JHLD

State Machine Design Techniques for Verilog and VHDL

Steve Golson, Trilobyte Systems

Designing a synchronous finite state machine (FSM) is a common task for a digital logic engineer. This paper discusses a variety of issues regarding FSM design using Synopsys Design Compiler™. Verilog™ and VHDL coding styles are presented, and different methodologies are compared using real-world examples.

A finite state machine has the general structure shown in Figure 1. The current state of the machine is stored in the *state memory*, a set of *n* flip-flops clocked by a single clock signal (hence "synchronous" state machine). The *state vector* (also *current state*, or just *state*) is the value currently stored by the state memory. The *next state* of the machine is a function of the state vector and the inputs. *Mealy outputs* are a function of the state vector and the inputs, while *Moore outputs* are a function of the state vector only.

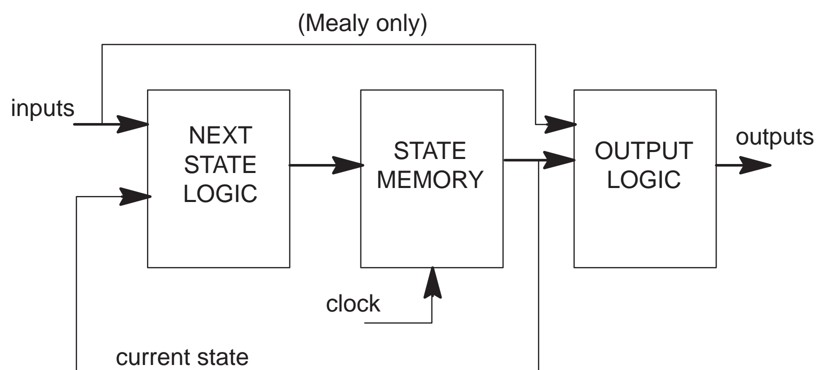

Figure 1. State Machine Structure

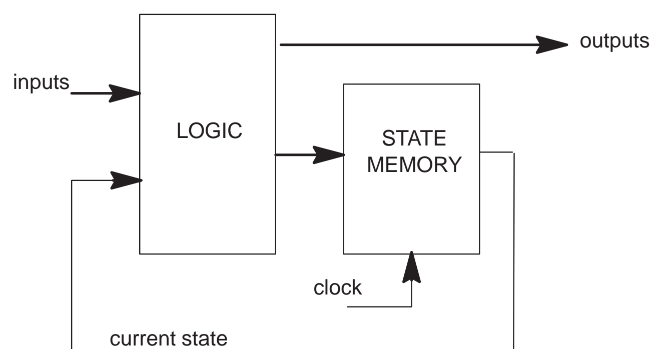

Figure 2. Alternative State Machine Structure

Another way of organizing a state machine uses only one logic block, as shown in Figure 2.

Basic HDL Coding

The logic in a state machine is described using a `case` statement or the equivalent (e.g., `if-else`). All possible combinations of current state and inputs are enumerated, and the appropriate values are specified for next state and the outputs.

A state machine may be coded as in Figure 1 using two separate `case` statements, or, following Figure 2, using only one. A single `case` statement may be preferred for Mealy machines where the outputs depend on the state transition rather than just the current state.

The listings in the Appendix show examples of both techniques. prep3 uses a single `case` whereas prep4 is coded with a separate logic block that generates the outputs.

Here are a few general rules to follow:

- Only one state machine per module
- Keep extraneous logic at a minimum (for example, try not to put other code in the same module as the FSM—this is especially important if you use `extract`)
- Instantiate state flip-flops separately from logic

State Assignment

Usually the most important decision to make when designing a state machine is what state encoding to use. A poor choice of codes results in a state machine that uses too much logic, or is too slow, or both.

Many tools and techniques have been developed for choosing an “optimal” state assignment. Typically such approaches use the minimum number of state bits (Ref. 1) or assume a two-level logic implementation such as a PLA (Ref. 2). Only recently has work been done on the multi-level logic synthesis typical of gate array design (Ref. 3).

Highly Encoded State Assignment

A highly encoded state assignment will use fewer flip-flops for the state vector; however, additional logic will be required simply to encode and decode the state.

One-Hot Encoding

In *one-hot encoding*, only one bit of the state vector is asserted for any given state. All other state bits are zero. So if there are *n* states, then *n* state flip-flops are required. State decode is simplified, since the state bits themselves can be used directly to indicate whether the machine is in a particular state. No additional logic is required.

History of One-Hot Encoding

The first discussion of one-hot state machines was given by Huffman (Refs. 4 and 5). He analyzed asynchronous state machines implemented with electromechanical relays, and introduced a “one-relay-per-row” realization of his flow tables.

Why Use One-Hot?

There are numerous advantages to using the one-hot design methodology:

- One-hot state machines are typically faster. Speed is independent of the number of states, and instead depends only on the number of transitions into a particular state. A highly encoded machine may slow dramatically as more states are added.
- You don’t have to worry about finding an “optimal” state encoding. This is particularly beneficial as the machine design is modified, for what is “optimal” for one design may no longer be best if you add a few states and change some others. One-hot is equally “optimal” for all machines.
- One-hot machines are easy to design. HDL code can be written directly from the state diagram without coding a state table.
- Modifications are straightforward. Adding and deleting states, or changing excitation equations, can be implemented easily without affecting the rest of the machine.
- Easily synthesized from VHDL or Verilog.

- There is typically not much area penalty over highly encoded machines.
- Critical paths are easy to find using static timing analysis.
- It is easy to debug. Bogus state transitions are obvious, and current state display is trivial.

Almost One-Hot Encoding

If a machine has two groups of states with almost identical functionality (e.g., for handling read and write access to a device), an "almost one-hot" encoding may be used where a single flag or state bit is used to indicate which of the two state groups the FSM is currently in. The remainder of the state bits are encoded one-hot. Thus to fully decode a given state we must look at two state bits. This scheme has most of the benefits of a pure one-hot machine but with less logic.

Although the flag bit is technically part of the state vector, it may be useful to consider the flag flip-flop output pin as just another input to the machine (and likewise the flag flip-flop input pin is a machine output). In the above example the flag might have a name like `RW`.

Another "almost one-hot" encoding uses the all-zeros or "no-hot" encoding for the initial state. This allows for easy machine reset since all flip-flops go to zero. This may be especially useful when a synchronous reset is needed.

Error Recovery and Illegal States

It is sometimes argued that state machines should have the minimum number of state flip-flops (i.e., a highly encoded state assignment) because this minimizes the number of illegal states. The hope is that if the machine malfunctions and makes an illegal transition, at least the erroneous destination will be a legal state, and the machine can recover.

This often turns out not to be the case. Just because the machine ends up in a "legal" state doesn't mean that it can recover from the error. Consider a WAIT state that the machine loops in until a particular signal is received. If the WAIT state is entered accidentally then the machine probably hangs.

Perhaps to facilitate error recovery the *maximum* number of state flip-flops should be used (i.e., one-hot). If a bad transition is made, then it will almost certainly put the machine in an illegal state (since the legal states are a small fraction of all possible state vector values). This illegal state can be detected by external logic, which may then take appropriate action (e.g., reset the FSM).

Coding State Transitions

State transitions are coded using a `case` structure to specify the next state values.

Highly Encoded Machine

For a highly encoded machine the `case` statement uses the state vector as the expression. In Verilog the `case` items are typically parameters that specify the state encoding for each state:

```
case (state)
    // synopsys parallel_case full_case

    START:
      if (in == 8'h3c)
          next_state = SA ;
      else
          next_state = START ;

    SB:
      if (in == 8'haa)
          next_state = SE ;
      else
          next_state = SF ;

    SC:
      next_state = SD ;
```

See Listing 1 and Listing 3 for more examples. Using `parameter` and the `full_case` directive in Verilog, we can specify arbitrary state encodings and still have efficient logic.

In VHDL the state encodings are declared as an enumerated `type` (see Listing 5). The actual numeric value of the enumerated elements is predefined by the VHDL language: the first element is 0, then 1, 2, etc. It is difficult to define arbitrary encodings in the VHDL language.[1]

To remedy this problem Synopsys has provided the attribute `enum_encoding`, which allows you to specify numeric code values for the enumerated types. Unfortunately, not all VHDL simulators will implement this vendor-specific extension, which means your behavioral and gate simulations will use different encodings.

