**Property 1**

- **English** : If a snoop hits a modified line in the I1 cache, then the next transaction must be a snoop writeback. (design 1)

- **Sugar**
  a. AG ((snoop & hitm) -> AX next_event(trans_start)(writeback))
  b. AG {snoop & hitm} |=> {!trans_start[*],trans_start & writeback}
  c. {[*],snoop & hitm} |=> {!trans_start[*],trans_start & writeback}

- **ForSpec**

  wnext_event(t,e) := !t WUNTIL t&e;

  c1 := always snoop & hitm -> NEXT wnext_event(trans_start, writeback);

  Alternatively, the property may be written as

  c1 := true*, snoop & hitm  TRIGGERS NEXT wnext_event(trans_start,writeback);

  /* Note: the same property definition can be used for either assumption or assertion */

  assert c1;

  assume c1;

- **E**

  expect p1 is (@snoop and @hitm) => ((@writeback or fail cycle) @trans_start);

- **CBV**

**Property 2**

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

- **Sugar**
  a. AG (rose(enable) -> AX next_event(transfer)[4](AX rose(pending)))
  b. AG {!enable,enable,{!transfer[*],transfer}[4]} |-> {!pending,pending}
  c. {[*],rose(enable),{!transfer[*],transfer}[4]} |=> {rose(pending)}

- **ForSpec**

  c2 := always a_rise(enable),(!transfer*,transfer){4} TRIGGERS b_rise(pending);

  assert c2; /* if you need c2 as an  assertion */

  assume c2; /*if you need c2 as an assumption  */

- **E**

  expect p2 is {rise('enable'); ([4] @transfer)} => rise('pending');

  The p2 illustrates the use of abstract/multiple locks. Event @transfer acts as a clock by sampling subexpression ([4]). Event @transfer has a granularity larger than that of

the default clock, which is sensitive to any event in the system. (The default clock defines the granularity of the ';' hidden under '=>'.)

Event @transfer acts as a strong clock because for ([4]*cycle)
@transfer to succeed four events @transfer must occur. Note that
event @transfer could be an atomic event such as true(``transfer'), or it could be a defined as a composite expression. In the latter case, the p2 would also illustrate the use of asbstract clocks.

- CBV

**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)

- **Sugar**
  a. AG (boff -> AX next_event(accepted)(!snoop_req -> write_req))
  b. AG {boff,!accepted[*],accepted} |-> {!snoop_req -> write_req}

- **ForSpec**
  first_acc_trans_type(t) := (!accepted*, accepted) TRIGGERS t;

  c3 := always boff -> next first_acc_trans_type(snoop_req|write_req);

  assert c3; /* if you need c3 as an assertion */
  assume c3; /* if you need c3 as an assumption */
- **E**

  expect p3 is @boff => ((@snoop_req or @write_req or fail cycle) @ accepted);

- **CBV**

**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)

- **Sugar:**
  a. AG ((hit & !pending) -> next_event(pending)(sel5))
  b. AG {hit & !pending, !pending[*], pending} |-> {sel5}

- **ForSpec**

  c4 := always (hit \ !pending+, pending) triggers sel5;

  assert c4; /* if you need c4 as an assertion */
  assume c4; /* if you need c4 as an assumption */

- **E**
  expect p4 is (@hit and (fail @pending)) => ((fail @sel5 or fail cycle) @pending);
- **CBV**

**Property 5**

- **English** : An urgent request should be the next handled. (design 2)
- **Sugar**
  a. AG (urgent_req -> AX next_event(grant)(urgent_answered))
  b. {[*], urgent_req, !grant[*], grant} |-> {urgent_answered}
- **ForSpec**

```
c5 := true*, urgent_req, !grant*, grant  TRIGGERS
                                  urgent_answered;

assert c5; /* if you need c5 as  an assertion  */
assume c5; /* if you need c5 as an assumption  */
```

- **E**

  expect p5 is @urgent_req => ((@urgent_anwered or fail cycle) @grant);

- **CBV**

**Property 6**

- **English:** Between a request and its acknowledge the busy signal must remain asserted. (design 2)
- **Sugar**
  a. AG (req -> (busy until ack))
  b. {[*], req} |-> {busy[*], ack}
- **ForSpec**

```
c6 := always req -> (busy until ack);

assert c6; /* if you need c6 as  an assertion  */
assume c6; /* if you need c6 as an assumption  */
```

- **E**

  expect p6 is (@req and fail @ack) => {[..]*@busy;@ack} or
  fail {[..];fail(@busy and fail @ack)};

- **CBV**

**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)
- **Sugar**
  a. AG {dp_start, !LAST[*], LAST}(next_event(dp_start)(FIRST))
  b. {[*], dp_start, !LAST[*], LAST ~ {!dp_start[*], dp_start}} |-> {FIRST}

- **ForSpec**

```
c7 := always (dp_start, !LAST*, LAST \ !dp_start*, dp_start)
triggers FIRST;

assert c7; /* if you need c7 as an assertion  */
assume c7; /* if you need c7 as  an assumption  */
```

- **E**

  expect p7 is {@dp_start; cycle @LAST} => ((@FIRST or fail cycle) @dp_start);

- **CBV**

**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)

- **Sugar**
  forall N: 1..8:
  AG within(write_command_start & size=N,LAST){gx_start[=N]}

- **ForSpec**
  bit[3] N;
  init N=0;
  assign N'= ( (write_command_start) ? size : N ) - zx(gx_start,3);

  (a) c8 := always write_command_start -> !LAST & ( N=0 -> !gx_start) wuntil LAST & N=0;

  (b) c8 := always write_command_start -> !LAST & N>0 wuntil !LAST & N=0 & !gx_start wuntil LAST;

  assert c8; // if c8 is used as assumption
  assume c8; // if c8 is used as assertion

- **E**
  // assuming that once write_command_start is asserted, it is not asserted
  // again before LAST is asserted (no overlapping commands)

  N: int;
  on write_command_start {
  N=size;
  };

  expect p8 is fail {@write_command_start;{[..];@LAST}} or
  {@write_command_start;{[..];@LAST}} and
  {[N] @gx_start;~[1..]};

- **CBV**

**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)

- **Sugar**
  forall x(3..0): boolean:
  AG ((addr_queue_ptr_p_q(3..0)=x(3..0)) ->
  next_event((addr_queue_ptr_p_q(3..0)!=x(3..0)))
  ((addr_queue_ptr_p_q(3..0)=(x(3..0) + 1))))

- **ForSpec**

```
addr := addr_queue_ptr_p_q[3:0];
c9 := NEXT always addr=past(addr) | addr=past(addr)+1;

assert c9; /* if you need c9 as an assertion  */
assume c9; /* if you need c9 as  an assumption  */
```

- E
- CBV

**Property 10**

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

- **Sugar**
  a. AG (data_grant -> next_event(dbusy & prev(dbusy))(dvalid))
  b. AG {data_grant, !dbusy[*], dbusy[2]} |-> {dvalid}

- **ForSpec**

```
Note: Sugar (a) and (b) are not equivalent, ForSpec follows (a)

c10 := ALWAYS (data_grant, !(dbusy & past(dbusy))*, dbusy{2})
TRIGGERS dvalid;
```

- **E**

```
expect p10 is {@data_grant; {[..];{@dbusy;@dbusy}}} => detach
{cycle;@dvalid};
```

- **CBV**

**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)

- **Sugar**

```
forall dbits(0..7): boolean:
forall addr(3..6): Boolean:
AG( {write_valid & reg_addr(11..14)=addr(3..6) &
    data_bus(0..7)=dbits(0..7),
     !(write_valid & reg_addr(11..14)=addr(3..6))[*],
```

```
        read_valid & reg_addr(11..14)=addr(3..6)} |->
{data_bus(0..7) = dbits(0..7)}
```

- **ForSpec**

```
memory_element(addr) :=
{
      write_cond := write_valid & reg_addr[14:11]=addr;

      bit[8] dbits;
      assign_on (write_cond) dbits'=data_bus[7:0];
      data := dbits';

      bit was_set;
      init was_set=false;
      assign_on (write_cond) was_set'=true;

      c11 := always read_valid & reg_addr[14:11]=addr & was_set'
      ->  data_bus[7:0]=dbits';

      assert c11;  // if the property has to be checked
      assume c11;  // if the property has to assumed
};

for i := 0 to 15 {  new memory_element(i); };
```

- **E**
- **CBV**

**Property 12**

- **English:** Every buffer will be read before it is overwritten. (design 6)

- **Sugar:**

```
forall addrbits(0..1): boolean:
AG ((write_enable & write_address(0..1)=addrbits(0..1)) ->
  AX ((read_enable  & read_address(0..1) =addrbits(0..1)) before
   (write_enable & write_address(0..1)=addrbits(0..1))))
```

