  - Assertion simulation semantics

What Is An Assertion?

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

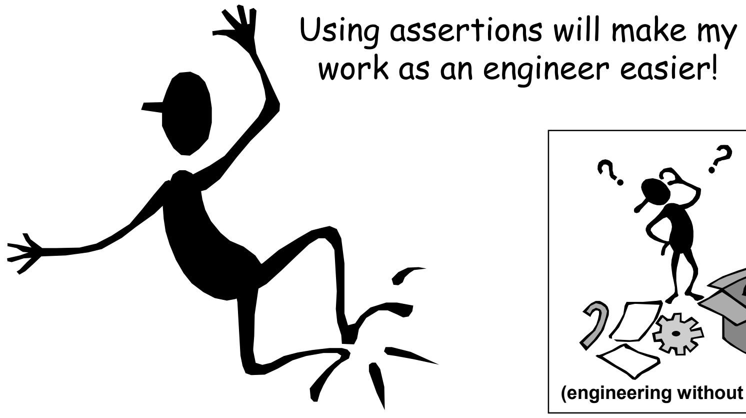


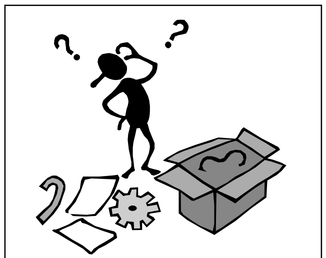
After the request signal is asserted, the acknowledge signal must arrive 1 to 3 clocks later

- Assertions are used to:
  - Document the functionality of the design
  - Check that the intent of the design is met over simulation time
  - Determine if verification tested the design (coverage)
- Assertions can be specified:
  - By the design engineer as part of the model
  - By the verification engineer as part of the test program

Is Assertion Based Verification Worth the Effort?

▪ Several papers have shown that Assertion-Based Verification (ABV) can significantly reduce the design cycle, and improve the quality of the design




(engineering without assertions)

Why Is Using SystemVerilog Assertions Important?

- It's a verification technique that is embedded in the language
  - Gives "white box" visibility into the design
- Enables specifying design requirements with assertions
  - Can specify design requirements using an executable language
- Enables easier detecting of design problems
  - In simulation, design errors can be automatically detected
    - Error reports show when error occurred with insight as to why
  - Formal analysis tools can prove that the model functionality does or does not match the assertion
    - Can generate "counter-examples" (test cases) for assertion failures
- Enables constrained random verification with coverage
  - Assertions can be used to report how effective random stimulus was at covering all aspects of the design

What is Formal Verification?

- Formal verification can statically (without using simulation) …
  - Exhaustively prove that design functionality complies with the assertions about that design
  - Find corner case bugs in complex hardware
    - It is not necessary to write a testbench to cover all possible behaviors
  - Demonstrate functional errors with counterexamples
    - A counterexample is a test case that causes an assertion failure
    - Formal tools can automatically create counterexamples
- Hybrid formal verification tools (such as Synopsys Magellan):
  - Combine random simulation with formal verification
    - Higher capacity than purely formal techniques
    - Better state-space coverage than random simulation alone

Assertion Coverage

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

- Assertion coverage helps answer the questions:
  - Are there enough assertions in the design?
  - Is the verification plan for simulation complete?
  - How thorough is the formal verification analysis?
- Assertion coverage can report on:
  - The number of assertions that never triggered
  - The number of assertions that only had vacuous successes

```
A |-> B;
```

If A is true then B must be true

If "A" is never true, then "B" is never tested (the assertion is always "vacuously true“)

  - The number of assertions that did not test every branch

```
A |-> ##[0:10] (B || C);
```

If A is true then either B or C must be true within 10 clock cycles

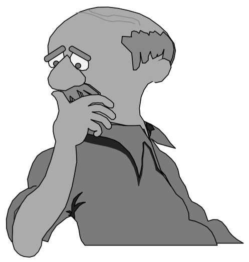
If "B" is true every time, the "C" is never tested

assertion succeeds if either B or C is true

Adopting an Assertion Based Verification Methodology

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

- An Assertion-Based Verification (ABV) methodology addresses several verification questions:
  - Who writes the assertions?
  - What languages should we use?
  - Are there assertion libraries?
  - How do we debug assertions?
  - How are assertions controlled in simulation?
  - Can we use assertions to measure functional coverage?
  - What about formal verification of assertions?
  - How do we know when we have written enough assertions?
- As we go through this tutorial, we will discuss and answer several of these questions



What's Next...

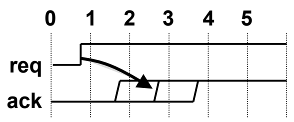
SUTHERLAND HDL
training engineers to be Verilog and SystemVerilog wizards
www.sutherland-hdl.com

- ✓ Why assertions are important
- ❑ **SystemVerilog Assertions overview**
  - Immediate assertions
  - Concurrent assertions
- ❑ Where assertions should be specified
  - Who should specify assertions
  - Developing an assertions test plan
- ❑ Assertions for Design Engineers
  - Verifying design assumptions
- ❑ Assertions for Verification Engineers
  - Verifying functionality against the specification
  - Specifying complex event sequences
- ❑ Special SystemVerilog Assertion features
  - Assertion system tasks and functions
  - Assertion binding
  - Assertion simulation semantics

Verilog Does Not Have An Assertion Construct

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

- Verilog does not provide an assertion construct
  - Verification checks must be coded with programming statements



Each request must be followed by an acknowledge within 2 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 several lines of Verilog code
- **Difficult to write, read and maintain**
- **Cannot be turned off during reset or other don't care times**

Checker's Written in Verilog Must be Hidden from Synthesis

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

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

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

Advantages of SystemVerilog Assertions

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

- 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

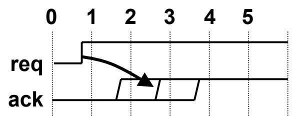
```
always @(state)
  assert (state == $onehot) else $fatal;
```

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!

