

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



ALJ003 WAIT! Describing implicit state machines with VHDL

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, what ever 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 than 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
        st1: WAIT ON send_data;
        req <= '1' after 5 ns;
        put_data <= put_data+1 after 6 ns;
        request: LOOP
            EXIT request WHEN ack='1';
            st2: WAIT ON ack;
        END LOOP;
        put_data <= (OTHERS => 'Z') after 6 ns;
        req <= '0' after 5 ns;
        st3: WAIT UNTIL ack='0';
    END PROCESS reqp;
    ackp: PROCESS IS
    BEGIN
        ack <= '0' after 10 ns;
        st1: WAIT UNTIL req='1';
        get_data <= put_data after 3 ns;
        ack <= '1' after 10 ns;
        st2: 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 **ST3** 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

[ALJ002 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 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 [ALJ002](#) used the traditional CASE statement style. In Example 2 the architecture `impli_c_i_t_s_i_g` 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_tpy;
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 r_i_s_i_n_g_e_d_g_e(ck);
            EXIT wait_hdr WHEN (rst = kRESET_ASSERTED);
            stf_ct <= to_unsigned(stfing, 4)-1;
            IF (fri(kINDI) = kELEMENT) THEN
                -- Incoming Hdr has been found so emit it as the header of the
                -- outgoing element stream.
                fro <= fri;
                ftr_scan : LOOP
                    WAIT UNTIL r_i_s_i_n_g_e_d_g_e(ck);
                    EXIT wait_hdr WHEN (rst = kRESET_ASSERTED);
                    IF (fri(kINDI) = NOT kELEMENT) THEN
                        -- Found the footer on the incoming element stream
                        -- to use it for the output stream saved_footer
                        saved_footer <= fri(nibble);
                    END IF;
                    stf2_lp : LOOP
                        fro <= kELEMENT & slv(stf_ct);
                        stf_ct <= stf_ct-1;
                        EXIT stf2_lp WHEN (stf_ct = 0);
                        -- Incoming footer detected and stf_ct > 0. So the input
                        -- stream was shorter then the output stream. Still need to
                        -- stuff the output stream...
                        WAIT UNTIL r_i_s_i_n_g_e_d_g_e(ck);
                        EXIT wait_hdr WHEN (rst = kRESET_ASSERTED);
                    END LOOP stf2_lp;
```

ALJ003 WAIT! Describing implicit state machines with VHDL

```

WAIT UNTIL ri s1 ng_edge(ck);
EXIT wait_hdr WHEN (rst = kRESET_ASSERTED);
fro <= NOT keLEMENT & saved_footer;
EXIT ftr_scan;
ELSE
stf_ct <= stf_ct-1;
fro <= keLEMENT & slv(stf_ct);
IF (stf_ct = 0) THEN
  WAIT UNTIL ri s1 ng_edge(ck);
  EXIT wait_hdr WHEN (rst = kRESET_ASSERTED);
  -- Have to decide what to output for the footer
  IF (fri(kINDI) = NOT keLEMENT) THEN
    fro <= fri;
    EXIT ftr_scan;
  ELSE
    -- Incoming footer hasn't arrived yet but the output
    -- element stream is already fully stuffed so just
    -- output a dummy footer.
    fro <= NOT keLEMENT & X"0";
    ftr_loop : LOOP
      -- Keep searching for the footer...
      WAIT UNTIL ri s1 ng_edge(ck);
      EXIT wait_hdr WHEN (rst = kRESET_ASSERTED);
      EXIT ftr_scan WHEN ((fri(kINDI) = NOT keLEMENT));
    END LOOP ftr_loop;
  END IF;
ELSE
  -- Neither the footer nor the last element to stuff has
  -- been reached yet.
END IF;
END IF;
END LOOP ftr_scan;
END IF; -- (fri(kINDI) = keLEMENT)
END LOOP wait_hdr;
END PROCESS implicit;
END ARCHITECTURE implicit_sig;

```

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 *which* **WAIT** 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 than 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, ftr_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**:

```
EXIT wait_hdr WHEN (rst = kRESET_ASSERTED);
```

What makes it synchronous? The sensitivity list of the preceding **WAIT** includes ck but not *ck* 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 is 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 fl ag as a boolean signal then

```
WAIT ON flag;
```

in one **PROCESS** would be sensitive to either possible change in the signal value of fl ag being changed in another **PROCESS**. Use this statement:

```
flag <= not flag;
```

ALJ003 WAIT! Describing implicit state machines with VHDL

as a simple way to signal an arbitrary event. No need to worry about initializing 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 easily 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 primary 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

WAIT; Suspend permanently.
WAIT ON a, b, c; Suspend then resume when a, b, or c changes.
WAIT UNTIL shark=byte; Suspend then resume when shark changes and matches the signal/constant/variable byte.
WAIT FOR 25 ns; Suspend then resume in 25 ns.
WAIT ON a, b UNTIL c=1; Suspend then resume when a or b changes and c=1. 'c' is not included in the sensitivity list
WAIT ON a, b FOR 25 ns; 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.
WAIT UNTIL zzz or qqg>3 FOR 50 ms; Suspend then resume after 50 ms or when a change on zzz or qqg occurs and zzz is true and qqg is greater than 3. The time-out does not reset should the sensitivity list get an event but the UNTIL condition is still false.
WAIT ON apple UNTIL orange FOR 100 ps; Suspend then resume after either 100 ps has passed or an event on apple has occurred and at that moment orange is true.

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 keep your test bench from failing to recognize failures that cause the simulator to run indefinitely.

LEVEL TESTING. It is not uncommon to test for a signal level before continuing to the next state. The **WAIT UNTIL** statement seems ideal for this but it doesn't work. The sensitivity list is built from the signals in the condition and if they don't change then the value checks in the condition aren't made. So, a loop is necessary.

Example 3 Level Testing

```
lv1: LOOP
  EXIT lv1 WHEN sig='1';
  WAIT ON sig;
END LOOP lv1;
```

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.)

Test Procedures

Sticking 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)¹. Several noteworthy approaches appear in this tiny example.

Example 4 Procedure to check DUT expected output over time

```
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
...
END PROCESS verify;
```

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 eliminates the need to pass DUT signals in 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 6) 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 **POSTIVE** or **NATURAL** range makes a better choice. Otherwise, the VHDL analyzer will assign the elements of the aggregate, specified with 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.

1. The code for the entire stuffer test bench is found at the very end of this article.

ALJ003 WAIT! Describing implicit state machines with VHDL

Example 5 Constructing an aggregate that configures the ISM

```
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(kl NDI)`, which for those uninitialized 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 than 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(kl NDI)` 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.)

[IEEE] IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis. IEEE Std 1076.6™-2004 (Revision of IEEE Std 1076.6-1999)

[ALJ002] "My Variable State of Mind." Timothy R. Davis. Copyright 2009 Aspen Logic, Inc. <http://www.aspenlogic.com/journal/alj002.html>

Feedback

To provide feedback (of any type) to the author of this article please send an E-mail to Journal@AspenLogic.com. With respect to this article, all feedback emails are considered as being in the public domain prior to disclosure to Aspen Logic, Inc. so *do not* send any confidential or proprietary information to us.

Subscribe

To get E-mail updates on new and/or revised articles visit the Aspen Logic Journal page at <http://www.aspenlogic.com/journal> and click the subscribe button. Quick. Easy. Free. What more can you ask for?

About the Author



Tim Davis is the night time janitor at Aspen Logic where he practices "janitorial engineering" on designs he finds lying around the office. Drop him a line if you need something cleaned up.

History

Revision History

Ver	Change
1.0	Initial version

Notes

ALJ003 WAIT! Describing implicit state machines with VHDL

Example Code

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

```

-----
-- 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;

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 down to 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.](#))

```

-----
-- 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 IS
  GENERIC (
    -- Number of nibbles to STuF in between header and footer
    stfing : integer RANGE 0 TO 15 := 8
  );
  PORT (
    -- Frame streams in&out
    fri : IN element_typ;
    fro : OUT element_typ;
    -- System

```

ALJ003 WAIT! Describing implicit state machines with VHDL

```

rst : IN std_logic;
ck  : IN std_logic
);
END ENTITY stuffer;

```

Example 8 The stuffer testbench

```

-----
-- 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 (
    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

```

ALJ003 WAIT! Describing implicit state machines with VHDL

```

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",
    cFOOTER));

  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",
    cFOOTER));

  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"D",
    '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;

```

ALJ003 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.
    );
```

ALJ003 WAIT! Describing implicit state machines with VHDL

```
-- 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
   kELEMENT & X"3",
   kELEMENT & X"2",
   kELEMENT & X"1",
   kELEMENT & 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
   kELEMENT & 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;
```