- **ForSpec**

```
If there is no requirement to use this property as an assumption,
the code below may also be used:

before(x,y) := !y wuntil x&!y;
rigid bit[2] addrbits;
assert c12 := ALWAYS write_enable & write_address[1:0]=addrbits ->
                     NEXT before (read_enable &
read_address[1:0]=addrbits,
                     write_enable & write_address[1:0]=addrbits);



If  both assumption and assertion are needed use the following
alternative version of the property:

bit[4] buffer;
```

```
init buffer=0;
assign buffer'= case {
     write_enable : buffer | (1 << write_address[1:0]);
     read_enable : buffer & ~(1 << read_address[1:0]);
     default : buffer };
write_legal := (buffer & (1 << write_address[1:0])) = 0;

c12 := ALWAYS write_enable -> write_legal;
```

- **E**
- **CBV**

**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.
- **Sugar**
  AG within(write_start, write_end)
  {{!write_en[*], write_en & !addr(0), !write_en[*], write_en & addr(0)}[+]}
- **ForSpec**

  weak_within(start,event,end) := start, event* TRIGGERS NEXT end | REJECT_ON (end) event;

  c14 := always weak_within (write_start, (!write_en*, write_en & !addr[0], !write_en*, write_en & addr[0]), write_end);
- **E**
- **CBV**

**Property 15**

- **English:** Two consecutive writes cannot be to the same address. Address appears one cycle after the write_valid.(design 6)
- **Sugar**
  a. forall addr(0..7): boolean:
  AG (write_valid -> AX ((addr_bus(0..7)=addr(0..7)) ->
  AX next_event(write_valid)(AX (addr_bus(0..7)!=addr(0..7)))))

  b. forall addr(0..7): boolean:
  {[*],write_valid,addr_bus(0..7)=addr(0..7),!write_valid[*],write_valid} |=>
  {addr_bus(0..7) != addr(0..7)}

- **ForSpec**

```
c15:= change_if (write_valid') wnext[2] always addr_bus[7:0] !=
past(addr_bus[7:0]);
```

- **E**
- **CBV**

**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.

- **Sugar**

a. forall addr(0..4): boolean:
AG ((write_en & data_valid & write_address(0..4)=addr(0..4)) ->
next_event (retire & retire_address(0..4)=addr(0..4))
(ABF[3..8] (write_en & !data_valid & write_address(0..4)=addr(0..4))))

b. forall addr(0..4): boolean:
AG {write_en & data_valid & write_address(0..4)=addr(0..4),
!(retire & retire_address(0..4)=addr(0..4))[*],
(retire & retire_address(0..4)=addr(0..4))} |->
{[3..8], write_en & !data_valid & write_address(0..4)=addr(0..4)}
•

- **ForSpec**

```
addr_behavior(addr) :=
{
        addr_write := write_en & write_address[4:0]=addr;
        addr_retire := retire & retire_address[4:0]=addr;
        c16 := always
             (addr_write & data_valid, !addr_retire*, addr_retire)
           TRIGGERS
              eventually[3,8] addr_write & !data_valid;
        assert c16;   /* if c16 is to be used as property to be verified */
        assume c16;   /* if c16 is to be used as assumption on the environment */
}

for (i=0 to 31) { new addr_behavior(i); }
```

- **E**

- **CBV**

**Property 17**

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

- **Sugar**

a. AG (rose(read_req) -> next_event_f(rose(out_en))[1..2](output_read))

b. AG {!read_req, read_req} |-> {{!out_en[+], out_en}[1..2]~
output_read}

- **ForSpec**

```
c17 := always a_rise(read_req) -> (!out_en+, out_en){1,2} \
output_read;
```

- **E**

- **CBV**

**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.)

- **Sugar:**
AG {{rose(read_req),[*],rose(write_req)} && !(out_en & output_read)[*]}
((out_en & output_read) before (out_en & output_write))

- **ForSpec**

```
strong_before(x,y) := !y*, x&!y;
before(x,y) := !y wuntil x&!y;

c18 := always (a_rise(read_req), strong_before(a_rise(write_req), out_en &
output_read))
                triggers before(out_en & output_read, out_en & output_write);
```

- **E**

- **CBV**

**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.

- **Sugar**

```
forall tag(0..73): boolean:
AG {fell(frame_) &
bustag(0..73)=tag(0..73),irdy_[1..8],trdy_[1..16],!stop_} |->
  {
    {[1..31999],fell(frame_) &
bustag(0..73)=tag(0..73),irdy_[1..8],trdy_[1..16],!trdy_} ||
    {[1..31999],fell(frame_) &
bustag(0..73)=tag(0..73),irdy_[1..8],trdy_[1..16],!stop_} ||
    {[32000..32100], BridgeControlRegister[10]}
  }
```

where bustag(0..73) is what you have written as {address, address_parity, command, byte_enables, REQ64#}.

however, i'm not sure that i understand your requirement of multiple scopes for variables. you list two kinds of variables other than variables in the design itself: assertion check variables and local assertion variables. what is the difference? it seems like you mean that tag(0..73) above is a "local assertion variable". so what do you mean by "assertion check variables"?

i do not understand at all your idea for checking this using procedural code. you say that at the end of the transaction, when the outcome is known, you would embed a call to a piece of code that handles all of the tag matching etc. however, the purpose of the property above is to check that transactions complete correctly. so how can you check whether a transaction completes correctly by waiting for it to complete?

- **ForSpec**

```
rigid bit tag[74];
rdy_sequence := a_fall(frame_) & bustag=tag, irdy_{1,8}, trdy_{1,16};
retry := rdy_sequence, !stop_;
```

```
good_completion := EVENTUALLY[1,31999] rdy_sequence, !trdy_;
retry_again := EVENTUALLY[1,31999] rdy_sequece, !stop_;
discard := EVENTUALLY[32000,32100] BridgeControlRegister[10];
spec := ALWAYS retry TRIGGERS
                good_completion | retry_again | discard;
```

The above is a straight translation of sugar. However looking at the waveform in property_20.pdf gives a different picture. The following are assumptions made when interpreting the waveform:

- signal frame_ falls iff a transaction tries to complete
- devsel_ is connected to the tag associated with each transaction

Two properties are needed to capture the model behavior:

- describing a transaction completion attempt
- when a transaction is referred to retry it will retry or discard

Only the second property is captured in the pseudo code above.

```
rigid bit[74] tag;
tagmatch := tag = (address % address_parity % command %
                   byte_enables % REQ64#);
start := a_fall(frame), tagmatch;
wait := irdy_{0,7}, !irdy_ & trdy_ & stop_{1,8};
end := !irdy & (trdy_ & !stop_ | !trdy_ & stop_);

// transaction tries to complete
ALWAYS start TRIGGERS wait, end;

// transaction retry
ALWAYS start, wait, end \ !stop_ TRIGGERS
  (EVENTUALLY[1,31999] start)
   | (EVENTUALLY[32000,32100] BridgeControlRegister[10];
```

- **E**
- **CBV**

**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 xx0011111100 to xx0100000000.

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 = (CachelineSize - (InitialAddress & (CachelineSize -1))) >> 2;

Using the example above

X = ( 000001000000 - (xx0011110100 & (000001000000 - 000000000001))) >> 2

= ( 000001000000 - (xx0011110100 & (000000111111))) >> 2

= ( 000001000000 - (000000110100)) >> 2

= ( 000000001100) >> 2

= 3

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

- **Sugar**

Assumptions: - legal cache line sizes are 32, 64, 128, and 256
bus is working in default 32 bit mode
Note: because of the way it is calculated (as (CachelineSize - (AD & (CachelineSize-1))) >> 2), N can range from 1 through the maximum cache line size divided by 4. in the example, the maximum cache line size is 256, so the maximum N is 64.

```
forall CachelineSize: {32, 64, 128, 256}:
forall N: 1..64:
AG {CachelineSizeRegSupported &
CachelineSizeReg=CachelineSize &
fell(frame_) & cbe_=MemoryReadLine &
((CachelineSize - (AD & (CachelineSize-1))) >> 2) = N} |->
{{(!IRDY_ & !TRDY_)[<N] && !frame_[*]}, !(!IRDY_ & !TRDY_)[*],frame_}
```

- **ForSpec**

```
rigid bit CachelineSize[9];
assume strong_mutex(CachelineSize) & CachelineSize > 31;

rdy := IRDY_ | TRDY_ ;
trigger :=  CachelineSizeRegSupported &
CachelineSizeReg=CachelineSize &
fell(frame_) & cbe_=MemoryReadLine;

bit N[7];
init N = 0;
assign N' = case {
        frame_ : 0;
        trigger : (CachelineSize - (AD & (CachelineSize-1))) >> 2;
        !rdy : N-1;
        default : N;
                };

spec := ALWAYS trigger, !frame_*, frame TRIGGERS (N > 0);
```

- **E**

- **CBV**

**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#.

- **Sugar**

  Assumptions: "disconnect with data" = !IRDY_ & !TRDY_ & !STOP_ "disconnect without data" = !IRDY_ & TRDY_ & !STOP_

