The American University in Cairo

CENTER OF NANOELECTRONICS & DEVICES

CND212: Digital Testing and Verification

SystemVerilog Assertions

- Assertions are primarily used to validate the behaviour of a design.
  ("Is it working correctly?")
- They may also be used to provide functional coverage information for a design.
  ("How good is the test?")
- Assertions can be checked dynamically by simulation, or statically by a separate property checker tool.
- There are two types of assertions: **immediate** (assert) and **concurrent** (assert property).

Who writes Assertions

- Design Engineers
  - _Capture design assumptions
  - _Record design intentions and internal error conditions
  - _Design behavior
- Verification Engineers
  - _Interface, cross-block, higher level behavior
- IP Providers

Immediate Assertions

- Immediate assertions are procedural statements and are mainly used in simulation.
- An assertion is basically a statement that something must be true, similar to the if statement.

```
if (A == B) ...  // Simply checks if A equals B
assert (A == B); // Asserts that A equals B; if not, an error is generated
```

- If the conditional expression of the immediate assert evaluates to X, Z or 0, then the assertion fails and the simulator writes an error message.

```
"testbench.sv", 16: tb.unnamed$$_0: started at 0ns failed at 0ns
        Offending '(A == B)'
```

Immediate Assertions

- An immediate assertion may include a pass and/or fail statements.
- Three severity system tasks can be included in the fail statement to specify a severity level: $error, $fatal, $warning.
- In addition, the system task $info indicates that the assertion failure carries no specific severity.

```
assert (A == B)
  $display ("OK. A equals B");
  else $error("It's gone wrong");
```

```
"testbench.sv", 16: tb.unnamed$$_0: started at 0ns failed at 0ns
        Offending '(A == B)'
Error: "testbench.sv", 16: tb.unnamed$$_0: at time 0 ns
It's gone wrong
```

Concurrent Assertions

- Concurrent assertions check the sequence of events spread over multiple clock cycles.
- SystemVerilog concurrent assertion statements can be specified in a module, interface or program block and **running concurrently with other statements**.
- Concurrent assertions usually appear outside any initial or always blocks in modules, interfaces and programs.
- Sampling of variables is done in the **proponed** region and evaluation of the expression is done in the **observed** region of the simulation scheduler.
- The Keyword “property” differentiates the immediate assertion from the concurrent assertion.

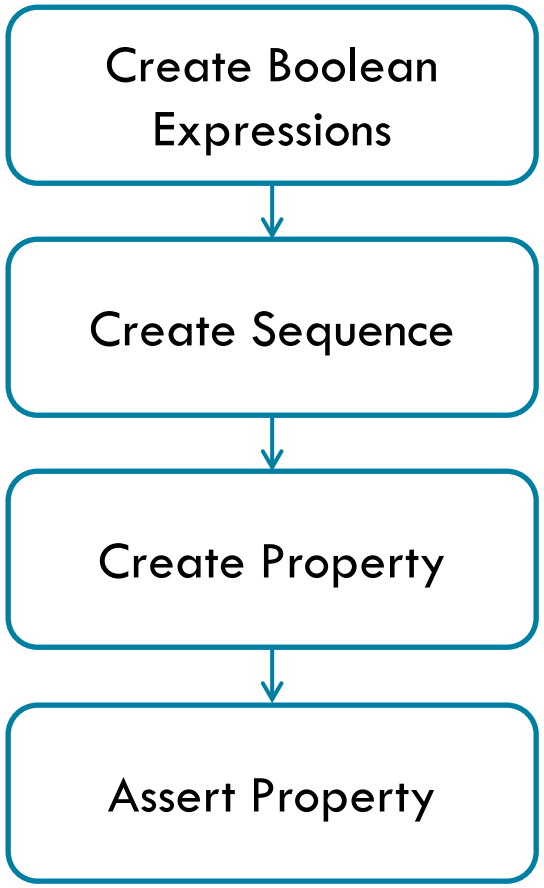
Concurrent Assertions

- **Property:** "A Request should be followed by an Acknowledge occurring no more than two clocks after the Request is asserted."

```
assert property (@(posedge clock) Req |-> ##[1:2] Ack);
```

- Concurrent assertion is only checked when a rising clock edge has occurred; the values of Req and Ack are sampled on the rising edge of the Clock.

Building blocks of SVA




```
sequence seq;
  a ##2 b;
endsequence

property p;
  @(posedge clk) seq;
endproperty

a_1 : assert property(p);
```

Implication

- The implication construct (|->) allows a user to monitor sequences based on satisfying some criteria.
- attach a precondition to a sequence and evaluate the sequence only if the condition is successful.
- The left-hand side of the implication is called the “antecedent” and the right-hand side is called the “consequent.”
- The antecedent is the gating condition. If the antecedent succeeds, then the consequent is evaluated.
- There are 2 types of implication:
    1. Overlapped implication |->
    2. Non-overlapped implication |=>

Overlapped implication

- The overlapped implication is denoted by the symbol |->.
- If there is a match on the antecedent, then the consequent expression is evaluated in the **same clock cycle**.

```
property p;
    @(posedge clk) a |-> b;
endproperty

a: assert property(p);
```

- The above property checks that, if signal “a” is high on a given positive clock edge, then signal “b” should also be high on the same clock edge.

Non-overlapped implication

- The non-overlapped implication is denoted by the symbol |=>.

