WAIT! Describing implicit state machines with VHDL

ALJ003 (V1.0) May 27, 2009

ABSTRACT • The VHDL PROCESS, written with multiple WAIT statements, provides a powerful way to develop implicit state machines (ISM). The ISM lacks an explicit state variable to track the current state thus making it easy and quick to add, delete or modify states with simple text edit commands. Synthesizable and behavioral state machine styles are covered with examples and suggestions for efficient application of this technique.

KEYWORDS • implicit state machine, FSM, VHDL, WAIT statement, RTL, behavioral modeling, test bench coding
The most common form of the VHDL process statement, with the sensitivity list written after the PROCESS keyword, is a short hand notation for the more powerful WAIT statement. Mastering the WAIT statement leads to satisfying descriptions through the construction of implicit state machines (ISM). Both behavioral and synthesizable expressions of this versatile style are presented.

Implicit finite state machines

I suspect, for the typical engineer, the design of finite-state machine controllers begins with some type of bubble, stick or flow diagram that graphically depicts each state. Either manually or using automation the diagram gets rendered down to a VHDL representation consisting of a state transition lookup table or a CASE statement next state decoder [ASHENDEN].

Both CASE and lookup methods use an explicit state variable to remember the current state of the controller. With these styles the emphasis is on the gate and flip flop description and less with the overall algorithmic process implemented by the state flow. Consequently, whatever the natural expression of algorithmic loops depicted graphically is lost.

The ISM style described in this article attempts to show an overlooked avenue for rich state machine descriptions that favor the algorithmic expression over the logic.

As the name implies the ISM description lacks an explicit state variable. How is that done? The state of the machine follows from the location in the description of each VHDL WAIT statement. (Recall that a PROCESS may have multiple WAIT statements in lieu of the (shortcut) sensitivity list.) Rather then jumping explicitly from state-to-state by directly modifying the state variable the ISM moves from state to state by:

1. The action of control constructs (IF, CASE, LOOP, EXIT, NEXT) that direct which sequential statement to evaluate next and,
2. The simple fact that sequential statements evaluate implicitly in sequence from top to bottom.

Example 1 illustrates a simple example of how the ISM tracks the state.

Example 1 REQ:ACK FSM as implicit state machine

| LIBRARY ieee; |
| USE ieee.std_logic_1164.ALL; |
| USE ieee.numeric_std.ALL; |
| ENTITY example_01 IS |
| PORT (send_data : in std_logic); |
| END ENTITY example_01; |
| ARCHITECTURE beh OF example_01 IS |
| SIGNAL req, ack, send_data : std_logic; |
| SIGNAL put_data, get_data : unsigned(1 TO 8); |
| BEGIN |
| reqp : PROCESS IS |
| BEGIN |
| t1: WAIT ON send_data; |
| reqp <= '1' after 5 ns; |
| put_data <= put_data+1 after 6 ns; |
| request: LOOP |
| EXIT request WHEN ack='1'; |
| t2: WAIT ON ack; |
| END LOOP; |
| put_data <= (OTHERS => '0') after 6 ns; |
| reqp <= '0' after 5 ns; |
| t3: WAIT UNTIL ack='0'; |
| END PROCESS reqp; |
| ackp: PROCESS IS |
| BEGIN |
| ackp <= '0' after 10 ns; |
| t1: WAIT UNTIL req='1'; |
| get_data <= put_data after 3 ns; |
| ackp <= '1' after 10 ns; |
| t2: WAIT UNTIL req='0'; |
| END PROCESS ackp; |
| END ARCHITECTURE beh; |

Each set of statements in between any two WAIT statements describes the logic forming the output decoder and the state transition logic. Note that the “states” (WAITs) are labeled st1, st2, st3 but those are purely descriptive and do not take part in the definition of the machine. Also note that no clock appears in the example – it is purely an asynchronous state machine (and so it isn’t synthesizable).

Clearly state st2 follows st1 and st3 follows st2. There is no way to get from st3 to st2 directly. Because of the signal assignment delays there is no practical way to get from st1 to st2. How does it get from st1 back to st1? The process body is itself an infinite loop and brings the sequential evaluation back to the BEGIN.

