SystemVerilog Assertions

structs, clking blocks, interfaces

Assertions

Stimulus Generation

Functional Coverage

OOP in SV

≡ Applications

What is an Assertion?

A piece of verification code used to check a property

- correct/illegal behavior
- assumptions/constraints
- coverage goals

Examples:

- Interface A must follow the PCI protocol
- Bus B must be one-hot
- The FIFO must never overflow
- I want to see back-to-back reads/writes
- Write will follow read 3 cycles later

2 Types of Assertions

- Immediate assertions
  - follow simulation event semantics for their execution and are executed like a statement in a procedural block.
- Concurrent assertions
  - based on clock semantics and use sampled values of their expressions

Immediate Assertions

```
always @(posedge clk) begin
   assert (req1 && req2)
   if (state == req1)
      next_state = req2;
   else if (state == req2)
      next_state = req2;
end
```

Concurrent Assertions

Concurrent assertions have special rules for sampling values of their expressions. The value of an expression sampled in one of these constructs is called a sampled value. In most cases the sampled value of an expression is its value in the Preponed region.

Value Sampling in SVA

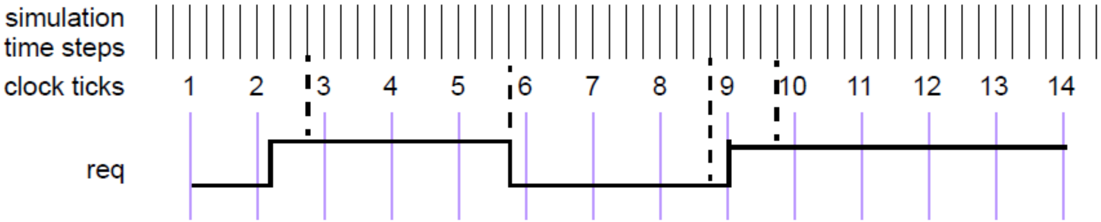


Figure 16-1—Sampling a variable in a simulation time step

Sequence, property, assert,..

Sequence

- Uses Boolean expressions with regular expressions, expressions are sampled at clock delays
- Every clock cycle, a new sequence is evaluated
- Useful as a “building block” , optional to use
- Can contain a cycle delay range , but no operators

Property

- Defines some behavior of a circuit

##n

##[1:5]

##[1:$]

assert, to specify the property as an obligation for the design that is to be checked to verify that the property holds. (MOST COMMON)

(a && cnt==3)

##10 (b && cnt==7)

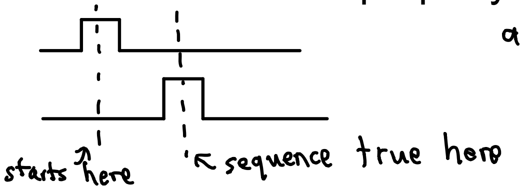
assume, to specify the property as an assumption for the environment. Simulators check that the property holds, while formal tools use the information to generate input stimulus.

cover, to monitor the property evaluation for coverage.

restrict, to specify the property as a constraint on formal verification computations. Simulators do not check the property.

SEQUENCE

a $\underbrace{\#\#1}_{\text{clock cycles}}$ b



a ##5 b

↑ a is true 5 cycles later, is b true?

Specifying Delays

Fixed Time

- ##1 ← a clock cycle
- ##0 – special case used to join two sequences

Time interval

- ##[0:3] or ##[1:5] ← delay from 1 to 5 clk cycles

Open ended, eventually

- ##[1:$] ← can never fail
- Between next clock cycle and the end of simulation
- This will continue matching which will potentially generate many incomplete events in simulation

Remember: the delay is in clock cycles, not nanoseconds

Example Sequences

sequence
property
assert

the same cycle

##0 a // means a

##1 a // means 1’b1 ##1 a ← at the next cycle, a must be true

##2 a // means 1’b1 ##1 1’b1 ##1 a

##[0:3]a // means

    (a) or

    (1’b1 ##1 a) or

    (1’b1##1 1’b1 ##1 a) or

    (1’b1 ##1 1’b1 ##1 1’b1 ##1 a)

a ##2 b // means a ##1 1’b1 ##1 b

```
sequence s1;
  @(posedge clk)
  a ##1 b
endsequence
```

assert property (@posedge clk) s1);

sampling mechanism

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

↑
holds at every clock cycle

```
sequence s1
  @(posedge clk)
    a ##1 a
```

boolean expr

(req==1'b1) ##1 (ack==1'b0)

Built-in Sample-value methods

$sampled (expression)

0→1 $rose (expression [, [clocking_event]])

1→0 $fell (expression [, [clocking_event]])

stayed same → $stable (expression [, [clocking_event]])

$changed (expression [, [clocking_event]])

#ofcycles ago ← $past (expression1 [, [number_of_ticks] [, [expression2] [, [clocking_event]]]])

For ALU

```
assert property((@ posedge clk) (opcode == MULT) && result == operand1 * operand2)
                                 || (opcode == ADD) && result = operand1 + operand2
```

