Introduction to SystemVerilog Assertions (SVA)

Harry D. Foster
Chief Scientist Verification
IC Verification Solutions Division

February 2020

Mentor A Siemens Business

Lecture Overview

In this lecture, you will. . .

- Learn the structure of the SVA language
- Learn how to construct sequence
- Learn how to construct properties
- Apply SVA on real examples
- Exercises
- Summary

LINEAR FORMALISM

Brief Review of LTL and Introduction of Regular Expressions

SystemVerilog Assertions

- SVA is based on linear temporal logic (LTL) built over sublanguages of regular expressions.
- Most engineers will find SVA sufficient to express most common assertions required for hardware design.

What We can Express in LTL

- **All Boolean logic propositions - p**

  "Process 2 is in the critical section"

- **X p – p holds in the next state.**

  "Process 2 will be in the critical section in the next state"

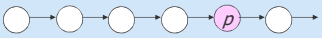
X *p*

What We can Express in LTL

- **F p – sometimes (i.e., eventually) p holds.**

  "eventually process 2 will enter the critical section"

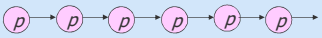
F *p*



- **G p – always (i.e., globally) p holds.**

  "process 1 and 2 are always mutually exclusive"

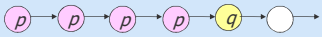
G *p*



What We can Express in LTL

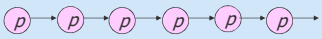
- **[*p* U *q*]** – "*q* holds now or sometime in the future and *p* holds from now until *q* holds" (strong)

p* U *q



- **[*p* W *q*]** – "*p* holds from now until *q* holds" (weak)

p* W *q



What We can Express in LTL

- Weak operators – **X, G, W**
  Used to express safety properties,
  i.e. *"something bad never happens"*

- Strong operators – **F, U**
  Used to express liveness properties,
  i.e. *"something good eventually happens"*

Safety properties put no obligation on the future, liveness properties do!

What We can Express in LTL

- LTL formulas can be combined using the ¬, ∧, ∨, → logic connectors *(negation, conjunction, disjunction, implication)*

For example….

G (*request* → F *grant*)

What We can Express in LTL

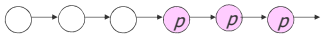
- LTL formulas can be combined using the ¬, ∧, ∨, → logic connectors *(negation, conjunction, disjunction, implication)*

For example….

G (*request* → F *grant*)

Temporal operators can be combined too…

FG *p*



What We Cannot Express in LTL

- Counting example:
  "p is asserted in every even cycle"

  All the following traces satisfy this property
  !p,p,!p,p,…
  p,p, p,p….
  p,p,!p,p,p,p…

- No LTL formula can express this property

Regular Expressions

- Regular expressions describe sets of finite words *w=a1,a2,…,an* .
  — a1,a2,… are letters in an alphabet.
- Regular expressions can express counting modulo n.
- The * operator – enables counting modulo n.
  — (ab)* - a regular expression describing the set of words:
  - ε - (the empty word)
  - ab
  - abab
  - ababab…..

Regular Expressions

- For reactive systems a letter in the alphabet is a Boolean expression
- The set of computations satisfying "*p is asserted in every even cycle*" **is described by the SVA regular expression**

  (1`b1 ## p)[*]
- A regular expression by itself is not a property

—Later: building properties from regular expressions in SVA

What Regular Expressions Cannot Express

- The behavior, *"eventually p holds forever"* cannot be expressed by a regular expression
- It can be expressed in LTL as : F G *p*

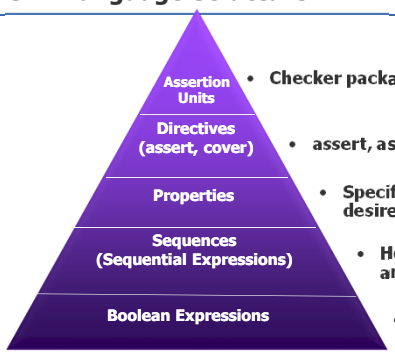
Linear Formalisms

- LTL and regular expressions are *linear formalisms*
  - Linear formalisms can be used to express mainly properties that are intended to hold on all computations (i.e., executions of a design model).
  - Most properties required for the specification of digital designs can be expressed using linear formalism
- What cannot express in linear formalisms:

