

1.0 Common Definitions

The following common definitions are used in many properties.

sequence next_event(e) = (!e * [0:inf]) ;[1] e;

sequence next_event_f(e) = (!e *[1:inf]) ;[1] e;

sequence before(x,y) = ((!y * [1:inf]) ;[1] (x && !y)) or (!y * [inf]);

sequence strong_before(x,y) = (!y *[1:inf]) ;[1] (x && !y);

2.0 Examples

Property 1

English

If a snoop hits a modified line in the l1 cache, then the next transaction must be a snoop writeback. (design 1)

SV

property p1 = (snoop & hitm) => [1] next_event(trans_start) ;[0] writeback;

Property 2

English

If signal "enable" rises, then a clock after the fourth transfer signal "pending" must rise. (design 1)

SV

sequence s2 = \$rose(enable) ;[1] (next_event(transfer) * [4]) ;

property p2 = ended s2 => [1] \$rose(pending);

Property 3

English

If signal "boff" is asserted, then if the first request which is accepted after the assertion of "boff" is not a snoop request, then it is a write request. (design 1)

SV

sequence s3 = boff ;[1] next_event(accepted) ;[0] !snoop_req;
property p3 = ended s3 => write_req;

Property 4

English

If signal "hit" is active and signal "pending" is not, then next time signal "pending" is active, signal "sel5" is active. (design 1)

SV

sequence s4 = (hit & !pending) and next_event(pending);
property p4 = ended s4 => sel5;

Property 5

English

An urgent request should be the next handled. (design 2)

SV

sequence s5 = urgent_req ;[1] next_event(grant);
property p5 = ended s5 => urgent_answered;

Property 6

English

Between a request and its acknowledge the busy signal must remain asserted. (design 2)

SV

property p6 = req => (busy * [1:inf]) ;[1] ack;

Property 7

English

If a data packet of any size starts and eventually gets a LAST bit, then next data packet must have the FIRST bit asserted. (design 3)

SV

sequence s7 = dp_start ;[1] next_event(LAST) ;[0] next_event(dp_start);
property p7 = ended s7 => FIRST;

Property 8

English

If a write command starts and size=N (N=1 through 8), then N assertions of signal "gx_start" should occur before the LAST bit goes active. (design 3)

SV

```
// generalized for any dynamic N
// clocking allows declaration of variables with synchronous assignments
// assumption is that gx_start and LAST do not happen at the same time
clocking p8_clk @(posedge clk);
    int count = 0;
    assign count <= write_command_start ? 0 : ( (gx_start) ? count+1 : count);
    property p8 = write_command_start ==>
        (int count_check=size; next_event(LAST) ;[0] (count == count_check));
endclocking
```

Property 9

English

The address queue ptr increment consecutively (cyclic). In other words, every time that an address is entered into the queue with queue ptr = N, the next time that an address is entered into the queue, the address will be N+1 (cyclically). (design 4)

SV

```
property p9 = [1] (addr_queue_ptr_p_q == $past(addr_queue_ptr_p_q)) ||
    (addr_queue_ptr_p_q == (addr_queue_ptr_p_q + 1));
```

Property 10

English

If data grant received then as soon as dbusy is high for two clocks, take the data bus. (design 5)

SV

```
property p10 = data_grant ==> next_event(dbusy&& $past(dbusy)) ;[0] dvalid;
```

Property 11

English

The data that returns for read is the last data that was written to the register before the read was issued. (design 5)

SV

```
sequence s11 = write_valid ;[0]
    (int addr = reg_addr[3:6], int dbits = data_bus[0:7];
    (!(write_valid && (addr == reg_addr[3:6]))) * [1:inf]);[0]
    (read_valid && (addr == reg_addr[3:6])) ;
sequence s11_a = write_valid ;[0]
```

```
(int dbits = data_bus[0:7];
  next_event(ended s11) ;[0] (dbits == data_bus[0:7]));
```

property p11 = ended s11 => ended s11_a;

Property 12

English

Every buffer will be read before it is overwritten. (design 6)

SV

```
property12 = write_enable =>
  (int addrbits = write_addr[0:1];
  before( (read_enable && (read_addr[0:1] == addrbits)),
    (write_enable && (write_address[0:1]==addrbits))) );
```

Property 14

English

For every write, data transfers must alternate between odd and even entries. In other words, if there is a write, then as long as we are transferring data belonging to this write, consecutive data transfers must alternate between even and odd addresses.