1. This still isn't fixed in VHDL '93 (Ref. 6).

One-Hot Machine

For one-hot encoding you need only look at one bit to determine if you are in a particular state. Thus the statement in Verilog looks as follows (see Listing 2 for more):

```
next_state = 8'b0 ;

case (1'b1)
    // synopsys parallel_case full_case

    state[START]:
        if (in == 8'h3c)
            next_state[SA] = 1'b1 ;
        else
            next_state[START] = 1'b1 ;

    state[SB]:
        if (in == 8'haa)
            next_state[SE] = 1'b1 ;
        else begin
            next_state[SF] = 1'b1 ;

    state[SC]:
        next_state[SD] = 1'b1 ;
```

The statement looks at each state bit in turn until it finds the one that is set. Then one bit of `next_state` is set corresponding to the appropriate state transition. The remaining bits of next_state are all set to zero by the default statement

```
  next_state = 8'b0 ;
```

Note the use of `parallel_case` and `full_case` directives for maximum efficiency. The `default` statement should *not* be used during synthesis. However `default` can be useful during behavioral simulation, so use compiler directives to prevent Design Compiler from seeing it:

```
// synopsys translate_off
default: $display("He's dead, Jim.") ;
// synopsys translate_on
```

For VHDL we use a sequence of `if` statements (see Listing 6 for more):

```
next_state <= state_vec'(others=>'0');

if state(1) = '1' then
  if (Iin(1) and Iin(0)) = '1' then
    next_state(0) <= '1';
  else
    next_state(3) <= '1';
  end if ;
end if ;
```

```
if state(2) = '1' then
  next_state(3) <= '1' ;
end if;
```

As before, all the bits of `next_state` are set to zero by the default assignment, and then one bit is set to 1, indicating the state transition.

For both the Verilog and VHDL one-hot machines, the behavioral simulation will exactly agree with the post-synthesis gate-level simulation.

Almost One-Hot Machine

The only difference from the pure one-hot machine is that you may look at more than one state bit to determine the current state:

```
case (1'b1)
    // synopsys parallel_case full_case

    state[START] && state[RW]:
        if (in == 8'h3c)
            next_state[SA] = 1'b1 ;
        else
            next_state[START] = 1'b1 ;
```

Outputs

Outputs are coded in a manner similar to the next state value. A `case` statement (or the equivalent) is used, and the output is assigned the appropriate value depending on the particular state transition or state value.

If the output is a don't care for some conditions, then it should be driven unknown (`x`). Design Compiler will use this don't care information when optimizing the logic.

Assigning the output to a default value prior to the `case` statement will ensure that the output is specified for all possible state and input combinations. This will avoid unexpected latch inference on the output. Also, the code is simplified by specifying a default value that may be overridden only when necessary. The default value may be 1, 0, or x.

It is best to have a default of 0 and occasionally set it to 1 rather than the reverse (even if this requires an external inverter). Consider an output that is 1 in a single state, and 0 otherwise. Design Compiler will make the output equal to the one-hot state bit for that state. Now consider an output that is 0 in only one state, and 1 otherwise. The output will be driven by an OR of all the other state bits! Using `set_flatten -phase true` will not help.

For a one-hot machine you can use the state bits directly to create outputs that are active in those states:

```
myout = state[IDLE] || state[FOO] ;
```

Sometimes it is easier to specify an output value as a function of the next state rather than of the current state.

Registered Outputs

Outputs can be registered. A simple D flip-flop may be used, but a JK functionality can be implemented as well. The output of the flip-flop is fed back as an input to the machine. The default next output value is the current flip-flop output:

```
next_myout = myout ; /* default */
```

With no further assignment the value will hold, or we can set, clear, and toggle:

```
next_myout = 1'b1 ; /* set */
next_myout = 1'b0 ; /* clear */
next_myout = !myout ; /* toggle */
```

This JK-type output is especially useful for pseudo-state flag bits (see the previous section titled “Almost One-Hot Encoding”).

Inputs

Asynchronous Inputs

Sometimes a state machine will have an input that may change asynchronously with respect to the clock. Such an input *must* be synchronized, and there must be *one and only one* synchronizer flip-flop.

The easiest way to accomplish this is to have the sync flip-flop external to the state machine module, and place a large[2] `set_input_delay` on that input to allow time for the sync flip-flop to settle.

If the sync flip-flop is included in the same module as the FSM, then you can place an input delay on the internal flip-flop output pin. Unfortunately this requires the flip-flop to be mapped prior to compiling the rest of the machine.

Rather than hand-instantiating the flip-flop we can use register inference as usual and simply map that one flip-flop before compiling. The following script will map the flip-flop:

2. “Large” means a large fraction of your clock period. For extra credit, ask your ASIC vendor about the metastability characteristics of their flip-flops. Try not to laugh.

```
/* get the name of the unmapped flip-flop */
theflop = signal_to_be_synced + "_reg"
/* group it into a design by itself */
group -cell flip-flop -design temp find(cell,theflop)
/* remember where you are */
top = current_design
/* push into the new design */
current_design = temp
/* set_register_type if necessary */
/* map the flip-flop */
compile -map_effort low -no_design_rule
/* pop back up */
current_design = top
/* ungroup the flip-flop */
ungroup -simple_names find(cell,flop)
/* clean up */
remove_design temp
remove_variable top
/* now set the internal delay */
set_input_delay 10 -clock clk find(pin,theflop/Q*)
/* now you can compile the fsm */
```

The will put an implicit `dont_touch` on the sync flip-flop.

If your ASIC vendor has a "metastable resistant" flip-flop then use `set_register_type` to specify it.

Unknown Inputs

A related problem occurs when an input is valid at certain well-defined times, and is otherwise unknown (and possibly asynchronous). Even if your code is written to only use this signal when you know it to be stable, Design Compiler may create optimized logic whereby changes in this input may cause glitches even when you are not "looking" at it.[3]

The only way to prevent this problem is to gate the input signal with an enable. This enable signal is usually a simple decode of the state vector; thus the gate output is non-zero only when the enable is true and will never be unknown. The gate output is used in place of the input signal in your FSM.

To implement this gating function, an AND gate (or other suitable logic) must be hand-instantiated and protected with `dont_touch` during compile.

3. An example would be where the signal is used as a mux select. If the data inputs to the mux are equal then Design Compiler assumes the mux output will have the same value regardless of the select value. Unfortunately, glitches on the select may nevertheless cause glitches on the mux output.

Rather than instantiating a specific gate from your vendor library, the gate can be selected from the Synopsys GTECH library. This keeps your HDL code vendor-independent. In Verilog this is done as follows:

```
GTECH_AND2 myand (
  .Z(signal_qualified),
  .A(signal_in), .B(enable)) ;
```

and for VHDL

```
myand : GTECH_AND2 port map(
   Z => signal_qualified,
   A => signal_in, B => enable) ;
```

Your compile script should contain

```
set_map_only find(cell,myand) ;
```

which prevents logic-level optimization during the compile. Design Compiler will attempt to map the gate exactly in the target library.

Sometimes this technique will create redundant logic in your module. This can cause a problem when generating test vectors, because some nodes may not be testable.

Verilog users may be tempted to use gate primitives:

```
and myand (signal_qualified, signal_in, enable) ;
```

making the reasonable assumption that this will initially map to a GTECH_AND2 as the Verilog is read in. Then `set_map_only` could be used as above. Unfortunately this does not work; gate primitives do not always map to a GTECH cell. Perhaps a future Synopsys enhancement will allow this.

In order to support behavioral simulation of your HDL, a behavioral description of the GTECH gates must be provided. Synopsys supplies such a library only for VHDL users. One hopes that a similar Verilog library will be provided in a future release.

FSM Extract

Design Compiler directly supports finite-state machines using the `extract` command. `extract` gives you the ability to change your state encodings during compile, thus allowing you to experiment with different FSM implementations (Ref. 7).

To use `extract` you must tell Design Compiler where your state vector is, and also any state names and encodings you may have. The easiest way to do this is with `attribute state_vector` in VHDL and via the `enum code` and `state_vector` synthetic comments in Verilog. (See Listing 1, Listing 3, and Listing 5 for examples.)

`extract` puts your design into a two-level PLA format before doing any optimizations and transformations. So if your design cannot be flattened, you cannot use `extract`.

Synopsys provides the `group -fsm` command to isolate your state machine from any other logic in its module. Unfortunately, the newly-created ports have Synopsys internal names like `n1234`. The resulting state table is difficult to understand. Therefore to efficiently use `extract` you should avoid `group -fsm.` This means you can have *no* extraneous logic in your module.

Your design must be mapped to gates before you can use `extract`. Synopsys suggests that you run `compile` on your design after reading in the HDL and before applying any constraints:

```
compile -map_effort low -no_design_rule
```

This isn't really necessary since most of your design will already be implemented in generic logic after reading the HDL, and `extract` can handle that fine. What you really must do is

```
replace_synthetic
```

to map all synthetic library elements into generic logic, followed by

```
ungroup -all -flatten
```

to get rid of any hierarchy. This will be considerably faster than using `compile`.

After using `extract`, always do `check_design` to get a report on any bad state transitions.

Advantages

You can get very fast results using `extract` with `set_fsm_coding_style one_hot`.

FSM design errors can be uncovered by inspecting the extracted state table.

Disadvantages

The world isn't a PLA, but `extract` treats your design like one.

Unless you are truly area constrained, the only interesting coding style that `extract` supports is one-hot. You might as well code for one-hot to begin with (see "One-Hot Machine" in the Coding State Transitions section of this paper).

You can be happily using `extract`, but one day modify your HDL source and then discover that you can no longer flatten the design. This precludes any further use of `extract`.

Compile scripts are more verbose and complicated.

