

# My VARIABLE state of mind

ALJ002 (V1.1) June 8, 2009

**ABSTRACT** • Since 1987 and my first exposure to VHDL, I have seen the humble VARIABLE relegated to the backwaters for use only as combinatorial logic. Verilog coders seem constrained to do the same. This article presents a convention for using blocking assignments effectively with locally declared REG objects in Verilog and variables in VHDL. It specifically addresses the case for inferring flip-flops and why it also eliminates the possibility of execution order dependent code.

**KEYWORDS** • VHDL, Verilog, variable, reg, memory, flip-flop, latch, always block, blocking assignment, nonblocking assignment, memory inference, logic synthesis, synchronous logic, finite state machine



(VHDL) VARIABLES and (Verilog) REGS, or to be more specific, locally declared objects with blocking assignment semantics (hereafter called VARIABLES -- all caps) do not get the credit they deserve. Engineers have been told that VARIABLES should only find use as combinatorial logic. Yet, I believe your HDL (Verilog or VHDL) descriptions will benefit in several ways when they are employed. For example:

1. Both PROCESS and ALWAYS blocks permit local declarations within the new scope each provides. This new scope provides an opportunity to bring those declarations close to where the action is thus improving descriptibility and readability.
2. Because of scoping, local Verilog REG declarations eliminate the possibility of simulation failures due to order of execution dependent blocking assignments.
3. VARIABLES have a lower simulation memory and execution time overhead. That is based on the idea that only one value needs to be stored and no overhead is needed for determining PROCESS sensitivity list changes or for keeping track of past values or future driver events.
4. For synchronous logic, VARIABLES permit inference of flip-flops safely and easily in addition to the common role of describing combinatorial logic.
5. The describer gets easy cut-paste of common blocks with no requirement to come up w/ clever names for replicated signals.

Expressing these ideas in one article with Verilog and VHDL examples has proven quite challenging. You may not find the conclusions earth shattering but I hope you will find a nugget or two of gold that will make your next HDL description more valuable.

The article progresses from a discussion of blocking assignments (necessary to put VHDL and Verilog on the same page) to an example of a register structures containing feedback (to show register inference from VARIABLES works everywhere), followed by a state machine.

## Blocking Assignments

Many authors and HDL style guide developers seem stuck on nonblocking assignments. I like a little more flexibility when describing hardware, and if you are game, then perhaps I can add a little bit to your HDL toolbox as well. To get there, in keeping with my mission to make this a joint VHDL+Verilog exercise, I need to melt the terminology used by both languages together -- this isn't easy and will probably upset people from both camps!

Verilog uses the terms "blocking" and "nonblocking" for the semantics of assignment to the REG object type using the "=" and "<=" operators respectively. VHDL makes the same distinction by object declaration type: "VARIABLE" and "SIGNAL" respectively with operators ":=" and "<=".

Recall (or learn as the case may be) that a *blocking* assignment is one where the right hand side ("RHS") expression is evaluated and the assignment to the left hand side ("LHS") target occurs without the evaluation of any intervening statements. In a *nonblocking* assignment the RHS expression gets computed but is stored for later assignment to the LHS at a point in the future determined by the implementation of the *simulator* kernel (typically at the conclusion of a time step). Nonblocking assignments represent the built-in means to express concurrency and distinguish Verilog and VHDL from languages like C.

Now, for the rest of this article, to keeps things easy and somewhat equivalent, when I mention "blocking assignment" I will assume it is always one to a *locally declared* ALWAYS block REG or PROCESS statement VARIABLE.<sup>1</sup> A nonblocking assignment will refer to an assignment using the "<=" operator to a REG declared at the top of a module or a SIGNAL declared at the top of an ARCHITECTURE. The two examples below illustrate what I mean for both Verilog and VHDL.

1. I'm not sure if this footnote will do it but I don't want to get a bunch of hate mail from Verilog coders who don't like this style. This isn't a Verilog versus VHDL religion article so if you have a better approach then feel free to write your *own* articles. Don't grind your axe on me!

