N26F300 VLSI SYSTEM DESIGN (GRADUATE LEVEL)

SystemVerilog Assertions

Outline

- Immediate assertions
- Concurrent assertions
- Sequence specifications
- Property specifications

Overview

- An assertion (斷言) specifies a behavior of the system.
- Assertions are primarily used to validate the behavior of a design.
- In addition, assertions can be used to provide functional coverage and to flag that input stimulus, which is used for validation, does not conform to assumed requirements.
- An assertion is a design condition that you want to make sure never violates.

Checker Function Description

- Desired checks
  - Verify if signal “a” is high in the current clock cycle, then signal “b” should be high within 1 to 3 clock cycles.

Verilog v.s. SystemVerilog

```
always @(posedge a)
begin
  repeat (1) @(posedge clk);
    fork: a_to_b










     join
end
```

```
a_to_b_chk:
  assert property
  @(posedge clk) $rose(a) |-> ##[1:3] $rose(b);
```

Overview

- An assertion appears as a statement that states the verification function to be performed. The statement shall be one of the following kinds:
  - `assert`, to specify the property as an obligation for the design that is to be checked to verify that the property holds. If violated, report it.
  - `assume`, to specify the property as an assumption for the environment. Simulators check that the property holds, while formal tools use the information to generate input stimulus. Tell the tool, what inputs are legal.
  - `cover`, to monitor the property evaluation for coverage. If hit, report it.
  - `restrict`, to specify the property as a constraint on formal verification computations. Simulators do not check the property.

Overview: Immediate Assertion

- There are two kinds of assertions: immediate and concurrent.
  - Immediate assertions follow simulation event semantics for their execution and are executed like a statement in a procedural block. There is no immediate restrict assertion statement. Test for a condition at the current time.

```
always_comb
begin
  a_ia: assert (a && b);
end
```

Source: C. M. Huang / SystemVerilog

Overview: Concurrent Assertions

- Concurrent assertions are based on clock semantics and use sampled values of variables. Test for a sequence of events over multiple clock cycles.

```
a_cc: assert property (@(posedge clk)
                              not (a && b));
```

Overview

- Immediate Assertions
- Concurrent Assertions
  - Boolean
  - Sequences
  - Property

Immediate Assertions

- The immediate assertion statement is a test of an expression performed when the statement is executed in the procedural code.
- The expression is nontemporal and is interpreted the same way as an expression in the condition of a procedural if statement.
- In other words, if the expression evaluates to X, Z, or 0, then it is interpreted as being false, and the assertion statement is said to fail. Otherwise, the expression is interpreted as being true, and the assertion statement is said to pass or, equivalently, to succeed.

Immediate Assertions

- There are two types of immediate assertions, simple immediate assertions and deferred immediate assertions.
- In a simple immediate assertion, pass and fail actions take place immediately upon assertion evaluation.
- In a deferred immediate assertion, the actions are delayed until later in the time step, providing some level of protection against unintended multiple executions on transient or "glitch" values.

Immediate Assertions

- There are three types of immediate assertions: immediate `assert`, immediate `assume`, and immediate `cover`.
- The immediate `assert` statement specifies that its expression is required to hold. Failure of an immediate assert statement indicates a violation of the requirement and thus a potential error in the design. If an assert statement fails and no else clause is specified, the tool shall, by default, call `$error`, unless `$assertfailoff` is used to suppress the failure.

Immediate Assertions

```
assertion_label : assert (expression)
  pass block code;
else
  fail block code;
```

- `assertion_label` : User defined assertion label.
- **`assert`** : SystemVerilog reserve word, used for assertion.
- `expression` : Any valid SystemVerilog expression.
- `pass block code`: Code that gets executed when assertion passes.
- `else` : Optional syntax to specify failed code.
- `fail block code`: Code that gets executed when assertion fails.