  "There exists a computation in which eventually *p* holds forever"
  - LTL implicitly quantifies universally over paths

SVA LANGUAGE STRUCTURE

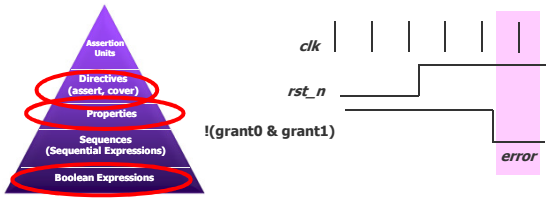
SVA Language Structure



- Checker packaging
- assert, assume, cover
- Specification of behavior; desired or undesired
- How Boolean events are related over time
- True or false

SVA Language Structure

```
assert property (@(posedge clk) disable iff (~rst_n)
                                  !(grant0 & grant1));
```



Note: rst_n is an active low reset in this example

SVA Language Structure

- SVA provides a mechanism to asynchronously disable a property during a reset using the SVA *disable iff* clause

```
assert property (@(posedge clk) disable iff (~rst_n)
                                  !(grant0 & grant1));
```

Note: rst_n is an active low reset in this example

MAPPING SVA INTO LTL

LTL Operators in SVA

- **All Boolean logic propositions - p**

  "Process 2 is in the critical section"

- **LTL: X p – p holds in the next state.**
- **SVA: nexttime [n] p – p holds in the next state.**

  "Process 2 will be in the critical section in the next state"

nexttime *p*

LTL Operators in SVA

- LTL: F p – eventually p holds.
- SVA: eventually p – eventually p holds (weak).

  "eventually process 2 will enter the critical section"

eventually *p*

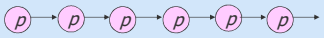
*Note: **s_eventually** is a strong version of this operator in SVA.*

LTL Operators in SVA

- **LTL: G p – always (i.e., globally) p holds.**
- **SVA: always p – always (i.e., globally) p holds.**

  "process 1 and 2 are always mutually exclusive"

always *p*



Note: there is an implicit always when asserting a property:
assert property(p);

LTL Operators in SVA

- LTL: [*p* U *q*] – "*q* holds now or sometime in the future and *p* holds from now until *q* holds" (strong)
- SVA: *p* s_until *q*

p s_until q

- LTL: [*p* W *q*] – "*p* holds from now until *q* holds" (weak)
- SVA: *p* until *q*

p until q

SVA with LTL Operator Example

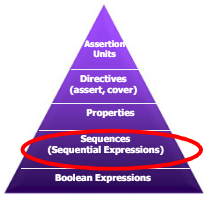
```
assert property (@posedge clk disable iff (reset)
                 $rose(req) implies !done s_until grnt);
```


SEQUENCES

SVA Language Structure

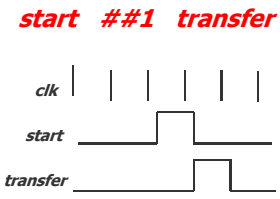
Sequences

- So far we have examined LTL-based assertions
- We now we introduce SVA sequences
  - Multiple Boolean expressions are evaluated in a linear order of increasing time



SVA Language Structure

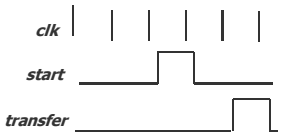
- **Sequence**
  - Temporal delay *##n* with an integer n.



SVA Language Structure

- **Sequence**
  - Temporal delay *##n* with an integer n.