Immediate Assertions

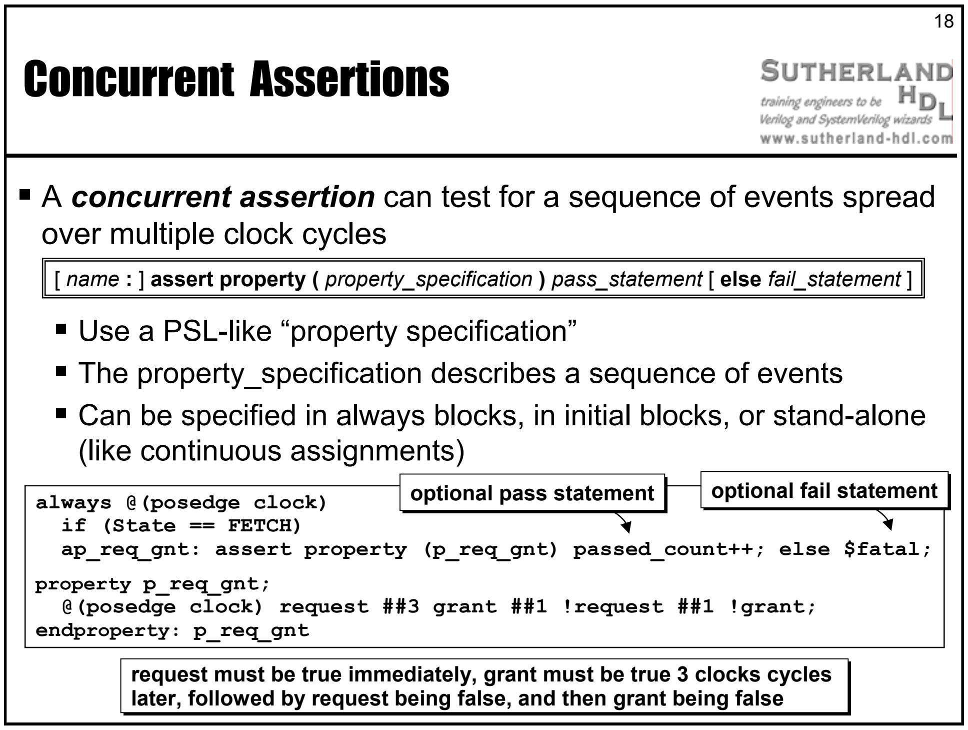
- An ***immediate assertion*** is a test of an expression the moment the statement is executed

[*name* :] **assert (** *expression* **)** [*pass_statement*] [**else** *fail_statement*]

  - May be used in initial and always procedures, tasks and functions
  - Performs a boolean true/false test
    - If the test result is true, execute the pass statement
    - If the test is result false or unknown, execute the fail statement
  - Evaluates the test at the instant the assert statement is executed

```
always @(negedge reset)
  a_fsm_reset: assert (state == LOAD)
    $display("FSM reset in %m passed");
  else
    $display("FSM reset in %m failed");
```

The name is optional:
- **Creates a named hierarchy scope that can be displayed with %m**
- **Provides a way to turn off specific assertions**



Concurrent Assertions

- A ***concurrent assertion*** can test for a sequence of events spread over multiple clock cycles

[*name* :] **assert property (** *property_specification* **)** *pass_statement* [**else** *fail_statement*]

  - Use a PSL-like "property specification"
  - The property_specification describes a sequence of events
  - Can be specified in always blocks, in initial blocks, or stand-alone (like continuous assignments)

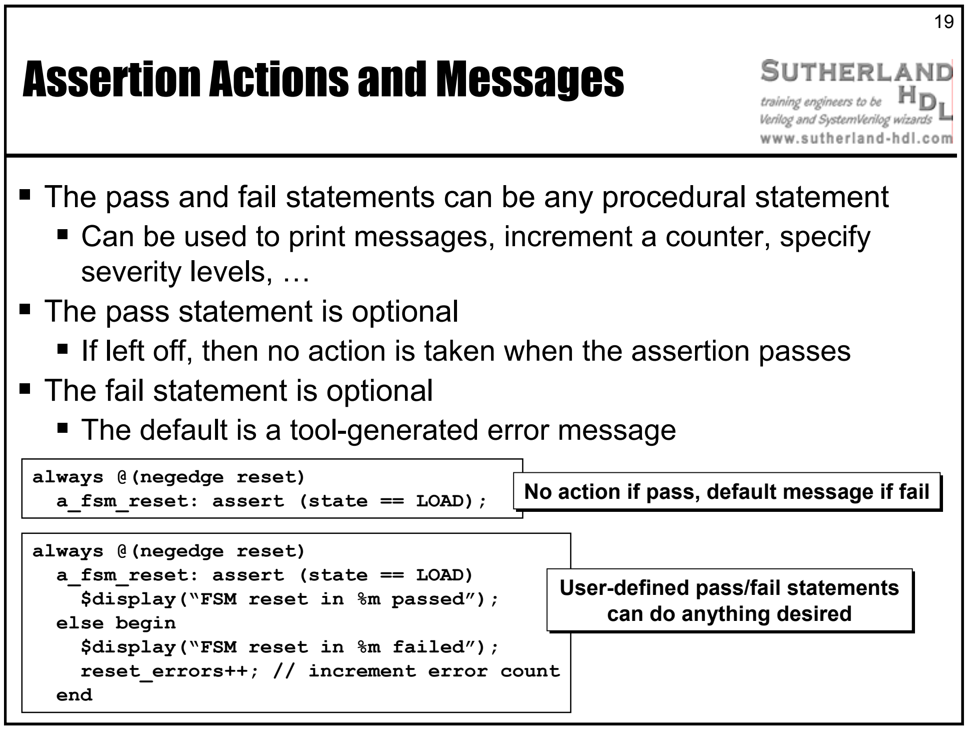
```
always @(posedge clock)
  if (State == FETCH)
  ap_req_gnt: assert property (p_req_gnt) passed_count++; else $fatal;

property p_req_gnt;
  @(posedge clock) request ##3 grant ##1 !request ##1 !grant;
endproperty: p_req_gnt
```

optional pass statement

optional fail statement

request must be true immediately, grant must be true 3 clocks cycles later, followed by request being false, and then grant being false