Immediate Assertions

- Normally we prefer to print some message for pass or fail block code. SystemVerilog provides various assertions levels for reporting messages as listed below.
  - `$fatal` is a run-time fatal.
  - `$error` is a run-time error.
  - `$warning` is a run-time warning, which can be suppressed in a tool-specific manner.
  - `$info` indicates that the assertion failure carries no specific severity.
- One can use `$display`, or any regular Verilog code, like triggering a event, or incrementing a counter, or calling function in pass/fail block code.

```
// assert01.sv

module assert01();

  logic clk = 0;
  integer a = 0;

  always #5 clk = ~clk;

  initial
    begin
      repeat(5)
        begin
          @(negedge clk);
          a = $random % 2;
        end
      $finish;
    end

  always @(posedge clk)
    begin
      CHECK : assert (a == 0)
        $info();
      else
        $error("a == 1");
    end
endmodule
```

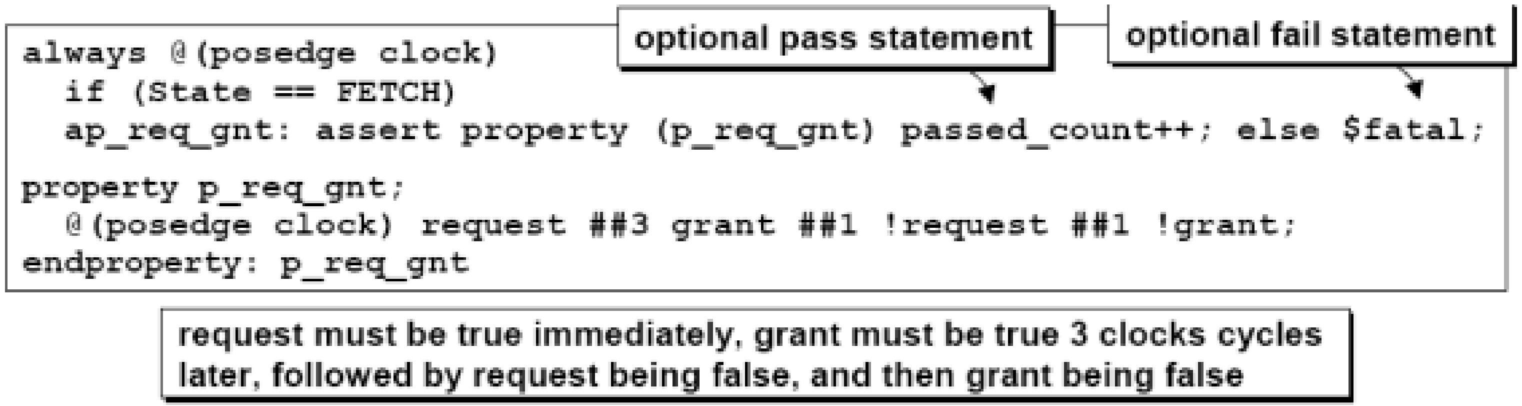
```
ncsim: (time 5 NS)  Assertion assert01.CHECK has passed
ncsim: (time 15 NS) Assertion assert01.CHECK has passed
ncsim: (time 25 NS) Assertion assert01.CHECK has failed
a == 1
ncsim: (time 35 NS) Assertion assert01.CHECK has failed
a == 1
ncsim: (time 45 NS) Assertion assert01.CHECK has failed
a == 1
```

```
// assert02.sv

module assert02();

  integer a, b;

  always_comb begin

    a1 : assert (a == b)
      $info("@%g a1: a = b = %0d", $time, a);
    else
      $error("@%g a1: a = %0d, b = %0d", $time, a, b);

  end

  initial begin
    a = 0; b = 0;
    #10 a = 10;
    #10 b = 10;
    #10 b = 99;
    #10 a = 99;
    #10 $finish;
  end

endmodule
```

```
ncsim: (time  0 FS) Assertion assert02.a1 has passed
@0 a1: a = b = 0
ncsim: (time 10 NS) Assertion assert02.a1 has failed
@10 a1: a = 10, b = 0
ncsim: (time 20 NS) Assertion assert02.a1 has passed
@20 a1: a = b = 10
ncsim: (time 30 NS) Assertion assert02.a1 has failed
@30 a1: a = 10, b = 99
ncsim: (time 40 NS) Assertion assert02.a1 has passed
@40 a1: a = b = 99
```

Concurrent Assertions

- Concurrent assertions describe behavior that spans over time. Unlike immediate assertions, the evaluation model is based on a clock so that a concurrent assertion is evaluated only at the occurrence of a clock tick. (cycle-based)
- The keyword `property` distinguishes a concurrent assertion from an immediate assertion.