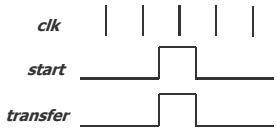
start ##2 transfer



SVA Language Structure

- **Sequence**
  - Temporal delay *##[m:n]* with range [m:n]

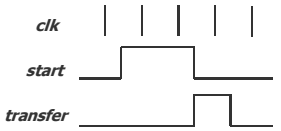
start ##[0:2] transfer



SVA Language Structure

- **Sequence**
  - Consecutive repetition [*m] or range [*m:n]
    - Use $ to represent infinity

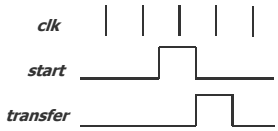
start[*2] ##1 transfer



SVA Language Structure

- **Sequence**
  - Consecutive repetition [*m] or range [*m:n]
    - Use $ to represent infinity

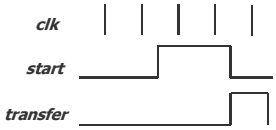
start[*1:2] ##1 transfer



SVA Language Structure

- **Sequence**
  - Consecutive repetition [*m] or range [*m:n]
    - Use $ to represent infinity

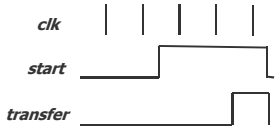
start[*1:2] ##1 transfer



SVA Language Structure

- **Sequence**
  - Consecutive repetition [*m] or range [*m:n]
    - Use $ to represent infinity

start[*1:2] ##1 transfer

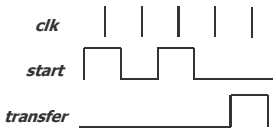


Note: This also matches the sequence specification!!!!

SVA Language Structure

- **Sequence**
  - Non-consecutive repetition [=m] or [=m:n]

start[=2] ##1 transfer



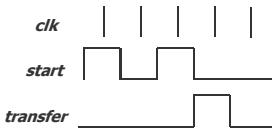
[*] represents zero to infinity

start[=2] ➔ !start[*] ##1 start ##1 !start[*] ##1 start ##1 !start[*]

SVA Language Structure

- **Sequence**
  - Goto non-consecutive repetition [->m] or [->m:n]

start[->2] ##1 transfer

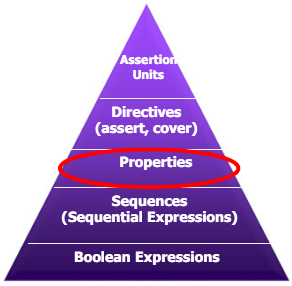


[*] represents zero to infinity

start[->2] ➔ !start[*] ##1 start ##1 !start[*] ##1 start

SVA Language Structure

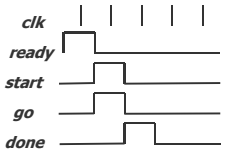
- Properties



SVA Language Structure

- Properties
  - Overlapping sequence implication operator |->

ready ##1 start |-> go ##1 done

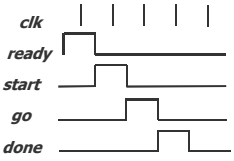



```
assertion property ( @(posedge clk) ready ##1 start |-> go ##1 done );
```


SVA Language Structure

- Properties
  - Non-overlapping sequence implication operator |=>

ready ##1 start |=> go ##1 done



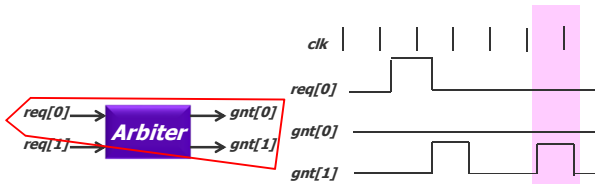
NOTE: A |=> B is the same as A |-> ##1 B

Fair Arbitration Scheme Example

- Asserting that an arbiter is fair
  - To be fair, a pending request for a particular client should never have to wait more than two arbitration cycles
  - Otherwise, the arbiter unfairly issued multiple grants to a different client

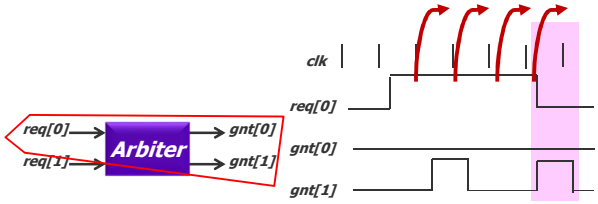
Fair Arbitration Scheme Example

```
a_0_fair:
    assert  property (@(posedge clk) disable iff (reset)
        $rose(req[0]) |-> not (!gnt[0] throughout (gnt[1])[->2]));
```



