Application Note

Synplicity

Designing Safe VHDL State Machines with Synplify

Introduction

One of the strengths of Synplify is the Finite State Machine compiler. This is a powerful feature that not only has the ability to automatically detect state machines in the source code, and implement them with either sequential, gray, or one-hot encoding. But also perform a reachability analysis to determine all the states that could possibly be reached, and optimize away all states and transition logic that can not be reached. Thus, producing a highly optimal final implementation of the state machine.

In the vast majority of situations this behavior is desirable. There are occasions, however, when the removal of unreachable states is not acceptable. One clear example is when the final circuit will be subjected to a harsh operating environment, such as space applications where there may be high levels of radiation. In the presence of high levels of radiation, storage elements (flip-flops) have been known to change state due to alpha particle hits. If a single bit of a state register were to suddenly change value, the resulting state may be invalid. If the invalid states and transition logic had been removed, the circuit may never get back to a valid state.

By default Synplify will create state machines that are optimized for speed and area. This application note will use an example state machine design to show the default small & fast implementation. It will also demonstrate how to trade-off some of that speed & area to produce highly reliable state machines using Synplify.

Example 1:
Assume that the transition diagram in figure 1 is to be implemented as a one-hot FSM:

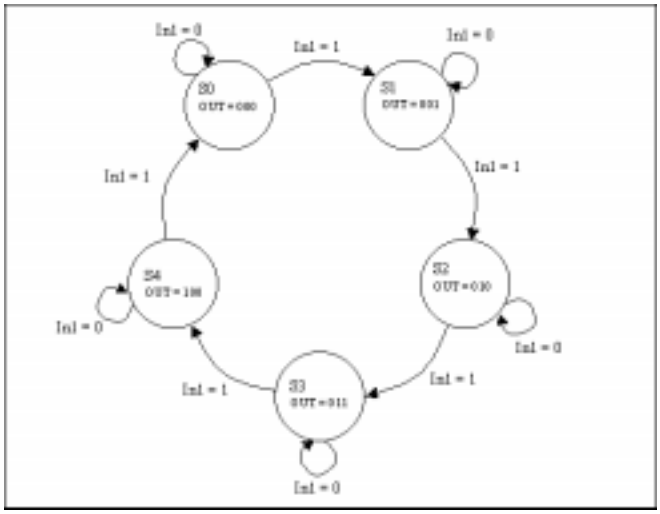

Figure 1.

One possible RTL implementation would be:

```
library ieee;
use ieee.std_logic_1164.all;

entity FSM1 is
     port(clk, in1, rst: in std_logic;
            out1: out std_logic_vector(2 downto 0));
end FSM1;

architecture RTL of FSM1 is
     type state_values is (s0, s1, s2, s3, s4);
     signal state, next_state: state_values;
     attribute syn_encoding : string;
     attribute syn_encoding of state : signal is "onehot";
begin

process (clk, rst)
begin
     if rst = '1' then
           state <= s0;
     elsif rising_edge(clk) then
           state <= next_state;
     end if;
end process;

process (state, in1)
     begin
           case state is
                when s0 =>
                     out1 <="000";
                     if in1 = '0' then
                           next_state <= s0;
                     else
                           next_state <= s1;
                      end if;
                when s1 =>
                     out1 <="001";
                     if in1 = '0' then
                           next_state <= s1;
                     else
                           next_state <= s2;
                      end if;
                when s2 =>
                      out1 <="010";
                     if in1 = '0' then
                           next_state <= s2;
                     else
                           next_state <= s3;
```



```
                    end if;
               when s3 =>
                    out1 <="011";
                     if in1 = '0' then
                          next_state <= s3;
                    else
                          next_state <= s4;
                     end if;
               when s4 =>
                    out1 <="100";
                    if in1 = '0' then
                          next_state <= s4;
                    else
                          next_state <= s0;
                    end if;
               when others =>
                    out1 <= "000";
                    next_state <= s3;
          end case;
     end process;
end RTL;
```

Note:

