FOR ENGINEERS AND ENGINEERING MANAGERS — WORLDWIDE

ELECTRONIC DESIGN

A PENTON PUBLICATION U.S. $5.00

SEPTEMBER 13, 1990

DESIGN APPLICATIONS

WHEN DESIGNING STATE MACHINES, A TECHNIQUE CALLED ONE-HOT ENCODING CREATES EFFICIENT CIRCUITS FOR TOP-PERFORMING FPGA MACROS.

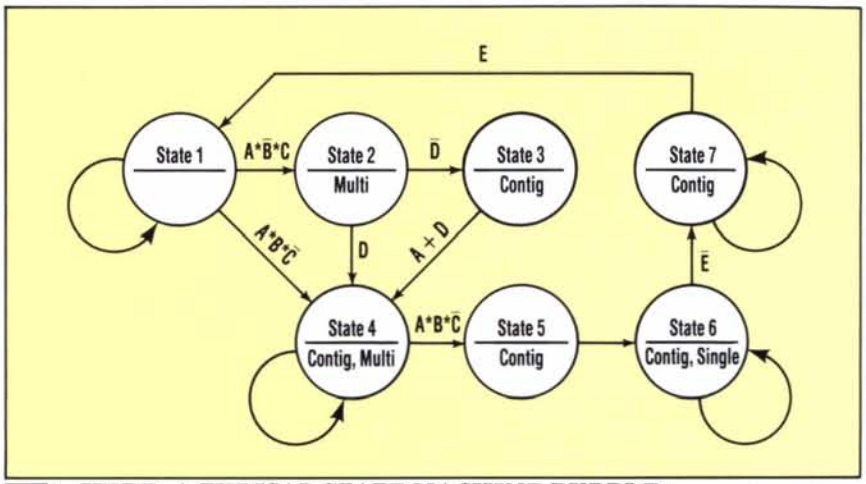
ACCELERATE FPGA MACROS WITH ONE-HOT APPROACH

STEVEN K. KNAPP
Xilinx Inc., 2100 Logic Dr.,
San Jose, CA 95124;
(408) 879-5172.

State machines—one of the most commonly implemented functions with programmable logic—are employed in various digital applications, particularly controllers. However, the limited number of flip-flops and the wide combinatorial logic of a PAL device favors state machines that are based on a highly encoded state sequence. For example, each state within a 16-state machine would be encoded using four flip-flops as the binary values between 0000 and 1111.

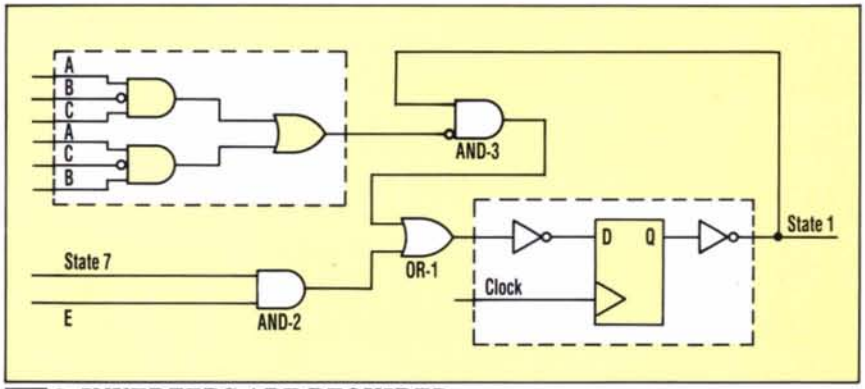
A more flexible scheme—called one-hot encoding (OHE)—employs one flip-flop per state for building state machines. Although it can be used with PAL-type programmable-logic devices (PLDs), OHE is better suited for use with the fan-in limited and flip-flop-rich architectures of the higher-gate-count field-programmable gate arrays (FPGAs), such as offered by Xilinx, Actel, and others. This is because OHE requires a larger number of flip-flops. It offers a simple and easy-to-use method of generating performance-optimized state-machine designs because there are few levels of logic between flip-flops.