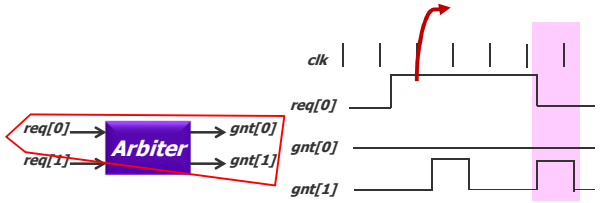
Fair Arbitration Scheme Example

```
a_0_fair:
    assert  property (@(posedge clk) disable iff (reset)
        req[0] |-> not (!gnt[0] throughout (gnt[1])[->2]));
```



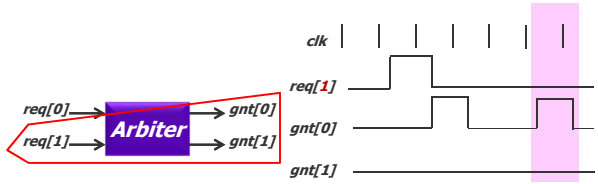
Fair Arbitration Scheme Example

```
a_0_fair:
    assert  property (@(posedge clk) disable iff (reset)
        $rose(req[0]) |-> not (!gnt[0] throughout (gnt[1])[->2]));
```



Fair Arbitration Scheme Example

```
a_1_fair:
    assert  property (@(posedge clk) disable iff (reset)
        $rose(req[1] |-> not (!gnt[1] throughout (gnt[0])[->2]));
```



SVA Language Structure

- Named sequences and properties
  - To facilitate reuse, properties and sequences can be declared and then referenced by name
  - Can be declared with or without parameters

```
sequence s_op_retry;
  (req ##1 retry);
endsequence
sequence s_cache_fill(req, done, fill);
  (req ##1 done [=1] ##1 fill);
endsequence
```


SVA Language Structure

- Named properties and sequences

```
sequence s_op_retry;
  (req ##1 retry);
endsequence
sequence s_cache_fill(rdy, done, fill);
  (rdy ##1 done [=1] ##1 fill);
endsequence
assert property ( @(posedge clk) disable iff (reset)
  s_op_retry |=> s_cache_fill (my_rdy,my_done,my_fill));
```


SVA Language Structure

- Named properties and sequences

```
property p_en_mutex(en0, en1);
  @(posedge clk) disable iff (reset)
      ~(en0 & en1);
endproperty

assert property (p_en_mutex(bus_en0, bus_en1));
```


SVA Language Structure

- Action blocks
  - An SVA action block specifies the actions that are taken upon success or failure of the assertion
  - The action block, if specified, is executed immediately after the evaluation of the assert expression

```
assert property ( @(posedge clk) disable iff (reset)
      !(grant0 & grant1) )
else begin // action block fail statement
    $error("Mutex violation with grants.");
end
```

SVA Language Structure

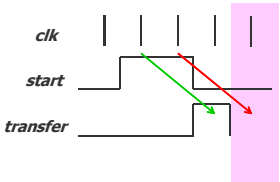
- System functions

  - **$rose**(expression)
  - **$fell**(expression)
  - **$stable**(expression)
  - **$past**(expression [, number_of_ticks])

The need for $rose system function

- You must be precise when specifying!

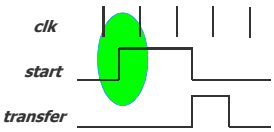
```
assertion property ( @(posedge clk) start |-> ##2 Transfer);
```



Eliminates multiple matches

- You must be precise when specifying!

```
assertion property ( @(posedge clk) $rose(start) |-> ##2 Transfer);
```



$rose(start) is a short cut for the sequence ***!start ##1 start***

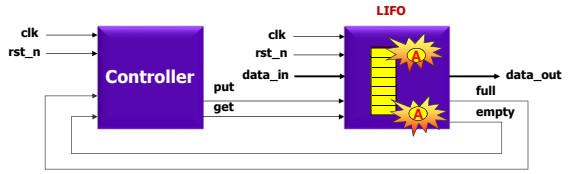
SVA Language Structure

- System functions

  - $onehot (<expression>)
    - Returns true if only one bit of the expression is high
  - $onehot0 (<expression>)
    - Returns true if at most one bit of the expression is high
  - $isunknown (<expression>)
    - Returns true if any bit of the expression is X or Z
    - This is equivalent to ^<expression> === 'bx