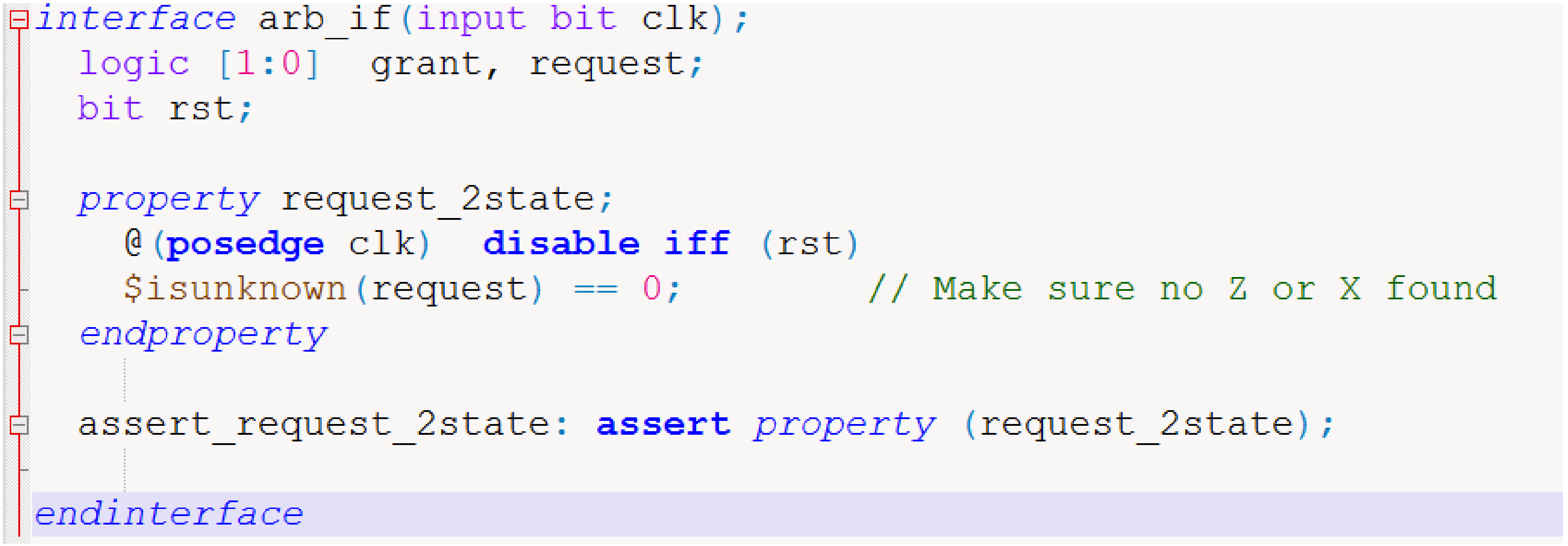


```
always @(posedge clock)
  if (State == FETCH)
  ap_req_gnt: assert property (p_req_gnt) passed_count++; else $fatal;

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

A Simple Example: Assertion in Interface



```
interface arb_if(input bit clk);
  logic [1:0]  grant, request;
  bit rst;

  property request_2state;
    @(posedge clk)  disable iff (rst)
    $isunknown(request) == 0;         // Make sure no Z or X found
  endproperty

  assert_request_2state: assert property (request_2state);

endinterface
```

disable iff (rst) : if the *rst* is detected high at any point during the period that *request* is not Z or X, the checker will stop.

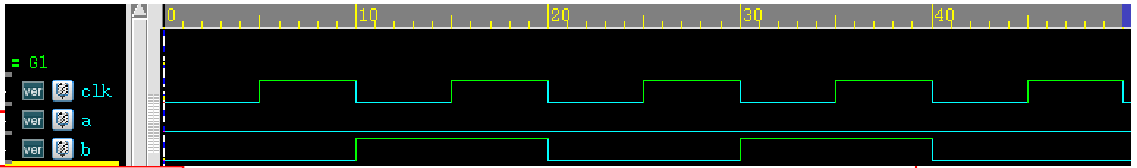
Boolean Layer

- Boolean layer is the lowest layer in concurrent layer, where boolean expression checking is done on variables.
- The result of boolean checking is either true or false. If the variables contain `x` or `z`, then the result is false.

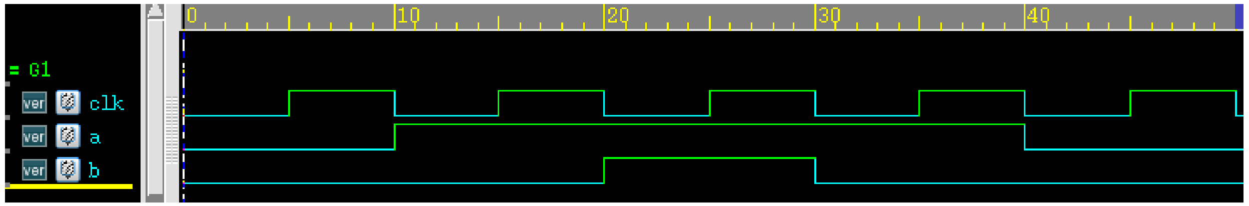
```
ncsim: (time  5 NS) Assertion assert03.a1 has failed
ncsim: (time  5 NS) Assertion assert03.a2 has failed
ncsim: (time 25 NS) Assertion assert03.a2 has failed
ncsim: (time 45 NS) Assertion assert03.a2 has failed
```

