Introduction to *Assertion Based verification* with of PSL

(VHDL flavor)

Bert Molenkamp

Jan Kuper

Implementation Coverage

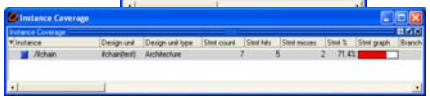
- Line coverage: is this line executed during simulation?
- Toggle Coverage: is this bit changed from 0 to 1 and 1 to 0?
- Combinational Coverage: are all possible combinations of an expression simulated?
  - y <= a and b; -- which percentage of the combination of a and b?
- FSM coverage: are all states reached and did it traverse all possible paths?

With code coverage enabled

```
clk <= NOT clk AFTER 5 ns;

a <= "010",
     "001" AFTER  12 ns,
     "011" AFTER  22 ns,
     "101" AFTER  32 ns;

y1 <= a(0) and a(1);

PROCESS(a)
BEGIN
  IF a(0)='1' THEN
    y2 <= '0';
  ELSIF a(1)='1' THEN
    y2 <= '1';
  ELSIF a="000" THEN
    y2 <= '1';
  ELSE
    y2 <= '0';
  END IF;
END PROCESS;
```

After initialization: all statements are missed

After 10 ns; only two statements are missed



Why functional verification?

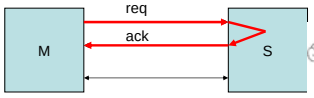
- Disadvantage of the *implementation coverage:* does not cover the functional intent.
- Survey of *Collett International* (2000): 74% of all re-spins are caused due to a functional error.
- Assertion Based Verification
  - Assertions capture the designers intent
  - Can specify legal and illegal behavior
  - During simulation assertions are verified
- Hardware description languages include PSL (Property Specification Language)
  - SystemVerilog (SystemVerilog Assertions (~PSL) is built-in)
  - In Verilog and VHDL it will be part of the new standard

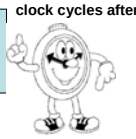
Assertion

- An assertion is a statement about the designers intended behavior
- VHDL already supports assertions.
  - assert (a>=b) report “message” severity note;
  - Use of VHDL assertions becomes complex in case of behavior over time.

communication

Requirement:
ack between 2 and 4 clock cycles after req





How can this timing constraint be verified?

```
ARCHITECTURE ....
  ...
BEGIN
  verify_protocol(req,ack,clk,2,4);
  ...
```

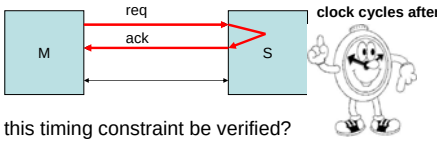
Concurrent procedure call


```
PROCEDURE verify_protocol (SIGNAL req, ack, clk : IN std_logic; min, max : IN natural) IS
  VARIABLE nmb : natural;
BEGIN
  ASSERT max >= min REPORT "max is lower than min";
  LOOP
    WAIT UNTIL rising_edge(clk);
    nmb := 0;
    IF req='1' THEN
      WHILE nmb <= max-1 LOOP
        nmb:=nmb+1;
        EXIT WHEN ack='1';
        WAIT UNTIL rising_edge(clk);
      END LOOP;
      IF      (nmb <= min)                THEN    REPORT "ack too early";
      ELSIF (nmb = max) AND (ack='0')  THEN    REPORT "ack too late";
      ELSE                                        REPORT "passed";
      END IF;
    END IF;
  END LOOP;
END verify_protocol;
```

Questions:
- how to handle a reset?
- how to handle a second request if the first did not receive an acknowledge?
- can max be infinite? (integer'HIGH ?)

communication

Requirement:
ack between 2 and 4 clock cycles after req



How can this timing constraint be verified?

```
ARCHITECTURE ....
  ...
BEGIN
  -- PSL assert
     always ... req ... 2..4 ... ack ...

  ...
```