Introduction to SVA

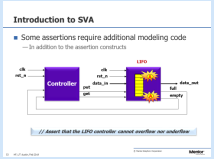
- Some assertions require additional modeling code
  - In addition to the assertion constructs



// Assert that the LIFO controller cannot overflow nor underflow

Introduction to SVA

```
// assertion modeling code – not part of the design
`ifdef ASSERT_ON
int cnt = 0;
always @(posedge clk)
 if (!rst_n)
    cnt <= 0;
 else
    cnt <= cnt + put – get;
// assert no LIFO overflow
assert property (@posedge clk disable iff (~rst_n)
                                !((cnt + put – get) > `DEPTH));
// assert no LIFO underflow
assert property (@posedge clk disable iff (!rst_n) !((cnt + put) < get));
`endif
```



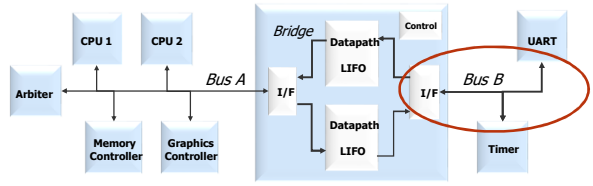
Note: rst_n is an active low reset in this example

SVA Does and Don’ts

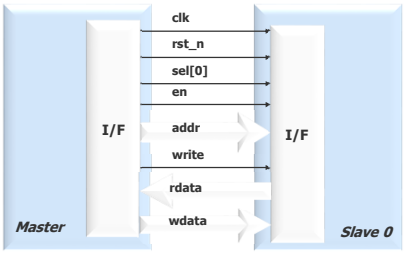
- Never *assert* a sequence!
  assert property (@posedge clk) (req ##1 grnt ##1 done));
  - This says every clock we see *req*, followed by *gnt*, followed by *done*
  - The correct way to do this is with an implication operator:
  assert property (@posedge clk) (req |=> grnt ##1 done));
- It’s ok to *cover* a sequence
- It’s ok to assert a forbidden sequence using *not*
  assert property (@posedge clk) not (req ##1 done ##1 grant));