  Note: I have only coded the first of the two requirements ("if A then C" in Bernard's mail). The second requirement is done similarly.

  a) (if the value of device registers are only visible by monitoring the bus during initialization)

  ```
  AG ((initializing & CachelineSizeRegSupported) ->
  AG ((fell(frame_) & (cbe_=MemoryReadLine |
  cbe_=MemoryReadMultiple)) ->
  next_event(!IRDY_ & !TRDY_)
     (!STOP_ | AX ((!STOP_ & TRDY_) before !TRDY_))))
  ```

  b) (if the value of device registers can be determined by looking into the design during verification)

  ```
  AG ((CachelineSizeRegSupported &
  ```

```
fell(frame_) & (cbe_=MemoryReadLine |
cbe_=MemoryReadMultiple)) ->
    next_event(!IRDY_ & !TRDY_)
        (!STOP_ | AX ((!STOP_ & TRDY_) before !TRDY_)))
```

- **ForSpec**

```
(a)
ready := !IRDY & !TRDY_;

true*, initializing & CachelineSizeRegSupported, true*,
fell(frame_) & (cbe_=MemoryReadLine | cbe_=MemoryReadMultiple) ->
!ready*, ready \
(!STOP_ | (STOP_, !ready*, ready & !STOP_));

(b)
ready := !IRDY & !TRDY_;

ALWAYS ( CachelineSizeRegSupported & fell(frame_) &
(cbe_=MemoryReadLine | cbe_=MemoryReadMultiple)) ->
!ready*, ready \
(!STOP_ | (STOP_, !ready*, ready & !STOP_));
```

- **E**
- **CBV**

**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 asseting (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.

(Definitions)

- Sugar

```
#define SPLCYC  = 01h
#define IOREAD  = 02h
#define IOWRITE = 03h
#define RESVD1  = 04h
#define RESVD2  = 05h
#define MEMYRD  = 06h
#define MEMYWR  = 07h
#define RESVD3  = 08h
#define RESVD4  = 09h
#define CONFRD  = 0ah
#define CONFWR  = 0bh
#define MEMRDM  = 0ch
#define DUADCY  = 0dh
#define MEMRDL  = 0eh
#define MEMWRI  = 0fh

define frame_  := PCI_FRAME;
define trdy_   := PCI_TRDY;
define irdy_   := PCI_IRDY;
define stop_   := PCI_STOP;
define devsel_ := PCI_DEVSEL;

define frame_chng := frame_ != prev(frame_);
define trdy_chng := trdy_ != prev(trdy_);
define irdy_chng := irdy_ != prev(irdy_);
define stop_chng := stop_ != prev(stop_);
define devsel_chng := devsel_ != prev(devsel);

define frame_steady := frame_ = prev(frame_);
define trdy_steady := trdy_ = prev(trdy_);
define irdy_steady := irdy_ = prev(irdy_);
define stop_steady := stop_ = prev(stop_);
define devsel_steady := devsel_ = prev(devsel);

define qclk := PCI_RST & rose(PCI_CLK);

define data_phase := !irdy_ & !trdy_;

define last_data := !irdy_ & (!trdy_ | !stop_) & frame_;

define target_abort := rose(devsel_) & fell(stop_);
```

```
define idle := frame_ & idry_;

define disconnect := !devsel_ & !stop_;

define read_transaction :=
            PCI_BE(3..0) in {IOREAD, MEMRDM, MEMYRD, CONFRD, MEMRDL};

define io_cmd := PCI_BE(3..0) in {IOREAD, IOWRITE};

var current_trans_ad(1..0): boolean;
assign next(current_trans_ad(1..0)) := AD(1..0);

define byte_enable_mismatch := (current_trans_ad(1..0)=1 & PCI_BE(0)!=1b) |
                        (current_trans_ad(1..0)=2 &
PCI_BE(1..0)!=11b) |
                        (current_trans_ad(1..0)=3 &
PCI_BE(2..0)!=111b);
```

- ForSpec

```
// Mappings
clk := a_rise(PCI_CLK);
qclk := PCI_RST & clk;
irdy := PCI_IRDY;
trdy := PCI_TRDY;
frame := PCI_FRAME;
devsel := PCI_DEVSEL;
stop := PCI_STOP;
BE := PCI_BE[3:0];

command := type { SPLCYC, … , MEMWRI };
pci_command( val ) := case val
{
0x1 : SPLCYC;
0x2 : IOREAD;
…
0xe : MEMRDL;
default : MEMWRI;
};

// Event templates
rise (a) := past(!a) & a;
fall (a) := past(a) & !a;
change (a) := rise(a) | fall(a);

// Definitions
data_phase := !irdy & !trdy;

io_cycle :=  data_phase & (pci_command(BE)= IOREAD | pci_command(BE)= IOWRITE);
target_abort := fall(frame), !frame*, fall(devsel), !devsel*, devsel & fall(stop);
last_data := !irdy & (!trdy | !stop) & frame;
```

- 
- E

```
type PCI_COMMAND        : [INTACK = 0x0,
 SPLCYC = 0x1,
 IOREAD = 0x2,
```

```
                IOWRITE = 0x3,
                RESVD1 = 0x4,
                RESVD2 = 0x5,
                MEMYRD = 0x6,
                MEMYWR = 0x7,
                RESVD3 = 0x8,
                RESVD4 = 0x9,
                CONFRD = 0xa,
                CONFWR = 0xb,
                MEMRDM = 0xc,
                DUADCY = 0xd,
                MEMRDL = 0xe,
                MEMWRI = 0xf] (bits:4);

// PCI_CLK
event clk is rise(`PCI_CLK') @sim;
// qualified clock
event qclk is true(`PCI_RST' == 1) @clk;


event frame_assr is true(`PCI_FRAME' == 0) @qclk;
event frame_rise is rise(`PCI_FRAME') @qclk;
event frame_fall is fall(`PCI_FRAME') @qclk;
event frame_chng is change(`PCI_FRAME') @qclk;

event irdy_assr is true(`PCI_IRDY' == 0) @qclk;
event irdy_fall is fall(`PCI_IRDY') @qclk;
event irdy_chng is change(`PCI_IRDY') @qclk;

event trdy_assr is true(`PCI_TRDY' == 0) @qclk;
event trdy_fall is fall(`PCI_TRDY') @qclk;
event trdy_chng is change(`PCI_TRDY') @qclk;

event stop_assr is true(`PCI_STOP' == 0) @qclk;
event stop_fall is fall(`PCI_STOP') @qclk;
event stop_chng is change(`PCI_STOP') @qclk;

event devsel_assr is true(`PCI_DEVSEL' == 0) @qclk;
event devsel_rise is rise(`PCI_DEVSEL') @qclk;
event devsel_fall is fall(`PCI_DEVSEL') @qclk;
event devsel_chng is change(`PCI_DEVSEL') @qclk;


// signal the data phase
event data_phase is (@irdy_assr and @trdy_assr)@qclk;

// signal last data phase
event last_data is (
   @irdy_assr
    and (@trdy_assr or @stop_assr)
    and not @frame_assr
)@qclk;

// signal target abort
event target_abort is {
   @frame_fall;
    // not a master abort
```

```
    {[..] * fail @frame_chng; @devsel_fall};
    {[..] * fail @devsel_chng; (@devsel_chng and @stop_fall)}
}@qclk;

    //: the expression "SIG.as_a(<type>)" does type conversion; the infix
    //: operator "in" is a range check (cf element _in_ set). the next
    //: three event definitions combine state formulae "true(<exp>)" with
    //: events.

// This event occurs when the pci transaction is an I/O cycle

event io_cycle is (@data_phase and true(`PCI_BE'.as_a(PCI_COMMAND)
                                in [IOREAD, IOWRITE])
)@qclk;
```

**Property 45.1**

**This requires that trdy_fall and stop_fall will not be asserted on the same cycle where frame_fall. Also it seems devsel_fall can appear *with* other behaviors, not *before* them.**

- English: A target must assert DEVSEL# before any other response within 1 to 3 clocks following the address phase. (Note: If no target responds, a Master-Abort should be performed no later than 15 cycles after the first clock where FRAME# is asserted.)
- Sugar
  AG (fell(frame_) ->
    ((!devsel_ | idle) before (!trdy_ | !stop_)))::clk=qclk

```
AG {fell(frame_)} -> {
                {[1..3],!devsel_} ||
                {[1..15], frame_}
              }::clk=qclk
```

- ForSpec

```
Option a:
other_response := fall(trdy) | fall(stop);
normal_behavior := !other_response {1,3} \ fall(devsel);
master_abort:= !( other_response | fall(devsel) ) {0,14} ,
rise(frame);
f := change_if (qclk) always fall(frame) -> next
normal_behavior | master_abort;

Option b:
other_response := fall(trdy) | fall(stop);
normal_behavior := !other_response until[0,2]
!other_response & fall(devsel);
master_abort:= !( other_response | fall(devsel) )
until[0,14] rise(frame);
f := change_if (qclk) always fall(frame) -> next
normal_behavior | master_abort;

assert f; /* if you need f as an assertion  */
```

```
assume f; /* if you need f as  an assumption  */
```