SV

```
sequence s14 = next_event(write_en) ;[0] !addr[0] ; next_event(write_en) ;[0] addr[0];
property p14 = write_start => (s14 *[1:inf]) within next_event(write_end);
```

Property 15

English

Two consecutive writes cannot be to the same address. Address appears one cycle after the write_valid.(design 6)

SV

```
property p15 = write_valid ; 1 =>
  (int addr = addr_bus[0:7];
  [1] next_event(write_valid) => [1] (addr != addr_bus[0:7]));
```

Property 16

English

If an address was set valid then the next time the "retire" signal is asserted for this address, the address will be invalidated 3-8 clocks later.

SV

```
property p16 = write_en & data_valid =>
    (int addr = write_addr[4:0];
     next_event(retire&&retire_address[0:4]==addr) =>
     [4:8] (write_en && !data_valid && (write_address[0:4]==addr)));
```

Property 17

English

If read req is received, then either the next output req to PLB is read, or the one after that.

SV

```
property p17 = $rose(read_req) =>
    next_event_f($rose(out_en)) * [1:2] ;[0] output_read;
```

Property 18

English

If read req is received and then write req is received before read req is output, then read req will be output before write req is output. (Assumption: reqs stay asserted until output.)

SV

```
sequence s18 = $rose(read_req) ;[1] before($rose(write_req),out_en &&output_read);
property p18 = ended s18 => before(out_en && output_read, out_en && output_write);
```

Property 20

English

"any number of transactions limited by the depth of the queue in the bridge may be split, and may be retried an unlimited number of times provided they are always each retried within a defined timeout period (typically 32,000 clock cycles) of the last attempt. If any of these transactions is not retried within this time period the bridge must set the Discard Timer Status bit 10 in the Bridge Control register. In addition, the bridge must assert SERR# on the primary interface if enabled to do so by the Discard Timer SERR# Enable bit 11 in the Bridge Control register and the SERR# Enable bit in the Command register"

Comments

many features of the bus are omitted, including error and abort conditions for clarity. note also - all the signals shown are active low, therefore "asserted" means =0.

Transactions can be 'stretched' by either the initiator or the target. The 'good' completion is denoted by irdy_ and trdy_ being asserted at the same time.

When a target is not able to complete a transaction quickly enough, it can ask for the transaction to be re-tried at a later time. It does this by asserting the stop_ signal instead of the trdy_ signal.

Where the target is a multi-function device or a bus-bridge it is possible for the target to have multiple "open" transactions at any instant. It is also important to understand there is no requirement for the transactions to be re-tried in the order in which they are started, nor for them ultimately to be completed in any particular order.

This property demonstrates the presence of multiple "open" transactions at any time, each transaction being matched by use of a 'tag' that is effectively the concatenation of { address, address_parity, command, byte_enables, REQ64# } (a total of about 74 bits if the bus is operating in the 64 bit mode). [[note : this is not strictly correct as the 'byte_enable' signals are not known until later in the transaction, therefore the concatenations ought to be spilt into two parts, but this explains the principle]] One issue we need to be aware of is that the 'address' and 'command' bits are known only in cycle 2 of the transaction, however we do not know a retry will be requested until between cycle 4 to cycle 17 for any transaction.

Note also : the scope of the 'tag' variable used in simulation must be local to the instance of the property since there will be multiple, concurrent instances of the property with different tag values, one per "open" transaction.

