Class 12 Topics

• What is an assertion?
• SystemVerilog assertions
• Immediate assertions
• Concurrent assertions
• Collecting functional coverage with assertions
• Properties
• Sequences
What is an assertion?

• An assertion specifies a behavior of the system.
• Assertions can be used for:
  • Verifying the behavior of a design
  • Means for functional coverage
  • Provide input stimulus for verification
• Assertions can be written in:
  • Verilog
  • SVA – SystemVerilog Assertions
  • PSL – Property Specification Language
• Does your current project use assertion languages or assertion libraries? (Yes/No) If yes, which one(s) are you using?
  don't use :  31.5%
  IBM Sugar/PSL :  14.9%
  0-In CheckerWare :  6.8%
  System Verilog SVA :  37.8%
  Synopsys Vera OVA :  9.6%
  Verplex OVL/IAL :  9.3%
  VHDL Assertions :  7.0%
  homegrown :  8.2%
Where can SVA be used?

• SystemVerilog
• Verilog 2001
• Even VHDL
• In RTL
• In a testbench
  • For the testbench
  • For the RTL
SVA in a Verification Strategy

• Provide added white box visibility to the design

• Useful for both the provider and user of IP
• Provides quick feedback if a simulation has gone awry
• Continuously running verification checkers in a task based testbench.
• Checkers that would be very difficult in Verilog.
SystemVerilog Assertions

• An assertion written in SVA requires less extra syntax to learn if your testbench is in SystemVerilog

• 2 types of assertions
  • Immediate - Operates in procedural code
  • Concurrent - Operates outside procedural code

Property to check: If request is 1 on the positive edge of the clock grant must be 1 on the next positive edge of the clock.

```verilog
always @(posedge clk) begin
  if (request) begin
    @(posedge clk);
    if (grant != 1)
      $display("%t: Error, grant != 1", $time);
  end
end
```
Immediate Assertions

• Immediate assertions operate in procedural code

Syntax:
<assertion name> : assert(expression)
    <action if true>;
else
    <action if false>;

always @(posedge clk) begin
    if (request) begin
        repeat (1) @(posedge clk);
        reg_grnt_assert: assert (grant == 1)
            $display("%t: Passed", $time);
    else
        $display("%t: Failed", $time);
    end
end
Do these Immediate Assertions work?

always @(posedge clk) begin
    if (request) begin
        @(posedge clk);
        if (grant != 1)
            $display("%t: Error,.....
    end
end

always @(posedge clk) begin
    if (request) begin
        repeat (1) @(posedge clk);
        reg_grnt_assert: assert (grant == 1)
            $display("%t: Passed", $time);
        else
            $display("%t: Failed", $time);
    end
end

Always block entered  Always block entered

# 250: Error, grant != 1
# 450: Error, grant != 1
Immediate Assertions (cont.)

The <action if true> expression is optional

```verilog
reg_grnt_assert: assert (grant == 1)
else
    $display("%t: Failed", $time);
```

The <action if false> expression is optional

```verilog
reg_grnt_assert: assert (grant == 1)
    $display("%t: Passed", $time);
```

Do not need <action if false> or <action if true>

```verilog
reg_grnt_assert: assert (grant == 1)
```
Immediate Assertion Severity

By default the severity of an assertion failure is error.

```
reg_grnt_assert: assert (grant == 1)
else
$error("Failed");
```

# ** Error: Failed

Other severities are $fatal, $warning or $info.
# ** Info: Failed
Exercise 1

Design an immediate assertion that:

1. If the signal $branch\_inst = 1$ on the negative edge of the clock, the signal $jump\_done$ must $= 1$ two cycles later on the negative edge of the clock.

2. If a failure occurs specify the severity to be a warning.

3. If a failure does not occur specify the severity to be an info.

4. Name the assertion $branch\_done$
Exercise 1

Design an immediate assertion that:
1. If the signal $\text{branch\_inst} = 1$ on the negative edge of the clock, the signal $\text{jump\_done}$ must $= 1$ two cycles later on the negative edge of the clock.
2. If a failure occurs specify the severity to be a warning.
3. If a failure does not occur specify the severity to be an info.
4. Name the assertion $\text{branch\_done}$

always @(negedge clk) begin
    if (branch_inst) begin
        repeat (2) @(negedge clk);
        branch_done: assert (jump_done == 1)
            $info("%t: jump\_done = 1 two cycle after branch\_inst", $time);
    else
        $warning("%t: jump\_done not= 1 two cycle after branch\_inst", $time);
    end
end
Concurrent Assertions

• Immediate assertions are no more powerful than coding in Verilog
• Concurrent assertions can describe behavior that spans time.
• Concurrent assertions evaluate on a defined edge.
• Defined outside procedural code.