- 
- E:

```
  expect p45_1a is @frame_fall
=> ({ [..] * (fail (@trdy_fall or @stop_fall));
      @devsel_fall}
   or
   fail { [..];(@trdy_fall or @stop_fall) }) @qclk
else
     dut_error("\nPCI_CHECKER: At time ",sys.time,
              " the target erroneously asserted STOP# or TRDY#",
              " before asserting DEVSEL#.");
```

```
expect p45_1b is @frame_fall
  => ( { [..2];  @devsel_fall}
      or
      {[..15]; (fail @frame_assr and fail @irdy_assr)}) @qclk
  else
       dut_error("\nPCI_CHECKER: At time ",sys.time,
              " master abort did not happen 15 cycles ");
```

- CBV

**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
- Sugar:
  AG {(fell(frame_) & io_cmd), byte_enable_mismatch}
   (target_abort before (!trdy_ | disconnect))::clk=qclk
- ForSpec:

```
f := change_if(qclk)
     io_cycle &
        ( (ad[1:0]=1 & be[0]=0) | (ad[1:0]=2 & (be[0]=0 | be[1]=0)) |
(ad[1:0]=3 &
         (be[0]=0 | be[1]=0 |  be[2]=0)) ) triggers !frame*,  target_abort;
```

- 

- **E**

```
!current_trans_ad: uint (bits:32);

on frame_fall {
    current_trans_ad = ^PCI_AD';
};
```

```
// Illegal-> AD[1:0]=1 BE0=0; AD[1:0]=2 BE0/BE1=0; AD[1:0]=3 BE0/BE1/BE2=0

expect (@io_cycle and true(
            (current_trans_ad[1:0] == 1
             and ``PCI_BE'[0:0] == 0)
          or
            (current_trans_ad[1:0] == 2
             and (``PCI_BE'[1:1] == 0 or
                 ``PCI_BE'[0:0] == 0))
          or
            (current_trans_ad[1:0] == 3
             and (``PCI_BE'[2:2] == 0 or
                 ``PCI_BE'[1:1] == 0 or
                 ``PCI_BE'[0:0] == 0))))
  => {
      [..] * @frame_assr;
      @target_abort
    }@qclk;
```

- CBV

**Property 45.3**

- **English:**
  During read transactions, the addressed target must keep TRDY# deasserted for one extra cycle during turnaround cycles.
- **Sugar**
  AG ((fell(frame_) & read_transaction) -> AX trdy_)::clk=qclk

- **ForSpec**

```
read_trans := case pci_command(BE) {
          ioread : true;
          memrdm: true;
          memyrd: true;
          confrd: true;
          memrdl: true;
          default : false;
          };

   f := change_if(qclk) always fall(frame) & read_trans
   triggers next trdy;

   assert f; /* if you need f as an assertion  */
   assume f; /* if you need f as  an assumption  */
```

- E

```
expect ((@frame_fall and true (``PCI_BE'.as_a(PCI_COMMAND)
  in [IOREAD, MEMRDM, MEMYRD, CONFRD, MEMRDL]))
  => fail @trdy_assr) @qclk;
```

- CBV

**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.
- **Sugar**

```
AG ((!trdy_ | !stop_) ->
  (!(trdy_chng | stop_chng | devsel_chng) until_ !irdy_))::clk=qclk
```

  - ForSpec

```
f := change_if(clk) always ( fall(trdy) | fall(stop) ) & irdy  triggers next
(devsel%trdy%stop)=past(devsel%trdy%stop) until !irdy & ( !trdy | !stop);

assert f; /* if you need f as an assertion  */
assume f; /* if you need f as  an assumption  */
```

  -
  - E

```
expect ((@trdy_fall or @stop_fall) and fail @irdy_assr)
 => {
      // it cannot change DEVSEL#, TRDY#, or STOP#
      [..] * fail (@trdy_chng or @stop_chng or @devsel_chng);
      // until the current data phase completes
      @irdy_assr and (@trdy_assr or @stop_assr)
   }@qclk;
```

  -
  - CBV

**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.
- **Sugar**

```
AG (fell(devsel_) ->
  (!devsel_ until ((last_data & !devsel_) |
 target_abort)))::clk=qclk
```

  - ForSpec

```
f := change_if(qclk) always fall(devsel) & !last_data ->
   !devsel until (last_data & !devsel) | (rise(devsel) & !stop) ;

assert f; /* if you need f as an assertion  */
assume f; /* if you need f as  an assumption  */
```

  -
  - E

```
expect (@devsel_fall and not @last_data)
 => ( {    // Target-Abort
        [..] * @devsel_assr;
        @devsel_rise and @stop_assr
    }
```

```
    or
     {     // last data completed
           [..] * @devsel_assr;
           @devsel_assr and @last_data
     }
) @qclk;
```

- CBV

**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#.
- **Sugar**

```
AG {fell(frame_)} |-> {((last_data
                {[1..16], !irdy_ & (!trdy_ | !stop_)} ||
                {devsel_[1..16], rose(frame_)}
              }::clk=qclk
```

- ForSpec

```
f := change_if(clk) always fall(frame) -> next
( true{0,15}, !irdy & ( !trdy | !stop ) ) | ( !fall(devsel){0,15} , rise(frame) );

assert f; /* if you need f as an assertion  */
assume f; /* if you need f as  an assumption  */
```

- E

```
expect @frame_fall

=> ( {
       [..15];
       @irdy_assr and (@trdy_assr or @stop_assr)
     }
     or
     {
       [..15] * (fail @devsel_fall);  //master abort
       @frame_rise
     }
    )@qclk;
```

- 
- CBV

**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.

- **Sugar**

```
AG (last_data -> AX[4] !PCI_PERR_en)::clk=qclk
```

- ForSpec

```
f := change_if(clk) always last_data -> next[4] PCI_PERR_en=0;

assert f; /* if you need f as an assertion  */
assume f; /* if you need f as  an assumption  */
```

- E

```
expect @last_data => { [3]; true(``PCI_PERR_en' == 0) } @qclk;
```

- CBV

**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.)
- **Sugar**

```
AG ((!irdy_ & !devsel_) ->
    ((!irdy_ & frame_steady) until_ (!stop_ | !trdy_)))::clk=qclk

AG ((!irdy_ &  devsel_) ->
    (!irdy_ until ((!devsel_ & !irdy_) | idle)))::clk=qclk
```

- ForSpec

```
f := change_if(qclk) always fall(irdy | devsel) -> next
( !change(irdy) & !change(frame) until !irdy & (!trdy | !stop) );

assert f; /* if you need f as an assertion  */
assume f; /* if you need f as  an assumption  */
```

- E

```
// case 1: DEVSEL# falls before IRDY#

expect p45_8a is (@devsel_assr and @irdy_assr)
      => { [..]* fail (@frame_chng or @irdy_chng);
           @irdy_assr and (@trdy_assr or @stop_assr)
         } @qclk;

   // case 2: IRDY# falls before DEVSEL#

expect p45_8b is (@irdy_fall and fail @devsel_assr)
      => { [..]* (fail @irdy_chng);
           (@irdy_assr and @devsel_assr or
            fail( @frame_assr or @irdy_assr))
         } @qclk;
```

- CBV

**Property 45.9**

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

```
AG ((fell(frame_) | (!frame_ & !irdy_ & !trdy_ )) ->
```

ABF[1..8] !irdy_)::clk=qclk

- **ForSpec**

```
// first and subsequent data phases
f2 := change_if(qclk) always (fall(frame) | !frame & !irdy & !trdy) -> eventually[1,8] !irdy
assert f; /* if you need f as an assertion  */
assume f; /* if you need f as  an assumption  */
```

  - E

```
// check first data phase
expect @frame_fall  => { [..7]; @irdy_assr } @qclk;

// check subsequent data phases

expect (@frame_assr and @irdy_assr and @trdy_assr)
   => {
        [..7];
        @irdy_assr
      } @qclk;
```

- **CBV**

**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 time-out period by at least the same number of wait states.
- **Sugar**

```
forall N: 1..7:
    AG {fell(frame_) & PCI_BE(3..0)=SPLCYC, irdy_[N], !irdy_} |->
   {!irdy_[N]}::clk=qclk
```

- **ForSpec**

```
special_cycle := fall(frame) & ( pci_command(be) = SPLCYC );
initiator_wait_cycle := !frame & (pci_command(be) = SPLCYC) & irdy;
bit [32] wait_cycles, abort_cycles;
init wait_cycles = 0;
assign_on (qclk) wait_cycles' =          (special_cycle) ? 0 :
(initiator_wait_cycle) ? inc(wait_cycles) : wait_cycles;
init abort_cycles = 0;
assign_on (qclk) abort_cycles' = (fall(irdy)) ? wait_cycles : (!irdy) ?
            dec(abort_cycles) : abort_cycles;

f := change_if(qclk) always special_cycle, true*, fall(irdy) triggers !irdy until
abort_cycles=0;

assert f;          // if you need f as an assertion
assume f;          // if you need f as an assumption
```