A state machine implemented with a highly encoded state sequence will



1. HERE, A TYPICAL STATE MACHINE BUBBLE diagram shows the operation of a seven-state state machine that reacts to inputs A through E as well as previous-state conditions.

STATE MACHINE DESIGN



2. INVERTERS ARE REQUIRED at the D input and the Q output of the state flip-flop to ensure that it powers on in the proper state. Combinatorial logic decodes the operations based on the input conditions and the state feedback signals. The flip-flop will remain in State 1 as long as the conditional paths out of the state are not valid.

generally have many, wide-input logic functions to interpret the inputs and decode the states. Furthermore, incorporating a highly encoded state machine in an FPGA requires several levels of logic between clock edges because multiple logic blocks will be needed for decoding the states. A better way to implement state machines in FPGAs is to match the state-machine architecture to the device architecture.

LIMITING FAN-IN

A good state-machine approach for FPGAs limits the amount of fan-in into one logic block. While the one-hot method is best for most FPGA applications, binary encoding is still more efficient in certain cases, such as for small state machines. It's up to the designer to evaluate all approaches before settling on one for a particular application.

FPGAs are high-density programmable chips that contain a large array of user-configurable logic blocks surrounded by user-programmable interconnects. Generally, the logic blocks in an FPGA have a limited number of inputs. The logic block in the Xilinx XC-3000 series, for instance, can implement any function of five or less inputs. In contrast, a PAL macrocell is fed by each input to the chip and all of the flip-flops. This difference in logic structure between PALs and FPGAs is important for functions with many inputs: Where a PAL could implement a many-input logic function in one level of logic, an FPGA might require multiple logic layers due to the limited number of inputs.

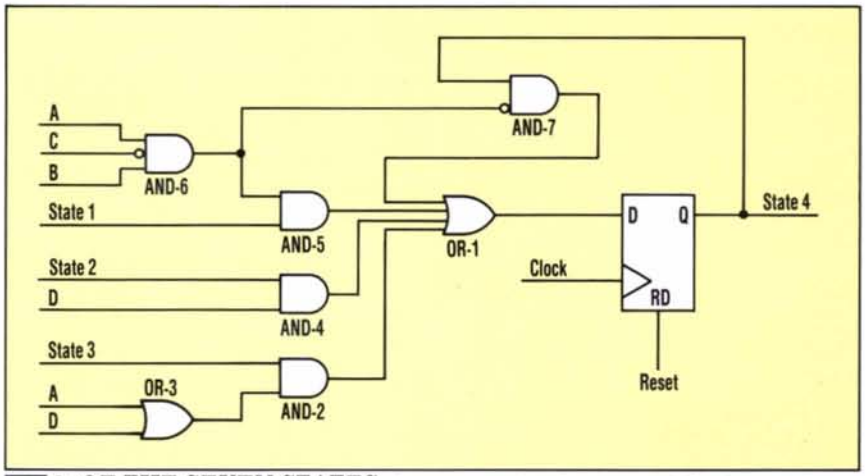
The OHE scheme is named so because only one state flip-flop is asserted, or "hot," at a time. Using the one-hot-encoding method for FPGAs was originally conceived by High-Gate Design—a Saratoga, Calif.-based consulting firm specializing in FPGA designs.

The OHE state machine's basic structure is simple—first assign an individual flip-flop to each state, and then permit only one state to be active at any time. A state machine with 16 states would require 16 flip-flops using the OHE approach; a highly encoded state machine would need just 4 flip-flops. At first glance, OHE may seem counter-intuitive. For designers accustomed to using PLDs, more flip-flops typically indicates either using a larger PLD or even multiple devices.

In an FPGA, however, OHE yields a state machine that generally requires fewer resources and has higher performance than a binary-encoded implementation. OHE has definite advantages for FPGA designs because it exploits the strengths of the FPGA architecture. It usually requires two or less levels of logic between clock edges than binary encoding. That translates into faster operation. Logic circuits are also simplified because OHE removes much of the state-decoding logic—a one-hot-encoded state machine is already fully decoded.

