Cliff Cummings' FSM Proposals - 20020301

Introduction: I believe the current proposals for the FSM syntax are flawed, but there is also much to be desired within pieces of the existing proposals. I submit for the committees consideration a modification to the FSM proposals that begin on page 31 of the Draft 4 SystemVerilog Preliminary Standard.

I have worked up another large FSM design example which I will include with this email, showing coding considerations. This FSM has 10 states, three outputs, and more transition arcs. This example shows more of the disparity between FSM coding styles and reveals that the current FSM proposal, unmodified, can actually be very verbose. See the diagram and example fsm_cc8sv1a.slg to view the verbose nature of the current porposal.

For those with access to the SystemSim simulator, the first eight examples will compile if you first make the following two global changes:

s/state/sttte/ - the annoying fact that state is now a SystemVerilog keyword
s/always @*/always_comb/ - @* not yet supported by SystemSim

One other note - SystemSim translates X's in enumerated types to 0's. I don't know if that is the intent of the SystemVerilog syntax or not. Creating a variable of all X's was turned into a variable of all 0's, which conflicted with the S0 state. That is why the XX variable is defined as 4'b11xx in the fsm_cc8b_enum2 model.

I have not run these designs through a testbench so there could still be functional errors in the designs.

Depending on what the committee approves, I would be willing to follow up, propose wording, make and test the appropriate changes in our simple examples found in Draft 4. Until I know the committee's mind, I would prefer to limit my time to what I have done so far.

Regards - Cliff

Included Example Files:

fsm_cc8.v - Verilog-2001, one always block style of FSM design. Similar to Draft 4 FSM styles (145 lines of code - registered outputs - combinational outputs would required another always block)

fsm_cc8a.v - Verilog-2001, two always block style of FSM design (79 lines of code - combinational outputs)

fsm_cc8b.v - Verilog-2001, three always block style of FSM design (82 lines of code - registered outputs)

fsm_cc8b_enum.slg - SystemVerilog, three always block style of FSM design with enumerated types for state and next - higher level design style (no state definitions - yet!) as envisioned by Simon Davidman (81 lines of code - registered outputs)

fsm_cc8b_enum2.slg - SystemVerilog, three always block style of FSM design with enumerated types for state and next, including state encodings for the enumerated labels (81 lines of code - registered outputs)

fsm_cc8sv1.slg - SystemVerilog, style of FSM design using one always_comb block, "state" declaration syntax and transition statement and tokens (**136** lines of code - registered outputs - combinational outputs would required another always block - no asynchronous reset possible and no state encodings possible)

fsm_cc8sv1a.slg - SystemVerilog, style of FSM design using one always_comb block, "state" declaration syntax and transition statement and tokens, combining transitions and output assignments where possible (**129** lines of code - registered outputs - combinational outputs would required another always block - no asynchronous reset possible and no state encodings possible)

fsm_cc8b_onehot.v - Verilog-2001, three always block onehot style of FSM design (85 lines of code - registered outputs)

fsm_cc8b_onehot2.slg - Proposed enhancement style - three always block onehot style of FSM design with enum_onehot types for state and next, including state encodings for the enumerated labels and state->>next transition expression to reduce next-state assignment verbosity (62 lines of code after removing comments - registered outputs)

fsm_cc8b_onehot3.slg - Proposed enhancement style - three always block onehot style of FSM design with enum_onehot types for state and next, including state encodings for the enumerated labels (62 lines of code after removing comments - registered outputs)

fsm_cc8a_goto.v - SystemVerilog proposed implicit FSM style using disable, labels and gotos (65 lines of code after removing comments - registered outputs)

PROPOSALS:

PROPOSAL: Delete the "state" keyword and declaration syntax

Reasons:
(1) "state" is a common identifier in 1,000's of existing Verilog designs. I had to do global search and replace from "state" to "sttte" to make my FSM designs work with SystemSim (what a pain!).

(2) I believe the capabilities we want in FSM design can be handled using enum (see more proposals and examples below).

(3) "state" does not permit value assignment. "enum" does permit value assignment, one of the capabilities we were seeking for FSM design.

PROPOSAL: Delete clocking and resetting from "state" syntax or any proposed replacement

Reasons:
(1) Reset handling in a two or three always block design is only three lines of code and can easily unambiguously handle either synchronous or asynchoronous resets. I think we have spent too much time trying to create a short-hand syntax for something that is already pretty code-efficient.

(2) The current syntax requires application of the clocking information to FSMs inside of an always_comb block, which is pretty strange and quite confusing.

PROPOSAL: do not require begin-end for transition-endtransition item-statements

Note: I can live with begin-end but removing them would be nice.

Reasons:
(1) VHDL does not require begin-end inside of VHDL case statements. Verilog does because a Verilog case item can be an expression, making it more difficult to detect where a case-item-statement ends and where a case-item-expression begins. I believe transition does not permit expressions in the tested items, which should make detection of the transition-item easier to find, even without the begin-end pair.