  - E

```
special_cycle := fall(frame)
event special_cycle_command is (
     true(`PCI_BE'.as_a(PCI_COMMAND) == SPLCYC)
   )@qclk;
```

```
event special_cycle is (
   @frame_fall
    and @special_cycle_command
)@qclk;

event initiator_wait_cycle is (
   @frame_assr
    and @special_cycle_command
    and not @irdy_assr
)@qclk;

// Create a counter to track initiator wait cycles
!initiator_wait_cnt: int;
on special_cycle {
    initiator_wait_cnt = 0;          // Reset counter at transaction start
};
on initiator_wait_cycle {
    initiator_wait_cnt +=1;          // Increment for every initiator wait cycle
};

expect @special_cycle
   => {
       [..];
       @irdy_fall;                          // After IRDY# asserted,
        [initiator_wait_cnt] * @irdy_assr;  // master abort should wait
                                            // same number of cycles
     }@qclk;
```

- 
- CBV

**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.
- Sugar

  ```
  AG {read} |=> {{busy[3],!busy} || {busy[1..3]~read}}
  ```
- ForSpec

  ```
  r1 := always read -> next accept_on (read) (busy{3},!busy);

  assert r1;
  assume r1;
  ```
- E:
  ```
  expect p46 is @read => {[3]*@busy;fail @busy} or {[..3]*@busy; @read};
  ```
- 
- CBV

**Property 47**

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

  forall a: 0..7:
  forall b: 0..7:
  AG ((a!=b) -> (x(a) -> (!x(b) & ABG[1..4] !x(b))))::clk=clk

- ForSpec

  /* at most 1 bit of x may be high when clk is high */

  assert m:= change_if (clk) always mutex(x);

  /*whenever an x bit q is high, then for the next 4 cycles, only the same bit may be high*/

  correct_x := change_if (clk) always x!=0 -> always[1,4] (x!=0 -> x=past(x,1,x!=0));

- E

  expect p47a is true( x[0]+x[1]+x[2]+x[3]+x[4]+x[5]+x[6]+x[7]<= 1) @clk;
  expect p47b is true(x>0) => {[1..4]*(true(x==0) or fail change(x))} @clk;

**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.
- **Sugar**

```
var ccount:  1..6;
assign init(ccount) := 0;
            next(ccount) := case
                        ccount = 6: 6;
                        clk: ccount + 1;
                        else:  ccount;
            esac;

define saw6clocks := ccount = 6;
 restrict {rst[5],!rst[+]}::clk=clk

fairness saw6clocks;
```

  where saw6clocks is a signal that is asserted when we have seen at least 6
  clocks.

- **ForSpec**

  r3 := change_on(clk) rst{5} seq next always !rst;

The directive that will enforce this property on the model without creating proof obligation looks like:

restrict r3;

- **E**
- **CBV**

**Property 49 (global variables)**

- **English :**
  datain - control signal, indicates that there is valid data on the data_input_bus.
  data_input_bus - 32 bit data input bus.
  dataout - control signal, indicates that there is valid data on the data_output_bus.
  data_output_bus - 32 bit data output bus.

  Write a specification of a fifo, where when datain rises data is entered into the fifo (thru the data_input_bus port) and when dataout rises data exits the fifo. The specification needs to check that for each data that enters the fifo, that data comes out at the anticipated dataout rising edge. That is, data exits in order of arrival.

- **Sugar**

```
var datacount: 0..256;
assign init(datacount) := 0;
        next(datacount) := case
             rise(datain) : datacount + 1;
             rise(dataout): datacount - 1;
             else         : datacount;
           esac;

forall D(0..31): boolean:
forall N: 1..256:
AG ((rise(datain) & data_input_bus(0..31)=D(0..31) & datacount=N-1) ->
            next_event(rise(dataout))[N](data_output_bus(0..31)=D(0..31)))
```

- **ForSpec**

```
bit[log_depth] observerCounter;
bit observerActive;
bit[32] register;


assume always (observerActive -> observerActive')  & (b_rise(observerActive) ->
datain);

observer_write := datain & !observerActive & !dataout;
observer_read := (dataout &  observerActive |  dataout & !observerActive & !datain) &
observerCounter > 0;

init observerCounter = 0;
assign observerCounter' = case {
                                observer_write :  inc(observerCounter);
                          observer_read :    dec(observerCounter);
                                default :              observerCounter;
                             };
```

```
assign_on(b_rise(observerActive)) register'=data_input_bus;

correctData :=  always ( observerActive & dataout & observerCounter = 1 ) ->
                                        (data_output_bus  = register );

assert correctData;
```

- **E**

```
struct checker {

  count: uint(bits:8);
  data: uint(bits:32);

  expect [count-1]*@sys.dout => ((true('data_output_bus' == data) @sys.dout
                          or fail(cycle @sys.dout)) exec{quit()});
};


extend sys {

  event din is rise('datain');
  event dout is rise('dataout');

  !datacount: uint(bits:8) = 0;
  !checkers: list of checker;

  on dout {
    // assert datacount > 0;
    datacount -= 1;
  };

  on din {
    // assert datacount < 255;
    datacount += 1;
    checkers.add( new checker
                with {count=datacount; data='data_input_bus'});
  };
};
```

- **CBV**

In CBV this is written using a global variable to count the number of data's that were in the fifo when a new data entered the fifo. With this counter each data item will wait the respective number of dataout's.

Assume that the fifo cannot have more than 256 data items at any time.

```
/* var declaration area. */
var prevdatain = datain;
var prevdataout = dataout;
var datacount[7:0] =
begin
```

```
   if(!prevdatain && datain)
   return(datacount + 1);
   else if (!prevdataout && dataout)
  return(datacount - 1);
end

/* Task declaration area. */
task check_data(const count[0:7], data[0:31])
begin
  if (count == 0)
      data_output_bus ==  data;
|else
   if +(0 to *) : (!dataout)
   if +(1) : (dataout)
check_data(count - 1, data);
end
endtask

/* Specfication area. */
if (!datain)
 if +(1) : (datain)
begin
     savedata[0:31] = data_input_bus;
     checkdata(datacount, savedata);
end
```

**Property 50 (suspend)**

- **English:** Same property as above with the addition of the 'wait' signal. When the 'wait' signal is asserted, all specification threads are suspended, then when the 'wait' signal is deasserted the specifications continue from the point they left off (before suspension).
- **Sugar**

```
var datacount: 0..256;
  assign init(datacount) := 0;
  next(datacount) := case
             wait         : datacount;
              rise(datain) : datacount + 1;
              rise(dataout): datacount - 1;
              else         : datacount;
            esac;
```

```
forall D(0..31): boolean:
forall N: 1..256:
AG ((!wait & rise(datain) & data_input_bus(0..31)=D(0..31) & datacount=N-1)
->
            next_event(!wait &
rise(dataout))[N](data_output_bus(0..31)=D(0..31)))
```

- **ForSpec**

```
bit[log_depth] observerCounter;
bit observerActive;
bit[32] register;

assume always (observerActive -> observerActive')  & (b_rise(observerActive) ->
datain & !wait); // I still think it should be b_rise.

observer_write := datain & !observerActive & !dataout;
observer_read := (dataout &  observerActive |  dataout & !observerActive & !datain) &
observerCounter > 0;

init observerCounter = 0;
assign_on(!wait) observerCounter' = case {
                                    observer_write :  inc(observerCounter);
                                observer_read :    dec(observerCounter);
                                    default :              observerCounter;
                                };
assign_on(b_rise(observerActive)) register'=data_input_bus;
correctData :=  change_on(!wait) always ( observerActive & dataout &
observerCounter = 1 ) ->
                                    (data_output_bus   = register );

assert correctData;
```

- **E**

```
struct checker {

  count: uint(bits:8);
  data: uint(bits:32);

  expect [count-1]*@sys.dout => ((true('data_output_bus' == data) @sys.dout
                            or fail(cycle @sys.dout)) exec{quit()});
};


extend sys {

  event din is rise('datain') and fail true('wait');
  event dout is rise('dataout') and fail true('wait');

  !datacount: uint(bits:8) = 0;
  !checkers: list of checker;

  on dout {
    // assert datacount > 0;
```

```
    datacount -= 1;
  };

  on din {
    // assert datacount < 255;
    datacount += 1;
    checkers.add( new checker
                with {count=datacount; data='data_input_bus'});
  };
};
```