OHE requires only one input to decode a state, making the next-state logic simple and well-suited to the limited fan-in architecture of FPGAs. In addition, the resulting collection of flip-flops is similar to a shift-register-like structure, which can placed and routed efficiently inside an FPGA device. The speed of an OHE state machine remains fairly constant even as the number of states grows. In contrast, a highly encoded state machine's performance drops as the states grow because of the wider and deeper decoding logic that's required.

To build the next-state logic for



3. OF THE SEVEN STATES, the state-transition logic required for State 4 is the most complex, requiring inputs from three other state outputs as well as four of the five condition signals (A - D).

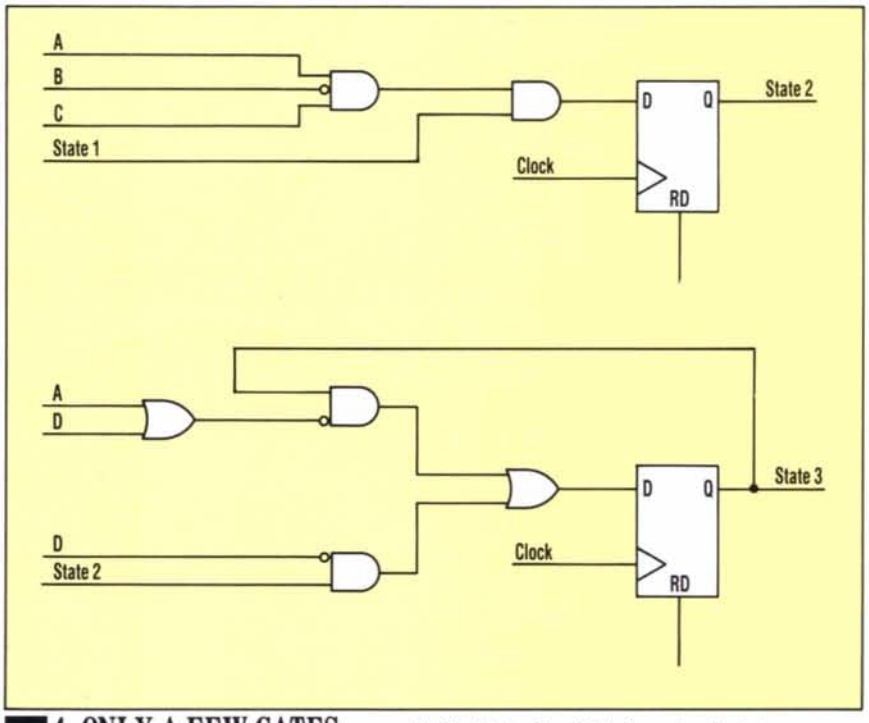
STATE MACHINE DESIGN

OHE state machines is simple, lending itself to a "cookbook" approach. At first glance, designers familiar with PAL-type devices may be concerned by the number of potential illegal states due to the sparse state encoding. This issue, to be discussed later, can be solved easily.

A typical, simple state machine might contain seven distinct states that can be described with the commonly used circle-and-arc bubble diagrams *(Fig. 1)*. The label above the line in each "bubble" is the state's name, the labels below the line are the outputs asserted while the state is active. In the example, there are seven states labeled State 1-7. The "arcs" that feed back into the same state are the default paths. These will be true only if no other conditional paths are true.

Each conditional path is labeled with the appropriate logical condition that must exist before moving to the next state. All of the logic inputs are labeled as variables A through E. The outputs from the state machine are called Single, Multi, and Contig. For this example, State 1, which must be asserted at power-on, has a doubly-inverted flip-flop structure *(shaded region of Fig. 2)*.

The state machine in the example was built twice, once using OHE and again with the highly encoded approach employed in most PAL designs. A Xilinx XC3020-100 2000-gate FPGA was the target for both implementations. Though the OHE circuit required slightly more logic than the highly-encoded state machine, the one-hot state machine operated 17% faster *(see the table)*. Intuitively, the one-hot method might seem to employ many more logic blocks than the highly encoded approach. But the highly encoded state machine needs more combinatorial logic to decode the encoded state values.