Assertion Actions and Messages

- The pass and fail statements can be any procedural statement
  - Can be used to print messages, increment a counter, specify severity levels, …
- The pass statement is optional
  - If left off, then no action is taken when the assertion passes
- The fail statement is optional
  - The default is a tool-generated error message

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

No action if pass, default message if fail

```
always @(negedge reset)
  a_fsm_reset: assert (state == LOAD)
    $display(“FSM reset in %m passed”);
  else begin
    $display(“FSM reset in %m failed”);
    reset_errors++; // increment error count
  end
```

User-defined pass/fail statements can do anything desired

Assertion Severity Levels

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

- The 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);
```

The user-supplied message is appended to a tool-specific message containing the source file location and simulation time

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

The message text is optional; if not specified the tool-specific message will still be printed

Assertion Terminology

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

- SystemVerilog supports three general categories of assertions…
  - Invariant assertions
    - A condition that should always be true (or never be true)
    - Example: A FIFO should never indicate full and empty at the same time
  - Sequential assertions
    - A set of conditions occuring in a specific order and over a defined number of clock cycles
    - Example: A request should be followed in 1 to 3 clock cycles by grant
  - Eventuality assertions
    - A condition should be followed by another condition, but with any number of clock cycles in between
    - Example: When an active-low reset goes low, it should eventually go back high

22

What's Next...

- ✓ Why assertions are important
- ✓ SystemVerilog Assertions overview
  - Immediate assertions
  - Concurrent assertions
- ☐ **Where assertions should be specified**
  - Who should specify assertions
  - Developing an assertions test plan
- ☐ Assertions for Design Engineers
  - Verifying design assumptions
- ☐ Assertions for Verification Engineers
  - Verifying functionality against the specification
  - Specifying complex event sequences
- ☐ Special SystemVerilog Assertion features
  - Assertion system tasks and functions
  - Assertion binding
  - Assertion simulation semantics

23

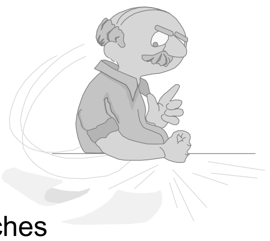
Where Assertions Can be Specified

As we will see, 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

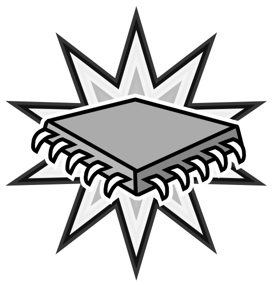
Who Should Write the Assertions?

- Assertions are verification constructs, but…
  - *Should assertions only be written by the verification team?*
- **Assertions are for design engineers, too!**
  - Designs are full of assumptions
    - Inputs to the module are valid values
    - Handshakes are always completed
    - Case statements never take unintended branches
  - Design engineers should add assertions as the code is written
    - Every assumption about the design should be an assertion
      - No X values on inputs
      - State machine sequencing is as intended
      - requests are followed by an acknowledge



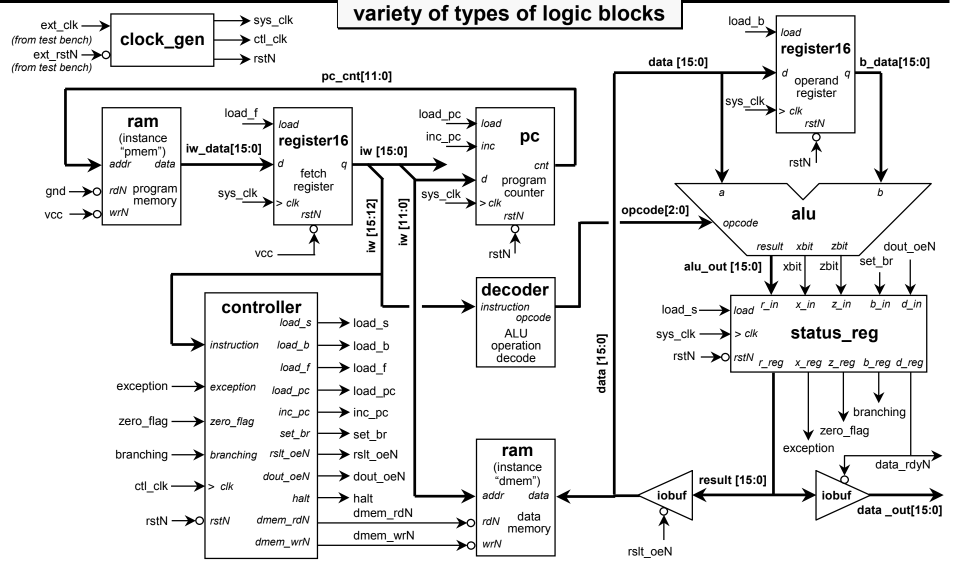
Case Study: Assertions for a Small DSP Design

- 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/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
  - The DSP is not a real design — it is scaled down for lab purposes



DSP Case Study: Block Diagram

The DSP design contains a variety of types of logic blocks



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?
    - 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?

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

An Assertions Test Plan Example

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

- RAM assertions

Functionality to Verify	Assertion Type	Assigned To
!rdN and !wrN are mutually exclusive	invariant	design team
address never has any X or Z bits when reading from or writing to the RAM	invariant	design team
data never has any X or Z bits when reading from or writing to the RAM	sequential	design team

- Program Counter assertions

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

An Assertions Test Plan Example

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

- ALU assertions

Functionality to Verify	Assertion Type	Assigned To
After reset, the A, input never have any X or Z bits	invariant	design team
After reset, the B input never have any X or Z bits	invariant	design team
After reset, the opcode input never have any X or Z bits	invariant	design team
All instructions are decoded	unique case	design team
zbit is always set if result == 0	invariant	verification team
zbit is never set if result != 0	invariant	verification team
xbit is always set if a mathematical operation results overflow or underflow	invariant	verification team
xbit is never set if a mathematical operation does not result in an overflow or underflow	invariant	verification team
xbit is never set for non-arithmetic operations	invariant	verification team
If load, then pc == d (must allow for clock-to-q delay)	sequential	verification team

What's Next...

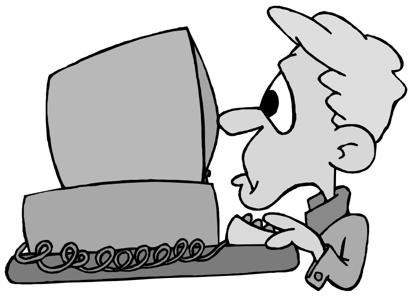
SUTHERLAND HDL
training engineers to be Verilog and SystemVerilog wizards
www.sutherland-hdl.com

A 10 Minute Break!

What's Next...

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

- ✓ Why assertions are important
- ✓ SystemVerilog Assertions overview
  - Immediate assertions
  - Concurrent assertions
- ✓ Where assertions should be specified
  - Who should specify assertions
  - Developing an assertions test plan
- ☐ **Assertions for Design Engineers**
  - Verifying design assumptions
- ☐ Assertions for Verification Engineers
  - Verifying functionality against the specification
  - Specifying complex event sequences
- ☐ Special SystemVerilog Assertion features
  - Assertion system tasks and functions
  - Assertion binding
  - Assertion simulation semantics



Guideline!

- Designer engineers should write assertions to verify assumptions that affect the functionality of a design block
  - Example: The ALU block assumes that the A, B and opcode inputs will never have a logic X or Z value
    - The RTL code depends on this assumption to function properly
    - When modeling the ALU, the designer should add assertions to the design block that verify these assumptions hold true
      - The assertion documents the designer's assumptions
      - Should the assumption prove false, the assertion failure will help isolate where a functional problem arose
- Assertions should not duplicate RTL logic!
  - RTL logic monitors input changes and causes an effect on an output
  - An assertion should monitor output changes, and verify that the input values will cause that effect
    - Poor assertion: If the ALU result is zero, then the zbit should be set
    - Good assertion: If the zbit is set, then the ALU result should be zero

Assertion Plan Example 1: Assertions on ALU Inputs

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

Functionality to Verify	Assertion Type	Assigned To
After reset, the A, input never have any X or Z bits	invariant	design team
After reset, the B input never have any X or Z bits	invariant	design team
After reset, the opcode input never have any X or Z bits	invariant	design team
All instructions are decoded	unique case	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: Assertions on RAM Inputs

- RAM ***design engineer*** assertions example

Functionality to Verify	Assertion Type	Assigned To
!rdN and !wrN are mutually exclusive	invariant	design team
...		