- **CBV**

```
 /* var declaration area. */
var prevdatain = datain;
var prevdataout = dataout;
var datacount[7:0] =
begin
  if (!wait)
   if(!prevdatain && datain)
  return(datacount + 1);
  else if (!prevdataout && dataout)
  return(dataout - 1);|
end

/* Task declaration area. */
task check_data(count[0:7], data[0:31])
begin
  if (count == 0)
   data_output_bus ==  data;
  else
  if +(0 to *) : (!dataout)
  if +(1) : (dataout)
  check_data(count - 1, data);
 end
endtask

/* Specfication area. */
when wait
  suspend
    if (!datain)
      if +(1) : (datain)
        begin
            savedata[0:31] = data_input_bus;
            checkdata(datacount, savedata);
       end
```

**Property 51 (zero time computation)**

- **English**

  **Missing !**

- **Sugar**

```
explanation:  we don't have functions in edl, but we do have
modules which can be instantiated.  a module can contain a
process, which is sequential code. The sequential code can contain
loops, but only if they are synthesizable. modules cannot be
called recursively.  therefore, as written, we can support
```

function inCache, but not function calcHec, unless it is rewritten in a synthesizable style.

declaration of module inCache:

```
module inCache(Tag(0..15),Cache(0.. CACHESIZE))(flag){
      process {
        var flag: boolean;

        flag := 0;
        %for i in 0.. CACHESIZE do
          if(Tag(0..15)=Cache(i)) then
            flag := 1;
          endif;
        %end
      }
}
```

instantiation of module inCache (the inputs appear in the first set of parentheses, the outputs (in this case the single output myflag) appear in the
second set):

```
instance inCache(mytag(0..15),mycache(0..9))(myflag);
```

- **ForSpec**
- **E**

```
inCache(Tag:uint(bits:16)) : bool is {

  result = FALSE;
  for i from 0 to CACHESIZE do {
    if Tag==Cache[i] then {
      result = TRUE;
    };
  };
};

calcHec(Header:uint(bits:32), index:uint(bits:3), Hec:uint(bits:8),coset:bit) :uint(bits:8) is
{

   if (index=<3 && !coset) then {
     result = calcHec(Header, index+1, hec(Header[8*index:8*(index+1)-1], Hec),0);
   } else if (index=<2 && coset) then {
     result = calcHec(Header, index+1,hec(Header[8*index:8*(index+1)-1], Hec),1);
   } else if (index==3 && coset) then {
     result = calcHec(Header, index+1, hec_coset(Header[8*index:8*(index+1)-1],
   Hec),1);
   } else {
     result = Hec;
   };
};

hec( byte:uint(bits:8), prev_Hec[7:0]) : uint(bits:8) is {
```

```
  result = %{
byte[7] ^ byte[6] ^ byte[5] ^ prev_Hec[5] ^ prev_Hec[6] ^ prev_Hec[7],
byte[6] ^ byte[5] ^ byte[4] ^ prev_Hec[4] ^ prev_Hec[5] ^ prev_Hec[6],
byte[5] ^ byte[4] ^ byte[3] ^ prev_Hec[3] ^ prev_Hec[4] ^ prev_Hec[5],
byte[4] ^ byte[3] ^ byte[2] ^ prev_Hec[2] ^ prev_Hec[3] ^ prev_Hec[4],
byte[7] ^ byte[3] ^ byte[2] ^ byte[1] ^ prev_Hec[1] ^ prev_Hec[2] ^ prev_Hec[3]
^ prev_Hec[7],
byte[6] ^ byte[2] ^ byte[1] ^ byte[0] ^ prev_Hec[0] ^ prev_Hec[1] ^ prev_Hec[2]
^ prev_Hec[6],
byte[6] ^ byte[1] ^ byte[0] ^ prev_Hec[0] ^ prev_Hec[1] ^ prev_Hec[6],
byte[7] ^ byte[6] ^ byte[0] ^ prev_Hec[0] ^ prev_Hec[6] ^ prev_Hec[7]
  };
};
hec_coset( byte:uint(bits:8), prev_Hec:uint(bits:8)) : uint(bits:8) is {

  result = %{
byte[7] ^ byte[6] ^ byte[5] ^ prev_Hec[5] ^ prev_Hec[6] ^ prev_Hec[7],
byte[6] ^ byte[5] ^ byte[4] ^ prev_Hec[4] ^ prev_Hec[5] ^ prev_Hec[6]^1'b1,
byte[5] ^ byte[4] ^ byte[3] ^ prev_Hec[3] ^ prev_Hec[4] ^ prev_Hec[5],
byte[4] ^ byte[3] ^ byte[2] ^ prev_Hec[2] ^ prev_Hec[3] ^ prev_Hec[4]^1'b1,
byte[7] ^ byte[3] ^ byte[2] ^ byte[1] ^ prev_Hec[1] ^ prev_Hec[2] ^ prev_Hec[3]
^ prev_Hec[7],
byte[6] ^ byte[2] ^ byte[1] ^ byte[0] ^ prev_Hec[0] ^ prev_Hec[1] ^ prev_Hec[2]
^ prev_Hec[6]^1'b1,
byte[6] ^ byte[1] ^ byte[0] ^ prev_Hec[0] ^ prev_Hec[1] ^ prev_Hec[6],
byte[7] ^ byte[6] ^ byte[0] ^ prev_Hec[0] ^ prev_Hec[6] ^ prev_Hec[7]^1'b1
  };
};
```

- **CBV**

A function that returns true if a given tag is in the cache tag memory.

```
function inCache(Tag[0:15):bool
begin
   flag:bool;
   flag = 0;
   for(i=0; i<CACHESIZE; i++)
   if (Tag == Cache[i])
   flag = 1;
   return(flag);
end
```

A function that claculates CRC (taken from some verification project).
Used when comparing the expected result with the designs result.

```
function calcHec(Header[0:31], index[2:0], Hec[7:0],coset[0:0]) [7:0]
begin
   if (index=<3 && !coset)
   return( calcHec(Header, index+1, hec(Header[8*index:8*(index+1)-1], Hec),0) );
        else if (index=<2 && coset) return (calcHec(Header, index+1,
        hec(Header[8*index:8*(index+1)-1], Hec),1));
  else if (index==3 && coset)
  return(calcHec(Header, index+1, hec_coset(Header[8*index:8*(index+1)-1],
  Hec),1)) ;
     else  return(Hec);
end
endfunction
```

```
function hec(byte[7:0], prev_Hec[7:0])[0:7]
begin
return {
byte[7] ^ byte[6] ^ byte[5] ^ prev_Hec[5] ^ prev_Hec[6] ^ prev_Hec[7],
byte[6] ^ byte[5] ^ byte[4] ^ prev_Hec[4] ^ prev_Hec[5] ^ prev_Hec[6],
byte[5] ^ byte[4] ^ byte[3] ^ prev_Hec[3] ^ prev_Hec[4] ^ prev_Hec[5],
byte[4] ^ byte[3] ^ byte[2] ^ prev_Hec[2] ^ prev_Hec[3] ^ prev_Hec[4],
byte[7] ^ byte[3] ^ byte[2] ^ byte[1] ^ prev_Hec[1] ^ prev_Hec[2] ^ prev_Hec[3]
^ prev_Hec[7],
byte[6] ^ byte[2] ^ byte[1] ^ byte[0] ^ prev_Hec[0] ^ prev_Hec[1] ^ prev_Hec[2]
^ prev_Hec[6],
byte[6] ^ byte[1] ^ byte[0] ^ prev_Hec[0] ^ prev_Hec[1] ^ prev_Hec[6],
byte[7] ^ byte[6] ^ byte[0] ^ prev_Hec[0] ^ prev_Hec[6] ^ prev_Hec[7]
};
end
endfunction

function hec_coset(byte[7:0], prev_Hec[7:0])[7:0]
begin
return {
byte[7] ^ byte[6] ^ byte[5] ^ prev_Hec[5] ^ prev_Hec[6] ^ prev_Hec[7],
byte[6] ^ byte[5] ^ byte[4] ^ prev_Hec[4] ^ prev_Hec[5] ^ prev_Hec[6]^1'b1,
byte[5] ^ byte[4] ^ byte[3] ^ prev_Hec[3] ^ prev_Hec[4] ^ prev_Hec[5],
byte[4] ^ byte[3] ^ byte[2] ^ prev_Hec[2] ^ prev_Hec[3] ^ prev_Hec[4]^1'b1,
byte[7] ^ byte[3] ^ byte[2] ^ byte[1] ^ prev_Hec[1] ^ prev_Hec[2] ^ prev_Hec[3]
^ prev_Hec[7],
byte[6] ^ byte[2] ^ byte[1] ^ byte[0] ^ prev_Hec[0] ^ prev_Hec[1] ^ prev_Hec[2]
^ prev_Hec[6]^1'b1,
byte[6] ^ byte[1] ^ byte[0] ^ prev_Hec[0] ^ prev_Hec[1] ^ prev_Hec[6],
byte[7] ^ byte[6] ^ byte[0] ^ prev_Hec[0] ^ prev_Hec[6] ^ prev_Hec[7]^1'b1
};
end
endfunction
```

**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.
- **Sugar**

  ```
  assume AG (power_down -> AX power_down)
  assume AG ((new_cycle & !power_down) -> AG (!power_down))
  ```