SV

```
bool bustag = {ad[0:63],par,c_be[0:3],req64};
sequence s20_1(data,en) = [1:32000] $fell(frame) && (bustag == data);
    [1] !frame*[1:inf];
    [1] !trdy &&(c_be[0:3] == en);
sequence s20_2(data,en) = [1:32000] $fell(frame) && (bustag == data);
    [1] !frame*[1:inf] ;
    [1] !stop &&(c_be[0:3] == en);
sequence s20_3 = [32001:32101] BridgeControlRegister[10];
sequence s20_a = $fell(frame) ; [1] (!frame * [1:inf]) ;[1] (!stop);

sequence s20_b = $fell(frame) ;
    [0] (int tag = bustag;
    [1] !frame * [1:inf];
    [1] (!stop);
    [0] (int enable = c_be[0:3];
        s20_1(tag,enable) or s20_2(tag,enable) or s20_3));
property 20 = (ended s20_a) => ended s20_b;
```

Property 21

English

"if an initiator begins a MemoryReadLine transaction, the last implied address of the last data phase of the transaction must not be in a different cache line than the initial address if the target implements the Cacheline Size Register feature. Note, that support of the Cacheline Size Register is optional and that each target may implement a different Cacheline size."

Comments

The PCI bus allows several forms of 'burst' transfer of data from successive words in memory. One of the burst modes is 'MemoryReadLine' which is recognised from the value on the C/BE# signal group during cycle 2.

Note : for simplicity we will assume the bus is working only in 32-bit mode

If the CachelineSize for this target has been set at, for example, 64 bytes, then the master may not attempt a burst that would cause the auto-incrementing address to rollover from xx001111100 to xx010000000.

There are two potential solutions to implement this property, either by watching for the rollover condition or by pre-calculating the maximum number of data transfers allowed in the burst as

$$X = (\text{CachelineSize} - (\text{InitialAddress} \& (\text{CachelineSize} - 1))) \gg 2;$$

Using the example above

$$X = (000001000000 - (\text{xx}0011110100 \& (000001000000 - 000000000001))) \gg 2$$

$$= (000001000000 - (\text{xx}0011110100 \& (000000111111))) \gg 2$$

$$= (000001000000 - (000000110100)) \gg 2$$

$$= (000000001100) \gg 2$$

$$= 3$$

Then limiting the non-deterministic burst transfer count to the range [1..X]

SV

```
clocking my_clk @posedge clk;
```

```
    bool burst_cond = CachelineSizeRegSupported && $fell(frame) &&
                      (cbe_ == MemoryReadLine);
```

```
    bool busy = (!IRDY_ && !TRDY_);
```

```
    logic var21 = (burst_cond) ? 1: var21 + 1;
```

```
    property 21 = (burst_cond) =>
```

```
        (int N = ((CachelineSize - (AD & (CachelineSizeReg-1))) >> 2);
```

```
        (busy&&!frame) *[1:inf] ; [0] (var21 < N) ;[1] !busy * [0:inf] ;[1] frame);
```

```
endclocking
```

Property 22

English

if a target does not implement the Cacheline Size Register feature, the target must respond to a MemoryReadLine or MemoryReadMultiple transaction using a Disconnect With Data during the first data phase or a Disconnect Without Data during the second data phase"

Comments

For information, the specification explains "This ensures that the transaction will complete (albeit slowly, since each request will complete as a single data phase transaction)."

For the purposes of demonstrating this property we can make a small simplification in the PCI specification and assume:

This property raises the issue of recognizing and reacting to different features of different targets. The property can be implemented by either

a) saving the value of device registers that are typically only visible by monitoring the bus during initialization

or

b) determining the value of the registers by looking into the design during verification, presumably based on some address translation or mapping scheme.

Disconnect With Data is signaled by the target by asserting both the STOP# signal in addition to TRDY# coincident with the initiators IRDY# to effect a data transfer.

Disconnect Without Data is signaled by the target by asserting the STOP# signal without asserting TRDY# coincident with the initiators IRDY#.