Syntax:
<assertion name> : assert property (expression)
    <action if true>;
else
    <action if false>;

```
reg_grnt_assert: assert property (@(posedge clk) (request |-> ##1 grant))
    $display("%t: Passed", $time);
else
    $display("%t: Failed", $time);
```

Equivalent to `reg_grnt_assert: assert property (@(posedge clk) (request |=> grant));`
Does this Concurrent Assertion work?

```
reg_grnt_assert: assert property (@(posedge clk) (request |-> ##1 grant))
  $display("%t: Passed", $time);
else
  $display("%t: Failed", $time);
```

Assertion inactive

Assertion begins
disabling with *disable iff*

- Continuous assertions can be turned off with *disable iff*
- Typical usage is during reset.

```vhdl
reg_grnt_assert: assert property (@(posedge clk) disable iff (reset) (request |-> ##1 grant));
```
Exercise 2

Re-write the immediate assertion in exercise 1 so that:
1. It is a concurrent assertion
2. It is disabled if input \textit{testmode} is asserted.
3. Remove the additional information on success or failure.

\begin{verbatim}
branch_done: assert property (@(negedge clk) disable iff (testmode) (branch_inst |-> ##2 jump_done));
\end{verbatim}
$\text{rose and } $fell

$\text{reg}_{\text{grnt}}$: assert property (@(posedge clk) (request -> ##1 grant));

- $\text{rose}$ detects a transition to 1
- $\text{fell}$ detects a transition to 0

$\text{reg}_{\text{grnt}}$: assert property (@(posedge clk) ($\text{rose(request)} |\rightarrow $ #1 $\text{rose(grant)}$));
Delay Expressions

- A ## followed by a number or range specifies the delay from the current clock to the beginning of the sequence that follows.

\[
\text{A_B_C_D: assert property (@(posedge clk) (A ##1 B ##1 C ##1 D))};
\]

- Specify large argument to # to span more clocks

\[
\text{A_D: assert property (@(posedge clk) (A ##3 D))};
\]
Implication

• The implication construct specifies that the checking of a property is performed conditionally
  1. \( |-\rightarrow \) Overlapped implication
  2. \( |=\rightarrow \) Non-overlapped implication

\[
\text{request} \, |=\rightarrow \, \text{grant} \quad \text{equivalent} \quad \text{request} \, |-\rightarrow \, ##1 \, \text{grant}
\]

\[
\text{A_B_C_D: assert property } \, @(\text{posedge clk}) \, (A \, |-\rightarrow \, B \, |=\rightarrow \, C \, |=\rightarrow \, D));
\]
Implication Can Pass Vacuously

- antecedent seq_expr |-> / |=> (<consequent seq_expr>);
  - If the antecedent matches, the consequent must too.
  - If the antecedent fails, the consequent is not tested and a true result is forced.
    - Such forced results are called “vacuous” and are usually filtered out.

Source: Mentor Graphics training
Implication Can Pass Vacuously

• The implication might never start which means it never can fail

A_B_C_D: assert property (@(posedge clk) (A |=> B |=> C |=> D));

• Or it can start and stop but never fail
Mixing Implication and Delay

- A mix of implication and delay expressions might be required

A_B_C_D: assert property (@(posedge clk) (A |=> (B ##1 C ##1 D)));
Coverage and Assertions

• Any concurrent assertion can be used to collect coverage.

```vhdl
req_grant: assert property (@(posedge clk) (request |=> grant));
```

```vhdl
req_grant_cover: cover property (@(posedge clk) (request |=> grant));
```
Exercise 3

Re-write the concurrent assertion in exercise 2 so that:

1. Coverage is collected
2. If you used an overlapped implication operator (|->) use a non-overlapped implication operator (|=>) or vise versa.
3. Remove the disable

```plaintext
branch_done: cover property (@(negedge clk) (branch_inst |=> ##1 jump_done));
or
branch_done: cover property (@(negedge clk) (branch_inst |-> ##2 jump_done));
```
Viewing SVA Coverage Directives
Properties

• A property can be defined separate from an assertion.

```verbatim
req_grant: assert property (@(posedge clk) (request |=> grant));
```

```verbatim
property req_grant;
    (request |=> grant);
endproperty
```

• A property can be used in both an assertion and a cover directive

```verbatim
assert property (@(posedge) (req_grant));
cover property (@(posedge) (req_grant));
```
Separating Clock from Assertion

- A default clock can be defined separate from an assertion.

```verilog
req_grant: assert property (@(posedge clk) (request |=> grant));
```

- A clock can be defined in a property

```verilog
property req_grant;
    @(posedge clk) (request |=> grant);
endproperty
assert property (req_grant);
cover property (req_grant);
```
Sequences

• Complex sequential behaviors can be described by a sequence
• A sequence is part of a property.