(2) The current proposal tried to allieviate this by permitting both a transition and an output assignment in the same statement. This works fine unless there are multiple conditional outputs that have to be assigned for that transition, then begin-end is again required (see example file ##). Concatenated some assignments would remove more begin-end pairs but that is a pretty silly approach to reducing begin-end pairs.

PROPOSAL: Add a new syntax for transition-blocks:

```
transition (test->>transition_destination)
  test1: ... code ... ->> transition_destination1;
         ... code ... ->> transition_destination2;
  test2: ... code ... ->> transition_destination3;
         ... code ... ->> transition_destination4;
endtransition
```

Reasons:
(1) The current FSM proposal uses the transition expression to compare against transition items. This should still be allowed. The above proposal also indicates in the transition expression what the default enumerated output type will be assigned.

(2) A hierarchical name could still be used to override the default output variable. This might be sufficient to support hierarchical FSM design (we need a good hierarchical FSM design to verify this claim).

PROPOSAL: Add enum_onehot syntax:

Reasons:
(1) enum_onehot would assist waveform viewers when coding onehot variables. Not even available in Verilog or VHDL today. This would be a leap forward to support hardware design.

(2) The transition expression of the form (test->>test_destination) would now compare the indexed test bit to a "1" and would assign all zeros to the output of the ->> transition except for the "test_destination" indexed bit, which would be set.

Requires indexing into a vector by an enumerated value. Currently, indexing can only be done with an integer value.

Note: state machines are not the only hardware that use onehot codes. Wide fast multiplexers are often coded using onehot selects wired to tristate drivers that drive a common bus. Control logic is often coded as either encoded or onehot, for the latter each control bit does not have to first be decoded.

GOALS:

I believe the goals of a new FSM syntax should address the following:

- Architectural exploration without making state assignments
- An enumerated style to facilitate state-name display in a waveform viewer
- Facilitate design of hierarchical FSMs (although we need a great example to prove this is importance - if anyone has a good hierarchical FSM design description or state diagram(s), I would appreciate that the description or state diagrams be sent to me by email or FAX (503-641-8486) and I will commit to coding the design in existing Verilog and SystemVerilog for comparison - we are sorely lacking a good example to show the need and advantage of this capability)
- A more concise syntax if possible
- Simple transition to an RTL style with state encodings
- Simple transition to other important FSM styles that offer synthesis and implementation advantages
- Ability to efficiently control output assignments

VHDL Complaints About Verilog FSM Coding

Whenever you teach VHDL coders to do Verilog FSM design, they have two big complaints:
(1) State assignments must be made (no enumerated types) - this is the same issue that has Simon raised.
(2) To display state names in a Verilog waveform display, the state names must be assigned to variables declared like "reg [6*8:1] statename; // ASCII display register" and then displayed as a string format in

the waveform viewer. This requires a significant amount of additional code to make all of the ASCII statename assignments.

Both of the above issues are solved using VHDL enumerated types.

VHDL Problems With Onehot FSM Coding

VHDL coders still have problems with an efficient onehot FSM coding style. An efficient onehot coding style is accomplished by turning the enumerated state type into an index into the state register instead of the onehot encoding itself.

In Verilog, the efficient onehot coding style uses:

```
// The parameters represent an index into the onehot state vector
parameter S0 = 0,   // not 0...0001
          // note S0 does not equal onehot state encoding
          S1 = 1,   // not 0...0010
          // note S1 does not equal onehot state encoding
          S2 = 2,   // not 0...0100
          // note S2 does not equal onehot state encoding
          S3 = 3,   // not 0...1000
          // note S3 does not equal onehot state encoding
          ...
          Sn = n-1; // not 1...0000

reg [n-1:0] state, next;
...
// Must add parallel_case directive to
// avoid an unnecessary priority encoder
case (1'b1) // 1-bit comparisons, not n-bit comparisons
  state[S0]: ... // 1-bit comparison
  state[S1]: ... // 1-bit comparison
  state[S2]: ... // 1-bit comparison
  state[S3]: ... // 1-bit comparison
  ...
  state[Sn]: ... // 1-bit comparison
endcase
```

In VHDL, the parameters are replaced with enumerated types and the case (1'b1) comparisons are replaced with parallel "if state(n) ..." statements.

Architectural Exploration Without Making State Assignments

An issue that Simon has raise frequently is having the capability to design a state machine without thinking about or making state assignments. I agree. In early architectural exploration, making state-encoding assignments is typically not warranted.

This can be accomplished by using enumerated types. The advantage of using enumerated types is that state assignments can be added later if state encodings do become important, or they can be left unassigned for behavioral simulation or left unassigned if a separate FSM tool is used to make the state assignments.

An Enumerated Style To Facilitate State-Name Display In A Waveform Viewer

Enumerated types address most of the issues related to waveform displays. As long as the enumerated types represent state encodings, displaying state names in a waveform viewer is easily accomplished.

A capability missing from the enumerated types (as described above for VHDL onehot FSMs) is an enumerated type that addresses designs of a onehot nature. Enumerated types work fine for encoded states, but they are unable to display state information when the enumeration is an index into the state vector as opposed to an encoded state.