Timing Constraints

When applying timing constraints, you should use the real clock only for the state flip-flops. Virtual clocks are then used to specify the input and output delays:

```
clk_period = 10
clk_rise   = 0
clk_fall   = clk_period / 2.0
/* create the clocks */
create_clock find(port,clk) \
   -period clk_period \
   -waveform {clk_rise clk_fall}
create_clock -name inclk \
   -period clk_period \
   -waveform {clk_rise clk_fall}
create_clock -name outclk \
   -period clk_period \
   -waveform {clk_rise clk_fall}
/* set the constraints */
set_input_delay  clk_fall \
   -clock inclk  find(port,in)
set_output_delay clk_fall \
   -clock outclk find(port,out)
```

This allows a clock skew to be applied to the state flip-flops without affecting the input and output timing (which may be relative to an off-chip clock, for example).

If you have any Mealy outputs, you generally need to specify them as a multicycle path using

```
set_multicycle_path -setup 2 \
  -from all_inputs() \
  -to   all_outputs()
set_multicycle_path -hold 1 \
  -from all_inputs() \
  -to   all_outputs()
```

Sometimes it is useful to group paths into four categories: input to state flip-flop, state flip-flop to output, input to output (Mealy path), and state flip-flop to state flip-flop. With the paths in different groups, they can be given different cost function weights during compile.

If you use separate clocks as suggested above, you might be tempted to try this:

```
/* put all paths in default group */
group_path -default -to \
  { find(clock) find(port) find(pin) } \
```

```
  > /dev/null
/* now arrange them */
group_path -name theins \
  -from find(clock,inclk) \
  -to   find(clock,clk)
group_path -name theouts \
  -from find(clock,clk) \
  -to   find(clock,outclk)
group_path -name thru \
  -from find(clock,inclk) \
  -to   find(clock,outclk)
group_path -name flop \
  -from find(clock,clk) \
  -to   find(clock,clk)
```

Unfortunately, this doesn't work! It seems that whenever you specify a clock as a startpoint or endpoint of a path, *all* paths with that clock are affected. You end up with the same path in more than one group.[4] So instead of using clocks, we can specify pins:

```
group_path -name theins \
  -from all_inputs() \
  -to   all_registers(-data)
group_path -name theouts \
  -from all_registers(-clock_pins) \
  -to   all_outputs()
group_path -name thru \
  -from all_inputs() \
  -to   all_outputs()
group_path -name flop \
  -from all_registers(-clock_pins)
  -to   all_registers(-data)
```

This works fine. You do get the paths where you want them.[5]

Regardless of the path groupings, we can specify timing reports that give us the information we want:

```
report_timing \
  -from all_inputs() \
  -to   all_registers(-data)
report_timing \
  -from all_registers(-clock_pins) \
  -to   all_outputs()
report_timing \
```

4. Hopefully this will be fixed in a future version.

5. This works even if you specify these groups on unmapped logic. As the flip-flops are mapped during the compile, Design Compiler automatically changes the flip-flop pin names used in the path groupings.

```
  -from all_inputs() \
  -to   all_outputs()
report_timing \
  -from all_registers(-clock_pins)   \
  -to   all_registers(-data)
```

One-Hot Timing Reports

A further advantage of one-hot state assignment is that critical path timing reports can be directly related to the state diagram. Consider the following timing report:

```
Point                              Path
-------------------------------------
clock clk (rise edge)              0.00
clock network delay (ideal)        0.00
state_reg[0]/CP (FD2)              0.00 r
state_reg[0]/QN (FD2)              5.25 f
U481/Z (NR2)                      11.39 r
U505/Z (IV)                       12.21 f
U474/Z (NR2)                      14.14 r
U437/Z (EO1)                      16.23 f
U469/Z (AO3)                      18.11 r
U463/Z (EO1)                      20.21 f
U480/Z (AO7)                      22.08 r
U440/Z (AO2)                      23.24 f
U495/Z (ND2)                      24.65 r
state_reg[1]/D (FD2)              24.65 r
data arrival time                 24.65
```

If this is a highly encoded machine, then it is very difficult to determine which state transition this path corresponds to. Worse, this may actually be a false path.

In contrast, if this is a one-hot machine, then this transition must start in state[0] because flip-flop `state_reg[0]` is set (pin `state_reg[0]/QN` falling), and must end in state[1] because flip-flop `state_reg[1]` is being set (`state_reg[1]/D` is rising).

Now that the particular transition has been identified, it may be recoded to speed up the path.

When using `extract`, the state flip-flops for one-hot machines are given the names of the corresponding states. This makes path analysis particularly straightforward.

Synthesis Strategies

If you are using `extract`, there aren't many useful compile options. Flattening is ignored, so all you can do is turn structuring on and off.

For a one-hot machine, flattening may provide some benefit. As usual, the improvements vary widely depending on your particular application (Ref. 8)

One interesting technique to experiment with is pipeline retiming with the `balance_registers` command. This is intended primarily for pipelined datapaths, but it does work with a single bank of flip-flops, as in a state machine. The drawbacks are:

- The flip-flops cannot have asynchronous resets
- Results may be affected by

  ```
  compile_preserve_sync_resets = "true"
  ```

- State encodings change in unpredictable ways

Compile Results

Four sample state machines were used to compare and illustrate the techniques outlined in this paper. The results are shown in Table 1.

Table 1. Compile results for Sample State Machines

	Compile for Maximum Speed			Compile for Minimum Area		
	Slack (ns)	Area	Run Time (minutes)	Slack (ns)	Area	Run Time (minutes)
prep3 8 states, 12 transitions, 8 inputs, 8 outputs						
coded for extract						
binary	–5.41	228	< 2	–8.84	166	< 2
one_hot	–5.19	227		–11.59	196	
auto_3	–5.95	214		–7.29	164	
auto_4	–5.01	234		–8.55	**159**	

	Compile for Maximum Speed			Compile for Minimum Area		
	Slack (ns)	Area	Run Time (minutes)	Slack (ns)	Area	Run Time (minutes)
prep3 (continued)						
8 states, 12 transitions, 8 inputs, 8 outputs						
auto_5	–5.43	229		–8.55	**159**	
no extract (binary)	–4.56	216		–12.22	169	
coded for one_hot						
structure	**–3.86**	221	< 2	–12.23	194	< 2
flatten	–4.35	341		–9.84	258	
flatten & structure	–4.38	239		–11.93	193	
prep4						
16 states, 40 transitions, 8 inputs, 8 outputs						
coded for extract						
binary	–7.33	298	< 7	–15.50	195	< 7
one_hot	**–4.34**	348		–10.96	255	
auto_4	–6.42	283		–13.16	190	
auto_5	–6.58	285		–14.63	**184**	
auto_6	–8.30	279		–12.81	191	
no extract (binary)	–8.87	299		–17.03	204	
coded for one_hot						
structure	–5.27	335	< 5	–10.30	259	< 5
flatten	–5.90	475		–13.79	370	
flatten & structure	–5.04	342		–10.94	260	

	Compile for Maximum Speed			Compile for Minimum Area		
	Slack (ns)	Area	Run Time (minutes)	Slack (ns)	Area	Run Time (minutes)
sm40 40 states, 80 transitions, 63 inputs, 61 outputs						
coded for extract						
binary	–5.19	931	100	–21.04	661	81
one_hot	**–2.82**	912	27	–16.48	737	21
auto_6	–5.39	885	87	–18.60	668	61
auto_7	–5.71	979	76	–29.54	683	51
auto_8	–5.63	933	69	–18.37	682	51
no extract (binary)	–7.72	889	35	–31.73	**604**	6
coded for one_hot						
structure	–4.78	882	12	–16.59	761	7
flatten	–7.38	3026	202	–49.68	2141	73
flatten & structure	–4.41	905	28	–16.86	753	22
sm70 69 states, 116 transitions, 27 inputs, 16 outputs						
coded for extract						
binary	–7.98	1030	17	–17.66	857	5
one_hot	**–3.12**	1200	10	–8.28	1121	5
auto_7	–5.51	996	15	–14.67	849	6
auto_8	–5.69	975	13	–11.33	**817**	6
auto_9	–4.55	1018	19	–11.84	827	5

	Compile for Maximum Speed			Compile for Minimum Area		
	Slack (ns)	Area	Run Time (minutes)	Slack (ns)	Area	Run Time (minutes)
sm70 (continued)						
69 states, 116 transitions, 27 inputs, 16 outputs						
no extract (binary)	–20.70	1249	49	–60.43	843	8
coded for one_hot						
structure	–7.92	1339	20	–35.77	1096	12
flatten	–6.51	1852	36	–26.20	1548	23
flatten & structure	–7.39	1326	29	–33.96	1104	18

Two Verilog versions of each machine were created: one highly encoded for use with `extract`, and the other one-hot encoded as described in this paper in the One-Hot Machine section under Coding State Transitions.

The highly encoded version was `extract`ed and compiled with a variety of state encodings: binary, one_hot, and auto with varying bit widths. In addition, the highly encoded version was compiled without using extract (thus using the binary encoding specified in the source).

The one-hot version was compiled using a selection of structuring and flattening options.