The Verilog example of Figure 1 shows a blocking assignment to a local REG for comparison to the same operation in VHDL. Note that to declare a local Verilog requires the label ('test' in this case) on the *begin..end* procedural block. Now, since a local to the *always* block it isn't (easily) possible for another procedural block to use a blocking assignment to it. That eliminates the possibility of creating an ordering problem that could cause a simulation mismatch.

```

module local_var(clk, ctr);
input clk;
output [7:0] ctr;
reg ctr;

always @(posedge clk)
begin: test
reg [7:0] alpha;
alpha = alpha+1;
ctr <= alpha;
end
endmodule
    
```

Figure 1 Verilog blocking assignment to locally scoped REG

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY local_var IS
PORT (clk : IN std_logic;
ctr : OUT unsigned(7 downto 0));
END ENTITY local_var;

ARCHITECTURE syn OF local_var IS
BEGIN
PROCESS (clk)
VARIABLE a : unsigned(7 DOWNTO 0);
BEGIN
IF rising_edge(clk) THEN
a := a+1;
ctr <= a;
END IF;
END PROCESS;
END ARCHITECTURE syn;
    
```

Figure 2 VHDL blocking assignment locally scoped VARIABLE

Now, what if I want three counters? Cut and paste three times and change the last nonblocking signal assignment. Voila! Writing a hardware description with a local REG and a blocking assignment gives you conveniently reproducible chunks of code.

## Mind your D's and Q's

### Feedback

Let's stretch the example a little bit more. Figure 3 illustrates a three bit ring counter. It allows each bit to be synchronously set with an initial hi/lo followed by the opposite state for the other trailing bits. The VHDL coding for the ring counter description appears in Figure 4.

You might ask, "What is with all the *\_d* and *\_q* suffixes?" Well, what you call *extra* I call *extra handy*. Without using the temporary VARIABLE *ff1\_d* how would you arrange the code to handle the feedback? You can't. No matter what order you put the blocking assignments (labeled *ba1*, *ba2* and *ba3*) one of the terms ends up being combinatorial and does not infer a flip-flop.

The *ff1\_d := ff3\_q* assignment represents the degenerate case of a "wire" being a "combinatorial" logic feedback path and it solves the feedback problem neatly. It has the added descriptive bonus that *most* readers will naturally grock the *\_d* suffix as meaning *combinatorial* and the *\_q* as meaning *flip-flop*.

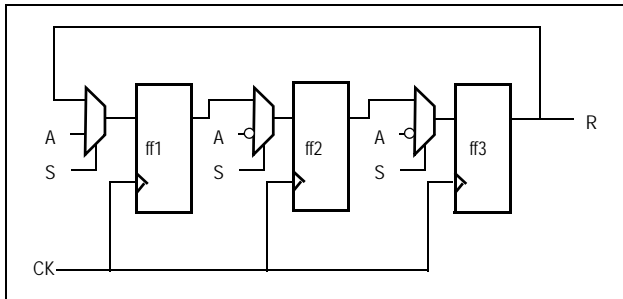


Figure 3 Ring Counter

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY ring_counter IS
    PORT (A1, S, CK : IN std_logic;
          R1 : OUT std_logic
          );
END ENTITY ring_counter;

ARCHITECTURE syn1 OF ring_counter IS
BEGIN
    -- State machine Style
    ring1: PROCESS (CK)
        VARIABLE ff1_d, ff1_q : std_logic;
        VARIABLE ff2_d, ff2_q : std_logic;
        VARIABLE ff3_d, ff3_q : std_logic;
    BEGIN
        IF rising_edge(CK) THEN
            -- Combinatorial logic
            ff1_d := ff3_q; -- DEFAULT assignment
            IF (S = '1') THEN
                ff1_d := A1;
            END IF;
            ff2_d := ff1_q; -- DEFAULT assignment
            IF (S = '1') THEN
                ff2_d := NOT A1;
            END IF;
            ff3_d := ff2_q; -- DEFAULT assignment
            IF (S = '1') THEN
                ff3_d := NOT A1;
            END IF;
            -- Sequential logic
            ba1: ff1_q := ff1_d;
            ba2: ff2_q := ff2_d;
            ba3: ff3_q := ff3_d;
            -- Nonblocking assignment to port signal
            R1 <= ff3_q;
        END IF;
    END PROCESS ring1;
END ARCHITECTURE syn1;
    
```

Figure 4 Ringer counter VHDL

I have to ask now, why would you want to declare a SIGNAL, or equivalently a global REG, outside the vicinity of the code were it is used merely to handle a simple feedback path? I say you should not, and now hopefully, would not want to. Using a blocking assignment, with a local declaration to keep everything close by, is as natural as the real hardware.