- If there is a match on the antecedent, then the consequent expression is evaluated in the **next clock cycle**.

```
property p;
  @(posedge clk) a |=> b;
endproperty

a: assert property(p);
```

- The above property checks that, if signal “a” is high on a given positive clock edge, then signal “b” should be high on the next clock edge.

Implication

- The implication with a fixed delay on the consequent.

```
property p;
  @(posedge clk) a |-> ##2 b;
endproperty
a: assert property(p);
```

- The above property checks that, if signal “a” is high on a given positive clock edge, then signal “b” should be high after 2 clock cycles.

Implication

- Below property checks that, if signal “a” is high on a given positive clock edge, then within 1 to 4 clock cycles, the signal “b” should be high.

```
property p;
  @(posedge clk) a |-> ##[1:4] b;
endproperty

a: assert property(p);
```

- Below property checks that, if signal “a” is high on a given positive clock edge, then signal “b” should be high in the same clock cycle or within 4 clock cycles.

```
property p;
  @(posedge clk) a |-> ##[0:4] b;
endproperty

a: assert property(p);
```

Repetition

```
property p;
  @(posedge clk) a |-> ##1 b ##1 b  ##1 b;
endproperty

a: assert property(p);
```

- The above property checks that, if the signal “a” is high on given posedge of the clock, the signal “b” should be high for 3 consecutive clock cycles.

```
property p;
  @(posedge clk) a |-> ##1 b[*3];
endproperty
a: assert property(p);
```

Repetition

```
property p;
  @(posedge clk) a |-> ##1 b[->3] ##1 c;
endproperty
a: assert property(p);
```

- The above property checks that, if the signal “a” is high on given posedge of the clock, the signal “b” should be high for 3 clock cycles followed by “c” should be high after ”b” is high for the third time.

Repetition

```
property p;
  @(posedge clk) a |-> ##1 b[->3] ##1 c;
endproperty
a: assert property(p);
```

- The above property checks that, if the signal “a” is high on given posedge of the clock, the signal “b” should be high for 3 clock cycles followed by “c” should be high after ”b” is high for the third time.

disable iff

- In certain design conditions, we don't want to proceed with the check if some condition is true. this can be achieved by using disable iff.

```
property p;
  @(posedge clk)
  disable iff (reset) a |-> ##1 b[->3] ##1 c;
endproperty

a: assert property(p);
```

- Below property checks that, if the signal "a" is high on given posedge of the clock, the signal "b" should be high for 3 clock cycles followed by "c" should be high after "b" is high for the third time. **During this entire sequence, if reset is detected high at any point, the checker will stop.**

Local variables in A sequence

- One of the powerful features of SVA, It enables information to be stored within the thread of a sequence and then tested at later cycles within this sequence. Variables can be used in sequence and property where it's declared.

```
sequence s1;
  int tmp_data;
  (##1 data_valid, tmp_data = data_in) ##1 (data_out == tmp_data);
endsequence

sequence s2;
  s1 ##2 (d_data == tmp_data); // error:tmp_data is not accessible even if s1 is
instantiated in the s2.
endsequence
```

SVA Methods

- **$rose:** used to detect the rising edge of a signal. It evaluates to true if the given signal transitions from low to high at the current clock tick.
- **$fell:** the opposite of $rose. It detects the falling edge of a signal. It evaluates to true if the given signal transitions from high to low at the current clock tick.
- **$stable**: returns true if the value of the expression did not change. Otherwise, it returns false.
- **$changed:** the opposite of $stable. It checks if a signal has changed its value between the current and previous clock ticks
- **$past:** The $past function is used to access the value of an expression in the previous clock tick.

Lab task

Lab Task

- Use your lab 5 files “**arbiter.v**”, “**arbiter_io.sv**”, “**test.v**”, and “**arbiter_test_top.sv**”

1) In the testbench (**test.v**), consider you want to check the grant signal is correct when a certain request is sent like shown in the example below.

   Replace the if statement above with an immediate assertion

```
arbif.cb.request <= 2'b01;
repeat (2) @arbif.cb;
if (arbif.cb.grant != 2'b01)
  $display("Error, grant != 2'b01");
```

2) Add a concurrent assertion in your interface (**arbiter_io.sv**) that checks that the arbiter request signal does not have X or Z values except during reset.

Assignment : (~60 mins)

Assignment

1. Write a SystemVerilog assertion to check if a signal valid is high whenever a signal ready is high

2. Write a SystemVerilog assertion to check if a sequence of events a, b, and c occurs in the given order with a onecycle gap between each event. Ensure that the sequence does not trigger if the reset signal is active.

3. Write a SystemVerilog cover directive to measure the coverage of a signal data being greater than zero.

4. Write a SystemVerilog sequence that checks if a signal a is high for at least two clock cycles, followed by a sequence of events b and c, and assert it.

5. Write a SystemVerilog sequence, s1, that checks for the following pattern in a signal sequence:

   _The sequence should start with a rising edge of a signal called data_valid.

   _The sequence should then wait for a delay of 7 clock cycles.

   _After the delay, the signal data_out should be equal to the value stored in data_in before the delay.

Assignment

6. Consider the following SystemVerilog sequence: S1 ##2 S2[=1:2] ##1 S3 ##1 S4[->1:3] ##1 S5.

Your task is to analyze the given sequence and identify the conditions that will result in the sequence succeeding or failing.

Write a comprehensive explanation that includes:

_The conditions under which the sequence will succeed.

_The conditions that will cause the sequence to fail.

_Provide specific examples for both success and failure cases to demonstrate your understanding.