```
// assert03.sv
module assert03();

  logic clk, a, b;

  initial
    begin
      clk = 0;
      forever #5 clk = ~clk;          // 5↑,15↑,25↑,35↑,45↑
    end

  initial
    begin
      a = 0; b = 0;                   // 0
      @(negedge clk) b = 1;           // 10↓
      @(negedge clk) b = 0;           // 20↓
      @(negedge clk) b = 1;           // 30↓
      @(negedge clk) b = 0;           // 40↓
      @(negedge clk) $finish;
    end

  initial
    a1 : assert property ( @(posedge clk) a ); // check once only

    a2 : assert property ( @(posedge clk) b ); // check continuously

endmodule
```

race condition



```
// assert04.sv

module assert04();

  logic clk, a, b;

  initial
    begin
      clk = 0;
      forever #5 clk = ~clk;
    end

  initial
    begin
      a = 0; b = 0;              // 0
      @(negedge clk) a = 1;      // 10↓
      @(negedge clk) b = 1;      // 20↓
      @(negedge clk) b = 0;      // 30↓
      @(negedge clk) a = 0;      // 40↓
      @(negedge clk) $finish;
    end

  a1 : assert property ( @(posedge clk) a |=> b );

endmodule
```



```
//5↑, 15↑, 25↑, 35↑, 45↑
```

a |=> b (non-overlapped implication operator)
if a is true, then it must be followed by b is true at the next clock tick. //The start point of the evaluation of consequent property_expr is the clock tick after the end point of the match.//

```
ncsim: (time 35 NS) Assertion assert04.a1 has failed (2 cycles, starting 25 NS)
ncsim: (time 45 NS) Assertion assert04.a1 has failed (2 cycles, starting 35 NS)
```

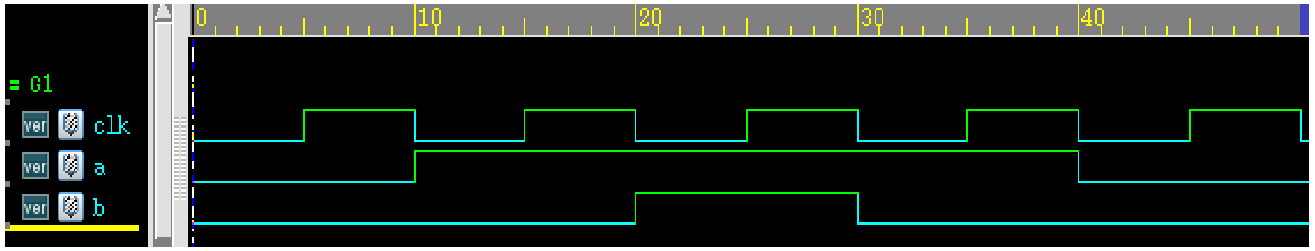
Sequence Layer

- Sequence Layer uses the boolean layer to construct valid sequence of events. The simplest sequential behaviors are linear.
- A linear sequence is a finite list of SystemVerilog boolean expressions in a linear order of increasing time.
- The linear sequence is said to match along a finite interval of consecutive clock ticks *provided the first boolean expression evaluates to true at the first clock tick, the second boolean expression evaluates to true at the second clock tick*, and so forth, up to and including the last boolean expression evaluating to true at the last clock tick.
- A sequence is a series of true/false expressions spread over one or more clock cycles.

## Operator

- A `##` followed by a number or range specifies the cycle delay from the current clock tick to the beginning of the sequence that follows.
- The delay `##1` indicates that the beginning of the sequence that follows is one clock tick later than the current clock tick.
- The delay `##0` indicates that the beginning of the sequence that follows is at the same clock tick as the current clock tick.
  - `req ##1 gnt` Means `gnt` happens one clock cycle later of `req`.
  - `req ##0 gnt` Means `gnt` happens on the same edge as `req` getting asserted. Normally this is used for merging two sequence.
  - `req ##[0:3] gnt` Means `gnt` will be asserted 0 to 3 clock cycles after `req` is asserted.