*Order independent assignments*

Isn't it nicer to show the left to right flow of Figure 4 with top to bottom coding? Using the D's and Q's approach it doesn't matter what order the set of D or set of Q assignments are made in as long as the D's are written before the Q's. That shouldn't be surprising because real hardware is wired that way. So breaking up the assignments with D's and Q's allows you to handle feedback easily AND allows you to naturally express the hardware dataflow in your description.

*Problems in the pipeline*

Lets look at some ways this style does not work. You might be tempted to arrange the code in a pipeline as shown below in Figure 5 with the \_q following the associated \_d but organized in stages

```

ARCHITECTURE syn2 OF ring_counter IS
BEGIN
    -- Pipeline style
    ring2: PROCESS (ck)
        VARIABLE ff1_d, ff1_q : std_logic;
        VARIABLE ff2_d, ff2_q : std_logic;
        VARIABLE ff3_d, ff3_q : std_logic;
    BEGIN
        IF rising_edge(ck) THEN
            -- Stage 1
            ff1_d := ff3_q;
            IF (S = '1') THEN
                ff1_d := A1;
            END IF;
            ff1_q := ff1_d;
            -- Stage 2
            ff2_d := ff1_q;
            IF (S = '1') THEN
                ff2_d := NOT A1;
            END IF;
            ff2_q := ff2_d;
            -- Stage 3
            ff3_d := ff2_q;
            IF (S = '1') THEN
                ff3_d := NOT A1;
            END IF;
            ff3_q := ff3_d;
            -- Nonblocking assignment to port signal
            R1 <= ff3_q;
        END PROCESS ring2;
    END ARCHITECTURE syn2;
    
```

No register here since no time passes between the assignment and it's use in a RHS expression!

Figure 5 Bad pipeline coding

Unfortunately using \_d and \_q doesn't cause flip flops to get synthesized but it was a nice try. This coding just creates a chain of combinational logic. A description infers flip-flops whenever hardware time passes between a value assignment to the LHS \_q and use of that value in the RHS of the \_d expressions. Placing all of the \_q assignments at the end of the PROCESS, right before the implicit WAIT statement ensures that hardware time advances between \_q and the \_d expressions.

You might also want to try the style of Figure 6. While it seems like the rising\_edge() tests represent the passage of time they do not. They only test whether the event that caused the sensitivity list to fire represents a rising edge.

Unfortunately, at the time of this writing, at least one major synthesis tool compiles this code with no errors and produces a chain of three flip-flops. But the simulation fails to match the synthesized result.

**State Machines**

The comments in Figure 4 hint at the state machine style which I present below with a design example called "stuffer". The goal is to introduce a sufficiently complicated example to be interesting but not one so complicated that it makes the underlying lesson difficult to appreciate. The design itself has no significant value other than as an example and thus is not tested -- I suspect it works but there are no guarantees.<sup>1</sup> Its main goal is to show the DQ style.

1. It does pass through synthesis.

```

ARCHITECTURE syn3 OF ring_counter IS
BEGIN
  -- Pipeline style
  ring3: PROCESS (ck)
    VARIABLE ff1_d, ff1_q : std_logic;
    VARIABLE ff2_d, ff2_q : std_logic;
    VARIABLE ff3_d, ff3_q : std_logic;
  BEGIN
    -- Stage 1
    IF rising_edge(ck) THEN
      ff1_d := ff3_q;
      IF (S = '1') THEN
        ff1_d := A1;
      END IF;
      ff1_q := ff1_d;
    END IF;
    -- Stage 2
    IF rising_edge(ck) THEN
      ff2_d := ff1_q;
      IF (S = '1') THEN
        ff2_d := NOT A1;
      END IF;
      ff2_q := ff2_d;
    END IF;
    -- Stage 3
    IF rising_edge(ck) THEN
      ff3_d := ff2_q;
      IF (S = '1') THEN
        ff3_d := NOT A1;
      END IF;
      ff3_q := ff3_d;
    END IF;
    -- Nonblocking assignment to port signal
    R1 <= ff3_q;
  END PROCESS ring3;
END ARCHITECTURE syn3;

```

Figure 6 Nice try at a pipeline coding style

The “stuffer” describes a packet filter with frames (aka packets) following a simple scheme. The frame consists of a contiguous sequence of elements called header, body and footer which always appear in that order. The “stuffer” removes all of the body elements in the incoming stream and replaces them with an ascending sequence of count values in the range 0 to 15. The output frames thus have the original header, new body elements, and original footer (most of the time.)

