One-hot state machine design for FPGAs

Steve Golson
Trilobyte Systems, 33 Sunset Road, Carlisle MA 01741
Phone: 508/369-9669
Email: sgolson@trilobyte.com

***Abstract*: One-hot state machines use one flop per state. They are particularly suited to today's register-rich FPGA architectures. This paper will discuss the advantages of one-hot state machines including ease of design, simple timing analysis, and high clock rates. An SBus master/slave interface will be used as a design example. VHDL and Verilog coding styles will be discussed.**

Introduction

Designing a synchronous state machine is a common task for a digital logic engineer. Usually the most important decision to make when designing a state machine is what state encoding to use. A poor choice of codes will result 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 encoding. Typically such approaches use the minimum number of state bits [7] or assume a two-level logic implementation such as a PLA [2]. Only recently has work been done on the multi-level logic synthesis typical of gate array (and FPGA) design [1].

In the *one-hot encoding* only one bit of the state vector is asserted for any given state. All other state bits are zero. Thus if there are *n* states then *n* state 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 [3],[4]. 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

State machine design for PAL devices generally requires highly-encoded state assignments because of the relatively small number of flops available. Further, the wide AND architecture allows any number of state bits to be included in each product term with no speed (or area) penalty.

Today's logic block FPGA architectures such as QuickLogic or Xilinx do not support such structured design techniques, and instead force more random logic implementations.

A highly-encoded state assignment implemented in such an FPGA will use fewer flops for the state vector, however additional logic blocks will be required simply to encode and decode the state. Since one or more flops are included in each logic block anyway, the total number of logic blocks used may in fact be larger for the highly-encoded case than in the one-hot case.

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

• Maps easily into register-rich FPGA architectures such as QuickLogic and Xilinx.

• 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.

• 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. Schematics can be captured and 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 no area penalty over highly-encoded machines.

• Critical paths are easy to find using static timing analysis.

• Easy to debug. Bogus state transitions are obvious, and current state display is trivial.

Example design

SBus is a synchronous IO expansion bus that supports 32-bit and 64-bit transfer widths. Both single and burst transfers are supported, allowing data rates of up to 168 Mbytes/sec. SBus devices may serve as masters on the bus, or as slaves providing data transfer in response to some other master, or both. All signals are generated and sampled with respect to CLK.

SBus was first developed by Sun Microsystems [8] and is now in the final stages of becoming an IEEE standard [5]. Lyle [6] gives a very useful introduction to SBus.

Figure 1 shows a simplified state diagram for an SBus device with both master and slave capability.