```
module ram (...);
 ...
  // write cycle
  always_latch begin
    if (!wrN) begin
      // assertion to check that no bits of address or data input are X or Z
      ai_addr_never_x: assert (^addr !==1'bx);
      ai_data_never_x: assert (^data !==1'bx);
      mem[addr] <= data;
    end
  end

  // assertion to check that read and write are never low at the same time
  always @(rdN or wrN)
    ai_read_write_mutex: assert (!(!rdN && !wrN));
```

This is so simple…why am I not already doing this in all my designs?

This check is written to run in parallel with the design logic

What's Next...

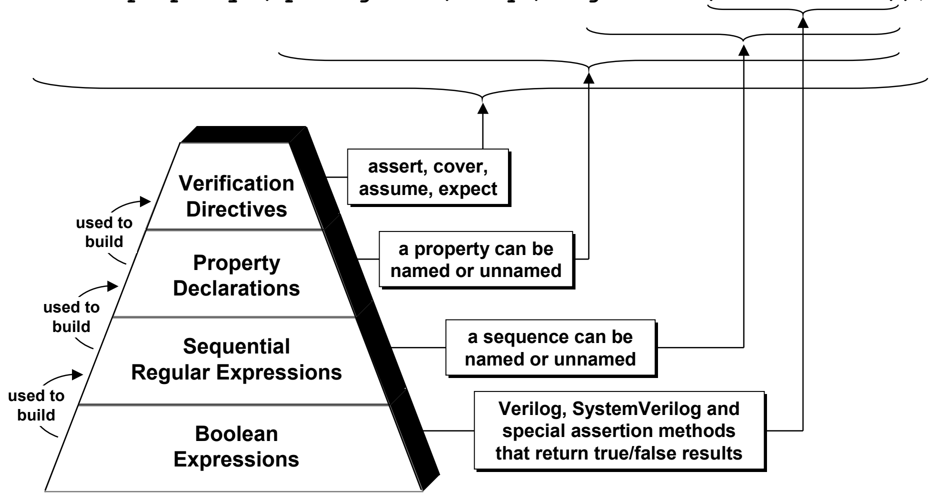
✓ Why assertions are important
✓ SystemVerilog Assertions overview
  - Immediate assertions
  - Concurrent assertions
✓ Where assertions should be specified
  - Who should specify assertions
  - Developing an assertions test plan
✓ Assertions for Design Engineers
  - Verifying design assumptions
❑ **Assertions for Verification Engineers**
  - Verifying functionality against the specification
  - Specifying complex event sequences
❑ Special SystemVerilog Assertion features
  - Assertion system tasks and functions
  - Assertion binding
  - Assertion simulation semantics

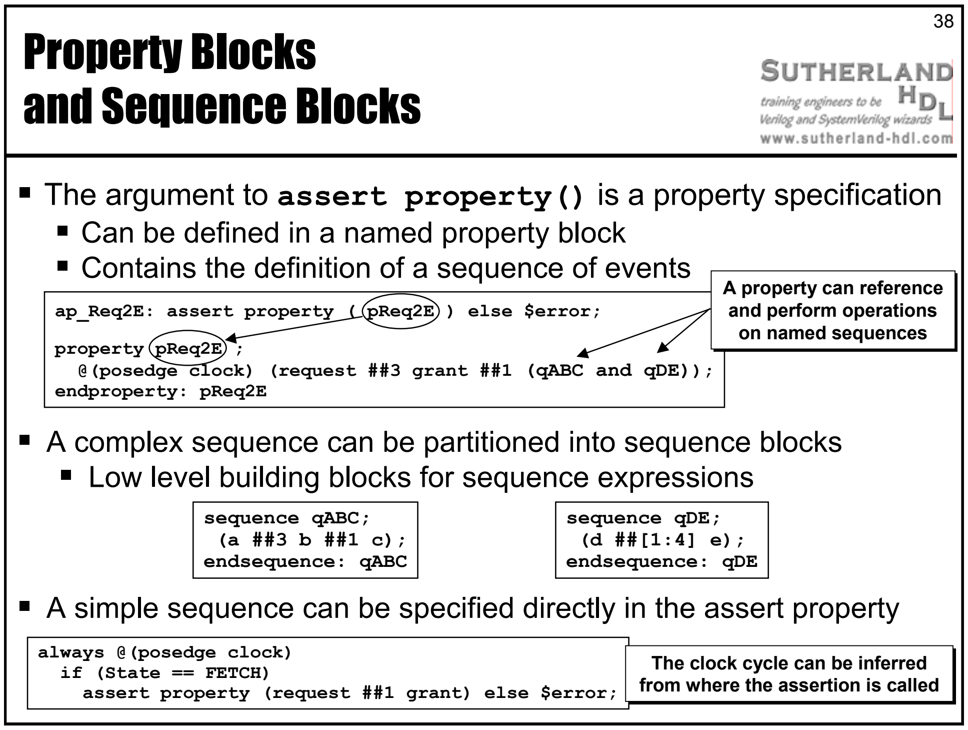
Guideline!

- Verification engineers should write assertions that verify design functionality meets the design specification
  - Example: The zero flag output of the ALU block should always be set if the ALU result output is zero
    - An assertion failure will help isolate the cause of a functional problem