The output frame consists of the header and footer elements of the incoming frame. However, the incoming body nibbles (everything in between the header and footer) get replaced with a static number of body element data nibbles. The nibble contains an ascending count value.

**VHDL FSM**

**STUFFER INTERFACE.** A **PACKAGE** and **ENTITY** define the stuffer’s interface. Since a frame consists of a number of elements and each element has two fields it is handy to “package” up this definition in one spot. The subtype declaration `nibble` acts like a “constant” by giving a name to the integer range used for slicing off the bits of the “nibble” payload.

**STUFFER ARCHITECTURE.** I’ll present the architecture in three parts: The asynchronous reset (Figure 9), the “clocked” portion containing combinatorial logic for the output and next state decoder (Figure 10) and finally the remaining portion (Figure 11) where the flip-flop get inferred.

First, notice there isn’t a single **SIGNAL** declared in the **ARCHITECTURE** as no nonblocking assignments are utilized for the combinatorial next state logic of the FSM. Everything is done with **VARIABLES**. Figure 9 shows the D&Q convention at work for the state variable, the stuff count, storage for the trailing footer, the filtered output element and a “convenience” variable called `fork` used in a case statement.

```

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, appearing 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 kINDI-1 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;

```

Figure 7 Stuffer interface frame definition

```

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

```

Figure 8 Stuffer Entity

Notice that the type declaration for `fsm_typ` is right there inside the **FSM** state machine **PROCESS** and not in the **ARCHITECTURE** declarative area. Nothing outside of the process needs to see this definition so putting it in the **ARCHITECTURE** declarative area seems out of place. Again, I like to place the declarations as close to the action as possible.

The typical **if-elsif** synthesis template for an asynchronous reset is employed here. The `_q` assignments in the reset clause are effectively at the “end” of the process even if they aren’t textually at the end.

```

ARCHITECTURE explicit_var OF stuffer IS
  -- LOOK MA ... NO SIGNALS!
  BEGIN
    varfsm : PROCESS (ck, rst) IS
      TYPE fsm_typ IS (HDR, STF1, STF2, FTR1, FTR2, F1 NI SH);
      VARIABLE state_d, state_q : fsm_typ := HDR;
      VARIABLE stf_ct_d, stf_ct_q : unsigned(nibble);
      VARIABLE trailer_d, trailer_q : nibble_typ;
      VARIABLE fro_d, fro_q : element_typ;
      VARIABLE fork : slv(0 TO 1);
    BEGIN
      IF (rst = '1') THEN
        state_q := HDR;
        fro_q := (OTHERS => '0');
        trailer_q := (OTHERS => '0');
        stf_ct_q := (OTHERS => '0');
      END IF;
    END PROCESS;
  END ARCHITECTURE;

```

Figure 9 FSM declarations and asynchronous resets

The next major section (Figure 10) contains the rising edge test in the ELSIF clause. It starts with a collection of default assignments from the current (q) values. Doing this provides a safety net to protect against accidental latches in some contexts but here, because of the (ck, rst) sensitivity list we are expecting flip-flops. Using default assignments like this is more a personal preference than anything because I like to have the default state transition be to *hold* the current state. That way I don't have to put in a bunch of ELSE clauses so that the state VARIABLE will hold its value. This may not be natural to you. So, because I do that with the state variable, I tend to have *all* my registers hold their "state".

This example is a little thin on combinatorial logic (other than the next state mux) but two pieces should stick out: the "stf\_ct\_d := stf\_ct\_q-1;" (in several places) and "fork := ...; CASE fork..." (inside the STF1 state). Most authors suggest the only acceptable use of a variable is for the latter case (no flip flop inferred). This article sets a firm position that the former case is overlooked and under utilized (flip flop *to be* inferred). Using the d convention and stf\_ct\_d as the target makes stf\_ct\_q-1 available for other uses like in condition tests. If you find yourself in an off-by-one clock situation because you adjusted a pipeline somewhere else it is always handy to have the combinatorial logic expression readily available to do something one clock early.

Notice that not a single q appears on the left hand side as the target of an assignment as this point. The code is purely combinatorial.