BUS-BASED DESIGN EXAMPLE

Bus-Based Design Example

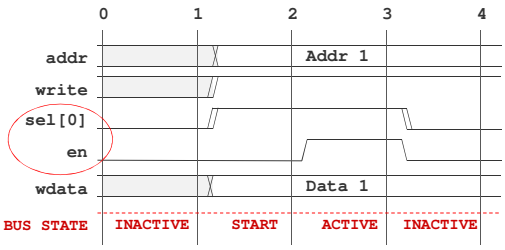


Nonpipelined Bus Interface

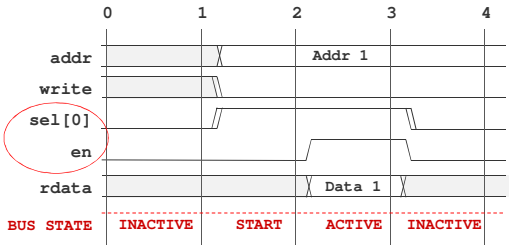


Note: rst_n is an active low reset in this example

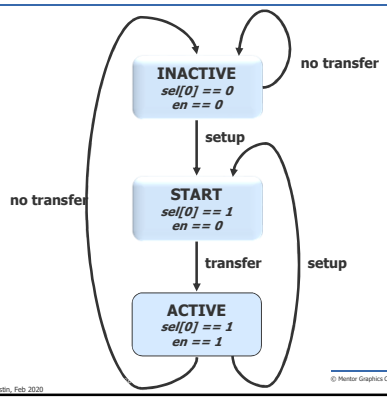
Non-Burst Write Transaction



Non-Burst Read Transaction

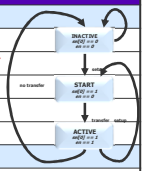


Conceptual Bus States



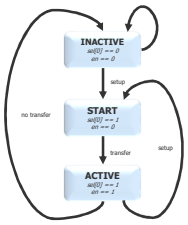
Interface Requirements

Property Name	Description
Bus legal treansitions	
p_state_reset_inactive	Initial state after reset is INACTIVE
p_valid_inactive_transition	INACTIVE is followed by INACTIVE or START
p_valid_start_transition	START is followed by ACTIVE
p_valid_active_transition	ACTIVE is followed by INACTIVE or START
p_no_error_state	Bus state must be valid: !(se==0 & en==1)
Bus stable signals	
p_sel_stable	Slave select signals remain stable from START to ACTIVE
p_addr_stable	Address remains stable from START to ACTIVE
p_write_stable	Control remains stable from START to ACTIVE
p_wdata_stable	Data remains stable from START to ACTIVE



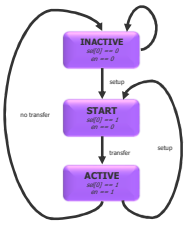
Use Modeling Code to Simplify Coding

```
`ifdef ASSERTION_ON
//Map bus control values to conceptual states
 if (~rst_n) begin
     bus_reset    = 1;
     bus_inactive = 1;
     bus_start    = 0;
     bus_active   = 0;
     bus_error    = 0;
 end
 else begin
     bus_reset    = 0;
     bus_inactive = ~sel & ~en;
     bus_start    =  sel & ~en;
     bus_active   =  sel & en;
     bus_error    = ~sel & en;
 end
`endif
```



SVA Examples

```
property p_valid_inactive_transition;
  @(posedge clk) disable iff (bus_reset)
   ( bus_inactive) |=>
              ((bus_inactive) || (bus_start));
endproperty
a_valid_inactive_transition:
  assert property (p_valid_inactive_transition);

property p_valid_start_transition;
  @(posedge clk) disable iff (bus_reset)
    (bus_start) |=> (bus_active);
endproperty
a_valid_start_transition:
  assert property (p_valid_start_transition);
```



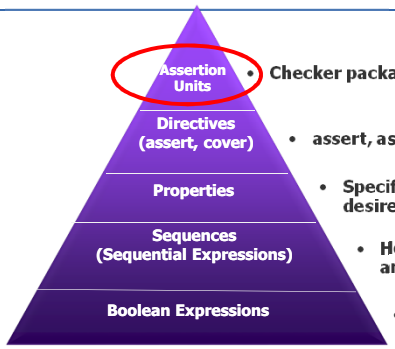
Instantiating Assertions within Modules

```
module bus_controller (. . .);
    . . .
    always (@posedge clk) begin
      . . . .
    end

    always (@posedge clk) begin
      . . . .
    end

    assert property (p_valid_start_transition);
endmodule
```

Implicit **always**

CHECKER PACKAGING

SVA Language Structure



- Checker packaging
- assert, assume, cover
- Specification of behavior; desired or undesired
- How Boolean events are related over time
- True or false

SVA Checker

```
checker seq_protocol (start, complete, dataIn, dataOut, event clk);

    default clocking @clk; endclocking
    var type(dataIn) data;

    property match (first, last);    first |=> !first until_with last;   endproperty

    always_ff @clk       if (start) data <= dataIn;

    a_data_check: assert property (complete |-> dataOut == data);
    a_no_start: assert property (match(start, complete));
    a_no_complete: assert property (match(complete, start));

    initial
        a_initial_no_complete: assert property (!complete throughout start[->1]);
endchecker : seq_protocol
```

Source: Dmitry Korchemny, "SystemVerilog Assertions for Formal Verification," HVC2013