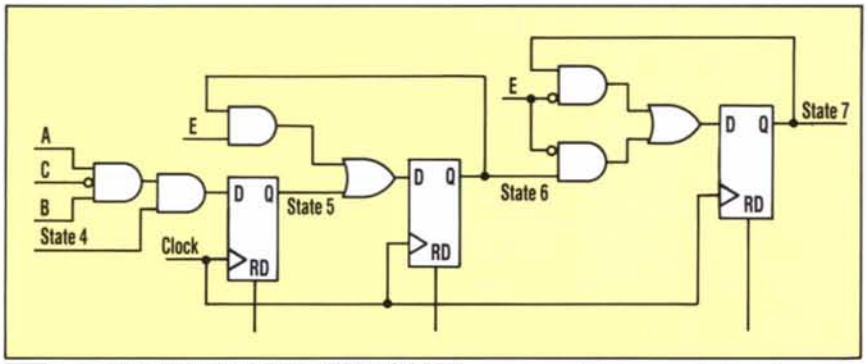
4. ONLY A FEW GATES are required by States 2 and 3 to form simple state-transition logic decoding. Just two gates are needed by State 2 (top), while four simple gates are used by State 3 (bottom).

The OHE approach produces a state machine with a shift-register structure that almost always outperforms a highly encoded state machine in FPGAs. The one-state design had only two layers of logic between flip-flops, while the highly encoded design had three. For other applications, the results can be far more dramatic. In many cases, the one-hot method yields a state machine with one layer of logic between clock edges. With one layer of logic, a one-hot state machine can operate at 50 to 60 MHz.

The initial or power-on condition in a state machine must be examined carefully. At power-on, a state machine should always enter an initial, known state. For the Xilinx FPGA family, all flip-flops are reset at power-on automatically. To assert an initial state at power-on, the output from the initial-state flip-flop is inverted. To maintain logical consistency, the input to flip-flop also is inverted.

All other states use a standard, D-type flip-flop with an asynchronous reset input. The purpose of the asynchronous reset input will be discussed later when illegal states are covered.

Once the start-up conditions are set up, the next-state transition logic can be configured. To do that, first examine an individual state. Then



5. LOOKING NEARLY THE SAME as a simple shift register, the logic for States 5, 6, and 7 is very simple. This is because the OHE scheme eliminates almost all decoding logic that precedes each flip-flop.

STATE MACHINE DESIGN

count the number of conditional paths leading into the state and add an extra path if the default condition is to remain in the same state. Second, build an OR-gate with the number of inputs equal to the number of conditional paths that were determined in the first step.

Third, for each input of the OR-gate, build an AND-gate of the previous state and its conditional logic. Finally, if the default should remain in the same state, build an AND-gate of the present state and the inverse of *all* possible conditional paths leaving the present state.

To determine the number of conditional paths feeding State 1, examine the state diagram—State 1 has one path from State 7 whenever the variable E is true. Another path is the default condition, which stays in State 1. As a result, there are two conditional paths feeding State 1. Next, build a 2-input OR-gate—one input for the conditional path from State 7, the other for the default path to stay in State 1 *(shown as OR-1 in Fig. 2)*.

The next step is to build the conditional logic feeding the OR-gate. Each input into the OR-gate is the logical AND of the previous state and its conditional logic feeding into State 1. State 7, for example, feeds State 1 whenever E is true and is implemented using the gate called AND-2 *(Fig. 2, again)*. The second input into the OR-gate is the default transition that's to remain in State 1. In other words, if the current state is State 1, and no conditional paths leaving State 1 are valid, then the state machine should remain in State 1. Note in the state diagram that two conditional paths are leaving State 1 *(Fig. 1, again)*.

The first path is valid whenever ($A^*\overline{B}^*C$) is true, which leads into State 2. The second path is valid whenever ($A^*B^*\overline{C}$) is true, leading into State 4. To build the default logic, State 1 is ANDed with the inverse of all of the conditional paths leaving State 1. The logic to perform this function is implemented in the gate labeled AND-3 and the logic elements that feed into the inverting input of AND-3 *(Fig. 2, again)*.