SV

```
bool ready = !IRDY_ & !TRDY_;
bool cond = CachelineSizeRegSupported && $fell(frame) &&
           ((cbe_==MemoryReadLine)||
            (cbe_==MemoryReadMultiple));

property p22_b = cond => next_event(ready) ;[0]
                !STOP_ or ([1] before(!STOP_ && TRDY_,!TRDY_));
```

Property 45

Comments

PCI compliant devices can behave as master or target agents, or target agents only. The required pins of a PCI agent include Interface Controls FRAME#, TRDY#, IRDY#, STOP#, DEVSEL#, and IDSEL (input), Error Lines PERR# and SERR# Arbitration Lines

REQ# (output) and ACK# (input), Clock/Reset CLK and RST# inputs, 32 bit wide Address/Data bus (AD), 4 bit wide Command/Byte Enable line (C/BE). There are also numerous optional signals. The Interface Signals are 'asserted' by holding them low. An "address phase" is marked by FRAME# fall; A "data phase" is indicated when TRDY# and IRDY# are high; An "i/o cycle" is a data phase in which the C/BE line indicates an i/o read or write;

After FRAME# has been asserted a target can claim the access cycle by asserting DEVSEL#; a "target abort" is executed by asserting the STOP# line after it has claimed the cycle by asserting DEVSEL#.

A "master abort" is executed by the master asserting (for at least one CLK period if not already asserted) and subsequently deasserting IDRY#; FRAME# is deasserted for at least one CLK period when IRDY# is asserted; the master abort is finally completed with deassertion of FRAME# and IRDY# lines.

The "last data phase" is indicated when IRDY# is asserted, FRAME# is deasserted and either TRDY# or STOP# is asserted.

The "last data phase" is indicated when IRDY# is asserted, FRAME# is deasserted and either TRDY# or STOP# is asserted.

data phase" completes when IRDY# is asserted with either STOP# or ir TRDY# asserted simultaneously.

The initial data phase is marked by FRAME# fall; subsequent data phases are indicated by FRAME#, IRDY# and TRDY# all being asserted.

SV

```
'define IOREAD  'h02
'define MEMYRD  'h06
```

```
'define CONFRD  'h0a
```

```
'define MEMRDM  'h0c
```

```
'define MEMRDL  'h0e
```

```
bool frame_ = PCI_FRAME;
bool devsel_ = PCI_DEVSEL;
bool trdy_ = PCI_TRDY;
bool stop_ = PCI_STOP;
bool byte_enable_mismatch = (current_trans_ad[1:0]==1 && PCI_BE[0] !=1) ||
                               (current_trans_ad[1:0]==2 && PCI_BE[1:0] !='b11) ||
```

```

                (current_trans_ad[1:0]==3 && PCI_BE[2:0]!='b111);
bool target_abort = $rose(devsel_) & $fell(stop_);
bool read_transaction = (PCI_BE[3:0]=='IOREAD) || (PCI_BE[3:0]=='MEMRDM) ||
                (PCI_BE[3:0]=='MEMYRD) || (PCI_BE[3:0]=='CONFRD) ||
                (PCI_BE[3:0]=='MEMRDL);
bool trdy_chng = (trdy_ != $past(trdy_));
bool stop_chng = (stop_ != $past(stop_));
bool devsel_chng = (devsel_ != $past(devsel_));
bool last_data = !irdy_ && (!trdy_ | !stop_) && frame_;

```

Property 45.1

English

A target must assert DEVSEL# before any other response within 1 to 3 clocks following the address phase. If no target responds, a Master-Abort should be performed after 15 cycles.

Note that this property deviates from the actual PCI spec by requiring that the Master-Abort *must* be completed in 15 cycles.

SV

```

property p45_1_a = @(posedge qclk)
                $fell(frame_) => before ((!devsel_ || idle), (!trdy_ || !stop_));
property p45_1_b = @(posedge qclk)
                $fell(frame_) => ([1:3] !devsel_) or ([1:15] frame_);

```

Property 45.2

English

For an IO cycle, if the initiator asserts byte-enables of lesser significance than what is indicated by AD[1:0] the target must terminate the transaction with target abort

SV

```

sequence s45_2 = ($fell(frame_) && io_cmd) ; [1] byte_enable_mismatch;
property p45_2 = @(posedge qclk)
                (ended s45_2) => before(target_abort, (!trdy_ || disconnect));

```

Property 45.3

English

During read transactions, no target may assert TRDY# in the first cycle following the assertion of FRAME#".