- Assertions should not duplicate RTL logic!
  - RTL logic monitors input changes and causes an effect on an output
  - An assertion should monitor output changes, and verify that the input values will cause that effect
    - Poor assertion: If the ALU result is zero, then the zbit should be set
    - Good assertion: If the zbit is set, then the ALU result should be zero

Concurrent Assertion Building Blocks

```
assert property (@posedge clk) req |-> gnt ##1 (done && !err));
```


assert, cover, assume, expect
a property can be named or unnamed
a sequence can be named or unnamed
Verilog, SystemVerilog and special assertion methods that return true/false results
Verification Directives
Property Declarations
Sequential Regular Expressions
Boolean Expressions
used to build
used to build
used to build



Property Blocks and Sequence Blocks

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

- The argument to **`assert property()`** is a property specification
  - Can be defined in a named property block
  - Contains the definition of a sequence of events

```
ap_Req2E: assert property ( pReq2E ) else $error;

property pReq2E ;
  @(posedge clock) (request ##3 grant ##1 (qABC and qDE));
endproperty: pReq2E
```

A property can reference and perform operations on named sequences

- A complex sequence can be partitioned into sequence blocks
  - Low level building blocks for sequence expressions

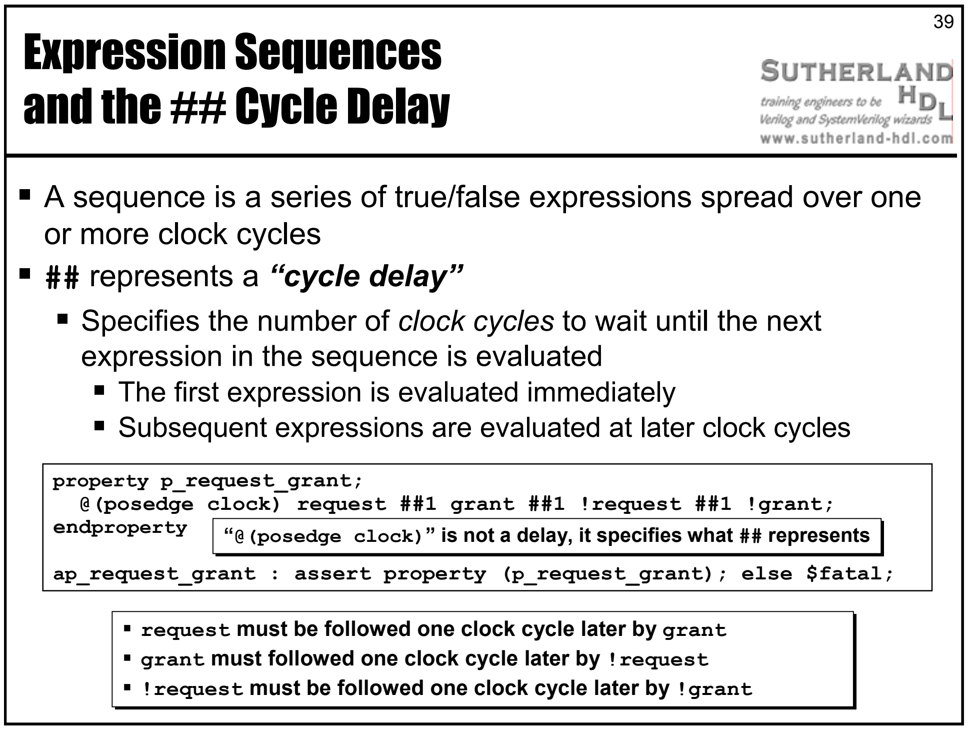
```
sequence qABC;
 (a ##3 b ##1 c);
endsequence: qABC
```