A_B_C_D: assert property (@(posedge clk) (A |=> (B ##1 C ##1 D)));
Exercise 4

Re-write the concurrent assertion in exercise 3 to:
1. Specify a default clock
2. Express the sequence, property and assertion separately.

```verilog
default clocking cb @(negedge clk); endclocking

sequence jump_done_seq;
  ##2 jump_done; or ##1 jump_done if using |=> on property
endsequence

property branch_done_prop;
  (branch_inst |-> jump_done_seq);
endproperty

branch_done: assert property (branch_done_prop);
```
Sequence Range

• A sequence range can be expressed

sequence B_C_D;
    (B ##1 C ##[1:3] D);
endsequence

A_B_C_D_assert: assert property (@(posedge clk) (A |=> B_C_D));
Sequence Repeat

• A sequence repeat can be expressed

sequence B_C_D;
  (B ##1 C ##1 D[*3]);
endsequence
A_B_C_D_assert: assert property (@(posedge clk) (A |=> B_C_D));
A mix of sequence range and repeat can be expressed

sequence B_C_D;
    (B ##1 C ##1 D[*2:3]);
endsequence

A_B_C_D_assert: assert property (@(posedge clk) (A |=> B_C_D));
Sequences with 0 Ranges

• A 0 range can be used if an event might not occur

sequence B_C_D;
  (B ##1 C[*0:2] ##1 D);
endsequence

A_B_C_D_assert: assert property (@(posedge clk) (A |=> B_C_D));
Sequences with Infinite Ranges

- A $ range can be used if a delay or event can occur infinitely long

```verilog
sequence B_C_D;
   (B ##1 C [2:$] D);
endsequence
A_B_C_D_assert: assert property @(posedge clk) (A |=> B_C_D);
```
Exercise 5

Re-write the assertion in exercise 4 to:

1. Require that jump_done can assert from 2 to 4 cycles after branch_done asserts for a
   A. Non-overlapped (|=>) property
   B. An overlapped (|->) property
Exercise 4

Re-write the concurrent assertion in exercise 3 to:
1. Specify a default clock
2. Express the sequence, property and assertion separately.

```
default clocking cb @(negedge clk); endclocking

sequence jump_done_seq;
    ##2 jump_done; or ##1 jump_done if using |=> on property
endsequence

property branch_done_prop;
    (branch_inst |-> jump_done_seq);
endproperty

branch_done: assert property (branch_done_prop);
```
Goto Repetition

• The [ operator indicates that the sequence is consecutive

\[(B \#\#1 C[*3] \#\#1 D);\]

• The [-> operator indicates that the sequence may not be consecutive

sequence B_C_D;

\[(B \#\#1 C[->3] \#\#1 D);\]
endsequence
A_B_C_D_assert: assert property \((@\text{(posedge clk)} \ (A |=> \ B_C_D))\);
Non-Consecutive Repetition

• [-] > indicates that the sequence **must** continue on the last match
  \[(B \##1 C[->3] \##1 D)\];

• [=] indicates that the sequence **may** not continue consecutively
  \[(B \##1 C[=3] \##1 D)\];
Sequences

• SystemVerilog sequence operators
  [* ] [= ] [ -> ]
  ##
  throughout
  within
  intersect
  and
  or
• A sequence can be declared in
  — A module
  — An interface
  — A program
  — A clocking block
  — A package
  — A compilation-unit scope
Sequences Examples: testbench.in

throughout

The throughout operator requires a Boolean to be true throughout a sequence
Sequences (cont.)

within

One sequence can fully contain another sequence

EXAMPLE:
Seq1 within seq2
Sequences (cont.)

intersect

Sequences must start at the same tie and end at the same time. Match done at end time.
Sequences (cont.)

and

Sequences must start at same time and end at any time. Match is done after last sequence has ended.

or

Sequence must start at the same time and can end at any time. Match done at both sequences’ ends.
Viewing Assertions in QuestaSim