SV

```
property p45_3 = @qclk ($fell(frame_) && read_transaction) => [1] trdy_;
```

Property 45.4

English

Once a target has asserted TRDY# or STOP# it cannot change DEVSEL#, TRDY#, or STOP# until the current data phase completes.

SV

```
bool phase_change = (!(trdy_chng | stop_chng | devsel_chng);  
property p45_4 = @(posedge qclk) (!trdy_ || !stop_) => phase_change * [1:inf] ;[0] !irdy_;
```

Property 45.5

English

Once DEVSEL# has been asserted, it cannot be deasserted until the last data phase has been completed, except to signal Target-Abort.

SV

```
property p45_5 = @(posedge qclk)  
    $fell(devsel_) => !devsel_ * [1:inf] ;[1] ((last_data && !devsel_) || (target_abort));
```

Property 45.6

English

All targets are required to complete the initial data phase of a transaction (read or write) within 16 cycles from the assertion of FRAME#.

Note that this property deviates from the actual PCI spec by ignoring the additional number of cycles a bus bridge is allowed.

SV

```
property p45_6 = (posedge qclk)  
    $fell(frame) => ([1:16] !irdy_ && (!trdy_ | !stop_) or  
    (devsel_ * [1:16] ;[1] $rose(frame_));
```

Property 45.7

English

PERR# has a turnaround cycle on the 4th clock after the last data phase, which is three clocks after the turnaround for AD# lines.

SV

```
property p45_7 = @qclk last_data => [4] !PCI_PERR_en;
```

Property 45.8

English

Once a master has asserted IRDY#, it cannot change IRDY# or FRAME# until the current data phase completes. (Note: DEVSEL# and IRDY# can go low in either order.)

SV

```
property p45_8_a = @(posedge qclk)
  (!irdy_ && !devsel_) => (!irdy_ && frame_steady) *[1:inf] ;
  [0] (!stop_ || !trdy_);
```

```
property p45_8_b = @(posedge qclk)
  ((!irdy_ && devsel_) => !irdy_ * [1:inf] ;[1] ((!devsel_ && !irdy_) || idle);
```

Property 45.9

English

A master is required to assert its IRDY# within 8 clocks from any given data phase (initial and subsequent).

SV

```
property p45_9 = @(posedge qclk)
  (($fell(frame_) || (!frame_ && !irdy_ && !trdy_ )) => [1:8] !irdy_;
```

Property 45.10

English

For a Special Cycle transaction, if the initiator inserted one or more wait states before asserting IRDY# with the message, the master must extend the master abort timeout period (that is to say the time before it terminates the transaction) by at least the same number of wait states.

SV

```
clocking @(posedge qclk);
bool special_cycle = $fell(frame) && ( PCI_BE[3:0] == SPLCYC );
```

```

logic wait_cycles = 0;

assign wait_cycles <= special_cycle ? 0 :
    (irdy ? wait_cycles + 1 :
    (!irdy_ ? wait_cycles - 1 : wait_cycles-1));
sequence s45_10 = special_cycle ;[1] irdy_ * [1:inf] ;[1] !irdy_;

property p45_10 = (ended s45_10) => !irdy_ * [1:inf] ;[1] (wait_cycles == 0);

endclocking;

```

Property 46

English

Whenever a “read” signal is asserted, “busy” has to be asserted for 3 cycles and then de-asserted. If an additional “read” arrives before “busy” has been de-asserted, then “busy” has to stay high for 3 cycles from the last “read”. The value of “busy” at the same cycle in which “read” is asserted is not important.

SV

```

// The following is written based on the problem description. It differs from Sugar
// implementation in that read should be aligned at the end
property p46 = read => [1] (busy * [3] ;[1] !busy) or (busy * [0:3] ;[0] read);

```

Property 47

English

bit vector “x[7:0]” is not allowed to contains more than one bit asserted. Moreover, if bit p ($0 \leq p \leq 7$) of x is asserted at time N and bit q of x is asserted at time M, then $|M-N| > 4$.