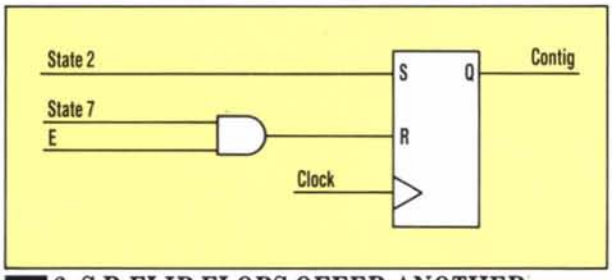
State 4 is the most complex state in the state-machine example. However, creating the logic for its next-state control follows the same basic method as described earlier. To begin with, State 4 isn't the initial state, so it uses a normal D-type flip-flop without the inverters. It does, however, have an asynchronous reset input, three paths into the state, and a default condition that stays in State 4. Therefore, a four-input OR-gate feeds the flip-flop *(OR-1 in Fig. 3)*.

The first conditional path comes from State 3. Following the methods established earlier, an AND of State 3 and the conditional logic, which is A ORed with D, must be implemented *(AND-2 and OR-3 in Fig. 3)*. The next conditional path is from State 2, which requires an AND of State 2 and variable D *(AND-4 in Fig. 3)*. Lastly, the final conditional path leading into State 4 is from State 1. Again, the State-1 output must be ANDed with its conditional path logic—the logical product, $A^*B^*\overline{C}$ *(AND-5 and AND-6 in Fig. 3)*.

Now, all that must be done is to build the logic that remains in State 4 when none of the conditional paths away from State 4 are true. The path leading away from State 4 is valid whenever the product, $A^*B^*\overline{C}$, is true. Consequently, State 4 must be ANDed with the inverse of the product, $A^*B^*\overline{C}$. In other words, "keep loading the flip-flop with a high until a valid transfer to the next state occurs." The default path logic uses AND-7 and shares the output of AND-6.

Configuring the logic to handle the remaining states is very simple. State 2, for example, has only one conditional path, which comes from State 1 whenever the product $A^*\overline{B}^*C$ is true. However, the state machine will immediately branch in one of two ways from State 2, depending on the value of D. There's no default logic to remain in State 2 *(Fig. 4, top)*. State 3, like States 1 and 4, has a default state, and combines the A, D, State 2, and State-3 feedback to control the flip-flop's D input *(Fig. 4, bottom)*.

State 5 feeds State 6 unconditionally. Note that the state machine waits until variable E is low in State 6 before proceeding to State 7. Again, while in State 7, the state machine waits for variable E to return to true before moving to State 1 *(Fig. 5)*.



6. S-R FLIP-FLOPS OFFER ANOTHER approach to decoding the Contig output. They can also save logic blocks, especially when an output is asserted for a long sequence of contiguous states.

OUTPUT DEFINITIONS

After defining all of the state transition logic, the next step is to define the output logic. The three output signals—Single, Multi, and Contig—each fall into one of three primary output types:

1. Outputs asserted during one state, which is the simplest case. The output signal Single, asserted only during State 6, is an example.

2. Outputs asserted during multiple, contiguous states. This appears simple at first glance, but a few techniques exist that reduce logic complexity. One example is Contig. It's asserted from State 3 to State 7, even though there's a branch at State 2.

3. Outputs asserted during multiple, non-contiguous states. The best solution is usually brute-force decoding of the active states. One

ONE-STATE VS. BINARY ENCODING METHODS

Method	Number of logic blocks	Worst-case performance
One-hot	7.5	40 MHz
Binary encoding	7.0	34 MHz

STATE MACHINE DESIGN

such example is Multi, which is asserted during State 2 and State 4.

OHE makes defining outputs easy. In many cases, the state flip-flop is the output. For example, the Single output also is the flip-flop output for State 6; no additional logic is required. The Contig output is asserted throughout States 3 through 7. Though the paths between these states may vary, the state machine will always traverse from State 2 to a point where Contig is active in either State 3 or State 4.

There are many ways to implement the output logic for the Contig output. The easiest method is to decode States 3, 4, 5, 6, and 7 with a 5-input OR gate. Any time the state machine is in one of these states, Contig will be active. Simple decoding works best for this state machine example. Decoding five states won't exceed the input capability of the FPGA logic block.