Finally, in Figure 11, the code marries the q with their d's right before the end of the process (and the implicit WAIT statement). No other blocking assignments appear after this to confuse the logic synthesis tool or the reader.

```

-- Inferred flip-flops
state_q := state_d;
fro_q := fro_d;
trailer_q := trailer_d;
stf_ct_q := stf_ct_d;
END IF;
-- Nonblocking PORT signal assignments
fro <= fro_q;
END PROCESS varfsm;
END ARCHITECTURE syn;

```

Figure 11 FSM State + Output registers

Using the if (rst) elsif (rising\_edge(ck)) statement has a simulation advantage: the large block of sequential statements making up the bulk of this description only eats simulation time when ck is rising. A more simulation intensive but possibly necessary modification is to use the approach in Figure 12 instead.

```

PROCESS (ck, rst, othersigs)
BEGIN
CASE state_q
..._d := calculation based on othersigs
END CASE;
IF rst='1' THEN
_q := reset defaults;
ELSIF rising_edge(ck) THEN
_q := _d;
END IF;
END PROCESS;

```

Figure 12 Alternate coding structure for FSM

The combinatorial CASE statement moves outside and before the IF statement. This is unavoidable if the combinatorial logic must drive an external block of logic via a nonblocking assignment AND that external logic drives signals used later by the remainder of the combinatorial logic. For example, an address SIGNAL to an external memory instance where the data output from the memory controls the FSM.

```

ELSIF (rising_edge(ck)) THEN
-- Next State decoder, Outputs
state_d := state_q; -- default is hold state
fro_d := fro_q;
trailer_d := trailer_q;
stf_ct_d := stf_ct_q;
CASE state_q IS
WHEN HDR =>
stf_ct_d := to_unsigned(stfing, 4)-1;
IF (fri(kINDI) = kELEMENT) THEN
state_d := STF1;
fro_d := fri;
END IF;
WHEN STF1 =>
-- Hdr has been found and emitted. Start stripping
-- the incoming elements and output the count elements
-- instead.
IF (stf_ct_q = 0) THEN
fork := fri(kINDI) & '0';
ELSE
fork := fri(kINDI) & '1';
END IF;
CASE fork IS
WHEN "00" => -- ftr & ct=0
-- Last element to stuff
state_d := FINISH;
fro_d := kELEMENT & slv(stf_ct_q);
trailer_d := fri(nibble);
WHEN "01" => -- ftr & ct>0
state_d := STF2;
fro_d := kELEMENT & slv(stf_ct_q);
trailer_d := fri(nibble);
stf_ct_d := stf_ct_q-1;
WHEN "10" => -- !ftr & ct=0
state_d := FTR1;
fro_d := NOT kELEMENT & slv(stf_ct_q);
WHEN "11" => -- !ftr & ct>0
fro_d := kELEMENT & slv(stf_ct_q);
stf_ct_d := stf_ct_q-1;
WHEN OTHERS =>
REPORT "Illegal framing bit" SEVERITY ERROR;
END CASE;
WHEN STF2 =>
-- Trailing ftr reached but there are still more count values to
-- stuff.
fro_d := kELEMENT & slv(stf_ct_q);
stf_ct_d := stf_ct_q-1;
IF (stf_ct_q = 0) THEN
state_d := FINISH;
END IF;
WHEN FTR1 =>
IF (fri(kINDI)=NOT kELEMENT) THEN
fro_d := fri;
state_d := HDR;
ELSE
fro_d := NOT kELEMENT & X"0";
state_d := FTR2;
END IF;
WHEN FTR2 =>
-- All of the count values have been stfed but the trailing
-- ftr has not been found yet. The trailing ftr of the output
-- stream has already been emitted. Just wait for the incoming
-- ftr and then quit
IF (fri(kINDI) = NOT kELEMENT) THEN
state_d := HDR;
END IF;
WHEN FINISH =>
-- Just emit stored ftr
state_d := HDR;
fro_d := NOT kELEMENT & trailer_q;
END CASE;

```

Figure 10 FSM Combinatorial Cloud

If it weren't for the simulation hit I would always use the structure of Figure 12 because I like all of the q assignments *together* at the *end* of the PROCESS.