SV

```

bool nobitchange(data) = (x != 0) ? (x == data) : 1;
property p47_b = @clk (x != 0) =>
    (int i = x; ($count(i) <= 1) ;
    [1] (throughout (nobitchange(i)) within [3] true);

```

Property 48

English

Put an assumption on the environment such that the run is initiated by 5 clk cycles of rst followed by rst staying low forever.

SV

```

// this only specifies a property. fairness/restrict are not supported
clocking clk48 @(posedge qclk);

```

```

property p48 = initial (rst * [5] ;[1] !rst * [1:inf]);
logic count_clk = 0;
count_clk <= (count_clk > 5) ? 6 : count_clk = count_clk + 1;
property p48_a = initial ([7] (count_clk == 6)*[inf]);
endclocking

```

Property 53

English

Write an assumption that specifies that once the design enters a power-down mode it stays in this mode and the design may choose to enter the power-down mode only during initialization period, that is, only until `new_cycle` is set for the first time.

SV

```

// assumptions are not supported, but are written as properties
property p53 = power_down ;[1] power_down;
property p53_a = (new_cycle && !power_down) => (!power_down) * [inf];

```

Property 54

English

Write an assumption that constrains the behavior of the “op-code instruction markers” (i.e., vectors `first_opcode_input` and `last_byte_input`). The required behavior is as follows: if bit `j` in vector `instr_val_input` is zero then both markers need to be zero in index `j`.

Remark: `first_opcode_input` is a vector such that its element `j` is set iff `j` is the first byte of an opcode. Similarly, `last_byte_input` is a vector such that its element `j` is set iff `j` is the last byte of an opcode.

SV

```

// assumptions are not supported, but are written as properties
`define instr_buffer_size 16;
`define msb_instr_buffer (`instr_buffer_size-1);

property p54 = (instr_val_input[`msb_inst_buffer:0] |
    (~(first_opcode_input[`msb_instr_buffer:0] |
        last_byte_input[`msb_instr_buffer:0])))
    == `instr_buffer_size `b1;

```

Property 55

English

Write a parametric assertion that states that every byte that gets into the queue will eventually be taken out of the queue. The parameter represents the width of the address of ele-

ments in the queue. This behavior is interrupted once the signals reset or raprep_cycle831 are set.

SV

```
bool [n:0] addr_in(n) = lin_addr_input[n+(instr_buffer_size-1):instr_buffer_size];
bool [n:0] addr_out(n) = lin_addr_output[n+(instr_buffer_size-1):instr_buffer_size];
property p55(n) = accept ( reset | raprep_cycle831 )
    @ (posedge qclk) valid_new_cycle =>
        (logic addr[n-1:0] = addr_in(n);
         [1:inf] new_load &&(addr_out(n) == addr));
```

Property 56

English

write an assumption that restricts the behavior of the lin_addr_input vector such that its 4 low level bits (called align) and its higher level bit (rest 28 bits, called addr) behave as follows:

In case of a reset in previous cycle addr is set to zero and align gets an arbitrary value.

In case of a valid new cycle addr and align are advanced to the next cache line.

In case of a branch advance to the branch address.

SV

```
// assumptions are not supported, but are written as properties
template generate_clip(m,n);
clocking clk56 @(posedge qclk);
    logic addr[n-1:0] = 0;
    logic align[m-1:0] = 0;
    assign addr <= $past(reset) ? n'b0 :
        (valid_new_cycle&& btclear13h ?
         br_target[n+(instr_buffer_size-1):instr_buffer_size] :
         (valid_new_cycle ? addr + 1 : addr));
    assign align <= $past(reset) ? $random() :
        (valid_new_cycle&& btclear13h ? br_target[m-1,0] :
         (valid_new_cycle ? m'b0 : align));
endclocking
endtemplate
generate_clip my_gen_clip(2,1);
property p56_a = @(posedge qclk) (my_gen_clip.addr == lin_addr_input[5:4]);
property p56_b = @(posedge qclk) (my_gen_clip.align == lin_addr_input[0:0]);
```

Property 57

English

Every request which is acknowledged (signal req must stay asserted

until ack is received) must be followed at some point in the future with a valid grant (a grant which is not aborted the following cycle)

SV

```
sequence s_57 = req *[1:inf] ;[1] ack;  
property p_57 = (ended s_57) => [1:inf] grant [1] !abort;
```

Property 58

English

Every request which is not retried (a retry happens or not two cycles after assertion of signal req) must be followed by a sequence in which busy is asserted until and is asserted (if end is never asserted, then busy must stay asserted forever).