PSL statement

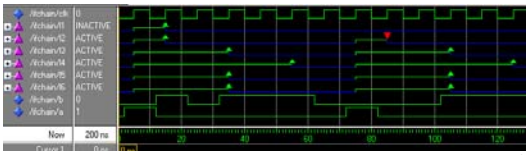
Logic

- *Proposition logic*
  - Not, and, or, implies, iff
- *Predicate logic*
  - Forall, exists, variables
- *Temporal logic*
  - Next, Always, Never, Until, …

Next operator

-- PSL default clock is rising_edge(clk);
-- PSL l1: assert (a -> next b);
-- PSL l2: assert always (a -> next b);
-- PSL l3: assert always (a -> next[3] (b));
-- PSL l4: assert always (a -> next_a[3 to 5] (b));
-- PSL l5: assert always (a -> next_e[3 to 5] (b));
-- PSL l6: assert always (a -> next_e![3 to 5] (b));

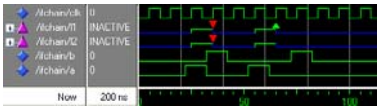
- L1: only at cycle 0; if a then next b
- L2: if a then next b
- L3: if a then 3 cycles later n
- L4: *next_all*. *b* true in cycles 3 until 5
- L5: *next_exists*. There exists at least one *b* in cycles 3 to 5



More variations of next

a -> next_event(b) (c)
a -> next_event(b)[n] (c)
a -> next_event_a(b)[i to j] (c)
a -> next_event_e(b)[i to j] (c)

Always/Never



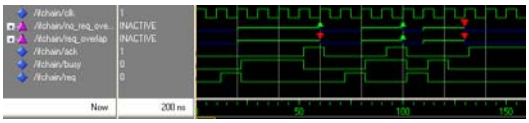
-- PSL l1: assert always (a -> next (not b));

-- PSL l2: assert never (a and (next b)) ;

until and until_

```
-- psl no_req_overlap: assert  always ( req -> next ( busy until ack ) );
-- psl req_overlap:    assert  always ( req -> next ( busy until_ ack ) );
```

- If req then busy should be true until ack is true.
- The until_ requires an overlap between the last busy and ack.
- Note: also a strong version is possible (until!)



before

```
-- psl  property ack_follows_req is
--      always ( req -> next (ack before req) )

-- psl ack_follows_req _lbl: assert ack_follows_req;
```

- How to prevent two consecutive requests?
  - **before**
    This requires that after a request first an acknowledge should occur before a new request is applied.
- If the simulation is finished this assertion will not respond that a request is still waiting for an acknowledge. But this is taken care of with the assertion on the next slide (eventually!).

eventually!

```
-- psl  property protocol is
--      always ( req -> eventually! ack )

-- psl protocol _lbl: assert protocol;
```

- **Eventually!**
  The request <u>must</u> always be honored with an acknowledge.
- But .. two requests can share the same acknowledge!
  - How to prevent two consecutive requests?
- What happens if a request is not honored with an acknowledge (at the end of the simulation run)?

Weak and Strong!

```
-- PSL l5: assert always ( a -> next_e[3 to 5] (b) );
-- PSL l6: assert always ( a -> next_e![3 to 5] (b) );
```

- **A simulation is time constrained**
- **Do the properties l5 and l6 hold when simulation finishes at 200 ns?**
- **A strong operator (!) requires that if the *enabling condition* is true then the *fulfilling condition* should occur**
- **Many operators have a weak and strong version**

Weak and strong!

next – next!

until – until!

before – before!

eventually! only strong

always, never: only weak

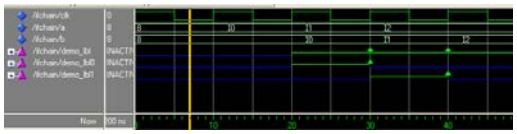
forall

Check values? Can be simplified with forall.

```
-- psl  demo_lbl0: assert always ( (a=0 -> eventually! b=0));
-- psl  demo_lbl1: assert always ( (a=1 -> eventually! b=1));
```

Is equivalent with:

```
-- psl  demo_lbl:  assert forall i IN {0 TO 1} : always ( (a=i -> eventually! b=i));
```



SEREs

- A sugared building block of PSL is a SERE (*Sequential Extended Regular Expression*). A SERE describes a *set of sequences* of states
- A sequence of states, in which req is high on the first cycle, busy on the second, and gnt on the third.