The example gave a quick flavor of the ISM description style. The next two sections deal with synthesizable ISM and behavioral ISM styles respectively.

Synthesizable ISM for the stuffer

ALJO02 My Variable State of Mind introduced the stuffer component as an example. The stuffer acts to remove body elements from an incoming frame and stuffs the output frame body elements with decrementing count values. It does this by parsing the input stream and emitting a new output stream.

The stuffer state machine coding in ALJO02 used the traditional CASE statement style. In Example 2 the architecture implicit_sig delivers the identical functionality using the ISM style. The big difference between Example 1 and Example 2 is that to be synthesizable the ISM must use a consistent clock signal in the WAIT statement. (Note that the full example with PACKAGE, ENTITY and ARCHITECTURE appears at the end of the article.)

The code also explicitly shows the loops necessary to wait for the incoming header, simultaneously receive and send body elements until either the input frame is empty or the output frame is complete. Then a new loop finishes watching the input or producing the necessary output. The code tracks the natural expression of the problem needing a solution.

Example 2 The stuffer rendered as an implicit FSM

| ARCHITECTURE implicit_sig OF stuffer IS |
| SIGNAL stf_ct : unsigned(nibble); |
| SIGNAL saved_footer : nibble_typ; |
| BEGIN |
| implicit: PROCESS IS |
| BEGIN |
| IF (rst = kRESET_ASSERTED) THEN |
| fro <= (OTHERS => '0'); |
| saved_footer <= (OTHERS => '0'); |
| stf_ct <= (OTHERS => '0'); |
| END IF; |
| wait_hdr : LOOP |
| WAIT UNTIL rising_edge(fk); |
| EXIT wait_hdr WHEN (rst = kRESET_ASSERTED); |
| stf_ct <= to_unsigned(stfng, 4) - 1; |
| END IF; |
| EXIT WHEN (rst = kRESET_ASSERTED); |
| stf2_lp LOOP |
| WAIT UNTIL rising_edge(fk); |
| EXIT wait_hdr WHEN (rst = kRESET_ASSERTED); |
| IF (fri(kINDI) = kELEMENT) THEN |
| -- Incoming Hdr has been found so emit it as the header of the -- outgoing element stream. |
| stf2_lp LOOP |
| WAIT UNTIL rising_edge(fk); |
| EXIT wait_hdr WHEN (rst = kRESET_ASSERTED); |
| END LOOP stf2_lp; |
WAIT! Describing implicit state machines with VHDL

A number of key aspects of this style are presented below.

States

A WAIT statement forms each state. In the form used in the example the \(ck\) signal appears in the sensitivity list. Since it is the only signal in the sensitivity list a change on \(ck\) with a rising edge will resume evaluation. Therefore only the clock edge causes the state transition to occur. The preceding statements determine whichWAIT becomes the next to suspend thus (implicitly) determining the current state.

The coding for the ISM style aligns the sequential evaluation of the VHDL (the spatial aspect) with the algorithmic evaluation necessary to implement the application’s functionality (the temporal aspect). It is this alignment that makes the ISM style more powerful, easier to understand and easier to debug then the explicit style.

Loops

The ISM style expresses the loops directly. The explicit FSM coding style just makes it difficult to grasp those intuitively.

There are a few VHDL syntax items to point out. The labels (wait_hdr, ftr_scan, stf2_lp, fr_loop) applied to the loops give the EXIT statements a way to identify which loop to leave when the condition is met. VHDL syntax permits unconditional exits as well. The NEXT keyword (unused in this example) provides a way to skip to the next iteration (or top) of the loop. Both permit escapes through multiple enclosing loop structures. This capability is best demonstrated by the synchronous reset exit that appears after every WAIT:

\[
\text{EXIT wait_hdr WHEN (rst = kRESET.Asserted);}
\]

What makes it synchronous? The sensitivity list of the preceding WAIT includes \(ck\) but not \(ct\) and \(rst\). The latter is required for asynchronous behavior modeling. Unfortunately, at this writing, the only synthesis tool I found that would synthesize the above description supports only synchronous resets.