SV

```
sequence s58 = req ;[2] !retry;  
property p58 = (ended s58) => [1] (end or (busy *[1:inf] ;[1] end));
```

Property 59

English

Every request which is not retried (a retry happens or not two cycles after assertion of signal req) must eventually receive an ack

SV

```
sequence s59 = req ;[2] !retry;  
property p59 = (ended s59) => [0:inf] ack;
```

Property 60

English

On every assertion of hi_pri_req, one of the next two assertions of gnt must be to a high priority requestor (dst=hi_pri).

SV

```
property p60 = (hi_pri_req) => next_event(gnt) * [1:2] ;[0] (dst==hi_pri);
```

Property 61

English

Every transaction must complete, and within every transaction, a full data transfer must occur.

SV

```
property p_61 = tr_start =>
    (data_start ;[1:inf] data_end) within next_event(tr_end);
```

Property 62

English

A sequence beginning with the assertion of signal go, containing eight not necessarily consecutive assertions of signal get, during which kill is not asserted, must be followed by a sequence of eight assertions of signal put before signal end can be asserted.

SV

```
sequence s62 = go ;[1] (throughout !kill within (get *= [8]));
property p62 = (ended s62) => [1] throughout !end within (put *= [8]);
```

Property 65

English

Reset can rise asynchronously, but falls on the rising edge of clk. Once asserted reset stays high at least 6 full clk cycles, where clk cycles are of indeterminate length. It is possible that reset eventually asserts forever.

SV

```
// lowest clock is a clock with frequency of the finest granularity
property p65 = @(posedge clk) $rose(reset) => reset * [6];
property p65_a = @(posedge lowest_clk) $fell(reset) => $rose(clk);
```

Property 66

English

When clk is high, if port1 has a v115 request to port2 then eventually port2 is granted to port1. The property holds after reset. Also write an assumption that for all 16 ports and for all 16 virtual-lanes, requests must keep pending until they are granted.

SV

```
genvar I,j;
property p_66 = @(posedge clk)
    (!reset && (p1_vl15_req_port == 2 )&& p1_vl15_req_valid) =>
        [1:inf] p2_vl15_gnt_issued && (p2_vl15_gnt_port == 1);
generate
    for (i=0; i<16; i = i+1;) begin : b1
        for (j=0;j<16; j = j+1;) begin : b2
            property p66[i,j] = @(posedge clk)
                p_vl_req_valid[I][j] =>
                    [1] (p_vl_req_valid[I][j] &&
                        (p_vl_req_port[I][j] == $past(p_vl_req_port[I][j])) * [1:inf]
                        [1] p_vl_0_arb_gnt[I][j];
        end
    end
endgenerate
// Assumptions must be specified the generated properties.
```

Property 67

English

Write an assumption that for each of the sixteen input ports, a port cannot request the same output port in both vl0 and vl15 simultaneously.

SV

```
genvar n;
generate
    for (n = 0;n<16; n = n+1) begin d1:
        property p_67[n] = (p_vl10_arb_req[5][n] && p_vl15_arb_req[5][n]) =>
            (p_vl10_arb_req[4:0][n] != p_vl10_arb_req[4:0][n]);
    end
endgenerate
```

Assumptions must be specified the generated properties.

Property 68

English

Put an assumption on the environment such that the run is initiated by rst cycle of at least 5 clk cycles, followed by rst staying low forever.