The Verilog listings for the prep3 and prep4 examples are given in the Appendix. Also listed are VHDL versions of the prep4 machine. These sources were also compiled, and the results were similar to the Verilog runs shown in the table.

For the max speed runs the prep3 and prep4 examples had a 10-ns clock, while the sm40 and sm70 examples used a 20-ns clock. The min area runs used a max area constraint of 0. The target library was the Synopsys class.db library.

All runs used Synopsys Design Compiler version 3.0b-12954 running on a SPARCstation 2™ under SunOS 4.1.3.

Hints, Tips, Tricks, and Mysteries

- `group -fsm` sometimes gives you a broken state machine. This doesn't happen if you code the FSM all alone in its module.
- `reduce_fsm` sometimes takes a long time, much longer than the `extract` itself.
- For Verilog users of `extract`, you have to define the `enum code` parameters *before* declaring the `reg` that uses it, and also before the `state_vector` declaration. In the `reg` declaration, `enum code` must be between `reg` and the name:

  ```
  reg [2:0] // synopsys enum code
     state, next_state ;
  ```

  or

  ```
  reg [2:0] /*synopsys enum code*/ state;
  ```

- A `set_false_path` from the asynchronous reset port of an extracted FSM will prevent the state flip-flops from being mapped during compile. Apparently an implicit `dont_touch` is placed on the flip-flops. This is no doubt a bug.
- When using auto state encoding, only the unencoded states are given new values. If you want to replace all current encodings then do this:

  ```
  set_fsm_encoding {}
  set_fsm_encoding_style auto
  ```

- When using `extract` with auto encoding, only the minimum number of state flip-flops are used. If you have specified a larger number, you may get a warning about "truncating state vector." Do a `report_fsm` to be sure.
- The encoding picked by `extract` does not depend on the applied constraints.
- Coding the same machine in Verilog and VHDL and using `extract` gives identical state tables, but the compile results are slightly different.
- If your HDL source specifies an output as don't care, this will not be reflected in the state table, because prior to the `extract` you have to map into gates and that collapses the don't care.
- Always do an `ungroup` before `extract`.
- `set_fsm_encoding` can't handle more than 31 bits if you are using the ^H format. Instead use ^B, which works fine.
- Remove any unused inputs from your module before doing an `extract`. Otherwise they will be included in the state table and it slows down the compile.

- Verilog users should infer flip-flops using non-blocking assignments with non-zero intra-assignment delays:

  ```
  always @ (posedge clk)
    myout <= #1 next_myout ;
  ```

  This is not necessary for Synopsys but should make your flip-flops simulate correctly in all Verilog simulators (e.g. Verilog-XL and VCS).

  When specifying the state flip-flop reset behavior in a one-hot machine (see Listing 2 and Listing 4) there are two assignments made to the state vector: the first clears all state bits to zero, and the second sets one particular state bit to one (indicating the reset state). The second assignment partly overrides the first assignment, so these two assignments *must* be executed in that order, and therefore *must* have different delay values:

  ```
  // build the state flip-flops
  always @ (posedge clk or negedge rst)
      begin
      if (!rst) begin
          state        <= #1 8'b0 ;
          state[START] <= #2 1'b1 ;
          end
      else
          state <= #1 next_state ;
      end
  ```

- Avoid using synchronous resets; it will probably add many additional transitions to your machine. For example, the sm40 machine adds 26 transitions for a total of 106, and the sm70 machine adds 60 for a total of 176.

  If you must use synchronous resets, then they should be implemented as part of the flip-flop inference and not in the state machine description itself. Here is a Verilog example modified from Listing 3:

  ```
    // build the state flip-flops
    always @ (posedge clk)
      begin
      if (!rst)
        state <= #1 S0 ;
      else
        state <= #1 next_state ;
      end
  ```

  and a VHDL example modified from Listing 5:

  ```
    -- build the state flip-flops
    process (clk)
    begin
  ```

```
      if clk='1' and clk'event then
        if rst='0' then
          state <= S0 ;
        else
          state <= next_state ;
        end if ;
      end if ;
    end process ;
```

- When using `extract` with auto encoding, only the minimum number of state flip-flops are used. Nevertheless, specifying more than the minimum will affect the state assignment and thus the compile results.
- Why is `extract` better at flattening a design than `compile` using `set_flatten`?

Acknowledgments

Thanks to my clients for providing access to design tools and for allowing the use of examples sm40 and sm70. Thanks to John F. Wakerly for finding the Huffman references.

References

1. James R. Story, Harold J. Harrison, Erwin A. Reinhard, "Optimum State Assignment for Synchronous Sequential Circuits," *IEEE Trans. Computers*, vol. C-21, no. 12, pp. 1365-1373, December 1972.

2. Giovanni De Micheli, Robert K. Brayton, Alberto Sangiovanni-Vincentelli, "Optimal State Assignment for Finite State Machines," *IEEE Trans. Computer-Aided Design*, vol. CAD-4, no. 3, pp. 269-285, July 1985.

3. Pranav Ashar, Srinivas Devadas, A. Richard Newton, *Sequential Logic Synthesis*, Kluwer Academic Publishers, 1992.

4. D. A. Huffman, "The Synthesis of Sequential Switching Circuits," *J. Franklin Institute*, vol. 257, no. 3, pp. 161-190, March 1954.

5. D. A. Huffman, "The Synthesis of Sequential Switching Circuits," *J. Franklin Institute*, vol. 257, no. 4, pp. 275-303, April 1954.

6. Jean-Michel Bergé, Alain Fonkoua, Serge Maginot, Jacques Rouillard, *VHDL '92*, Kluwer Academic Publishers, 1993.

7. Synopsys, *Finite State Machines Application Note*, Version 3.0, February 1993.

8. Synopsys, *Flattening and Structuring: A Look at Optimization Strategies Application Note*, Version 3.0, February 1993.

9. Programmable Electronics Performance Corporation, *Benchmark Suite #1*, Version 1.2, March 28, 1993.

Related Reading

Steve Golson, “One-hot state machine design for FPGAs,” *Proc. 3rd Annual PLD Design Conference & Exhibit*, p. 1.1.3.B, March 1993.

John F. Wakerly, *Digital Design: Principles and Practices*, Prentice-Hall, 1990.

Appendix

The following example state machines are taken from the PREP benchmark suite (Ref. 9).

prep3

prep3 is a Mealy machine with eight states and 12 transitions. It has eight inputs and eight registered outputs. The state diagram is shown in Figure 3.

Listing 1 is a Verilog implementation for use with Synopsys FSM `extract`.

Listing 2 is a Verilog implementation that is one-hot coded.