The EXIT statement permits arbitrary loop constructs. There is no explicit WHILE or UNTIL loop construct in VHDL. Just put the EXIT at the top of the loop for a WHILE loop (pre-test) and at the bottom for an UNTIL loop (post-test). Put it in the middle when you need some initialization code prior to the exit test that is used in each loop iteration.

The FOR loop, with a static range seems easily synthesizable yet it was not accepted by the synthesis tool I used. If a FOR loop contained a WAIT statement then it would have the effect of generating states. Perhaps that makes the synthesis process too complex. But as I explain later the ability to generate states on demand makes the for-loop indispensable in test benches.

Clock and reset expressions

The synthesis tool I used requires that all WAIT statements describing the state machine states be identical. IEEE standard 1076-6-2004, section 6.1.3.4, states this as the requirement: “c) Each wait statement shall specify the same clock edge of a single clock.”

The necessity to maintain consistent clock edge and reset loop exit statements just begs for a macro or procedure. Unfortunately a procedural approach fails for two reasons: 1) procedures containing a WAIT statement are not synthesizable (although 1076.6-2004 permits them) and 2) the label of the outer reset loop would not be visible from within the procedure. That leaves the macro preprocessor approach as the only viable option to save some typing effort.

Surprised?

Were you surprised by the synthesizability of the ISM style? If you want to use it then hammer on your synthesis tool vendor to support 1076.6-2004. No vendor will support it until there is wide spread adoption. Wide spread adoption won’t happen because there is an acceptable alternative (explicit FSM style). So, your voice is needed to drum up support.

Even if your synthesis vendor doesn’t support it you can still take advantage of the principles in your test benches.

The stuffer test bench

This article fell out of the need to test the stuffer presented in ALJ002 My Variable State of Mind. While I’m not always keen to write tests I always enjoy the freedom to use the entire language without the restrictions of synthesizability or other tool limitations. There is a lot you can do with the WAIT statement to write great test benches using ISMs.

WAIT statements

THINK GENERAL. It’s typical to consider the condition on the arc in a bubble diagram as being the event that causes the transition. (It would be if the state machine was purely asynchronous but I’ll bet you don’t typically write VHDL descriptions for those!) The arc condition selects which state is next but doesn’t cause the jump to occur. Only the event trigger for that state can actually cause the state change to occur.

To write a truly general ISM however you need to forget about clock edges and concentrate on events (as was done in Example 1). Each state may have a distinct event that permits entry. The RTL ISM only allowed one event -- the clock edge.

To discover and diagnose bugs using your test bench may require a state machine that controls the testing itself. Perhaps you will find it necessary to synchronize your stimulus and verification processes. So a general purpose signalling event is called for. Boolean signals support this nicely.

For example If your description declares \(flag\) as a boolean signal then

WAIT ON \(flag\);

in one PROCESS would be sensitive to either possible change in the signal value of \(flag\) being changed in another PROCESS. Use this statement:

\(flag <= \neg flag;\)
as a simple way to signal an arbitrary event. No need to worry about initial-
izing flag or making a convention of '1' means do something and '0'
means do nothing. Now your test bench controller ISM has a way to eas-
ily transition between states simply by inverting a boolean signal.

This is what I mean by thinking general about WAIT statements. 

**Syntax.** What can the WAIT statement do for you? There are three pri-
mary clauses in the WAIT statement and all are optional strangely
enough. Lets take a look at all the combinations to get a feel for what
tools an ISM will have to work with to change states.