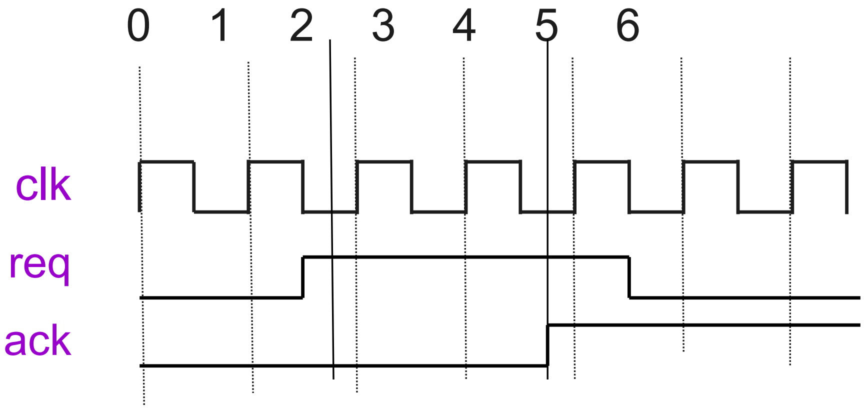
$ rose = ! a ## a

Sequence Example

```
sequence s1;
  @(posedge clk) $rose(
req);
 endsequence


 sequence s2;
  @(posedge clk) $rose(
ack);
 endsequence


 sequence s3;
 @(posedge clk) s1 ##2 s2
;
 endsequence
```



Simple Examples

```
sequence s1;
  @(posedge clk) $rose(
req);
 endsequence
 sequence s2;
  @(posedge clk) $rose(
ack);
 endsequence
 sequence s3;
 @(posedge clk) s1 ##2 s2
;
 endsequence
assert property ( s3 );
```

```
 property p1;
    @(posedge clk) $rose(
req) ##2 $rose(ack);
endproperty
assert property (p1);
```

```
assert property ( @(posedge clk)  $rose(req) ##2
$rose (ack)  );
```

Implication in SVA

- Implication is equivalent to if-then structure
- Implication can be used inside a property only
  - Implications can not be used inside a sequence
- Overlapped implication, denoted by “|->”
  - If there is a match on the antecedent, the consequent is evaluated beginning at the end time of the match
- Non-overlapped implication, denoted by “|=>”
  - If there is a match on the antecedent, the consequent is evaluated beginning one clock tick after the end time of the match
  - This form is used in multi-clock properties
- If not is used in the consequent, it means the result of the consequent evaluation is reversed

Try ALU assert again |=>

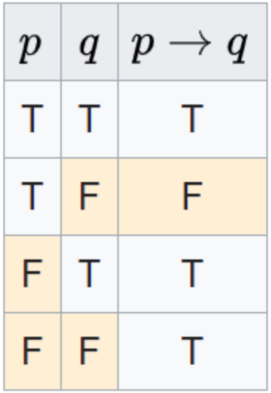
```
(opcode == MULT) |-> results==op1 * op2
```

Implication

p -> q
p is termed the antecedent of the conditional, and
q is termed the consequent of the conditional.

equivalent to

~p | q



p	q	$p \to q$
T	T	T
T	F	F
F	T	T
F	F	T

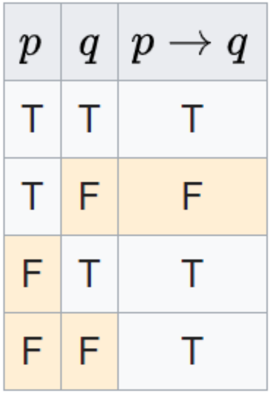
```
assert property ( @(posedge clk)  p |-> q);
```

Non-overlapping Implication

p => q
p is termed the antecedent of the conditional, and
q is termed the consequent of the conditional.

equivalent to

```
~p | ##1 q
```



p	q	$p \to q$
T	T	T
T	F	F
F	T	T
F	F	T

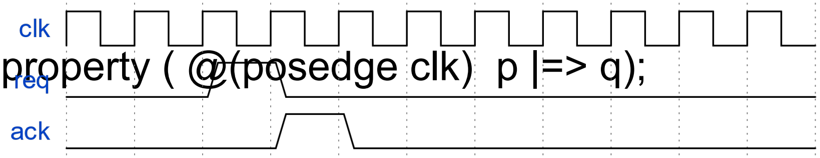
```
assert property ( @(posedge clk)  p |=> q);
```

sequence → no implication
property → implication

assert → DIRECTIVE

Example req/ack - 1

if request is asserted, ack must come 1 cycle later



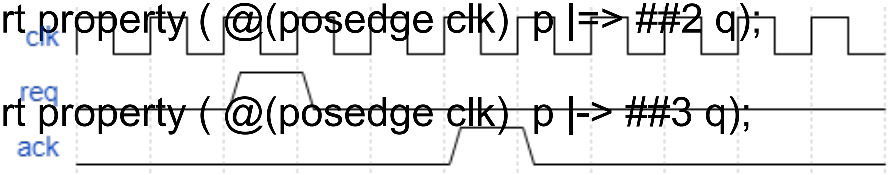
assert property (@(posedge clk) p |=> q);

assert property (@ posedge clk) req |⇒ ack);