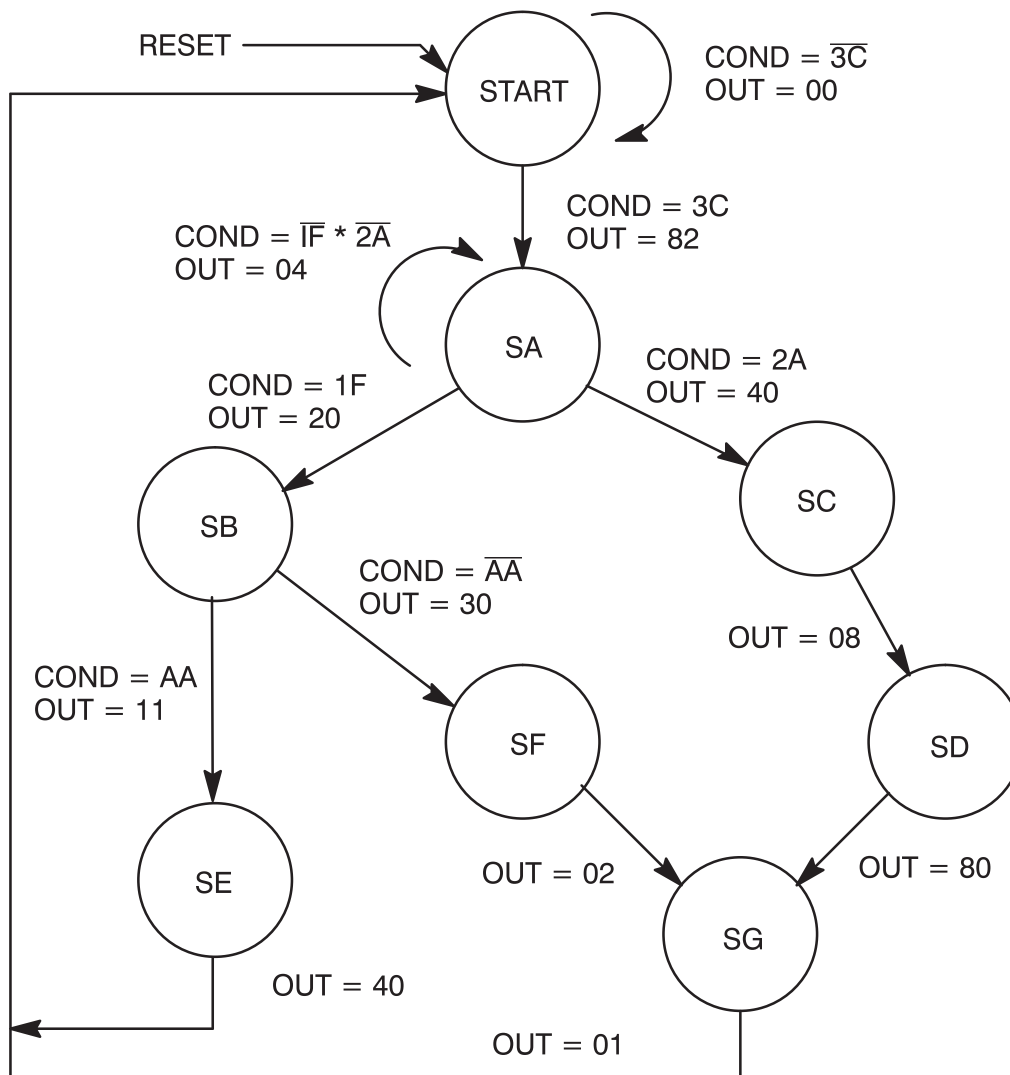

Figure 3. prep 3 State Transition Diagram

Listing 1—prep3.v

```
/*
** prep3.v
**
** prep benchmark 3 -- small state machine
** benchmark suite #1 -- version 1.2 -- March 28, 1993
** Programmable Electronics Performance Corporation
**
** binary state assignment -- highly encoded
*/

module prep3 (clk, rst, in, out) ;

input clk, rst ;
input [7:0] in ;
output [7:0] out ;

parameter [2:0] // synopsys enum code
    START = 3'd0 ,
    SA    = 3'd1 ,
    SB    = 3'd2 ,
    SC    = 3'd3 ,
    SD    = 3'd4 ,
    SE    = 3'd5 ,
    SF    = 3'd6 ,
    SG    = 3'd7 ;

// synopsys state_vector state
reg [2:0] // synopsys enum code
    state, next_state ;
reg [7:0] out, next_out ;

always @ (in or state) begin

    // default values

    next_state = START ;
    next_out = 8'bx ;

    // state machine

    case (state) // synopsys parallel_case full_case

    START:
        if (in == 8'h3c) begin
            next_state = SA ;
            next_out = 8'h82 ;
            end
        else begin
```

```
        next_state = START ;
        next_out = 8'h00 ;
        end

SA:
    case (in) // synopsys parallel_case full_case
        8'h2a:
            begin
            next_state = SC ;
            next_out = 8'h40 ;
            end
        8'h1f:
            begin
            next_state = SB ;
            next_out = 8'h20 ;
            end
        default:
            begin
            next_state = SA ;
            next_out = 8'h04 ;
            end
        endcase

SB:
    if (in == 8'haa) begin
        next_state = SE ;
        next_out = 8'h11 ;
        end
    else begin
        next_state = SF ;
        next_out = 8'h30 ;
        end

SC:
    begin
    next_state = SD ;
    next_out = 8'h08 ;
    end

SD:
    begin
    next_state = SG ;
    next_out = 8'h80 ;
    end

SE:
    begin
    next_state = START ;
    next_out = 8'h40 ;
```

```
        end

    SF:
        begin
        next_state = SG ;
        next_out = 8'h02 ;
        end

    SG:
        begin
        next_state = START ;
        next_out = 8'h01 ;
        end

    endcase

    end

// build the state flip-flops
always @ (posedge clk or negedge rst)
    begin
    if (!rst)   state <= #1 START ;
    else        state <= #1 next_state ;
    end

// build the output flip-flops
always @ (posedge clk or negedge rst)
    begin
    if (!rst)   out <= #1 8'b0 ;
    else        out <= #1 next_out ;
    end

endmodule
```

Listing 2—prep3_onehot.v

```
/*
** prep3_onehot.v
**
** prep benchmark 3 -- small state machine
** benchmark suite #1 -- version 1.2 -- March 28, 1993
** Programmable Electronics Performance Corporation
**
** one-hot state assignment
*/

module prep3 (clk, rst, in, out) ;

input clk, rst ;
input [7:0] in ;
output [7:0] out ;
```

```
parameter [2:0]
    START = 0 ,
    SA    = 1 ,
    SB    = 2 ,
    SC    = 3 ,
    SD    = 4 ,
    SE    = 5 ,
    SF    = 6 ,
    SG    = 7 ;

reg [7:0] state, next_state ;
reg [7:0] out, next_out ;

always @ (in or state) begin

    // default values

    next_state = 8'b0 ;
    next_out = 8'bx ;

    case (1'b1) // synopsys parallel_case full_case

    state[START]:
        if (in == 8'h3c) begin
            next_state[SA] = 1'b1 ;
            next_out = 8'h82 ;
            end
        else begin
            next_state[START] = 1'b1 ;
            next_out = 8'h00 ;
            end

    state[SA]:
        case (in) // synopsys parallel_case full_case
            8'h2a:
                begin
                next_state[SC] = 1'b1 ;
                next_out = 8'h40 ;
                end
            8'h1f:
                begin
                next_state[SB] = 1'b1 ;
                next_out = 8'h20 ;
                end
            default:
                begin
                next_state[SA] = 1'b1 ;
                next_out = 8'h04 ;
```

```
                end
            endcase

    state[SB]:
        if (in == 8'haa) begin
            next_state[SE] = 1'b1 ;
            next_out = 8'h11 ;
            end
        else begin
            next_state[SF] = 1'b1 ;
            next_out = 8'h30 ;
            end

    state[SC]:
        begin
        next_state[SD] = 1'b1 ;
        next_out = 8'h08 ;
        end

    state[SD]:
        begin
        next_state[SG] = 1'b1 ;
        next_out = 8'h80 ;
        end

    state[SE]:
        begin
        next_state[START] = 1'b1 ;
        next_out = 8'h40 ;
        end

    state[SF]:
        begin
        next_state[SG] = 1'b1 ;
        next_out = 8'h02 ;
        end

    state[SG]:
        begin
        next_state[START] = 1'b1 ;
        next_out = 8'h01 ;
        end

    endcase

    end

// build the state flip-flops
always @ (posedge clk or negedge rst)
```

```
    begin
    if (!rst) begin
        state <= #1 8'b0 ;
        state[START] <= #2 1'b1 ;
        end
    else
        state <= #1 next_state ;
    end

// build the output flip-flops
always @ (posedge clk or negedge rst)
    begin
    if (!rst)   out <= #1 8'b0 ;
    else        out <= #1 next_out ;
    end

endmodule
```

prep4

prep4 is a Moore machine with sixteen states and 40 transitions. It has eight inputs and eight unregistered outputs. The state diagram is shown in Figure 4.

Listing 3 is a Verilog implementation for use with Synopsys FSM `extract`.

Listing 4 is a Verilog implementation that is one-hot coded.

Listing 5 is a VHDL implementation for use with Synopsys FSM `extract`.

Listing 6 is a VHDL implementation that is one-hot coded.