```
sequence qDE;
 (d ##[1:4] e);
endsequence: qDE
```

- A simple sequence can be specified directly in the assert property

```
always @(posedge clock)
  if (State == FETCH)
    assert property (request ##1 grant) else $error;
```

The clock cycle can be inferred from where the assertion is called



Expression Sequences and the ## Cycle Delay

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

- 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 !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`

Multiple Cycle Clock Delays

- **`##n`** specifies a fixed number of clock cycles
  - n must be a non-negative constant expression

```
request ##3 grant;
```

After evaluating request, skip 2 clocks and then evaluate grant on the 3rd clock

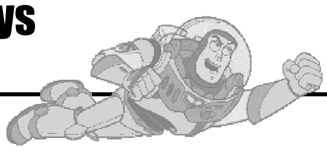
- **`##[min_count:max_count]`** specifies a range of clock cycles
  - min_count and max_count must be non-negative constants

```
request ##[1:3] grant;
```

After evaluating request, grant must be true between 1 and 3 clocks later

This sequence would evaluate as true for:
```
   (request ##1 grant);
or (request ##2 grant);
or (request ##3 grant);
```

Infinite Cycle Delays



- The dollar sign (**`$`**) is used to specify an infinite number of cycles

```
request ##[1:$] grant;
```

request must true at the current cycle, then grant must become true sometime between now and the end of time

  - In simulation, the end of time is when simulation finishes
    - Simulators might report an assertion that never completed as a failure or as an uncompleted assertion
  - In formal verification, there is no end of time
    - Formal tools might keep trying to find a success until told to stop

Repeated Regular Expressions

- A sequence of events can be repeated using a repeat count, in the form **`[* n ]`** (n must be a non-negative constant expression)

```
a ##1 (b[*3]);
```

is equivalent to:

```
(a ##1 b ##1 b ##1 b)
```

- A range of steps can be repeated using a count, in the form **`[*`** min_count **`:`** max_count **`]`** (must be a non-negative constants)

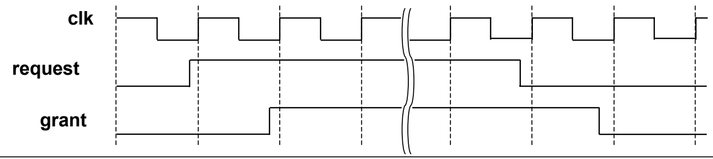
```
(a[*0:3] ##1 b ##1 c);
```

is equivalent to:

```
   (b ##1 c)
or (a ##1 b ##1 c)
or (a ##1 a ##1 b ##1 c)
or (a ##1 a ##1 a ##1 b ##1 c)
```

Infinite Repeated Expressions

- An infinite number of repetitions can be specified using **`[*1:$]`**



- `request` **must be followed one clock later by** `grant`
- `grant` **must followed** ***any number of clock cycles later*** **by** `!request`
  - `grant` ***must remain true until*** `!request`
- `!request` **must be followed one clock later by** `!grant`

```
property p_request_grant;
  @(posedge clk)
  request ##1 grant[*1:$] ##1 !request ##1 !grant;
endproperty
ap_request_grant: assert property (p_request_grant);
```

Declarative and Procedural Concurrent Assertions

- Procedural concurrent assertions
  - Specified within an initial or always procedure
  - Runs when the procedural block calls the assert statement
  - Runs as a separate, parallel thread to the procedure

```
module top(input bit clk);
  logic req, grant;
  property p2;
    req ##3 gnt;
  endproperty
  always @(posedge clk)
    if (State == FETCH)
     ap_p2: assert property (p2);
  ...
endmodule
```

- Declarative concurrent assertions
  - Specified outside of initial or always procedural blocks
  - Runs throughout the simulation
  - "Fires" (starts a new evaluation) every clock cycle

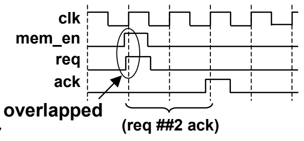
```
module top(input bit clk);
  logic req, grant;
  property p1;
    @(posedge clk) req |-> ##3 gnt;
  endproperty
  ap_p1: assert property (p1);
  ...
endmodule
```

Declarative assertions are the most common type

Conditioning Sequences Using Implication Operators

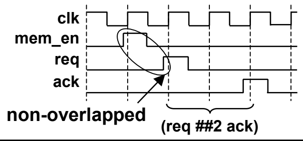
- Evaluation of a sequence can be preconditioned with an implication operator
  - `|->` overlapped implication operator
    - If the condition is true, sequence evaluation starts immediately
    - If the condition is false, the sequence acts as if it succeeded

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



  - `|=>` non-overlapped implication operator
    - If the condition is true, sequence evaluation starts at the next clock
    - If the condition is false, the sequence acts as if it succeeded

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



A "Gotcha" With Simple Sequence Expressions

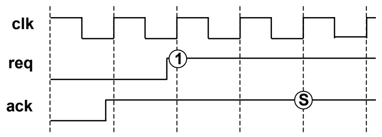
- A simple sequence expression can test as true even if the expressions changed in the wrong sequence
  - Given the following assertion:

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

ap_req_ack: assert property (p_req_ack);
```

ack must be true 2 clock cycles after req

  - 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

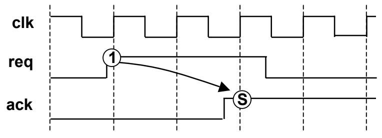
If the design requires an acknowledge must *follow* a request, then the assertion must verify that ack does not become true until *after* req went true

Sequence Value Change Functions

- Special system functions are provided to detect if a value changed between two adjacent clock ticks:
  - `$rose   (expression);`
    - returns true if the least significant bit of the expression changed to 1
  - `$fell   (expression);`
    - returns true if the least significant bit of the expression changed to 0
  - `$stable (expression);`
    - returns true if the value of the expression did not change

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

ap_req_ack: assert property (p_req_ack);
```



$rose and $fell should only be used with 1-bit wide signals; if a vector is used, only the LSB is monitored for changes

A "Gotcha" With Declarative Assertions

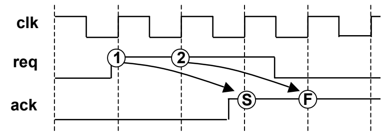
- A declarative assertion fires every single clock cycle
  - Given the following assertion:

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

ap_req_ack: assert property (p_req_ack);
```

ack must be true 2 clock cycles after req

  - Why is there an assertion failure on the cycle after acknowledge?