SV

```
// assumptions are not supported. Assumptions are described as a properties
clocking clk68 @(posedge clk);
property p48 = initial (rst * [5:inf] ;[1] !rst * [1:inf]);
logic count_clk = 0;
assign count_clk <= (count_clk > 5) ? 6 : count_clk = count_clk + 1;
property p48_a = initial ([7] (count_clk == 6)*[inf]);
endclocking
```

Property 69

English

If within 8 cycles from the beginning of the transaction, 'p_start_reg' and 'discard_rx_' both appear, then the valid signal should be deasserted in the next cycle.

SV

```
sequence s69 = start_pack ;[1] ((p_start_reg within (discard_rx_ within [8] true));
property p69 = ended s69 => [1] !valid;
```

Property 70

English

transaction sequence with two data-transfers, during which an error occurs, should be retried.

SV

```
sequence s70 = (!reset ;[1] trans_start ;[1] data * [=2] ;[1] trans_end) ;
sequence s70_a = error within s70;
property p70 = (ended s70_a) => [0:4] retry;
```

Property 71

English

The number of retries cannot be greater than 20, if the number of retries is greater then 20 a special error flag will be asserted.

SV

```
bool bustag = { ad[0:63], par, c_be[0:3],req64_ };
sequence s71 = $fell(frame_) ;
[0] (int tag = bustag ;
[1] (!frame_)*[0:inf] ;[1] !stop_ ;
[1] !($fell(frame)&&(bustag == tag)) * [1:inf]);
```

```
sequence s71_a = s71 * [21];
property p71 = (ended s71_a) => [1] retry_error;
```

Property 72

English

In this example, the behavior of a simple cache is specified. In this cache each address has a given number of lives. Each time an address is read, the lives of all other addresses in the cache is reduced by one. When the life of an address reaches zero, the address is removed from the cache's memory. When an address is reread it is returned all of its lives.

SV

```
sequence 72(data) = [1] !($rose(read) && (data == address[7:0])) * [1:inf];
sequence s72_a(data) = s72(data) intersect ($rose(read) *= [CACHESIZE]);
property p72 = $rose(read) =>
    (int tag = address[7:0];
     (s72_a(tag)) => [1] !inCache(tag));
```

Property 74

English

We model a bus with a Bus No Request (BNR) signal. Generally, the bus is either: (a) free, requests may be sent. (b) stalled, requests may not be sent. (c) throttled, one request may be sent.

A request is signaled by asserting ADS. In state free ADS may be asserted freely (but at least 3 clks apart). In state stalled ADS may not be asserted. In state throttled ADS may be asserted once.

If the bus is stalled or throttled, BNR is sampled 2 clocks after the previous sampling. If the bus is free, BNR is sampled 3 clocks after ADS is asserted.

Transition from state to state:

In state free, if BNR is asserted move to state stalled.

In state stalled, if BNR is not asserted move to state throttled.

In state throttled, if BNR is asserted return to state stalled, otherwise move to state free.

SV

```
// assuming state values free, throttled and stalled are defined elsewhere
function next_state logic[1:0] (logic in_state, bnr_in);
```

```

logic[1:0] state_out;
case (state_in)
  free: if (bnr_in) state_out = stalled;
        else state_out = free;
  stalled: if (!bnr_in) state_out = throttled;
           else state_out = stalled;
  throttled: if (bnr) state_out = stalled;
            else state_out = free;
next_state = state_out;
endfunction

clocking fsm_clk @(posedge clk);
  logic sample,prev2_sample,prev1_sample,prev3_ads,prev2_ads,prev1_ads;
  logic state[1:0] ;
  assign sample <= (state==free) ? prev3_ads : prev2_sample;
  assign prev2_sample <= prev1_sample;
  assign prev1_sample <= sample;
  assign prev3_ads <= prev2_ads;
  assign prev2_ads <= prev2_ads;
  assign prev1_ads <= ads;
  assign state <= (sample) ? next_state(state,sample,bnr): state;

  property p74_a = ads => [1] (!ads * [1:3])
  property p74_b = ((state==throttled) &&ads) =>
    [1] (!ads *[1:inf]) ;[1] (state != throttled);
  property p74_c = (state==stalled) => !ads;
endclocking

```