- **ForSpec**

  ```
  either_power_on_or_off:=  always (
   (power_down -> power_down') & (new_cycle & !power_down -> always(
  !power_down)));

  assume either_power_on_or_off;
  ```

- **E**

  ```
  assume p53a is @power_down => @power_down;
  ```

```
assume p53b is @new_cycle and fail(@power_down) => fail
{[..];@power_down};
```

- **CBV**

**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.

- **Sugar**

```
#define instr_buffer_size 16

#define msb_instr_buffer instr_buffer_size-1

assume AG ((inst_valid_input(msb_instr_buffer ..0) |
          !(first_opcode_input(msb_instr_buffer ..0)|
           last_byte_input(msb_instr_buffer
..0)))=ones(instr_buffer_size))
```

- **ForSpec**

```
instr_buffer_size:=16;
msb_instr_buffer := instr_buffer_size-1;

correct_markers := always (
    (
     instr_val_input   |
    (~(first_opcode_input[(msb_instr_buffer-1),0] |
last_byte_input[(msb_instr_buffer-1),0]))
     )
          = repeat(1, msb_instr_buffer));
/* In the above  bit-wise OR among the bits of the vectors are performed
*/

assume correct_markers;
```

- **E**

```
define instr_buffer_size 16
var msb_instr_buffer := instr_buffer_size - 1;
var ones :uint(bits:instr_buffer_size) = ~(0);

expect p54 is true(( instr_val_input   |
          (~(first_opcode_input[(msb_instr_buffer-1):0] |
            last_byte_input[(msb_instr_buffer-1):0]))) == ones);
```

- **CBV**

**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 elements in the queue. This behavior is interrupted once the signals reset or raprep_cycle83l are set.
- **Sugar**
  we don't have parameterized assertions in sugar. what i usually do if I want something like this is to define the assertion in a separate file, then include the file. the "parameter" effect is achieved by a macro which is redefined before each inclusion of the file. so:... file property_55 would look like this:

```
define byte_is_in := valid_new_cycle &
     lin_addr_input(n+(instr_buffer_size-1)  ..
instr_buffer_size)=line_addr_copy(n-1 .. 0);
define byte_is_taken := new_load &
     lin_addr_output(n+(instr_buffer_size-1) ..
instr_buffer_size)=line_addr_copy(n-1 .. 0));

forall lin_addr_copy(n-1 ..0): boolean:
AG (byte_is_in -> AF (reset | raprep_cycle83l | byte_is_taken))

... end of file property_55
```

  then, to use the property, define n and instr_buffer_size and include the file like this:

```
#define n 5
#define instr_buffer_size 32
#include "property_55"
```

- **ForSpec**

```
assert_byte_out (n) :=
{
rigid bit [n] lin_addr_copy;   /*  n >= 1  */
byte_is_in :=
          valid_new_cycle &
            (lin_addr_input[n+(instr_buffer_size-1), instr_buffer_size] =
                                                          lin_addr_copy);
                                  /* the rigid vector lin_addr_copy remembers
                                     the address of
                                      the byte which was put in the queue */
     byte_is_taken :=
         new_load &
          (lin_addr_output[n+(instr_buffer_size-1), instr_buffer_size] =
                                                           lin_addr_copy);


     byte_eventually_out := always (byte_is_in -> (accept_on (reset
                              |raprep_cycle83l) eventually byte_is_taken));
};

in_out_2 := new assert_byte_out (2);
```

```
assert in_out_2/byte_eventually_out;
```

- **E**

```
struct checker {

  n: uint;                  // 5
  instr_buffer_size: uint; // 32

  !cell_checkers: list of cell_checker;

  init() is {
    for i from 0 to instr_buffer_size - 1 do {
      cell_checkers.add( new cell_checker
                        with {.n=n;.instr_buffer_size=instr_buffer_size;
                             .line_add_copy = i});
    };
  };
};

struct cell_checker like checker{

  line_add_copy; uint;

  event byte_is_in is
    @sys.valid_new_cycle and
    true('lin_addr_input'[n+(instr_buffer_size-1):instr_buffer_size]==
         line_addr_copy[n-1:0]);
  event byte_is_taken is
    @sys.new_load and
    true(lin_addr_output[n+(instr_buffer_size-1):instr_buffer_size]==
         line_addr_copy[n-1:0]);

  expect (@byte_is_in and (@reset | @raprep_cycle83l |
@byte_is_taken))
     or  (@byte_is_in => {[..];(@reset or @raprep_cycle83l or
@byte_is_taken)});
};

To instantiate the checker:

new checker with {.n=5; .instr_buffer_size=32};
```

- **CBV**

**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.
- **Sugar**
  .. start file property_56:

```
var addr(n-1 ..0): boolean;
var align(m-1 ..0): boolean;

assign next(addr(n-1 ..0)) :=
            case
              prev(reset)              : zeroes(n);
               valid_new_cycle & btclear13h:
                  br_target(n+(instr_buffer_size-1) ..
instr_buffer_size);
               valid_new_cycle          : addr(n-1 ..0)+1;
              else                      : addr(n-1 ..0);
            esac;
assign next(align(m-1 ..0)) :=
            case
              prev(reset)              : nondets(m);
               valid_new_cycle & btclear13h: br_target(m-1 ..0);
              valid_new_cycle          : zeroes(m);
              else                     : align(m-1 ..0);
            esac;

... end of file property_56

to use property 56, do:

#define n 2
#define m 1
#include "property_56"

restrict {(addr(n-1 ..0) = lin_addr_input(5..4))[*]}
restrict {(align(m-1 ..0) =  lin_addr_input(0))[*]}
```

- **ForSpec**

```
//Parameterized block definition, used in address generation
generate_clip (n,m) :=

 {  /* 1 <= n <= 28 , 1 <= m <= 4 */
bit[n]  addr;
bit[m] align, new_align;

assign_on (!clk) addr' = case {
        past(reset) : zx(0b0,n);
        valid_new_cycle & btclear13h :
                br_target[n+(instr_buffer_size-1),instr_buffer_size]';  //note
prime on rhs
        valid_new_cycle : addr + 1;
        default : addr;};

assign_on (!clk) align' = case {
       past(reset) : new_align;
        valid_new_cycle & btclear13h : br_target[m-1,0]';
       valid_new_cycle : zx(0b0,m);
      default : align;};
};
////////////////////////////////////////////////////////////
```

```
//Instantiation

my_gen_clip := new generate_clip (2, 1);

//Restrictions

restrict always (my_gen_clip/addr = lin_addr_input[5,4]);

restrict always (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)
- **Sugar:**
  AG {req[+], ack} |=> {[*],grant,!abort}!
- **ForSpec:**
  always req+,ack triggers next eventually grant,!abort;
- E
  expect p57 is {[1..]*@req;@ack} => {[..];{@grant;fail @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 end is asserted (if end is never asserted, then busy must stay asserted forever).
- **Sugar:**
  AG {req, true, !retry} |=> {busy[*], end}
- **ForSpec:**
  always req,true,!retry triggers next (busy wuntil end);
- **E**
  expect p58 is {@req;cycle;fail @retry} =>
  ({[..]*@busy; @end} or fail {[..];fail @busy});

**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

- **Sugar:**
  AG {req, true, !retry} (AF ack)
- **ForSpec:**
  always (req,true,!retry) triggers eventually ack;
- E
  expect p59 is {@req;cycle;fail @retry} => {[..];@ack}

**Property 60**

4. 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).

- Sugar: AG (hi_pri_req -> next_event_f(gnt)[1..2](dst=hi_pri))
- ForSpec: always hi_pri_req-> ((!gnt*,gnt){1,2} \ dst=hi_pri);
- E

```
event hi_pre_req_assr is rise('hi_pri_req');
event gnt_assr is rise('gnt');

expect p60 is
    @hi_pri_req_assr and (true(dst=hi_pri) @gnt_assr) or
    @hi_pri_reg_assr => {~[..1]*@gnt_assr; (true(dst=hi_pri) @gnt_assr or
                                fail (cycle @gnt_assr))};
```

**Property 61**

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

  - Sugar: AG within!(tr_start, tr_end)
    {[*], data_start, [*], data_end}
- ForSpec: always tr_start -> (!tr_end+ \ data_start) , (!tr_end+ \ data_end) , tr_end;
- E

```
expect p61a is @tr_start and fail(@data_start) =>
  {[..];@tr_end} and {{[..];data_start};{[..];@data_end};~[..]};
expect p61b is @tr_start and @data_start =>
  {[..];@tr_end} and {[..];@data_end;~[..]};
```

**Property 62**

6. 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.

- Sugar: AG ({go, {get[=8]} && {kill[=0]}} |=> {{put[=8]} && {end[=0]}})
- ForSpec: always Go, ((!get & !kill)*,(get & kill)){8} triggers reject_on(end)(!put*,put){8}
- E