```
// assert06.sv

module assert06();

  logic clk, a, b;

  initial
    begin
      clk = 0;
      forever #5 clk = ~clk;          // 5, 15, 25, 35, 45
    end

  initial
    begin
      a = 0; b = 0;
      @(negedge clk) a = 1;          // 10
      @(negedge clk) b = 1;          // 20
      @(negedge clk) b = 0;          // 30
      @(negedge clk) a = 0;          // 40
      @(negedge clk) $finish;
    end

  a1 : assert property ( @(posedge clk) a ##1 b );

endmodule
```

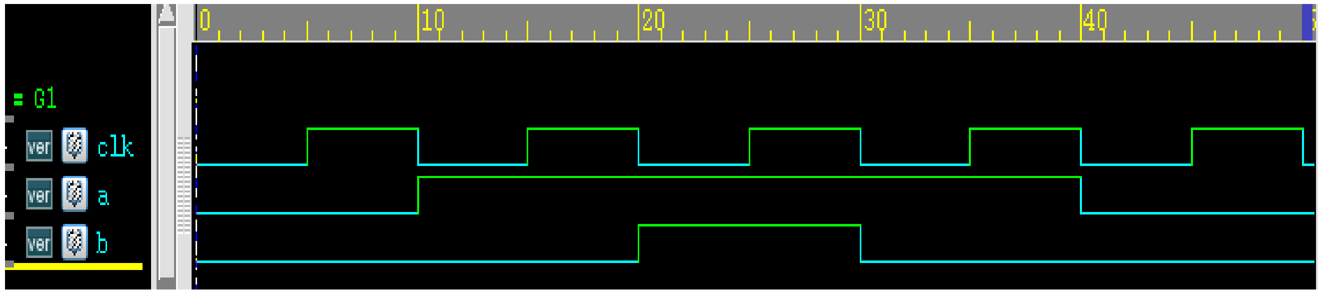


$$\begin{tikzcd} 5 \arrow[r, bend left, "b"] & 15 \arrow[r, bend left, "b"] & 25 \arrow[r, bend left, "b"] & 35 & 45 \end{tikzcd}$$

```
a ##1 b
a is true, followed by b is true
at the next clock tick
```

```
ncsim: (time  5 NS) Assertion assert06.a1 has failed (1 cycles, starting  5 NS)
ncsim: (time 35 NS) Assertion assert06.a1 has failed (2 cycles, starting 25 NS)
ncsim: (time 45 NS) Assertion assert06.a1 has failed (2 cycles, starting 35 NS)
ncsim: (time 45 NS) Assertion assert06.a1 has failed (1 cycles, starting 45 NS)
```

```
// assert07.sv

module assert07();

  logic clk, a, b;

  initial
    begin
      clk = 0;
      forever #5 clk = ~clk;
    end

  initial
    begin
      a = 0; b = 0;
      @(negedge clk) a = 1;
      @(negedge clk) b = 1;
      @(negedge clk) b = 0;
      @(negedge clk) a = 0;
      @(negedge clk) $finish;
    end

  a1 : assert property ( @(posedge clk) not (a ##1 b) );

endmodule
```



```
not (a ##1 b)
A negation property is same as
logic not operator on outcome of a
property. So if the outcome is
true, then negation makes it false.
```

```
ncsim: (time 25 NS) Assertion assert07.a1 has failed (2 cycles, starting 15 NS)
```

Implication Operators

- *antecedent_expr* ||-> *consequent _expr*
  - If there is no match of the *antecedent sequence expression*, implication succeeds vacuously by returning true. If there is a match, for each successful match of the *antecedent sequence expression*, the *consequent sequence expression* is separately evaluated, beginning at the end point of the match.

Types of Implication Operators

- Overlapped Implication Operator (|->)

```
s1 |-> s2;
```

  - If the sequence *s1* matches, then sequence *s2* must also match. If sequence *s1* does not match, then the result is true.

- Non-overlapped Implication Operator (||->)

```
s1 ||-> s2;
```

  - the first element of the *consequent sequence expression* is evaluated on the next clock tick.