1. The “syn_encoding” attribute is used to specify that this state machine should be encoded as one-hot.
2. There are 5 defined states (S0, S1, S2, S3, and S4), all of which are reachable.
3. Since the encoding style is one-hot, there are 27 undefined (and unreachable) states that are covered by the “others” branch of the case statement.
4. The state register resets to state S0.
5. The “others” case specifies a transition to state S3. Keep in mind that this circuit will never reach the “others” branch without some external influence such as an alpha particle hit or a physical defect in the target part.
6. The syn_encoding attribute directs the FSM compiler to implement this design as a one-hot state machine. The final circuit will have the state encodings: S0 = 00001, S1 = 00010, S2 = 00100, S3 = 01000, S4 = 10000.
7. The material covered in this application note applies to all supported encoding styles, one-hot, sequential, and gray.

Default Implementation:

If Synplify is used to synthesize this design as is, the result is an optimized state machine with the transition logic for unreachable states removed. The final implementation is basically a shift register. Where the state register resets to the 00001 state (S0), and the output of state bit 4 is the input to state bit 3, the output of state bit 3 is the input to state bit 2, and so on. This is shown in figure 2 using the Technology View in HDL Analyst.

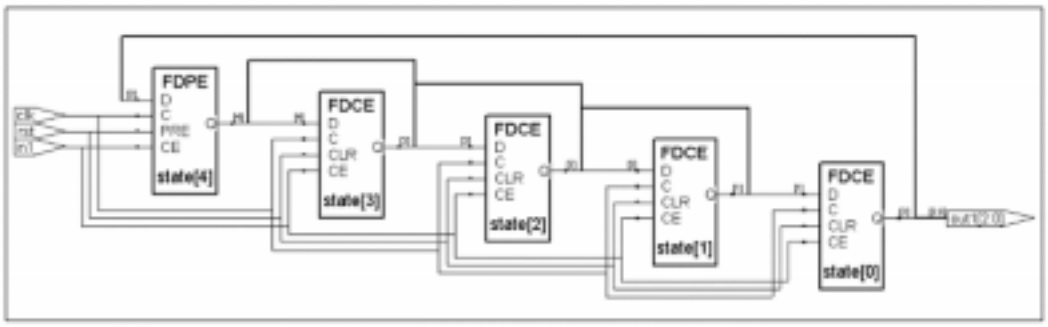

Fig 2

This is a very optimal result for both timing and area. In a normal operating environment this circuit will function perfectly. Suppose, however, that this circuit is to be placed in a hostile operating environment where a register could spontaneously change value due to an alpha particle hit, or some other reason. What would happen if this state machine ended up in the 00000 state? The next transition would shift all the state bits resulting in the state 00000. The result being that this FSM would effectively be stuck in the 00000 state.

"Safe" Implementation:
To handle this type of problem, the FSM compiler in Synplify has a special encoding directive, "safe", that will add logic such that if the state machine should ever reach an invalid state, it will be forced to the reset state. This behavior has the advantage of avoiding any possible "hang" conditions, where the state machine is unable to get back to a valid state, while having minimal impact on the timing of the circuit.

To enable this feature simply change the value of the syn_encoding attribute from:

```
attribute syn_encoding of state : signal is "onehot";
```

to:

```
attribute syn_encoding of state : signal is "safe,onehot";
```

Note: The syn_encoding attribute can also be applied in the SCOPE graphical constraint editor or directly in the constraint (.sdc) file. Using the following syntax:

```
define_attribute {state[*]} syn_encoding {safe,onehot}
```