### Table 1: WAIT statement possibilities

<table>
<thead>
<tr>
<th>WAIT</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suspend permanently.</td>
<td></td>
</tr>
<tr>
<td>a, b, c</td>
<td>Suspend then resume when a, b, or c changes.</td>
</tr>
<tr>
<td>shark=byte;</td>
<td>Suspend then resume when shark changes and matches the signal/constant/variable byte.</td>
</tr>
<tr>
<td>25 ns</td>
<td>Suspend then resume in 25 ns.</td>
</tr>
<tr>
<td>a, b until c=1;</td>
<td>Suspend then resume when a or b changes and c=1. 'c' is not included in the sensitivity list</td>
</tr>
<tr>
<td>a, b for 25 ns;</td>
<td>Suspend then resume when either a or b changes or 25 ns have passed. This is a common test bench idiom when expecting something to happen but you need a time-out to prevent the simulation from running forever when it doesn't get the expected result.</td>
</tr>
<tr>
<td>250 ns or qqq&gt;3 for 50 ms</td>
<td>Suspend then resume after 50 ms or when a change on zzz or qqq occurs and zzz is true and qqq is greater then 3. The time-out does not reset should the sensitivity list get an event but the UNTIL condition is still false.</td>
</tr>
<tr>
<td>apple until orange for 100 ps</td>
<td>Suspend then resume after either 100 ps has passed or an event on apple has occurred and at that moment orange is true.</td>
</tr>
</tbody>
</table>

All of the combinations above will prove useful in your test benches at
one time or another. The time-out variations are particularly useful to
All of the combinations above will prove useful in your test benches at
enough. Let's take a look at all the combinations to get a feel for what
tools an ISM will have to work with to change states.

### Test Procedures

#### Stickling a WAIT statement inside a procedure or function body makes a
powerful, reusable, state machine fragment. This is fundamental to
writing good tests efficiently. But where should you declare the procedure?
That depends on whether it needs to be reused by multiple tests or by multiple people.

The following code demonstrates a state machine fragment to check the
expected value coming from the stuffer (the DUT). Several notewor-
thy approaches appear in this tiny example.

#### Example 4 Procedure to check DUT expected output over time

```vhdl
verify: process
    begin
        wait until fro_tb(kINDI) = kELEMENT for 1000 ns;
        assert (fro_tb(kINDI) = kELEMENT)
            report "Test &test& - no frame appeared"
            severity failure;
        end
    begin
        wait until rising_edge(clock_tb);
        assert (fro_tb = frame(i))
            report "test &test& - frame mismatch"
            severity failure;
    end
    loop
        for i in frame_range loop
            wait for 25 ns;
            assert (frame(i) /= stuffer)
                report "&test& - stuffer failed"
                severity failure;
        end loop;
    end process
end PROCEDURE
```

### Procedure Scope. 
First the PROCEDURE declaration appears inside the
PROCESS. Why put it here? Simply because all of the signal declarations
outside the process itself are visible within the PROCEDURE. That elimi-
nates the need to pass DUT signals as parameters to the procedure. It
is easy to control tens of signals within such a procedure and passing the
same DUT signals over and over makes for tedious amounts of typing.

Unfortunately, now you can’t share the procedure with somebody else in
a package.

### Unconstrained Parameters. 
The procedure is configured in this case using a constant named TEST which improves diagnostic messages from internal assert statements and a constant named frame which holds the expected sequence of values. The alj002 package (see Example 5) declares frame_typ as an unconstrained range of vectors to make it possible for the caller to give a variable number of values to check. The for loop then iterates over range of values returned by the `range`
attribute.

So, this little bit of trickery allows you to have a variable number of
states in your implicit state machine. Try to do that with an explicit
state machine using an enumerated type for the state variable!

### Using the Procedure. 
Pass in the frame as a literal value composed from
an aggregate as shown in Example 5. The declaration for frame_typ defines the index range to be an INTEGER range. Practically a positive
or natural range makes a better choice. Otherwise, the vhdl analyzer
will assign the elements of the aggregate, specified with the positional
notation below, to indexes -2147483648, -2147483647, ... which makes
debugging an annoying exercise. Or you can specify the index values
explicitly using 0, 1, 2, ... but then you must ensure that no gaps appear
or the for loop will fail to operate as expected.

### Example 3 Level Testing

```vhdl
lvl: loop
    exit lvl when sig='1';
    when sig='1';
    end loop lvl;
```

This while loop is skipped entirely if sig='1' upon entry. Ideally you
would stick this loop into its own subprogram. (No restrictions here like
there are for synthesis.)