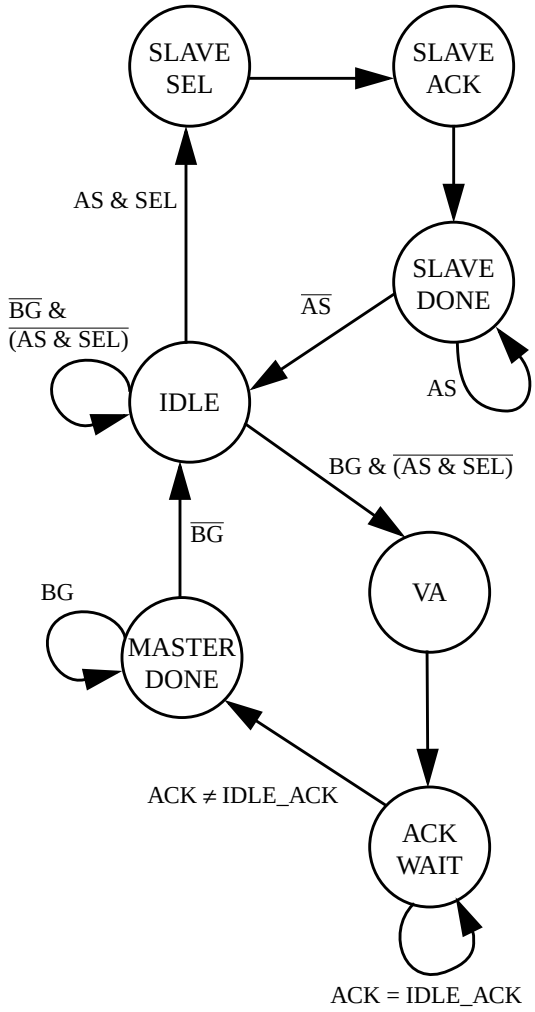

Figure 1 -- State diagram

While in state IDLE, if AS and SEL are asserted then the device is being accessed as a slave. During state SLAVE_ACK the appropriate acknowledgment is returned to the master, either indicating that the requested data has been transferred, or asking that the transfer be retried, or signifying an error. In state SLAVE_DONE the slave waits for AS to be negated indicating the end of the access.

Alternatively if BG is asserted while in IDLE then the device has been granted the bus for a master access. In state VA the virtual address is driven by the master indicating which data is to be accessed. In ACK_WAIT the master waits for an acknowledgment from the selected slave, and then in MASTER_DONE waits for BG to be negated indicating the access has completed.

Schematic capture

An advantage of one-hot encoding is that the logic design may be captured directly from the state diagram without first requiring the generation of a state table.

Each state in the state diagram has a corresponding flop in the state vector. The output of that flop is given the name of the corresponding state. To determine when any given transition is taken we AND the transition condition with the signal representing the previous state. Then the D input of a state flop is the logical OR of all possible transitions which have that state as their destination.

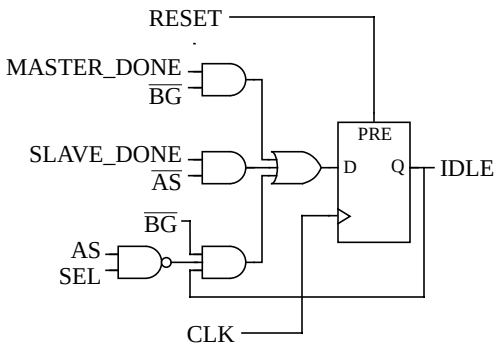

Figure 2 -- Schematic for IDLE state flop

Consider the IDLE state in Figure 1. There are three transitions into IDLE: from MASTER_DONE if BG is negated, from SLAVE_DONE if AS is negated, and from IDLE itself. Thus the D input of the IDLE flop is driven by a three-input OR gate with each input representing one of the possible transitions. Figure 2 shows the schematic for the IDLE state flop.

When RESET is asserted we want the machine to enter the IDLE state. Therefore the IDLE flop is preset when RESET is asserted. All other state flops will be cleared by RESET.

Using a QuickLogic FPGA allows almost all of the circuit of Figure 2 to be mapped into a single logic cell. The remaining logic (the NAND) can be incorporated into a logic block whose state flop requires little or no input logic (e.g. SLAVE_ACK) or into a logic block that requires a similar gate anyway (e.g. SLAVE_SEL). The area required to implement a one-hot machine with *n* states is typically close to *n* logic blocks. This allows for easy area estimation early in the design process.

Logic synthesis

Logic synthesis has recently become a popular way to design FPGAs. As available gate counts have climbed, traditional PLD schematic capture or PALASM descriptions have become too limiting. Further, using a hardware description language such as Verilog or VHDL allows some measure of vendor independence, facilitates system-level behavioral modeling, and more easily supports the translation of FPGA designs into gate arrays. Traditional gate array synthesis tools such as Synopsys have added more support for FPGA architectures, and FPGA-specific synthesis tools such as Exemplar have become available.

One-hot state machines may be easily described using either Verilog or VHDL. Using the proper coding style can result in logic approaching the quality of hand-captured schematics.

Some vendors have state machine optimization tools which will automatically extract a state machine from an HDL description, and pick an "optimal" state encoding. The results may not be as good as coding for one-hot to begin with. Further, the techniques outlined here require no non-standard HDL extensions, and thus should work well with any synthesis tool.