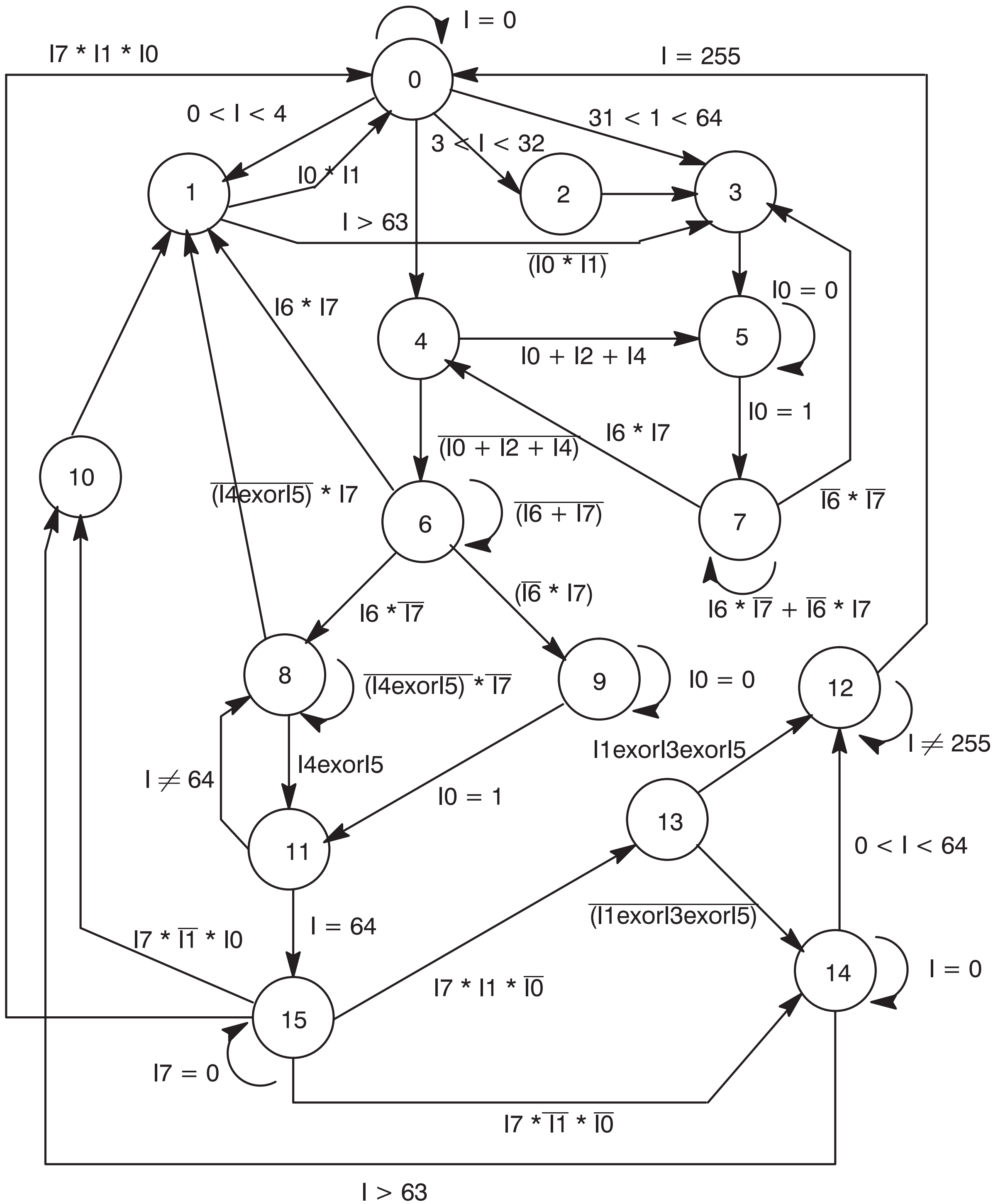

Figure 4. prep4 State Transition Diagram

Listing 3—prep4.v

```
/*
** prep4.v
**
** prep benchmark 4 -- large state machine
** benchmark suite #1 -- version 1.2 -- March 28, 1993
** Programmable Electronics Performance Corporation
**
** binary state assignment -- highly encoded
*/

module prep4 (clk, rst, in, out) ;

input clk, rst ;
input [7:0] in ;
output [7:0] out ;

parameter [3:0] // synopsys enum code
  S0  = 4'd0 ,  S1  = 4'd1 ,  S2  = 4'd2 ,  S3  = 4'd3 ,
  S4  = 4'd4 ,  S5  = 4'd5 ,  S6  = 4'd6 ,  S7  = 4'd7 ,
  S8  = 4'd8 ,  S9  = 4'd9 ,  S10 = 4'd10 , S11 = 4'd11 ,
  S12 = 4'd12 , S13 = 4'd13 , S14 = 4'd14 , S15 = 4'd15 ;

// synopsys state_vector state
reg [3:0] /* synopsys enum code */ state, next_state ;
reg [7:0] out ;

// state machine

always @ (in or state) begin

  // default value
  next_state = S0 ; // always overridden

  case (state) // synopsys parallel_case full_case

    S0: case(1'b1) // synopsys parallel_case full_case
          (in == 8'd0):               next_state = S0 ;
          (8'd0 < in && in < 8'd4):   next_state = S1 ;
          (8'd3 < in && in < 8'd32):  next_state = S2 ;
          (8'd31 < in && in < 8'd64): next_state = S3 ;
          (in > 8'd63):               next_state = S4 ;
          endcase

    S1: if (in[0] && in[1])   next_state = S0 ;
        else                  next_state = S3 ;

    S2: next_state = S3 ;
```

```
S3: next_state = S5 ;

S4: if (in[0] || in[2] || in[4])  next_state = S5 ;
    else                          next_state = S6 ;

S5: if (in[0] == 1'b0)    next_state = S5 ;
    else                  next_state = S7 ;

S6: case(in[7:6]) // synopsys parallel_case full_case
      2'b11:  next_state = S1 ;
      2'b00:  next_state = S6 ;
      2'b01:  next_state = S8 ;
      2'b10:  next_state = S9 ;
      endcase

S7: case(in[7:6]) // synopsys parallel_case full_case
      2'b00:  next_state = S3 ;
      2'b11:  next_state = S4 ;
      2'b10,
      2'b01:  next_state = S7 ;
      endcase

S8: if(in[4] ^ in[5])       next_state = S11 ;
    else if (in[7])         next_state = S1 ;
    else                    next_state = S8 ;

S9: if (in[0] == 1'b0)      next_state = S9 ;
    else                    next_state = S11 ;

S10:  next_state = S1 ;

S11:  if (in == 8'd64)      next_state = S15 ;
      else                  next_state = S8 ;

S12:  if (in == 8'd255)     next_state = S0 ;
      else                  next_state = S12 ;

S13:  if (in[1] ^ in[3] ^ in[5])  next_state = S12 ;
      else                        next_state = S14 ;

S14:  case(1'b1) // synopsys parallel_case full_case
        (in == 8'd0):               next_state = S14 ;
        (8'd0 < in && in < 8'd64):  next_state = S12 ;
        (in > 8'd63):               next_state = S10 ;
        endcase

S15:  if (in[7] == 1'b0)      next_state = S15 ;
      else
        case (in[1:0])
```

```
              // synopsys parallel_case full_case
              2'b11:  next_state = S0 ;
              2'b01:  next_state = S10 ;
              2'b10:  next_state = S13 ;
              2'b00:  next_state = S14 ;
              endcase
    endcase
  end

// outputs

always @ (state) begin

  // default value
  out = 8'bx ;

  case (state) // synopsys parallel_case full_case
    S0: out = 8'b00000000 ;
    S1: out = 8'b00000110 ;
    S2: out = 8'b00011000 ;
    S3: out = 8'b01100000 ;
    S4: begin
        out[7] = 1'b1 ; out[0] = 1'b0 ;
        end
    S5: begin
        out[6] = 1'b1 ; out[1] = 1'b0 ;
        end
    S6: out = 8'b00011111 ;
    S7: out = 8'b00111111 ;
    S8: out = 8'b01111111 ;
    S9: out = 8'b11111111 ;
    S10:  begin
          out[6] = 1'b1 ; out[4] = 1'b1 ;
          out[2] = 1'b1 ; out[0] = 1'b1 ;
          end
    S11:  begin
          out[7] = 1'b1 ; out[5] = 1'b1 ;
          out[3] = 1'b1 ; out[1] = 1'b1 ;
          end
    S12:  out = 8'b11111101 ;
    S13:  out = 8'b11110111 ;
    S14:  out = 8'b11011111 ;
    S15:  out = 8'b01111111 ;
    endcase
  end

// build the state flip-flops
always @ (posedge clk or negedge rst)
  begin
```

```
  if (!rst) state <= #1 S0 ;
  else      state <= #1 next_state ;
  end

endmodule
```