{req; busy; gnt}

SEREs

{a ; b ; c} close, but not equal to:

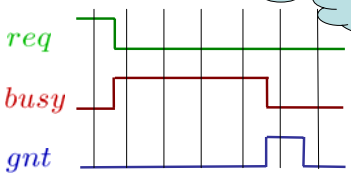
a and (next b) and (next (next c))

SEREs - Examples

$\{req; busy; busy; busy; busy; gnt\}$

$\{req; busy[*4]; gnt\}$

signal *busy* holds 4 times



SEREs - Example

{req; busy[*3 to 5]; gnt}

req;busy;busy;busy;gnt
req;busy;busy;busy;busy;gnt
req;busy;busy;busy;busy;busy;gnt

signal *busy* holds any number of times between 3 to 5

{req; busy[*]; gnt}

req;gnt
req;busy;gnt
req;busy;busy;gnt
req;busy;busy;busy;gnt
req;busy;busy;busy;busy;gnt
....

Signal *busy* holds any number of times

{req; busy[+]; gnt}

Signal *busy* holds any number of times; but at least ones

SEREs - Example

{req; busy[*3 to inf]; gnt}

Signal *busy* holds any number of times between 3 to infinite

{req; busy[=2]; gnt}

req;busy;not busy;busy;gnt
req;busy;not busy;busy;not busy; gnt
req;busy;busy;gnt

....

signal *busy* holds exactly twice before *gnt* (not necessarily consecutive)

SEREs - Example

{req; busy[=2]; gnt}

req;busy;not busy;busy;gnt
req;busy;not busy;busy;not busy; gnt
req;busy;busy;gnt
....

signal *busy* holds exactly twice before *gnt* (not necessarily consecutive)

{req; busy[->2]; gnt}

req;busy;not busy;busy;gnt
req;busy;busy;gnt
....

busy[->1] is equal to busy[->]

signal *busy* holds exactly twice before gnt (not necessarily consecutive) But *gnt* should occur immediately after the last occurrence of *busy*.

SERE; boolean

{req; busy; gnt}

- The strong typing mechanism is relaxed:

  req, busy and gnt are booleans but also type bit, std_logic is allowed with

  '1' is TRUE and '0' is FALSE.
- If req, busy and gnt are of type std_logic the previous SERE is equal to

  {(req='1'); (busy='1'); (gnt='1')}

 25

Synchronous systems

{req; busy; busy; busy; gnt}

- Simulation tools use the SEREs during simulation (*assertion based verification*).
- What is the difference between:
  - {req; busy; busy; busy; gnt}
  - {req; busy; busy; gnt}
- Therefore @rising_edge(clk) is added. The first SERE required three consecutive busy signals at the rising edge of clk.

 26

suffix implication operators

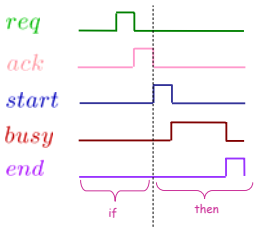
```
--    SERE_A |=> SERE_B
--    SERE_A |-> SERE_B
```

- if the path starting matches SERE_A
- then its continuation should match SERE_B
- |=> overlapping
- |-> non-overlapping

 27

PSL Sugar Properties – Example1

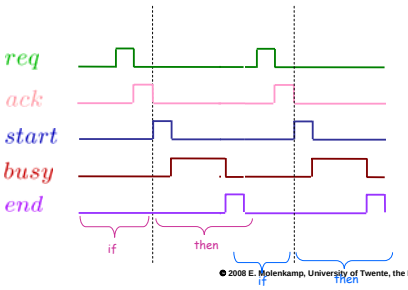
```
-- psl property name_of_property is
--    always ( {req;ack} |=> {start; busy[*]; end1} )
--    @rising_edge(clk);
```