VHDL examples

The "classic" VHDL description of our example state machine first defines the states as an enumeration type:

```
type STATE_TYPE is
  (IDLE, SLAVE_SEL, SLAVE_ACK,
   SLAVE_DONE, VA, ACK_WAIT,
   MASTER_DONE) ;
```

Then the main state machine logic is implemented with a `case` statement:

```
case STATE is
 when IDLE =>
  if (AS = '1' and SEL = '1') then
   NEXT_STATE <= SLAVE_SEL ;
  elsif (BG = '1') then
   NEXT_STATE <= VA ;
  else
   NEXT_STATE <= IDLE;
   end if;

 when SLAVE_SEL =>
  NEXT_STATE <= SLAVE_ACK ;

 when SLAVE_ACK =>
  NEXT_STATE <= SLAVE_DONE ;

 when SLAVE_DONE =>
  if (AS = '1') then
   NEXT_STATE <= SLAVE_DONE ;
  else
   NEXT_STATE <= IDLE ;
   end if ;

 when VA =>
  NEXT_STATE <= ACK_WAIT ;

 when ACK_WAIT =>
  if (ACK = IDLE_ACK) then
   NEXT_STATE <= ACK_WAIT ;
  else
   NEXT_STATE <= MASTER_DONE ;
   end if ;

 when MASTER_DONE =>
  if (BG = '1') then
   NEXT_STATE <= MASTER_DONE ;
  else
   NEXT_STATE <= IDLE ;
   end if ;
 end case;
```

Finally, the state flops are inferred from an `if` statement:

```
if (RESET = '1') then
 STATE <= IDLE ;
elsif (CLK'event and CLK = '1') then
 STATE <= NEXT_STATE ;
 end if ;
```

The first enumeration literal in the `type` statement is given the value 0, the next literal is assigned 1, and so forth. Thus we have a highly-encoded state assignment using the minimum number of state bits, where the encoding can be modified only by changing the order of enumeration.

So how can we describe a one-hot encoding? If we are using Synopsys synthesis tools it is tempting to use the ENUM_ENCODING attribute:

```
attribute ENUM_ENCODING of
 STATE_TYPE: type is
  "0000001 0000010 0000100 0001000
   0010000 0100000 1000000" ;
```

Although this does result in a one-hot encoding, the synthesis tool may not recognize all the don't care conditions. Some unnecessary state decode logic may be built, except now using wider gates, which is exactly the opposite of the desired result! Furthermore the ENUM_ENCODING attribute is a non-standard extension of VHDL and is not portable to VHDL simulators or other synthesis tools.

A better way is to use a coding style that directly gives the one-hot results we want. Such a style is illustrated in Listing 2 which shows a complete VHDL description of our example state machine.

Listing 2 looks superficially similar to the "classic" version above. The major difference is the use of a sequence of `if` statements rather than a single `case`. The statements within each `if` are identical to the statements within the `case`, except that the assignment to `NEXT_STATE` affects only the single bit representing the destination state. All other bits of `NEXT_STATE` will be assigned to zero by the default

```
NEXT_STATE <= ZERO_STATE ;
```

statement at the beginning of the process.

Verilog examples

The same coding style can be used in Verilog. An initial assignment of `next_state` to zero is followed by a series of `if` statements. Listing 1 shows a Verilog description of the example design.

Timing analysis

Static timing analysis may be used to speed up a slow design. Once the slowest state transition is found, the machine may be sped up by changing the state diagram or modifying the input dependencies.

A static timing report on a highly-encoded machine may show that the critical path (the slowest path) is from `state[3]/Q` falling to `state[0]/D` falling. It is difficult to determine exactly which state transition this corresponds to. Worse, it may actually be a false path that doesn't correspond to any legal transition.