Listing 4—prep4_onehot.v

```
/*
** prep4_onehot.v
**
** prep benchmark 4 -- large state machine
** benchmark suite #1 -- version 1.2 -- March 28, 1993
** Programmable Electronics Performance Corporation
**
** one-hot state assignment
*/

module prep4 (clk, rst, in, out) ;

input clk, rst ;
input [7:0] in ;
output [7:0] out ;

parameter [3:0]
  S0  = 4'd0 ,  S1  = 4'd1 ,  S2  = 4'd2 ,  S3  = 4'd3 ,
  S4  = 4'd4 ,  S5  = 4'd5 ,  S6  = 4'd6 ,  S7  = 4'd7 ,
  S8  = 4'd8 ,  S9  = 4'd9 ,  S10 = 4'd10 , S11 = 4'd11 ,
  S12 = 4'd12 , S13 = 4'd13 , S14 = 4'd14 , S15 = 4'd15 ;

reg [15:0] state, next_state ;
reg [7:0] out ;

// state machine

always @ (in or state) begin

  // default value
  next_state = 16'b0 ;  // only one bit overridden

  case (1'b1) // synopsys parallel_case full_case

   state[S0]:
     case(1'b1) // synopsys parallel_case full_case
       (in == 8'd0):              next_state[S0] = 1'b1 ;
       (8'd0 < in && in < 8'd4):  next_state[S1] = 1'b1 ;
       (8'd3 < in && in < 8'd32): next_state[S2] = 1'b1 ;
       (8'd31 < in && in < 8'd64): next_state[S3] = 1'b1 ;
       (in > 8'd63):              next_state[S4] = 1'b1 ;
       endcase
```

```
state[S1]:  if (in[0] && in[1]) next_state[S0] = 1'b1 ;
            else                next_state[S3] = 1'b1 ;
state[S2]:  next_state[S3] = 1'b1 ;

state[S3]:  next_state[S5] = 1'b1 ;

state[S4]:
  if (in[0] || in[2] || in[4])  next_state[S5] = 1'b1 ;
  else                          next_state[S6] = 1'b1 ;

state[S5]:  if (in[0] == 1'b0)  next_state[S5] = 1'b1 ;
            else                next_state[S7] = 1'b1 ;
state[S6]:
  case(in[7:6]) // synopsys parallel_case full_case
    2'b11:  next_state[S1] = 1'b1 ;
    2'b00:  next_state[S6] = 1'b1 ;
    2'b01:  next_state[S8] = 1'b1 ;
    2'b10:  next_state[S9] = 1'b1 ;
    endcase

state[S7]:
  case(in[7:6]) // synopsys parallel_case full_case
    2'b00:  next_state[S3] = 1'b1 ;
    2'b11:  next_state[S4] = 1'b1 ;
    2'b10,
    2'b01:  next_state[S7] = 1'b1 ;
    endcase

state[S8]:  if(in[4] ^ in[5])   next_state[S11] = 1'b1 ;
            else if (in[7])     next_state[S1] = 1'b1 ;
            else                next_state[S8] = 1'b1 ;

state[S9]:  if (in[0] == 1'b0)  next_state[S9] = 1'b1 ;
            else                next_state[S11] = 1'b1 ;

state[S10]: next_state[S1] = 1'b1 ;

state[S11]: if (in == 8'd64)  next_state[S15] = 1'b1 ;
            else              next_state[S8] = 1'b1 ;

state[S12]: if (in == 8'd255) next_state[S0] = 1'b1 ;
            else              next_state[S12] = 1'b1 ;

state[S13]:
  if (in[1] ^ in[3] ^ in[5])  next_state[S12] = 1'b1 ;
  else                        next_state[S14] = 1'b1 ;

state[S14]:
  case(1'b1) // synopsys parallel_case full_case
```

```
      (in == 8'd0):                next_state[S14] = 1'b1 ;
      (8'd0 < in && in < 8'd64):  next_state[S12] = 1'b1 ;
      (in > 8'd63):                next_state[S10] = 1'b1 ;
      endcase

  state[S15]:
    if (in[7] == 1'b0)      next_state[S15] = 1'b1 ;
    else
      case (in[1:0]) // synopsys parallel_case full_case
        2'b11:  next_state[S0] = 1'b1 ;
        2'b01:  next_state[S10] = 1'b1 ;
        2'b10:  next_state[S13] = 1'b1 ;
        2'b00:  next_state[S14] = 1'b1 ;
        endcase
  endcase
 end

// outputs

always @ (state) begin

  // default value
  out = 8'bx ;

  case (1'b1) // synopsys parallel_case full_case
    state[S0]:  out = 8'b00000000 ;
    state[S1]:  out = 8'b00000110 ;
    state[S2]:  out = 8'b00011000 ;
    state[S3]:  out = 8'b01100000 ;
    state[S4]:  begin
        out[7] = 1'b1 ; out[0] = 1'b0 ;
        end
    state[S5]:  begin
        out[6] = 1'b1 ; out[1] = 1'b0 ;
        end
    state[S6]:  out = 8'b00011111 ;
    state[S7]:  out = 8'b00111111 ;
    state[S8]:  out = 8'b01111111 ;
    state[S9]:  out = 8'b11111111 ;
    state[S10]: begin
        out[6] = 1'b1 ; out[4] = 1'b1 ;
        out[2] = 1'b1 ; out[0] = 1'b1 ;
        end
    state[S11]: begin
        out[7] = 1'b1 ; out[5] = 1'b1 ;
        out[3] = 1'b1 ; out[1] = 1'b1 ;
        end
    state[S12]: out = 8'b11111101 ;
    state[S13]: out = 8'b11110111 ;
```

```
    state[S14]: out = 8'b11011111 ;
    state[S15]: out = 8'b01111111 ;
    endcase
  end

// build the state flip-flops
always @ (posedge clk or negedge rst)
  begin
  if (!rst) begin
            state <= #1 16'b0 ;
            state[S0] <= #2 1'b1 ;
            end
  else      state <= #1 next_state ;
  end

endmodule
```

Listing 5—prep4.vhd

```
-- prep4.vhd
--
-- prep benchmark 4 -- large state machine
-- benchmark suite #1 -- version 1.2 -- March 28, 1993
-- Programmable Electronics Performance Corporation
--
-- binary state assignment, highly encoded

library IEEE ;
use IEEE.std_logic_1164.all ;
use IEEE.std_logic_arith.all ;

package typedef is
  subtype byte is std_logic_vector (7 downto 0) ;
  subtype bytein is bit_vector (7 downto 0) ;
end typedef ;

library IEEE ;
use IEEE.std_logic_1164.all ;
use IEEE.std_logic_arith.all ;
use work.typedef.all ;

entity prep4 is
  port ( clk,rst : in std_logic ;
    I : in byte ;
    O : out byte) ;
end prep4 ;

architecture behavior of prep4 is
  type state_type is (S0, S1, S2, S3,
    S4, S5, S6, S7, S8, S9, S10, S11,
    S12, S13, S14, S15) ;
```

```
  signal state, next_state : state_type ;
  attribute state_vector : string ;
  attribute state_vector of behavior :
    architecture is ”state” ;
  signal Iin : bytein ;
begin

  process (I)
  begin
    Iin <= to_bitvector(I);
  end process ;

  -- state machine

  process (Iin, state)
  begin
    -- default value
    next_state <= S0 ;

    case state is

    when S0 =>
      if (Iin = X”00”) then
        next_state <= S0;
        end if ;
      if (x”00” < Iin) and (Iin < x”04”) then
        next_state <= S1;
        end if;
      if (x”03” < Iin) and (Iin < x”20”) then
        next_state <= S2;
        end if;
      if (x”1f” < Iin) and (Iin < x”40”) then
        next_state <= S3;
        end if;
      if (x”3f” < Iin) then
        next_state <= S4;
        end if;

    when S1 =>
      if (Iin(1) and Iin(0)) = ’1’ then
        next_state <= S0;
      else
        next_state <= S3;
        end if ;

    when S2 =>
      next_state <= S3 ;

    when S3 =>
```

```
  next_state <= S5 ;

when S4 =>
  if (Iin(0) or Iin(2) or Iin(4)) = '1' then
    next_state <= S5 ;
  else
    next_state <= S6 ;
    end if ;

when S5 =>
  if (Iin(0) = '0') then
    next_state <= S5 ;
  else
    next_state <= S7 ;
    end if ;

when S6 =>
  case Iin(7 downto 6) is
    when b"11" => next_state <= S1 ;
    when b"00" => next_state <= S6 ;
    when b"01" => next_state <= S8 ;
    when b"10" => next_state <= S9 ;
    end case ;

when S7 =>
  case Iin(7 downto 6) is
    when b"00" => next_state <= S3 ;
    when b"11" => next_state <= S4 ;
    when b"01" => next_state <= S7 ;
    when b"10" => next_state <= S7 ;
    end case ;

when S8 =>
  if (Iin(4) xor Iin(5)) = '1'  then
    next_state <= S11 ;
  elsif Iin(7) = '1' then
    next_state <= S1 ;
  else
    next_state <= S8 ;
    end if;

when S9 =>
  if (Iin(0) = '1') then
    next_state <= S11 ;
  else
    next_state <= S9 ;
    end if;

when S10 =>
```

```
      next_state <= S1 ;

    when S11 =>
      if Iin = x"40" then
        next_state <= S15 ;
      else
        next_state <= S8 ;
        end if ;

    when S12 =>
      if Iin = x"ff" then
        next_state <= S0 ;
      else
        next_state <= S12 ;
        end if ;

    when S13 =>
      if (Iin(1) xor Iin(3) xor Iin(5)) = '1' then
        next_state <= S12 ;
      else
        next_state <= S14 ;
        end if ;

    when S14 =>
      if (Iin > x"3f") then
        next_state <= S10 ;
      elsif (Iin = x"00") then
        next_state <= S14 ;
      else
        next_state <= S12 ;
        end if ;

    when S15 =>
      if Iin(7) = '0' then
        next_state <= S15 ;
      else
        case Iin(1 downto 0) is
          when b"11" => next_state <= S0 ;
          when b"01" => next_state <= S10 ;
          when b"10" => next_state <= S13 ;
          when b"00" => next_state <= S14 ;
          end case ;
        end if ;
    end case ;
  end process;

  -- outputs

  process (state)
```

```
    begin

      -- default value is don't care
      O <= byte'(others => 'X') ;

      case state is
        when S0 =>  O <= "00000000" ;
        when S1 =>  O <= "00000110" ;
        when S2 =>  O <= "00011000" ;
        when S3 =>  O <= "01100000" ;
        when S4 =>
            O(7) <= '1' ;
            O(0) <= '0' ;
        when S5 =>
            O(6) <= '1' ;
            O(1) <= '0' ;
        when S6 =>  O <= "00011111" ;
        when S7 =>  O <= "00111111" ;
        when S8 =>  O <= "01111111" ;
        when S9 =>  O <= "11111111" ;
        when S10 =>
            O(6) <='1' ;
            O(4) <='1' ;
            O(2) <='1' ;
            O(0) <='1' ;
        when S11 =>
            O(7) <='1' ;
            O(5) <='1' ;
            O(3) <='1' ;
            O(1) <='1' ;
        when S12 => O <= "11111101" ;
        when S13 => O <= "11110111" ;
        when S14 => O <= "11011111" ;
        when S15 => O <= "01111111" ;
        end case ;
    end process;

    -- build the state flip-flops
    process (clk, rst)
    begin
      if rst='0' then
        state <= S0 ;
      elsif clk='1' and clk'event then
        state <= next_state ;
      end if ;
    end process ;

  end behavior ;
```