1. The code for the entire stuffer test bench is found at the very end of
this article.
**Example 5**  Constructing an aggregate that configures the ISM

```vhdl
expect("1.5",
  cHEADER, -- same as was sent
  kELEMENT & X"3",
  kELEMENT & X"2",
  kELEMENT & X"1",
  kELEMENT & X"0",
  "00000")  -- footer isn’t available so default
            -- of 0 is used.
);
```

Of course you could construct the frame by reading in data from a file by passing in the name of a file and letting the procedure read and parse it directly. Such approaches can occasionally make it difficult to diagnose problems. It might be better to use PERL or TCL to generate the VHDL necessary to explicitly produce a sequence of procedure calls. Make the VHDL analyzer do all the parsing work!

**TIME-OUT.** Lastly, notice the use of the time-out as the first line of the procedure. It is sensitive to `fro_tb(kINDI)`, which for those uninitiated to the stuffer example, represents the appearance of the first element of the frame. For verification purposes this signal is a qualifier valid at the clock edge. For the test bench I just care that I see the edge asserted in less then 1000 ns. If it isn’t seen the test bench can gracefully quit and report a failure. Without this test the simulation might run forever and never report anything wrong. (Guess why I added this…) Protecting expect procedures like this is easy with the **WAIT**.

By the way, there is no side effect to using the **WAIT** this way. `fro_tb(kINDI)` will change then the clock edge will appear. But it pays to be careful and document complex side effects that occur in general ISM coding.

**KEEP LOOKING.** There are more goodies in the test bench. But they are easy enough of find now that you are thinking about how to use **WAIT** statements in general.

**Conclusion**

I hope this article increased your appreciation for the power of the implicit state machine coding style. You can write synthesizable and behavioral code with it so there is wide design applicability. Your code gets the following benefits too:

1. Looping structures appear explicitly in the code which dramatically improves readability.
2. Adding, moving, and removing states is easily accomplished with copy, cut and paste text editor operations because there is no state variable encoding to mess with keeping up to date.
3. In test benches, procedures provide for encapsulation of frequently used state machine fragments and,
4. provide a simple mechanism for creating arbitrary numbers of states as necessary.

So, get busy, write a few test benches and then write your elected officials and demand they require all EDA tool vendors to support the synthesis of RTL implicit state machines. You will be glad you did!

**References**

[ASHENDEN] The Designer’s Guide to VHDL, Peter J. Ashenden. Copyright 1996 by Morgan Kaufmann Publishers, Inc. (IMHO if you write VHDL and do not have a copy of this book then you MUST go out and buy one today.)


Example Code

Example 6 The stuffer package (from ALJ002 My VARIABLE state of mind.)

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

PACKAGE alj002 IS
SUBTYPE slv IS std_logic_vector;
CONSTANT kRESET_ASSERTED : std_logic := '1';
CONSTANT kRESET_DEASSERTED : std_logic := '0';

-- A frame consists of header, body and footer elements.
-- The header is the first element with element(indi)='1'.
-- The footer is the first element, anytime after the header,
-- with element(indi)='0'. There can be any number of elements,
-- in the body including zero, between the header and footer.
-- The least significant bits of the element carry data in
-- bits 3 downto 0 and are called the nibble.

CONSTANT kINDI   : integer   := 4;  -- element indicator bit
CONSTANT kELEMENT : std_logic := '1';

SUBTYPE  nibble IS integer RANGE 3 DOWNTO 0;
SUBTYPE  nibble_typ IS std_logic_vector(nibble);
SUBTYPE  element_typ IS std_logic_vector(kINDI DOWNTO 0);
TYPE frame_typ IS ARRAY(integer RANGE <>) OF element_typ;
END PACKAGE alj002;

Example 7 The stuffer entity (from ALJ002 My VARIABLE state of mind).