A one-hot machine is much easier to analyze. In our example we might get a critical path from `state[6]/Q` rising to `state[0]/D` rising. Since bit 6 corresponds to MASTER_DONE and bit 0 to IDLE, this path occurs when MASTER_DONE is entered and we transition to IDLE on the next clock. This transition depends on input BG.

Other design considerations

The transitions leaving a given state must be mutually exclusive and all inclusive [9]. If more than one transition is active then more than one state bit will be set. If the transitions are not all inclusive then there is some value of inputs which will result in *no* state bits being set (i.e. all will be cleared). Both conditions are illegal states.

Thus, when drawing schematics directly from the state diagram be sure that all possible input conditions for each state are correctly specified.[1] If you are using an HDL this is easy to guarantee. In any given state an `if-elsif-else` construct is used. Mutual exclusion is ensured by asserting only one bit of `NEXT_STATE` in any `if` branch. All inclusion is ensured by having a default `else` branch. See Listing 2 for examples.

Illegal states may also be entered because of poorly synchronized inputs, illegal input combinations, or hardware failures. One technique for recovering from illegal states is to use an asserted state bit to synchronously reset *other* state bits. In our example we could use SLAVE_SEL to reset all bits except SLAVE_ACK (don't reset a state bit which may possibly be the next state). In our VHDL example we can implement this by adding:

```
if (STATE(SLAVE_SEL) = '1') then
 NEXT_STATE(IDLE) = '0' ;
 NEXT_STATE(SLAVE_SEL) = '0' ;
 NEXT_STATE(SLAVE_DONE) = '0' ;
 NEXT_STATE(VA) = '0' ;
 NEXT_STATE(ACK_WAIT) = '0' ;
 NEXT_STATE(MASTER_DONE) = '0' ;
 end if ;
```

just before the first `end process` in Listing 2.

Another illegal state is the all zeroes condition. A wide (but slow) NOR gate may be used to detect this condition and set the IDLE state bit.

All of these techniques will provide some measure of recovery from illegal states, but cost area and

1. This is one situation where it might be better to use a state table rather than a state diagram.

speed, thus negating the very strengths of one-hot encoding. If such a minimal risk design is required by your application then perhaps a larger, slower, more highly-encoded state machine should be used.

Acknowledgments

Thanks to QuickLogic for providing access to design tools. Thanks to John F. Wakerly for finding the Huffman references. Thanks to Michiel Ligthart of Exemplar for synthesizing the VHDL examples.

References

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

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

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

[5] IEEE, *Standard for a Chip and Module Interconnect Bus: SBus*, P1496, Draft 2.3 of January 11, 1993.

[6] James D. Lyle, *SBus: Information, Applications, and Experience*, Springer-Verlag, 1992.

[7] 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.

[8] Sun Microsystems, Inc., *SBus Specification B.0*, Part Number 800-5922-10, Revision A of December 1990.

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

Listing 1 -- Partial Verilog code for one-hot SBus state machine example

```
always @ (state or BG or AS or
          SEL or ACK)
 begin

  next_state = `ZERO_STATE ;

  if (state[`IDLE]) begin
   if (AS && SEL)
    next_state[`SLAVE_SEL] = 1'b1 ;
   else if (BG)
    next_state[`VA] = 1'b1 ;
   else
    next_state[`IDLE] = 1'b1 ;
   end

  if (state[`SLAVE_SEL])
   next_state[`SLAVE_ACK] = 1'b1 ;

  if (state[`SLAVE_ACK])
   next_state[`SLAVE_DONE] = 1'b1 ;

  if (state[`SLAVE_DONE]) begin
   if (AS)
    next_state[`SLAVE_DONE] = 1'b1 ;
   else
    next_state[`IDLE] = 1'b1 ;
   end

  if (state[`VA])
   next_state[`ACK_WAIT] = 1'b1 ;

  if (state[`ACK_WAIT]) begin
   if (ACK == `IDLE_ACK)
    next_state[`ACK_WAIT] = 1'b1 ;
   else
    next_state[`MASTER_DONE] = 1'b1 ;
   end

  if (state[`MASTER_DONE]) begin
   if (BG)
    next_state[`MASTER_DONE] = 1'b1 ;
   else
    next_state[`IDLE] = 1'b1 ;
   end
  end

always @ (posedge CLK or posedge RESET)
 begin
  if (RESET) begin
   state = `ZERO_STATE ;
   state[`IDLE] = 1'b1 ;
   end
  else
   state = next_state ;
  end
```