By the way, the EMACS text editor (with Reto Zimmerman's excellent VHDL MODE) will automatically fill in the sensitivity list (to add *others*) to eliminate any synthesis tool warnings about missing sensitivity list signals and guarantee simulation to synthesis coherence.<sup>1</sup>

### Verilog FSM

As a community service, for those who suffer from VHDL allergic reactions, I've provided the same state machine coded in Verilog in Figure 13 and Figure 14.

```

`timescale 1ns / 1ps
module stuffer (fri, fro, rst, ck);

  input [4:0] fri;
  output reg [4:0] fro;
  input rst;
  input ck;

  parameter stfing = 4'd8;
  parameter kINDI = 4;
  parameter ELEMENT = 1'b1;

```

Figure 13 Verilog coding structure for FSM

## Simulation & debugging concerns

Some have avoided VARIABLES simply because viewing them with a waveform viewer was impossible. I can't speak for all simulators out there but my name brand simulator permits VARIABLE objects to be shown on the waveform viewer. If your simulator supports Verilog I can't believe it would be unable to show VHDL VARIABLES in its waveform viewer.

Also, I find it somewhat easier to debug a PROCESS which uses VARIABLES. Single stepping through the statements provides immediate feedback on what changes so that bugs seem more readily apparent.

## Conclusion

Well there you have it. I hope the idea of using `_d` and `_q` convention will encourage you to develop PROCESS/ALWAYS blocks that utilize local VARIABLES and blocking assignments for both combinatorial AND flip-flop logic. Your description will keep declarations and logic together in one tight package, your sims will be faster, and you can code for simple cut-n-paste reuse.

If nothing else you now understand my VARIABLE state of mind!

1. Verilog 2005 supports a smart solution: `always @(*)` says, "I want everything on the sensitivity list".

```

always @(posedge ck)
begin: varfsm
  parameter HDR=3'b000, STF1=3'b001, STF2=3'b010,
            FTR1=3'b011, FTR2=3'b100, FINISH=3'b101;
  reg [2:0] state_d, state_q;
  reg [3:0] stf_ct_d, stf_ct_q;
  reg [3:0] trailer_d, trailer_q;
  reg [4:0] fro_d, fro_q;
  if (rst==1'b1) begin
    state_q = HDR;
    stf_ct_q = 0;
    trailer_q = 0;
    fro_q = 0;
  end else begin
    case (state_q)
    HDR: begin
      stf_ct_d = 8;
      if (fri[kINDI] == ELEMENT) begin
        state_d = STF1;
        fro_d = fri;
      end
    end
    STF1: begin
      case ({fri[kINDI], stf_ct_q ? 1'b1 : 1'b0})
      2'b00: begin
        state_d = FINISH;
        fro_d = {ELEMENT, stf_ct_q};
        trailer_d = fri[3:0];
      end
      2'b01: begin
        state_d = STF2;
        fro_d = {ELEMENT, stf_ct_q};
        trailer_d = fri[3:0];
        stf_ct_d = stf_ct_q-1;
      end
      2'b10: begin
        state_d = FTR1;
        fro_d = {~ELEMENT, stf_ct_q};
      end
      2'b11: begin
        fro_d = {ELEMENT, stf_ct_q};
        stf_ct_d = stf_ct_q - 1;
      end
    end
  endcase
  end
  STF2: begin
    fro_d = {ELEMENT, stf_ct_q};
    stf_ct_d = stf_ct_q-1;
    if (stf_ct_q == 0) state_d = FINISH;
  end
  FTR1: begin
    if (fri[kINDI] == ~ELEMENT) begin
      fro_d = fri;
      state_d = HDR;
    end else begin
      fro_d = {~ELEMENT, 4'b0};
      state_d = HDR;
    end
  end
  FTR2: begin
    if (fri[kINDI] == ~ELEMENT)
      state_d = HDR;
  end
  FINISH: begin
    state_d = HDR;
    fro_d = {~ELEMENT, trailer_q};
  end
endcase // case (state_q)
state_q = state_d;
fro_q = fro_d;
trailer_q = trailer_d;
stf_ct_q = stf_ct_d;
end // else: !if(rst)
fro <= fro_q;
end // block: varfsm
endmodule // stuffer

```

Figure 14 Verilog coding structure for FSM

**Feedback**

To provide feedback (of any type) to the author of this article please send an E-mail to [Journal@AspenLogic.com](mailto: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.

**Notes**

**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**

Ver	Change
1.0 30-Apr-2009	initial version
1.1 8-Jun-2009	Figures 7, 8, 9 and 13 were all updated to reflect bugs found in the state machine. See also: <a href="#">ALJ003: WAIT! Describing implicit state machines with VHDL</a> for details on the testbench used to find the bugs.