Binding Checkers

```
module top;
logic clock, snda, sndb, sndc, rcva, rcvb, rcvc;
...
trans ta (clock, snda, rcva);
trans tb (clock, sndb, rcvb);
trans #(2) tc (clock, sndc, rcvc);
endmodule : top
```

```
checker eventually_granted (req, gnt, ...);
...
endchecker : eventually_granted

checker request_granted (req, gnt, n, ...);
...
endchecker : request_granted
```

```
module trans #(DEL=1) (input logic clock, in, output logic out);
    if (DEL == 1) begin : b
        always @(posedge clock) out <= in;
    end
    else begin : B
        logic [DEL - 2: 0] tmp;
        always @(posedge clock) begin
            tmp[0] <= in;
            for (int i = 1; i < DEL - 1; i++) tmp[i] <= tmp[i-1];
            out <= tmp[DEL - 2];
        end
    end
endmodule : trans
```

```
bind trans eventually_granted check_in2out(in, out, posedge clock);
bind trans: ta, tb request_granted delay1(in, out,, posedge clock);
bind trans: tc request_granted delay2(in, out, 2, posedge clock);
```

Source: Dmitry Korchemny, "SystemVerilog Assertions for Formal Verification," HVC2013

EXERCISES

Ex.1: Simple Shift Buffer Example

- After reset, the input *d_in* should never be unknown.

Ex.1: Signal is Valid After Reset

- After reset, the input *d_in* should never be unknown.

```
a_d_in_never_x:  assert property (@(posedge clk) disable iff (reset)
                                  (d_in !== 1'bx));
```

Ex.2: One-Cold State Machine

- After reset, *state[7:0]* must have only a single bit low.

state: 11101111, 1011111, 0111111, 11111110, …

Ex.2: One-Cold FSM

- After reset, *state[7:0]* must have only a single bit low.

state: 11101111, 1011111, 0111111, 11111110, …

```
a_one_cold_fsm: assert property (@(posedge clk) disable iff (reset)
                                $onehot(~state));
```


Ex.3: Simple Handshaking Protocol

- Whenever *start* is high, then *start* must be low in the next cycle and remain low until after the next strictly subsequent cycle in which *complete* is high.
- *complete* may not be high unless *start* was high in a preceding cycle and *complete* was not high in any of the intervening cycles.

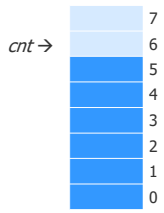
Ex.3: Simple Handshaking Protocol

- Whenever *start* is high, then *start* must be low in the next cycle and remain low until after the next strictly subsequent cycle in which *complete* is high.
- *complete* may not be high unless *start* was high in a preceding cycle and *complete* was not high in any of the intervening cycles.

```
a_no_start: assert property (@(posedge clk) disable iff (reset)
    start |=> !start throughout complete[->1]
);
a_no_complete: assert property (@(posedge clk) disable iff (reset)
    complete |=> !complete throughout start[->1]
);
```

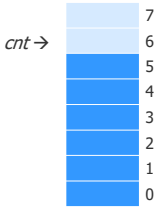
Ex.4 Stack (LIFO)

- A LIFO contains the following controls:
  - *put* : add data to LIFO
  - *get* : remove data from LIFO
  - *cnt* counter that points to the next available location in the LIFO (4'b1000 represents full)
- It is not possible to overflow the LIFO
- It is not possible to underflow the LIFO



Ex.4 Stack (LIFO)

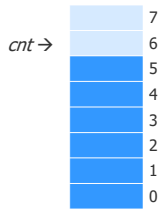
- A LIFO contains the following controls:
  - *put* : add data to LIFO
  - *get* : remove data from LIFO
  - *cnt* counter that points to the next available location in the LIFO (4'b1000 represents full)




```
a_no_overflow: assert property
    (@(posedge clk) disable iff (reset)
      !(cnt == 4'b1000 & put & !get)
    );
```


Ex.4 Stack (LIFO)

- A LIFO contains the following controls:
  - *put* : add data to LIFO
  - *get* : remove data from LIFO
  - *cnt* counter that points to the next available location in the LIFO (4'b1000 represents full)




```
a_no_underflow: assert property
    (@(posedge clk) disable iff (reset)
      !(cnt == 4'b0000 & !put & get)
    );
```


SUMMARY

Lecture Recap

In this lecture, I discussed. . .

- Discussed the structure of the SVA language
- Discussed how to construct sequence
- Discussed how to construct properties
- Demonstrate SVA on real examples
- Discussed Checkers and Bind
- Exercises
- Summary

More Info on Industry Verification Trends

- http://go.mentor.com/55d6T

Mentor®
A Siemens Business

www.mentor.com