Note: clk is omitted

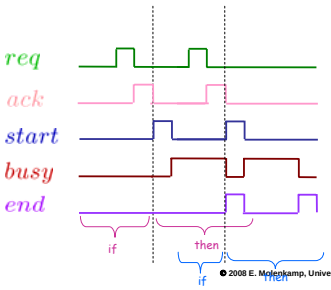
 28

PSL Sugar Properties – Example1



 29

PSL Sugar Properties – Example1

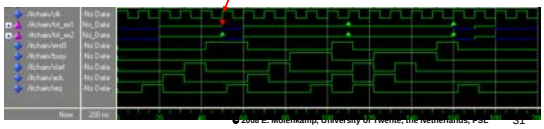


 30

Example1

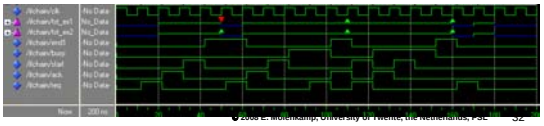
```
ARCHITECTURE …
BEGIN
-- psl property ex1 is
--     always ( {req;ack} |=> {start; busy[+]; end1} )
--     @rising_edge(clk);

-- psl  tst_ex1: assert ex1;

-- psl property ex2 is
--     always ( {req;ack} |=> {start; busy[*]; end1} )
--     @rising_edge(clk);

-- psl  tst_ex2: assert ex2;
```

- Blue line indicates assertion is inactive.
- Red triangle indicates that tst_ex1 fails (no busy signal). Green indicates a pass.
- Note:
  - the overlapping of {req;ack} start time 60 ns and 100 ns
  - {req;ack} start time 100 ns and 120 ns have the same end1 (at 160 ns).
  - Since the req (at 170 ns) is not followed with ack the starting SERE does not occur and the assertion is inactive.



Also informative tables are generated

- In this table the same end of {req;ack} start time 100 ns and 120 ns is counted.
- Notice the translation of the assertion [*] is equal to [*0 to inf]

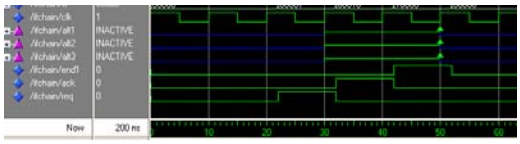


PSL Sugar Properties

- SERE1 |-> SERE2
  - If a sequence matches SERE1, then SERE2 should be matched, starting from the last element of the sequence matching SERE1
  - So there is one cycle of overlap in the middle
- SERE1 |=> SERE2
  - If a sequence matches SERE1, then continuation SERE2 is
  - So there is no overlap of SERE1 and SERE2
- As a consequence:

  SERE1 |=> SERE2 ≡
  SERE1 |-> { true; SERE2} ≡
  SERE1 |-> { [*1]; SERE2}

Example; |=>, |->

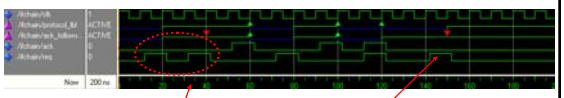


ARCHITECTURE …
BEGIN
-- PSL default clock is rising_edge(clk);

-- PSL alt1:assert always ({req;ack} |=> {end1});
-- PSL alt2:assert always ({req;ack} |-> { [*1]; end1});
-- PSL alt3:assert always ({req;ack} |-> {true; end1});

- Default clock is possible
- Assertions may be labeled.

Assertions for a protocol



For communication the following protocol can be used. After a request the system has to wait for an acknowledge.

- Two consecutive request without an acknowledge is not allowed
- A request must always be served

PSL

```
-- psl  property protocol is
--     always ( req -> eventually! ack )
--     @rising_edge(clk);

-- psl protocol _lbl: assert protocol;
```

- **Eventually!**
  The request must always be honored with an acknowledge.
- But .. two requests can share the same acknowledge!
  - How to prevent two consecutive requests?
- What happens if a request is not honored with an acknowledge (at the end of the simulation run)?
  - If the simulation is finished (QuestaSim: quit –sim) the simulator responds with:

```
quit -sim
# ** Error: Assertion failed
#    Time: 200 ns Started: 150 ns  Scope: /ifchain/ protocol_lbl File:
….Line: 23 Expr: ack
```


PSL, cont.