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
USE work.alj002.ALL;

ENTITY stuffer IS
GENERIC (
  -- Number of nibbles to STuF inbetween header and footer
  stfing : integer RANGE 0 TO 15 := 8
);
PORT (
  -- Frame streams in&out
  fri : IN  element_typ;
  fro : OUT element_typ;
  -- System
**Example 8** The `stuffer` testbench

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
USE work.alj002.ALL;

ENTITY stuffer_tb IS
  GENERIC (
    gClOCK_PERIOD_LOW  : time := 5 NS;
    gClOCK_PERIOD_HIGH : time := 5 NS
  );
END ENTITY stuffer_tb;

ARCHITECTURE test OF stuffer_tb IS
  CONSTANT cCLOCKPERIOD : time := gClOCK_PERIOD_HIGH + gClOCK_PERIOD_LOW;

  -- component generics
  -- Number of nibbles to STuF inbetween header and footer
  CONSTANT stfing : integer RANGE 0 TO 15 := 4;
  -- Just for test purposes. The "F" and "E" are arbitrary
  CONSTANT cHEADER : slv := kELEMENT & X"F";
  CONSTANT cFOOTER : slv := NOT kELEMENT & X"E";

  COMPONENT stuffer IS
    GENERIC (
      -- Number of nibbles to STuF inbetween header and footer
      stfing : integer RANGE 0 TO 15);
    PORT (  
      -- Frame streams in&out
      fri : IN  element_typ;
      fro : OUT element_typ;
      -- System
      rst : IN  std_logic;
      ck  : IN  std_logic);
    END COMPONENT stuffer;

  -- component ports
  -- Frame streams in&out
  SIGNAL fri_tb : element_typ;  -- [IN]
  SIGNAL fro_tb : element_typ;  -- [OUT]
  SIGNAL reset_tb : std_logic;
  SIGNAL clock_tb : std_logic;
  SIGNAL done : boolean := FALSE;  -- Indicates verification complete

BEGIN
  -- component instantiation
```

---

**Copyright (c) 2009, Aspen Logic, Inc. All Rights Reserved.**

**DISCLAIMER:**

THIS VHDL MODEL IS PROVIDED "AS IS" AND "AS THEY STAND". ASPEN LOGIC, INC.

MAKES NO WARRANTIES OR REPRESENTATIONS, EITHER EXPRESS OR IMPLIED,

INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF QUALITY, PERFORMANCE,

MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR AGAINST INFRINGEMENT

OF ANY KIND. THERE ARE NO WARRANTIES WHICH EXTEND BEYOND THE FACE HEREOF.

---

**NOTES:**

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
USE work.alj002.ALL;

ENTITY stuffer_tb IS
  GENERIC (  
    -- Number of nibbles to STuF inbetween header and footer
    stfing : integer RANGE 0 TO 15 := 4;
  );
END ENTITY stuffer_tb;

ARCHITECTURE test OF stuffer_tb IS
  CONSTANT cCLOCKPERIOD : time := gClOCK_PERIOD_HIGH + gClOCK_PERIOD_LOW;

  -- component generics
  -- Number of nibbles to STuF inbetween header and footer
  CONSTANT stfing : integer RANGE 0 TO 15 := 4;
  -- Just for test purposes. The "F" and "E" are arbitrary
  CONSTANT cHEADER : slv := kELEMENT & X"F";
  CONSTANT cFOOTER : slv := NOT kELEMENT & X"E";

  COMPONENT stuffer IS
    GENERIC (  
      -- Number of nibbles to STuF inbetween header and footer
      stfing : integer RANGE 0 TO 15);
    PORT (  
      -- Frame streams in&out
      fri : IN  element_typ;
      fro : OUT element_typ;
      -- System
      rst : IN  std_logic;
      ck  : IN  std_logic);
    END COMPONENT stuffer;

  -- component ports
  -- Frame streams in&out
  SIGNAL fri_tb : element_typ;  -- [IN]
  SIGNAL fro_tb : element_typ;  -- [OUT]
  SIGNAL reset_tb : std_logic;
  SIGNAL clock_tb : std_logic;
  SIGNAL done : boolean := FALSE;  -- Indicates verification complete

BEGIN
  -- component instantiation
```
DUT : COMPONENT stuffer
  GENERIC MAP (  
    -- Number of nibbles to STuF inbetween header and footer
    stfing => stfing)  -- [integer RANGE 0 TO 15]
PORT MAP (  
    -- Frame streams in&out
    fri => fri_tb,  -- [IN  element_typ]
    fro => fro_tb,  -- [OUT element_typ]
    -- System
    rst => reset_tb,  -- [IN  std_logic]
    ck => clock_tb);  -- [IN std_logic]