Listing 6—prep4_onehot.vhd

```
-- prep4_onehot.vhd
--
-- prep benchmark 4 -- large state machine
-- benchmark suite #1 -- version 1.2 -- March 28, 1993
-- Programmable Electronics Performance Corporation
--
-- one-hot state assignment

library IEEE ;
use IEEE.std_logic_1164.all ;
use IEEE.std_logic_arith.all ;

package typedef is
  subtype state_vec is std_logic_vector (0 to 15) ;
  subtype byte is std_logic_vector (7 downto 0) ;
  subtype bytein is bit_vector (7 downto 0) ;
end typedef ;

library IEEE ;
use IEEE.std_logic_1164.all ;
use IEEE.std_logic_arith.all ;
use work.typedef.all ;

entity prep4 is
  port ( clk,rst : in std_logic ;
      I : in byte ;
      O : out byte) ;
end prep4 ;

architecture behavior of prep4 is
  signal state, next_state : state_vec ;
  signal Iin : bytein ;
begin
  process (I)
  begin
    Iin <= to_bitvector(I);
  end process ;

  -- state machine

  process (Iin, state)
  begin
    -- default value
    next_state <= state_vec'(others => '0') ;

    if state(0) = '1' then
      if (Iin = X"00") then
        next_state(0) <= '1';       end if ;
```

```
  if (x"00" < Iin) and (Iin < x"04") then
    next_state(1) <= '1';       end if;
  if (x"03" < Iin) and (Iin < x"20") then
    next_state(2) <= '1';       end if;
  if (x"1f" < Iin) and (Iin < x"40") then
    next_state(3) <= '1';       end if;
  if (x"3f" < Iin) then
    next_state(4) <= '1';       end if;
  end if;

if state(1) = '1' then
  if (Iin(1) and Iin(0)) = '1' then
    next_state(0) <= '1';
  else
    next_state(3) <= '1';       end if ;
  end if ;

if state(2) = '1' then
  next_state(3) <= '1' ;
  end if;

if state(3) = '1' then
  next_state(5) <= '1' ;
  end if;

if state(4) = '1' then
  if (Iin(0) or Iin(2) or Iin(4)) = '1' then
    next_state(5) <= '1' ;
  else
    next_state(6) <= '1' ;       end if ;
  end if;

if state(5) = '1' then
  if (Iin(0) = '0') then
    next_state(5) <= '1' ;
  else
    next_state(7) <= '1' ;       end if ;
  end if;

if state(6) = '1' then
  case Iin(7 downto 6) is
    when b"11" => next_state(1) <= '1' ;
    when b"00" => next_state(6) <= '1' ;
    when b"01" => next_state(8) <= '1' ;
    when b"10" => next_state(9) <= '1' ;
    end case ;
  end if;

if state(7) = '1' then
```

```
  case Iin(7 downto 6) is
    when b"00" => next_state(3) <= '1' ;
    when b"11" => next_state(4) <= '1' ;
    when b"01" => next_state(7) <= '1' ;
    when b"10" => next_state(7) <= '1' ;
    end case ;
  end if;

if state(8) = '1' then
  if (Iin(4) xor Iin(5)) = '1'  then
    next_state(11) <= '1' ;
  elsif Iin(7) = '1' then
    next_state(1) <= '1' ;
  else
    next_state(8) <= '1' ;        end if;
  end if;

if state(9) = '1' then
  if (Iin(0) = '1') then
    next_state(11) <= '1' ;
  else
    next_state(9) <= '1' ;        end if;
  end if;

if state(10) = '1' then
  next_state(1) <= '1' ;
  end if ;

if state(11) = '1' then
  if Iin = x"40" then
    next_state(15) <= '1' ;
  else
    next_state(8) <= '1' ;        end if ;
  end if ;

if state(12) = '1' then
  if Iin = x"ff" then
    next_state(0) <= '1' ;
  else
    next_state(12) <= '1' ;       end if ;
  end if ;

if state(13) = '1' then
  if (Iin(1) xor Iin(3) xor Iin(5)) = '1' then
    next_state(12) <= '1' ;
  else
    next_state(14) <= '1' ;       end if ;
  end if ;
```

```
  if state(14) = '1' then
    if (Iin > x"3f") then
      next_state(10) <= '1' ;
    elsif (Iin = x"00") then
      next_state(14) <= '1' ;
    else
      next_state(12) <= '1' ;        end if ;
    end if ;

  if state(15) = '1' then
    if Iin(7) = '0' then
      next_state(15) <= '1' ;
    else
      case Iin(1 downto 0) is
        when b"11" => next_state(0) <= '1' ;
        when b"01" => next_state(10) <= '1' ;
        when b"10" => next_state(13) <= '1' ;
        when b"00" => next_state(14) <= '1' ;
        end case ;
      end if ;
    end if ;
end process;

-- outputs

process (state)
begin

  -- default value is don't care
  O <= byte'(others => 'X') ;

  if state(0) = '1' then  O <= "00000000" ; end if ;
  if state(1) = '1' then  O <= "00000110" ; end if ;
  if state(2) = '1' then  O <= "00011000" ; end if ;
  if state(3) = '1' then  O <= "01100000" ; end if ;
  if state(4) = '1' then
    O(7) <= '1' ;
    O(0) <= '0' ;
    end if ;
  if state(5) = '1' then
    O(6) <= '1' ;
    O(1) <= '0' ;
    end if ;
  if state(6) = '1' then  O <= "00011111" ; end if ;
  if state(7) = '1' then  O <= "00111111" ; end if ;
  if state(8) = '1' then  O <= "01111111" ; end if ;
  if state(9) = '1' then  O <= "11111111" ; end if ;
  if state(10) = '1' then
    O(6) <='1' ;
```

```
      O(4) <='1' ;
      O(2) <='1' ;
      O(0) <='1' ;
      end if ;
    if state(11) = '1' then
      O(7) <='1' ;
      O(5) <='1' ;
      O(3) <='1' ;
      O(1) <='1' ;
      end if ;
    if state(12) = '1' then O <= "11111101" ; end if ;
    if state(13) = '1' then O <= "11110111" ; end if ;
    if state(14) = '1' then O <= "11011111" ; end if ;
    if state(15) = '1' then O <= "01111111" ; end if ;

  end process;

  -- build the state flip-flops
  process (clk, rst)
  begin
    if rst='0' then
      state <= state_vec'(others => '0') ;
      state(0) <= '1' ;
    elsif clk='1' and clk'event then
      state <= next_state ;
    end if ;
  end process ;

end behavior ;
```

About the Author

Steve Golson has been an independent consultant for the past eight years. His areas of expertise include VLSI design (full-custom, semi-custom, gate array, FPGA); computer architecture, especially memory systems; and digital hardware design. Mr. Golson also provides services in reverse engineering, in patent infringement analysis, and as an expert witness.

Prior to striking out on his own, Mr. Golson worked for five years at General Computer Company of Cambridge, Mass. While at GCC he designed several microprocessor-controlled advanced graphics systems for real-time applications (video games). He has a B.S. in Earth, Atmospheric, and Planetary Sciences from MIT.

You may contact the author at sgolson@trilobyte.com or (508) 369–9669.

Synopsys Server Copyright

Copyright 1994 Synopsys, Inc. 700 East Middlefield Road, Mountain View, CA 94043–4033. All rights reserved.

Any person is hereby authorized to view, copy, print, and distribute these documents subject to the following conditions:

1. This document may be used for informational purposes only.

2. Any copy of this document or portion thereof must include the copyright notice.

Restricted Rights Legend

Use, duplication, or disclosure by the United States Government is subject to the restrictions set forth in DFARS 252.227–7013 (c)(1)(ii) and FAR 52.227–19.

Trademarks

Synopsys and the Synopsys logo are trademarks or registered trademarks of Synopsys, Inc. All other product names and company logos mentioned herein are the trademarks of their respective owners.

Warranties

THIS PUBLICATION IS PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON–INFRINGEMENT.

REFERENCES TO CORPORATIONS, THEIR SERVICES AND PRODUCTS, ARE PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED. IN NO EVENT SHALL SYNOPSYS, INC. BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT OR CONSEQUENTIAL DAMAGES OF ANY KIND, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER OR NOT ADVISED OF THE POSSIBILITY OF DAMAGE, AND ON ANY THEORY OF LIABILITY, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS INFORMATION.

Descriptions of, or references to, products or publications within the Synopsys Information Server does not imply endorsement of that product or publication. Synopsys, Inc. makes no warranty of any kind with respect to the subject matter included herein, the products listed herein, or the completeness or accuracy of this catalog. Synopsys specifically disclaims all warranties, express, implied or otherwise, includ-

ing without limitation, all warranties of merchantability and fitness for a particular purpose.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SYNOPSYS MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.