Additional Logic

However, when an output must be asserted over a longer sequence of states (six or more), additional layers of decoding logic would be required. Those additional logic layers reduce the state machine's performance.

Employing S-R flip-flops gives designers another option when decoding outputs over multiple, contiguous states. Though the basic FPGA architecture may not have physical S-R flip-flops, most macrocell libraries contain one built from logic and D-type flip-flops. Using S-R flip-flops is especially valuable when an output is active for six or more contiguous states.

The S-R flip-flop is set when entering the contiguous states, and reset when leaving. It usually requires extra logic to look at the state just prior to the beginning and ending state. This approach is handy when an output covers multiple, non-contiguous states, assuming there are enough logic savings to justify its use.

In the example, States 3 through 7 can be considered contiguous. Contig is set after leaving State 2 for either States 3 or 4, and is reset after leaving State 7 for State 1. There are no conditional jumps to states where Contig isn't asserted as it traverses from State 3 or 4 to State 7. Otherwise, these states would not be contiguous for the Contig output.

The Contig output logic, built from an S-R flip-flop, will be set with State 2 and reset when leaving State 7 *(Fig. 6)*. As an added benefit, the Contig output is synchronized to the master clock. Obvious logic reduction techniques shouldn't be overlooked either. For example, the Contig output is active in all states except for States 1 and 2. Decoding the states where Contig isn't true, and then asserting the inverse, is another way to specify Contig.

The Multi output is asserted during multiple, non-contiguous states—exclusively during States 2 and 4. Though States 2 and 4 are contiguous in some cases, the state machine may traverse from State 2 to State 4 via State 3, where the Multi output is unasserted. Simple decoding of the active states is generally best for non-contiguous states. If the output is active during multiple, non-contiguous states over long sequences, the S-R flip-flop approach described earlier may be useful.

One common issue in state-machine construction deals with preventing illegal states from corrupting system operation. Illegal states exist in areas where the state machine's functionality is undefined or invalid. For state machines implemented in PAL devices, the state-machine compiler software usually generates logic to prevent or to recover from illegal conditions.

In the OHE approach, an illegal condition will occur whenever two or more states are active simultaneously. By definition, the one-hot method makes it possible for the state machine to be in only one state at a time. The logic must either prevent multiple, simultaneous states or avoid the situation entirely.

Synchronizing all of the state-machine inputs to the master clock signal is one way to prevent illegal states. "Strange" transitions won't occur when an asynchronous input changes too closely to a clock edge. Though extra synchronization would be costly in PAL devices, the flip-flop-rich architecture of an FPGA is ideal.

Even off-chip inputs can be synchronized in the available input flip-flops. And internal signals can be synchronized using the logic block's flip-flops (in the case of the Xilinx LCAs). The extra synchronization logic is free, especially in the Xilinx FPGA family where every block has an optional flip-flop in the logic path.

Resetting State Bits

Resetting the state machine to a legal state, either periodically or when an illegal state is detected, gives designers yet another choice. The Reset Direct (RD) inputs to the flip-flops are useful in this case. Because only one state bit should be set at any time, the output of a state can reset other state bits. For example, State 4 can reset State 3.

If the state machine did fall into an illegal condition, eventually State 4 would be asserted, clearing State 3. However, State 4 can't be used to reset State 5, otherwise the state machine won't operate correctly. To be specific, it will never transfer to State 5; it will always be held reset by State 4. Likewise, State 3 can reset State 2, State 5 can reset State 4, etc.—as long as one state doesn't reset a state that it feeds.

This technique guarantees a periodic, valid condition for the state machine with little additional overhead. Notice, however, that State 1 is never reset. If State 1 were "reset," it would force the output of State 1 high, causing two states to be active simultaneously (which, by definition, is illegal).□

Steve Knapp, new product development manager at Xilinx, spent the last four years as a field applications engineer aiding customers in FPGA designs. He received a BS in materials science and engineering from Massachusetts Institute of Technology, Cambridge, Mass.