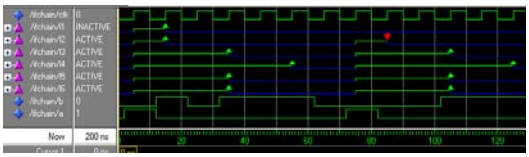
```
-- psl  property ack_follows_req is
--     always ( req -> next (ack before req) )
--     @rising_edge(clk);

-- psl ack_follows_req _lbl: assert ack_follows_req;
```

- How to prevent two consecutive requests?
  - **Before**
    This requires that after a request first an acknowledge should occur before a new request is applied.
- If the simulation is finished (QuestaSim: quit –sim) this assertion will not respond that a request is still waiting for an acknowledge. But this is taken care of with the assertion on the previous slide (eventually!)

Next operator

```
-- PSL default clock is rising_edge(clk);
-- PSL l1:assert          ( a -> next b);
-- PSL l2:assert always ( a -> next b);
-- PSL l3:assert always ( a -> next[3] (b) );
-- PSL l4:assert always ( a -> next_a[3 to 5] (b) );
-- PSL l5:assert always ( a -> next_e[3 to 5] (b) );
-- PSL l6:assert always ( a -> next_e![3 to 5] (b) );
```

- L1: only at cycle 0; if a then next b
- L2: if a then next b
- L3: if a then 3 cycles later b
- L4: *next_all*. *b* true in cycles 3 until 5
- L5: *next_exists*. There exists at least one *b* in cycles 3 to 5



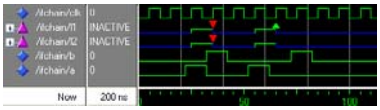
Weak and Strong!

-- PSL l5:assert always (a -> next_e[3 to 5] (b));
-- PSL l6:assert always (a -> next_e![3 to 5] (b));

- A simulation is time constraint
- Do the properties l5 and l6 hold when simulation finishes at 200 ns?
- A strong operator (!) requires that if the *enabling condition* is true there should the *fulfilling condition* should occur
- Many operators have a weak and strong version

```
quit -sim
# ** Error: Assertion failed
#    Time: 200 ns Started: 185 ns  Scope: /ifchain/l6 File: ….. Line: 21
# ** Error: Assertion failed
#    Time: 200 ns Started: 195 ns  Scope: /ifchain/l6 File: ….. Line: 21
```


Always/Never

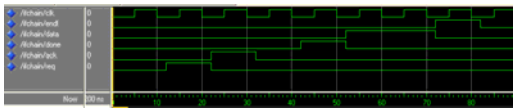


-- PSL default clock is rising_edge(clk);
-- PSL l1:assert always (a -> next not(b));
-- PSL l2:assert never {a; b} ;

example

Assume the protocol for reading data is:

- Request for data (req)
- next cycle (ack), then after 1 but at most 4 clock cycles done followed with
- data (any number) finalized with
- endl



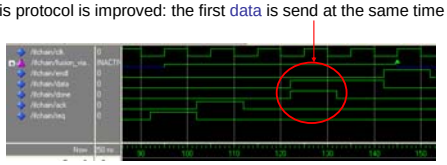
How would you write the assertion?

Easy …

-- PSL default clock is rising_edge(clk);
-- PSL sequence ack_seq is { ack;[*1 to 4]; done }; -- named sequence!!
-- PSL sequence data_transfer is { data[*]; endl };

-- PSL concatenation:assert always (req |=> {{ack_seq}; {data_transfer}}) ;

This protocol is improved: the first data is send at the same time as done.



How would you write the assertion?