Example req/ack - 2

if request is asserted, ack must come 3 cycles later

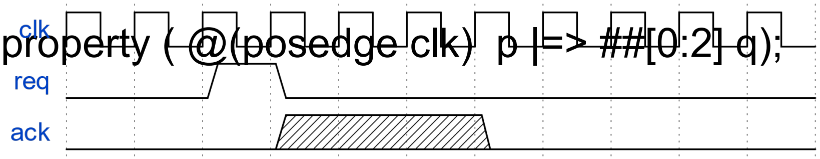
```
assert property ( @(posedge clk)  p |=> ##2 q);
Or
assert property ( @(posedge clk)  p |-> ##3 q);
```



Example req/ack - 3

if request is asserted, ack must come within 3 cycles

assert property (@(posedge clk) p |=> ##[0:2] q);



Notes

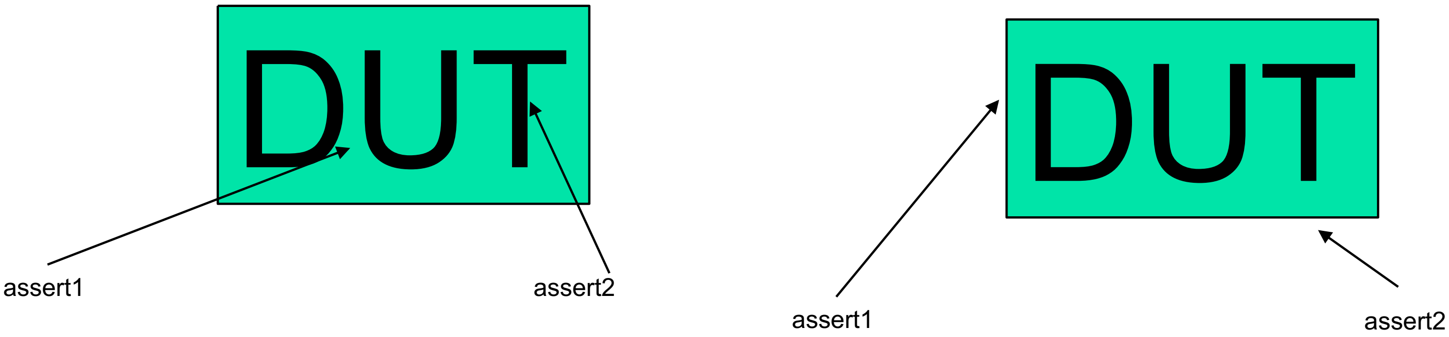
Most assertions must use some kind of implication

```
assert property ( @(posedge clk)  p & q);
```

This assertion will fail every cycle that

(p & q) == 0

White vs Black box assertions



DUT

assert1

assert2

DUT

assert1

assert2

Fifo Black-box assertions

```
interface t_fifo_intf(input clk,reset);
  logic full, empty;
  t_fifo_data dout,din;
  logic push,pop;

  assert property (@(posedge clk) !(full && push) );
  assert property (@(posedge clk) !(empty && pop) );
  assert property (@(posedge clk) ^{push,pop} !==1'bx );

  property data_consistency;
    t_fifo_data x;
    @(posedge clk) (push,x=din)  |-> ##[1:$] (pop && x==dout) ;
  endproperty
  assert property ( data_consistency);      can't fail

endinterface
```

Note: data consistency property not very efficient, and failure time is

Alternative Solution

```
interface t_fifo_intf(input clk,reset);
  logic full, empty;
  t_fifo_data dout,din;
  logic push,pop;

  assert property (@(posedge clk) !(full && push) );
  assert property (@(posedge clk) !(empty && pop) );
  assert property (@(posedge clk) ^{push,pop} !==1'bx );

  t_fifo_data tmp[$];
  t_fifo_data exp;
  always @(posedge clk) if (push) tmp.push_back(din);
  always @(posedge clk) if (pop)  exp <= tmp.pop_front();
  property push_pop;                          actual popped value
    @(posedge clk)  pop |=> ((exp) == $past(dout)) ;
  endproperty

  assert property ( push_pop )
      else $display("%p %p",$sampled(exp),$past(dout));
endinterface
```

3 ready pulses ↙ repeat 3 times

```
(rdy ##1 !rdy) [*3]
```

"throughout" keyword

Assertions for

- ALU Assignment
- Simple flip Flop

Flip Flop

```
always @(posedge clk) Q <= D;

assert prop((@(posedge clk)          if (x)
       q == $past(d)))
       else $display(~~q, d~~)        else
       $sampled(q)
       $past(d)

                 $past(d, 10)
```