The assertion will pass the first check when req is high, but a second check is started on the next clock, because req is still high

If the design requires an req stay high until ack, then the assertion should check for the rising edge of ack instead of the logic level of ack:
`$rose(req) |-> ##2 $rose(ack);`

Testing for a Cause

- Many assertions test if a *cause* resulted in an *effect*

Every request should be followed by an acknowledge within 1 to 6 clock cycles

```
property pReqAck;
  @(posedge clk) req |-> ##[1:6] $rose(ack);
endproperty: pReqAck
```

- Sometimes it is necessary to test if an *effect* had a *cause*

Every acknowledge should have been preceded by a request in the last 1 to 6 clock cycles

This means I have to look back in time — How do I do that?

- SVA provides three ways to look back into the past
  - `$past()` function
  - `.ended` method (for single clock assertions)
  - `.matched` method (for multi-clock assertions)

Looking Back In Time for a Cause

- An assertion can use the sampled value of an expression any number of clock cycles in the past

```
$past ( expr [, number_of_cycles] [, gating_expr] [, clocking_event] );
```
- Returns the sampled value of expr any number of clock cycles prior to the time of the evaluation of $past
  - number_of_cycles (optional) specifies the number of clock cycles in the past
    - If number_of_cycles is not specified, then it defaults to 1
  - gating_expr (optional) is used as a gating expression for the clocking event
  - clocking_event specifies the clocking event for sampling expr
    - If not specified, the clocking event of the property or sequence is used

```
property pReqCausedAck;
  @(posedge clk) $rose(ack) |-> $past(req, 6);
endproperty: pReqCausedAck
```

if `ack` became true, then was `req` true 6 clock cycles previously

Vector Analysis Functions

- Vector analysis system functions provide a means to test the bits of vectors for specific bit patterns

- **`$onehot (expression);`**
  - returns true if *only* one bit of a vector is high
- **`$onehot0 (expression);`**
  - returns true if *at most* one bit of a vector is high
- **`$isunknown (expression);`**
  - returns true if any bit of a vector is X or Z
- **`$countones (expression);`**
  - returns the number of bits that are set to 1 in a vector (X and Z bits are not counted)

```
property pCheckState;
  $onehot(state);
endproperty: pCheckState

a_pCheckState: assert property (@(posedge clk) pCheckState);
```

the assertion will fail if no bits are set or more than one bit is set at each clock cycle

Assertion Plan Example 3: Assertions on the Program Counter

- Program Counter ***verification engineer*** assertions example

Functionality to Verify	Assertion Type	Assigned To
…		
If !load and !increment, then on posedge of clock, pc does not change (must allow for clock-to-q delay)	sequential	verification team
If increment, then pc increments by 1 (must allow for clock-to-q delay)	sequential	verification team
If load, then pc == d input (must allow for clock-to-q delay)	sequential	verification team

```
property p_no_change_if_not_enabled; // no change if not loading or incrementing
  @(posedge clk)
   (!load && !inc) |-> ##1 $stable(pc_cnt);
endproperty
ap_no_change_if_not_enabled: assert property (p_no_change_if_not_enabled);

property p_increment_if_enabled; // if increment is enabled, then PC increments
  @(posedge clk)
   inc |-> ##1 pc_cnt == ($past(pc_cnt) + 1);
endproperty
ap_increment_if_enabled: assert property (p_increment_if_enabled);
```

the PC load check is similar to this check

Assertion Plan Example 4: Assertions on the State Machine

- FSM ***verification engineer*** assertions example

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

What's Next...

- ✓ Why assertions are important
- ✓ SystemVerilog Assertions overview
  - Immediate assertions
  - Concurrent assertions
- ✓ Where assertions should be specified
  - Who should specify assertions
  - Developing an assertions test plan
- ✓ Assertions for Design Engineers
  - Verifying design assumptions
- ✓ Assertions for Verification Engineers
  - Verifying functionality against the specification
  - Specifying complex event sequences
- ☐ **Special SystemVerilog Assertion features**
  - Assertion system tasks and functions
  - Assertion binding
  - Assertion simulation semantics

You can't do these tricks in Verilog or PSL!

Controlling Assertions

- Special system tasks are used to control assertions

```
$assertoff ( levels [ , list_of_modules_or_assertions ] ) ;
```
  - Stops the evaluation and execution of the specified assertions
  - Assertions currently being executed when $assertoff is called will complete execution

```
$assertkill ( levels [ , list_of_modules_or_assertions ] ) ;
```
  - Stops the evaluation and execution of the specified assertions
  - Assertions currently being executed when $assertoff is called are aborted

```
$asserton ( levels [ , list_of_modules_or_assertions ] ) ;
```
  - re-enables the evaluation and execution of the specified assertions

- Modules are specified using a relative or full hierarchy path name
- Assertions are specified using the name of the assertion
- levels indicates how many levels of hierarchy below the specified module(s) in which to turn assertions on or off
  - 0 indicates all levels of hierarchy below the reference