Easy …

-- PSL default clock is rising_edge(clk);
-- PSL fusion_via_concat:assert always (req |=> { {ack;[*1 to 4]};
-- (done AND data) ; {data[*];endl}}) ;

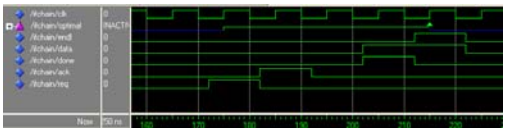
Alternative: FUSION (:)
It is similar to concatenation but a single cycle overlap is required
(last element of first sequence and first element of second sequence)

-- PSL default clock is rising_edge(clk);
-- PSL sequence ack_seq is { ack;[*1 to 4]; done };
-- PSL sequence data_transfer is { data[*]; endl }; -- endl after last done

-- PSL fusion:assert always (req |=> {{ack_seq}:{data_transfer}}) ;

The protocol is improved again: endl is send at the the same as last data.
How would you write the assertion?

Fusion



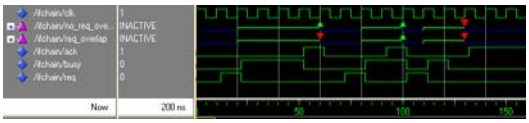
-- PSL default clock is rising_edge(clk);
-- PSL sequence ack_seq is { ack;[*1 to 4]; done };
-- PSL sequence data_transfer_f is { data[*]: endl }; -- endl with last data

-- PSL optimal:assert always (req |=> {{ack_seq}:{data_transfer_f}}) ;

until and until_

```
-- psl no_req_overlap: assert always ( req -> next ( busy until ack ) );
-- psl req_overlap:    assert always ( req -> next ( busy until_ ack ) );
```

- If req then busy should be true until ack is true.
- The until_ requires an overlap between the last busy and ack.
- Note: also a strong version is possible (until!)



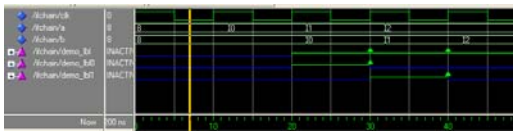
forall

Check values? Can be simplified with forall.

```
-- psl  demo_lbl0: assert always ( (a=0 -> eventually! b=0));
-- psl  demo_lbl1: assert always ( (a=1 -> eventually! b=1));
```

Is equivalent with:

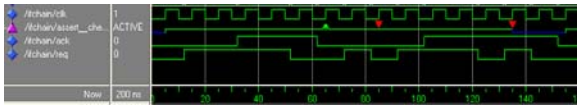
```
-- psl  demo_lbl:  assert forall i IN {0 TO 1} : always ( (a=i -> eventually! b=i));
```



PSL embedded in VHDL

- This was only a brief introduction to PSL.
- Simulation tools do not yet support all of PSL.
- Try to use it .. you can start with a simple assertion.

four-phase handshake



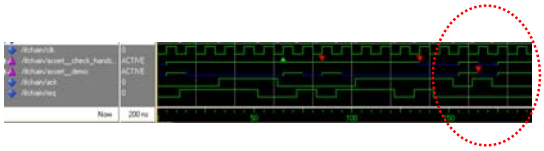
Write an assertion for the four-phase handshake protocol

A solution

```
ARCHITECTURE ..
BEGIN

 -- PSL sequence request  IS {not (req or ack); req};
 -- PSL sequence start_s  IS {req[*]; req AND ack};
 -- PSL sequence middle_s IS {(req AND ack)[*]};
 -- PSL sequence end_s    IS {(not req)[*]; (not req) and (not ack)};

 -- PSL property check_handshake is
 --   always ({request} |=> {start_s; middle_s; end_s}) @rising_edge(clk);
 -- PSL assert check_handshake;
```

But what if an ack occurs when there was not req at all?

```
  -- psl  property illegal_ack is
  --      never { (not (req or ack));ack }
  --      @rising_edge(clk);

  -- psl  assert illegal_ack;
```

- Next_event(_e _a)
- Simple subset (ifchain22.vhd)
- Ended (file28); not yet supported (new in std 1850-2005)
- Async/sync not supported yet (2005)
- forall

 55