ASSERT (FALSE) REPORT "stuffer testbench" & LF SEVERITY NOTE;

clk : Clock_tb <= '1' AFTER gCLOCK_PERIOD_LOW WHEN Clock_tb = '0' AND NOT Done ELSE
  '0' AFTER gCLOCK_PERIOD_HIGH;

rst : reset_tb <= kRESET_ASSERTED, kRESET_DEASSERTED AFTER 2.5 * cCLOCKPERIOD;

-----------------------------------------------------------------------------
--stimulus : PROCESS
PROCEDURE framein (  
  CONSTANT frame : IN frame_typ) IS
BEGIN
  FOR i IN frame'range LOOP
    WAIT UNTIL rising_edge(clock_tb);
    fri_tb <= frame(i);
  END LOOP;
END PROCEDURE framein;
BEGIN
  fri_tb <= (OTHERS => '0');
  WAIT UNTIL rising_edge(clock_tb) AND reset_tb = kRESET_DEASSERTED;
  REPORT LF &  "test 1.1: case of header followed immediately by footer" SEVERITY NOTE;
  framein((1 => cHEADER, 2 => '0' & X"E"));
  WAIT UNTIL fro_tb(kINDI) = kELEMENT;
  WAIT UNTIL fro_tb(kINDI) = NOT kELEMENT;
  REPORT LF &  "test 1.2: case of header followed by 1 element" SEVERITY NOTE;
  framein((-1 => cHEADER, 0 => kELEMENT & X"D", 1 => '0' & X"E"));
  WAIT UNTIL fro_tb(kINDI) = NOT kELEMENT;
  REPORT LF &  "test 1.3: case of header followed by 3 elements" SEVERITY NOTE;
  framein((cHEADER,  
    kELEMENT & X"D", kELEMENT & X"A", kelement & X"D",  
    cfFOOTER ));
  WAIT UNTIL fro_tb(kINDI) = NOT kELEMENT;
  REPORT LF &  "test 1.4: case of header followed by 4 elements" SEVERITY NOTE;
  framein((cHEADER,  
    kELEMENT & X"D", kELEMENT & X"A", kelement & X"D", kELEMENT & X"A",  
    cfFOOTER ));
  WAIT UNTIL fro_tb(kINDI) = NOT kELEMENT;
  REPORT LF &  "test 1.5: case of header followed by 5 elements" SEVERITY NOTE;
  framein((cHEADER,  
    kELEMENT & X"D", kELEMENT & X"A", kelement & X"D", kELEMENT & X"A", kelement & X"A", kELEMENT & X"A",  
    '0' & X"0"));
  WAIT UNTIL fro_tb(kINDI) = NOT kELEMENT;
  REPORT LF &  "test 1.6: case of header followed by 6 elements" SEVERITY NOTE;
  framein((cHEADER,  
    kELEMENT & X"D", kELEMENT & X"A", kelement & X"D", kELEMENT & X"A", kelement & X"D", kelement & X"A",  
    '0' & X"0"));
  WAIT UNTIL fro_tb(kINDI) = NOT kELEMENT;
WAIT! Describing implicit state machines with VHDL

REPORT "Stimulus complete" SEVERITY NOTE;
WAIT;
END PROCESS stimulus;