By default, all assertions are turned on

Assertion Control Example

- The following example:
  - Disables all assertions when reset is active (active low)
  - Re-enables all assertions after reset is complete

```
module assert_control ();
  initial begin : disable_assertions_during_reset
    @(negedge top_tb.reset_n)  // active low reset
      $display ("%0t %m Disabling assertions during reset", $time);
      $assertoff(0, top_tb.cpu_rtl_1);
    @(posedge top_tb.reset_n)
      $display ("%0t %m Enabling assertions after reset", $time);
      $asserton(0, top_tb.cpu_rtl_1);
  end
endmodule : assert_control
```

```
module top_tb;
  ...
  cpu_rtl cpu_rtl_1(clk, reset_n, .*); // instance of cpu module
  assert_control assert_control();     // instance of assertion control
  ...
endmodule : top_tb
```

Binding Assertions to Design Blocks

- Assertions and properties can be defined outside of the design models, and “bound” to the design

bind *design-block-name_or_instance-name   design-block-with-assertions* ;

  - SystemVerilog assertions can be bound to a specific instance of a module or interface
  - SystemVerilog assertions can be bound to all instances of a module or interface
  - The assertions can be defined in separate design blocks (modules, interfaces, or programs)
- Binding allows the verification engineer to add assertions to a design without modifying the design files!

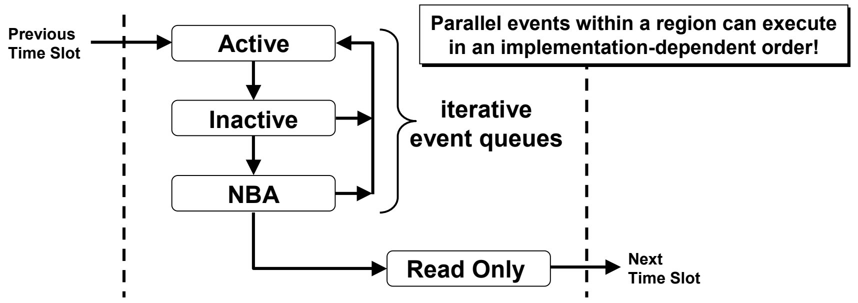
SystemVerilog assertions can also be bound to VHDL models (requires a mixed language simulator or formal analysis tool)

Behind the Curtains: How Assertions are Simulated

- The problem…
  - Assertion-like checks written in Verilog are just programming statements
    - The checks execute with the same simulation semantics as the RTL code
    - You must be very careful to avoid race conditions between the RTL code and the checking routines
  - Assertions written in PSL are just comments
    - Comments have no standard simulation semantics — how a simulator should execute PSL assertions is not defined!
- The solution…
  - SVA defines concurrent assertion execution semantics
    - Race condition avoidance is built in!
    - All simulators will evaluate SVA in the same way!

Verilog Simulation Event Scheduling

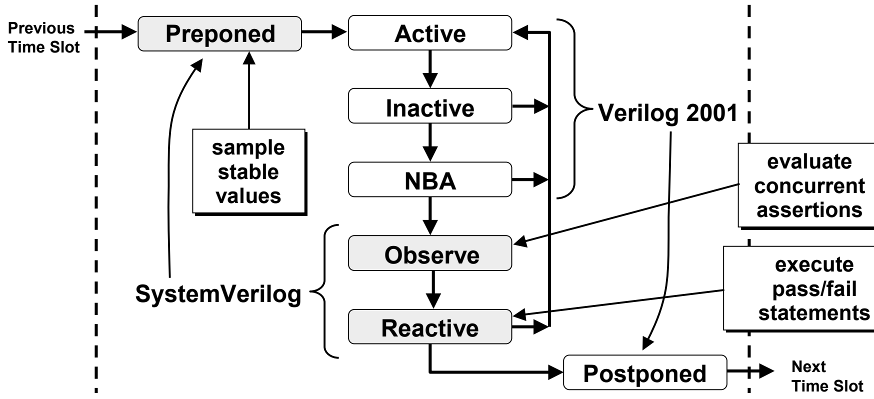
- Events within a simulation time step are divided into 4 regions
  - Execute all active events, then inactive events, then non-blocking assignment update (NBA) events
    - Active events include procedural statements and assignment statements
  - Re-iterate the three queues until all are empty


Previous Time Slot
Active
Inactive
NBA
iterative event queues
Parallel events within a region can execute in an implementation-dependent order!
Read Only
Next Time Slot

Concurrent Assertions and Simulation Event Scheduling

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

- Concurrent assertion expressions are:
  - Sampled in a preponed region
  - Evaluated in an observe region, using the sampled values
  - Execute assertion pass or fail statements in a reactive region



It's Time to Wrap Things Up...

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

- ✓ Why assertions are important
- ✓ SystemVerilog Assertions overview
  - Immediate assertions
  - Concurrent assertions
- ✓ Where assertions should be specified
  - Who should specify assertions
  - Developing an assertions test plan
- ✓ Assertions for Design Engineers
  - Verifying design assumptions
- ✓ Assertions for Verification Engineers
  - Verifying functionality against the specification
  - Specifying complex event sequences
- ✓ Special SystemVerilog Assertion features
  - Assertion system tasks and functions
  - Assertion binding
  - Assertion simulation semantics

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

Additional Resources

- IEEE 1800-2005 SystemVerilog Language Reference Manual
  - 2005, published by the IEEE, ISBN 0-7381-4811-3 (PDF version)
- SystemVerilog Assertions Handbook
  - Cohen, Venkataramanan, Kumari, 2004, ISBN: 0-9705394-7-9
- Assertion-Based Design, Second Edition (PSL and SVA)
  - Harry Foster, Adam Krolnik, David Lacey, 2004, ISBN: 1-4020-8027-1
- SystemVerilog for Design (synthesizable SystemVerilog)
  - Sutherland, Davidmann and Flake, 2001, ISBN: 1-4020-7530-8
- SystemVerilog Assertions Are For Design Engineers, Too
  - Sutherland and Mills, SNUG Conference paper, March 2006

Sutherland HDL offers comprehensive training on SystemVerilog Assertions!

Questions & Answers...

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

What will assertions reveal about my design?

A copy of this presentation will be available at www.sutherland-hdl.com/papers