```
expect p62 is {@go;([8]*true(kill==0) @get)} => ([8]*true(end==0) @put);
```

**Property 63**

7. Lack of deadlock:
AG EF (state = idle)

**Property 64**

8. Lack of dependency: In one version of PCI, TRDY must not be dependent on IRDY
AG (frame_fall -> E[IRDY U !TRDY])

- 

**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.
- Sugar:
  AG ( (fell(reset) -> rose(clk)) &
  (rose(reset) >next_event![1..6](rose(clk))(reset)))
- ForSpec:
  reset_stable := a_fall(reset) -> a_rise(clk);
  reset_cycle := a_rise(reset) TRIGGERS
  CHANGE_ON(a_rise(clk)) ALWAYS[0,5] reset;
  reset_spec := ALWAYS reset_stable & reset_cycle;
- E

expect p65a is (fail fall(reset)) or rise(clk);
expect p65b is rise(reset) => ([6]*true(reset) @rise(clk))

**Property 66**

- English: When clk is high, if port1 has a vl15 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.
- Sugar:
  %for port in 0..15 do
  %for vl in 0..15 do
  assume { AG (p.%{port}._vl.%{vl}._req_valid ->
  AX (p.%{port}._vl.%{vl}._req_valid &
  (p.%{port}._vl.%{vl}._req_port
  =prev(p.%{port}._vl.%{vl}._req_port))

```
Until
 p.%{port}._vl.0_arb_gnt))  :: clk = clk  }
%end
 %end
 AG (!reset -> AG (p1_vl15_req_port = 2 & p1_vl15_req_valid  ->
                          AF ( p2_vl15_gnt_issued  &
                        p2_vl15_gnt_port = 1 ) :: clk = clk
```

- 
- ForSpec:
- req_until_gnt (port, vl) := change_if(clk) always “p”.port.”_vl”.vl.”_req_valid” ->
  next (“p”.port.”_vl”.vl.”_req_valid” & “p”.port.”_vl”.vl.”_req _port = past (“p”.port.”_vl”.vl.”_req _port)) wuntil “p”.port.”_vl”.0_arb_gnt;

```
for port := 0 to 15
{
   for vl := 0 to 15
   {
      assume req_until_gnt (port, vl);
   };
};

assert gnt_after_req := always change_if(clk) !reset-> always
                                  p1_vl15_req_port=2 & p1_vl15_req_valid ->
eventually
                                   p2_vl15_gnt_issued & p2_vl15_gnt_port=1;
```

**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.

- Sugar:

```
 %for  n  in 0..15 do
    assume   { AG (p.%{n}._vl10_arb_req(5) &  p.%{n}._vl15_arb_req(5) ->
p.%{n}._vl10_arb_req(4..0)  != p.%{n}._vl10_arb_req(4..0)   ) } %end
```

- ForSpec:

```
input_port (n) :=
{
      vl0_req :=    "/p".n."_vl0_arb_req[5:0]";
      vl15_req :=   "/p".n."_vl15_arb_req[5:0]";

      vl0_req_port  := vl0_req[4:0];
      vl0_req_valid := vl0_req[5];
      vl15_req_port := vl15_req[4:0];
      vl15_req_valid:= vl15_req[5];

      assume always (vl0_req_valid & vl15_req_valid)->(vl0_req_port !=
      vl15_req_port);
};
```

```
        for i := 0 to 15  { new input_port(i); }
```

- E

The assumption:

```
struct checkPair {

  !port: uint [0..15];
  !vl: uint [0..15];

  event req_valid is true('p(port)_vl(vl)_req_valid') @clk;
  event arb_gnt is true('p(port)_vl.0_arb_gnt') @clk;
  event same_port is fail change('p(port)_vl(vl)_req_port') @clk;

  assume req_until_gnt is
      @req_valid => {[..]*(@req_valid and @same_port);@arb_gnt} or
                fail {[..];fail (@req_valid and @same_port)} @clk;
};
```

To instantiate the checker for port 2 and vl 15:

```
  new checkPair with {.port = 2; .vl=15};
```

The property:

```
  event clk is true('clk') @sim;
  event req_port2 is true('p1_vl15_req_port==2') @clk;
  event gnt_port1 is true('p1_vl15_gnt_port==1') @clk;
  event req_valid is true('p1_vl15_req_valid') @clk;
  event gnt_issued is true('p2_vl15_gnt_issued') @clk;

  expect gnt_after_req is
     @req_port2 and @req_valid and fail(@gnt_issued and @gnt_port1) =>
      {[..];@gnt_issued and @gnt_port1} @clk;
```

**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.
- Sugar
  restrict {rst[5..],!rst[+]}::clk=clk

```
var ccount:  1..6;
assign init(ccount) := 0;
next(ccount) := case
     ccount = 6: 6;
     clk: ccount + 1;
     else:  ccount;
esac;

                    define saw6clocks := ccount = 6;
                    fairness saw6clocks;
```

- ForSpec

  f := change_on(clk) rst{5},rst* seq next always !rst;

- Restrict f;  /* assumption on the environment  */

- E

```
assume p68a is fail( @startRun) or true('rst') @clk;
assume p68b is @startRun => {[4..]*true('rst');fail{[..];true('rst')}} @clk;
```

- CBV

**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.
- Sugar

  AG { start_pack,
  { true[8] } && { p_start_reg[>0] } && { discard_rx_ [>0] } } (AX !valid)

- ForSpec

  always start_pack & (eventually[1,8] p_start_reg) & (eventually[1,8] discard_rx_) -> next[9] !valid;

- E
- CBV

**Property 70**

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

- Sugar

  AG { !reset,
  { trans_start, data[=2], trans_end} && {error [>0] }} |-> { [.. 4], retry}

- ForSpec

  trans := trans_start, (!data*,data){2}, !data*, trans_end;

  p := (next !reject_on(error) trans) -> ( (!reset,trans) triggers eventually[0,4] retry );

- E
- CBV

**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.
- Sugar
- ForSpec
- E
- CBV

**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.

- Sugar
- ForSpec
- E
- CBV

In this example you can see how a CBV thread follows the existence of a specific address within the cache.

```
type  AddressType = stream[15:0];
type  TagType = stream[7:0];
type  DataType = stream[31:0];
type  CacheEntryType = record
tag:  TagType;
data: DataType;
end
type CacheType =  array[0..'CACHESIZE-1] of CacheEntryType;

cbvmodule cache(hit, miss, write, read:bool, address:AddressType,
cache:CacheEntryType)
`define CACHESIZE 16'h7fff

function getTag(cacheAddress:AddressType):TagType
begin
  return(cacheAddress[7:0]);
end
endfunction

function inCache(cacheAddress:AddressType):bool
begin
  e:integer;
  flag:bool;
  flag = 0;

  for(e=0; e<= 'CACHESIZE; e++)
    if (getTag(cacheAddress) == cache[e].tag)
      flag = 1;
  return(flag);
end
endfunction

task rise(signal:bool)
begin
  if +(0 to *):(!signal)
    if +(1):(signal)
      return;
end
endtask

task followAddress(saveAddress:stream[0:31],lifeLeft:int)
```

```
begin
  if rise(read)
    begin
      if (address == saveAddress) /* Address used again. */
        begin
          hit;
          /* Address renewed all its lives. */
          followAddress(saveAddress, `CACHESIZE);
        end
      else
        if (lifeLeft == 0)
          +(1):(!inCache(saveAddress);
        else
          /* Another address was read therefore this
             address lost one of its lives. */
          followAddress(saveAddress, lifeLeft - 1);
    end
end
endtask
```

**Property 73 (strong liveness)**

- **English:** The finite state machine LockDet cannot stay in state UNSAT forever if the oldest uop keeps replaying.
- Sugar
- ForSpec

  !eventually ( (always unsat) & always eventually replay )

- E
- CBV
  -

**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.

- **ForSpec:**

```
// Mapping
ads := /bus/ADS_bus;
rise_clk := b_rise(/bus/clk);
```

```
past_bnr := past(/bus/BNR_bus, 1, rise_clk);

// Request Stall State FSM
bit sample;
request_stall_state := FSM
{
        init: free;
        clock: past(sample,1,rise_clk);
        transition:
                free: !past_bnr ? free;
                        past_bnr ? stalled;
                stalled: !past_bnr ? throttled;
                        past_bnr ? stalled;
                throttled: ! past_bnr ? free;
                        past_bnr ? stalled;
}
assume i1 := new request_stall_state;

// BNR sample points
assume always sample = (i1/state = i1/free) ? past(ads,3,rise_clk) : past(sample,2,rise_clk));

// Properties
p1 := change_if (rise_clk) always ads -> always[1,3] !ads;
p2 := change_if (rise_clk) always (i1/state = i1/throttled) & ads
-> NEXT !ads WUNTIL i1/state != i1/throttled);
p3 := change_if (rise_clk) always (i1/state = i1/stalled)
-> !ads;

//To use the properties as assertions /assumptions do:
assume / assert p1 & p2 & p3;
```