There is opportunity here to really enhance SystemVerilog. Perhaps make two different enumerated types: the keyword **enum** for the typical enumerated type and a new keyword **enum_onehot** for onehot enumerated types. More on this idea in the proposals section at the bottom of this message.

Facilitate Design Of Hierarchical FSMs

Again, a useful example is needed here. Simon referred to a hierarchical PCI bus FSM that engineers worked on when he was at Virtual Chips. The PCI spec is not proprietary to any one company, so if Simon can remember the design problem, perhaps he could send it out. I would be willing to code the example for comparison purposes.

Simon has mentioned not needing to require output assignments in an FSM design. I think that statement might be applicable to a hierarchical FSM design, where one FSM simply detects the state of the other FSM instead of handshaking an output signal from one FSM to an input signal of another FSM. In this I could see a potentially more concise coding style using some proposed FSM enhancements.

Hierarchically referencing a state instead of passing an output signal from that state to another FSM works well if each output is unique to just one state. Now all we have to do is recognize that the state was entered instead of examining an output from the other state machine.

If the output is set in multiple states, then referencing the multiple states from the other state machine can actually become more verbose than if a simple output signal is tested that is set in all of the other states.

This is FSM dependent, so now we are going to coerce multiple coding styles based on an engineering judgement of how many states set specific outputs.

A useful example is needed!

A More Concise Syntax If Possible

The goal of a more concise syntax is a worthy goal.

The problem with the proposed syntax enhancements is that only synchronous resets benefit from the "state (...) iff posedge clk;" syntax. Change the reset from synchronous to asynchronous and you now have to add in the old Verilog code.

In Verilog, synchronous and asynchronous FSMs use just three almost identical lines of Verilog to code the state register:

```
always @(posedge clk <or negedge rst_n>)
  if (!rst_n) state <= IDLE;
  else        state <= next;
```

That's it! add the negedge rst_n to the sensitivity list and synchronous reset becomes asynchronous. That's it! This is not terribly verbose! If there is an elegant way to do this even more concisely with a new syntax, I am not opposed to reducing three more lines of code, but these three lines of code do not impact my RTL coding schedule.

Enumerated types will remove the requirement to make separate state and next declarations. Enumerated types will also make statename display easier (most of the time) which again reduces diagnostic coding efforts. Enumerated types are one step in the direction of a more concise syntax.

Simple Transition To An RTL Style With State Encodings

As a problem, architectural exploration pales to the problem of timing closure in a high speed design. I have yet to find a synthesis tool that can take a general purpose coding style, push the onehot button, and beat the size and speed of a Verilog design coded with the case 1'b1 syntax.

I know of no synthesis tool that automatically infers the high-speed registered output-encoded style.

Synplicity has a "safe FSM" setting that is not "safe" (although it is generally fast because it is a modified onehot style with one state assigned the state encoding of all zeros - I referred to this style in one of my conference papers as a "onehot with 0-Idle" style).

Any satellite circling the Earth with a Synplicity "safe" coding style might have to be periodically reset to re-allign the rest of the hardware to the state machine. Any medical-implant company that uses the Synplicity "safe" coding style could kill their patients! Synplicity has addressed the minor problem of a onehot state machine losing a bit and therefore going to a known state of all 0's. Unless the all 0's state is an error state that resets the rest of the logic to the same known state, the FSM thinks it is in one state while the rest of the hardware might think it is in another state with dire consequences.

If a onehot FSM turns on an extra bit, then extra hardware activity will occur if or until the FSM is restored to a onehot state. This type of state machine would require significant extra logic to test for multiple bits being set, effectively killing the objective of using a onehot FSM to reduce combinational logic circuitry and increase overall speed (especially in an FPGA design).

A truly "safe" FSM coding style would require encodings that would employ a SEC/DED (Single Error Correct / Double Error Detect) encoding to fix the introduction of single bit changes and detect double bit changes and then go to an error recovery-reset state.

Onehot (or onehot with 0-Idle) FSMs should not be used for any design where there is danger of state bits changing value. As language designers, we can sit back and blame the FSM synthesis tools but that is of little comfort to the guy whose pacemaker malfunctioned due to a poor synthesized FSM implementation.

Simple Transition To Other Important FSM Styles

This is lacking in the enhanced FSM syntax proposal.

But quite frankly, this is also a problem in Verilog. When transitioning from a binary-encoded FSM to the efficient onehot style, there are about 10 coding changes that have to be made to transition the standard binary-encoded style to a onehot style. It would be nice if it were as easy as changing "enum" to "enum_onehot" and "case(state)" to case_onehot(state)."

Ability To Efficiently Control Output Assignments

I believe the value of this capability was entirely overlooked in the enhanced FSM syntax proposal.

Sometimes FSM outputs are also ASIC outputs. Sometimes FSM outputs are control-logic outputs. And sometimes outputs must be registered to avoid output glitching. My Boston SNUG-2000 paper addressed reasons for and techniques to register FSM outputs.