Listing 2 -- Complete VHDL code for one-hot SBus state machine

```
package TYPES is
 constant IDLE:        NATURAL := 0 ;
 constant SLAVE_SEL:   NATURAL := 1 ;
 constant SLAVE_ACK:   NATURAL := 2 ;
 constant SLAVE_DONE:  NATURAL := 3 ;
 constant VA:          NATURAL := 4 ;
 constant ACK_WAIT:    NATURAL := 5 ;
 constant MASTER_DONE: NATURAL := 6 ;

 subtype STATE_TYPE is
  BIT_VECTOR (6 downto 0) ;

 constant ZERO_STATE:
  STATE_TYPE := "0000000" ;

 type ACK_TYPE is (IDLE_ACK, ERROR_ACK,
  BYTE_ACK, RETRY_ACK, WORD_ACK,
  DOUBLE_ACK, HALF_ACK, RESERVED_ACK) ;

 type SIZ_TYPE is (WORD_SIZ, BYTE_SIZ,
  HALF_SIZ, EXTENDED_SIZ, BURST4_SIZ,
  BURST8_SIZ, BURST16_SIZ, BURST2_SIZ);

 end ;

use WORK.TYPES.ALL ;

entity ONE_HOT is
 port
  (CLK, RESET, BG, AS, SEL : in BIT ;
   ACK : in ACK_TYPE ;
   STATE : buffer STATE_TYPE );
 end;

architecture BEHAVIOR of ONE_HOT is
 signal NEXT_STATE: STATE_TYPE ;
begin
 process(STATE, BG, AS, SEL, ACK)
 begin

  NEXT_STATE <= ZERO_STATE ;

  if (STATE(IDLE) = '1') then
   if (AS  = '1' and SEL  = '1') then
    NEXT_STATE(SLAVE_SEL) <= '1' ;
   elsif (BG = '1') then
    NEXT_STATE(VA) <= '1' ;
   else
    NEXT_STATE(IDLE) <= '1' ;
    end if ;
   end if ;

  if (STATE(SLAVE_SEL) = '1') then
   NEXT_STATE(SLAVE_ACK) <= '1' ;
   end if ;

  if (STATE(SLAVE_ACK) = '1') then
   NEXT_STATE(SLAVE_DONE) <= '1' ;
   end if ;

  if (STATE(SLAVE_DONE) = '1') then
   if (AS = '1') then
    NEXT_STATE(SLAVE_DONE) <= '1' ;
   else
    NEXT_STATE(IDLE) <= '1' ;
    end if ;
   end if ;

  if (STATE(VA) = '1') then
   NEXT_STATE(ACK_WAIT) <= '1' ;
   end if ;

  if (STATE(ACK_WAIT) = '1') then
   if (ACK = IDLE_ACK) then
    NEXT_STATE(ACK_WAIT) <= '1' ;
   else
    NEXT_STATE(MASTER_DONE) <= '1' ;
    end if ;
   end if ;

  if (STATE(MASTER_DONE) = '1') then
   if (BG = '1') then
    NEXT_STATE(MASTER_DONE) <= '1' ;
   else
    NEXT_STATE(IDLE) <= '1' ;
    end if ;
   end if ;
  end process;

process(CLK, RESET, NEXT_STATE)
begin
 if (RESET = '1') then
  STATE <= ZERO_STATE ;
  STATE(IDLE) <= '1' ;
 elsif (CLK'event and CLK = '1') then
  STATE <= NEXT_STATE ;
  end if ;
 end process;
end BEHAVIOR;
```