---------------------------------------------------------------------
--
---------------------------------------------------------------------
verify : PROCESS
PROCEDURE expect (CONSTANT test : IN string;
CONSTANT frame : IN frame_typ) IS
BEGIN
WAIT UNTIL fro_tb(kINDI) = kELEMENT FOR 1000 NS;
ASSERT (fro_tb(kINDI) = kELEMENT)
REPORT "Test &test& - no frame appeared"
SEVERITY FAILURE;
FOR i IN frame'range LOOP
WAIT UNTIL rising_edge(clock_tb);
ASSERT (fro_tb = frame(i))
REPORT "test &test& - frame mismatch"
SEVERITY FAILURE;
END LOOP;
END PROCEDURE expect;
BEGIN
WAIT UNTIL rising_edge(clock_tb) AND reset_tb = kRESET_DEASSERTED;

-- In test 1.1 the expected result is a header, 4 body elements then a footer
-- (Incoming frame is shorter in length then outgoing frame)
expect'1.1',
  (cHEADER,      -- same as was sent
  kELEMENT & X'3',
  kELEMENT & X'2',
  kELEMENT & X'1',
  kELEMENT & X'0',
  cFOOTER) -- same as was sent
);

-- In test 1.2 the expected result is a header, 4 body elements then a footer
-- (Incoming frame is shorter in length then outgoing frame)
expect'1.2',
  (cHEADER,      -- same as was sent
  kELEMENT & X'3',
  kELEMENT & X'2',
  kELEMENT & X'1',
  kELEMENT & X'0',
  cFOOTER) -- same as was sent
);

-- In test 1.3 the expected result is a header, 4 body elements then a footer
-- (Incoming frame is shorter in length then the outgoing frame)
expect'1.3',
  (cHEADER,      -- same as was sent
  kELEMENT & X'3',
  kELEMENT & X'2',
  kELEMENT & X'1',
  kELEMENT & X'0',
  cFOOTER) -- same as was sent
);

-- In test 1.4 the expected result is a header, 4 body elements then a footer
-- (Incoming frame is equal in length to the outgoing frame)
expect'1.4',
  (cHEADER,      -- same as was sent
  kELEMENT & X'3',
  kELEMENT & X'2',
  kELEMENT & X'1',
  kELEMENT & X'0',
  cFOOTER) -- footer isn't available so default
  -- of 0 is used.
);
-- In test 1.5 the expected result is a header, 4 body elements then a footer
-- (Incoming frame is longer in length then the outgoing frame)
expect("1.5",
(cHEADER, -- same as was sent
ekELEMENT & X"3",
ekELEMENT & X"2",
ekELEMENT & X"1",
ekELEMENT & X"0",
"00000") -- footer isn't available so default
-- of 0 is used.
);

-- In test 1.6 the expected result is a header, 4 body elements then a footer
-- (Incoming frame is longer in length then the outgoing frame)
expect("1.6",
(cHEADER, -- same as was sent
ekELEMENT & X"3",
kELEMENT & X"2",
kELEMENT & X"1",
kELEMENT & X"0",
"00000") -- footer isn't available so default
-- of 0 is used.
);

WAIT FOR 100 NS;
REPORT "Verification complete" SEVERITY NOTE;
done <= TRUE;
WAIT;
END PROCESS verify;
END ARCHITECTURE test;

-- -- Configure to test the stuffer architecture with the explicit state machine
CONFIGURATION stuffer_tb_test1_explicit_var_cfg OF stuffer_tb IS
FOR test
FOR dut : stuffer
USE ENTITY work.stuffer(explicit_var);
END for;
END FOR;
END stuffer_tb_test1_explicit_var_cfg;

-------------------------------------------------------------------------------

-- Configure to test the stuffer architecture with the implicit state machine
CONFIGURATION stuffer_tb_test1_implicit_var_cfg OF stuffer_tb IS
FOR test
FOR dut : stuffer
USE ENTITY work.stuffer(implicit_var);
END for;
END FOR;
END stuffer_tb_test1_implicit_var_cfg;

-------------------------------------------------------------------------------

-- Configure to test the stuffer architecture with the implicit state machine
CONFIGURATION stuffer_tb_test1_implicit_sig_cfg OF stuffer_tb IS
FOR test
FOR dut : stuffer
USE ENTITY work.stuffer(implicit_sig);
END for;
END FOR;
END stuffer_tb_test1_implicit_sig_cfg;

-------------------------------------------------------------------------------