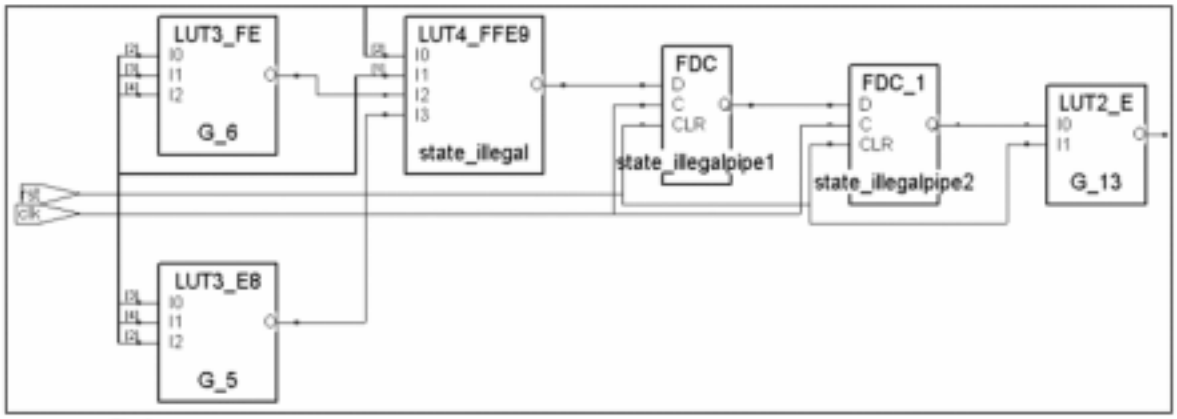

Fig 3

Synthesizing this design will result in a circuit that has the state transition logic implemented exactly as shown in figure 2 above, with the addition of the circuitry in figure 3 added to the reset logic.

If an invalid state is detected, the state_illegalpipe1 register is set on the next rising clock edge. On the falling edge of the clock, the state_illegalpipe2 register is set. Instance G_13 ORs the original reset signal “rst” with the new recovery logic. The output of instance G_13 drives the clear/preset pins of the state bits, forcing the circuit to the (valid) reset state. Once this valid state is reached, the next rising edge of the clock will clear the state_illegalpipe1 register, the next falling edge of the clock will clear the state_illegalpipe2 register and normal operation will begin. Note that the result of this recovery logic, the output of state_illegalpipe2, is registered on the falling edge of the clock to prevent any harzardous conditions that could result from removing the reset signal too close to the active clock edge of the state registers.

The recovery logic discussed above is generated for the example circuit which happens to have an asynchronous reset. If the circuit had a synchronous reset instead, the logic implemented would be slightly different. Suppose the register definition was changed from:

```
       process (clk, rst)
       begin
          if rst = '1' then
              state <= s0;
          elsif rising_edge(clk) then
              state <= next_state;
          end if;
       end process;
to:
       process (clk)
       begin
            if rising_edge(clk) then
```

```
            if rst = '1' then
                 state <= s0;
            else
                 state <= next_state;
            end if;
       end if;
  end process;
```

For this synchronous reset implementation, the circuitry in figure 4 would be added to the reset logic:

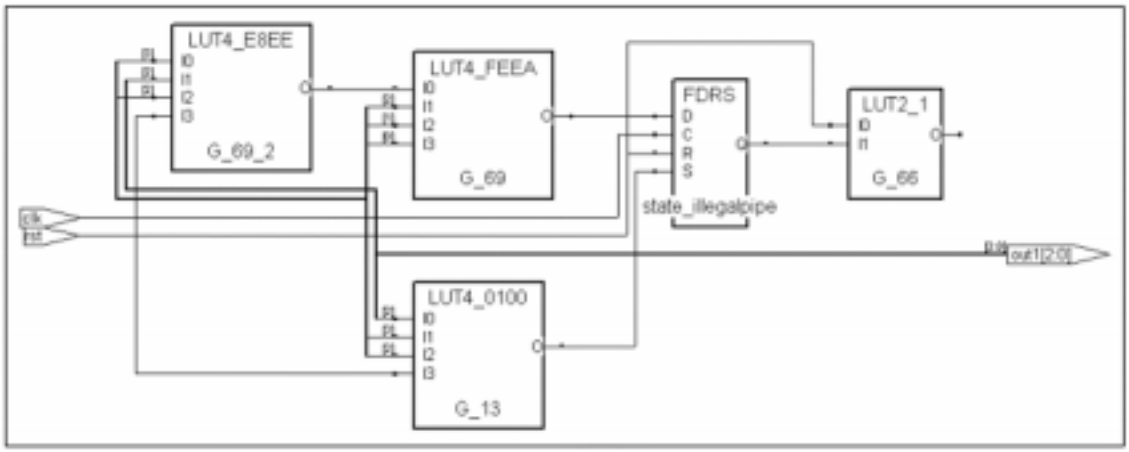

Fig 4

If an invalid state is detected, the state_illegalpipe register is set on the next rising clock edge. Instance G_66 ORs the original reset signal “rst” with the new recovery logic. On the next positive clock edge, the state register will switch to the (valid) reset state. Once this valid state is reached, the next rising edge of the clock will clear the state_illegalpipe register, and normal operation will begin.

In both the asynchronous and synchronous reset case, if the circuit should ever reach an invalid state (state 00000 for example), the recovery logic will be activated reseting the state register back to the 00001 state (S0). Once the FSM is back to the valid state of 00001 (S0), normal operation of the state machine can resume. Notice that upon entering an invalid state this circuit will recover to the 00001 state (S0) not the 01000 state (S3) as described in the “others” branch of the case statement.

This implementation eliminates the possibility of the state machine getting “stuck” in an invalid state and not returning to a valid state. This problem is handled with very minimal impact on the timing of the circuit. However, as pointed out above, the transition out of an invalid state is not implemented exactly as described in the “others” branch of the source code. This deviation from the defined “others” branch behavior only occurs for invalid states. If the “others” case contained any valid state transitions they would be implemented as described in the source code.

"Exact" Implementation:

It is possible to get an implementation of the circuit that fully implements the "others" branch if it is necessary to do so. This requires disabling the reachability analysis of the state machine, which is done by turning off the FSM compiler, defining state codes as constants instead of enumerated types. This can have a significant affect on the area and timing of the circuit.

To get a full implementation of the "others" case change the state register description from:

```
type state_values is (s0, s1, s2, s3, s4);
            signal state, next_state: state_values;
            attribute syn_encoding : string;
            attribute syn_encoding of state : signal is "onehot";
```

to:

```
signal state, next_state: std_logic_vector(4 downto 0);
       constant s0 : std_logic_vector(4 downto 0) := "00001";
       constant s1 : std_logic_vector(4 downto 0) := "00010";
       constant s2 : std_logic_vector(4 downto 0) := "00100";
       constant s3 : std_logic_vector(4 downto 0) := "01000";
       constant s4 : std_logic_vector(4 downto 0) := "10000";
       attribute syn_preserve : boolean;
       attribute syn_preserve of state : signal is true;
```

Note:

1. The state register is defined as a std_logic_vector rather than an enumerated type.
2. The state encodings have been defined as constants. In this case the states are defined such that they implement a one-hot state machine. Any other encoding would work just as well.
3. A syn_preserve attribute is applied to the state register to disable the FSM compiler.
4. The syn_encoding attribute is no longer needed because the FSM compiler disabled.
5. The rest of the code remains unchanged.

Figure 5 uses the RTL view of HDL Analyst to show that the "others" case is fully implemented.

The instances next_state14, next_state13, next_state12, next_state11, and next_state10 decode the current state (S4, S3, S2, S1, S0 respectively). The instances un13, un14, and un20 implement the next state logic for state S0 (bit 0 of the state register). The function is ((~In1 & S0) | (In1 & S4)). The function for state bits 1, 2, and 4 are very similar. Bit 3, however, has an extra term generated by instance un16. This term checks if the FSM is currently in a valid state. If so, the function ((~In1 & S3) | (In1 & S2)) is used. If not, bit 3 is forced high making the next state 01000 (S3) as described in the "others" branch of the original source code.

Summary:

To summarize, Synplify contains a powerful FSM compiler which by default will produce state machine implementations that are highly optimial in regards to area and timing. If recovery from an invalid state is important the “safe” feature can be used to force the state machine to the reset state if an invalid state is reached, with minimal impact on timing and area of the circuit. This implementation of transitioning out of an invalid state may differ from what is explicitly described in the source code. For most designs this is an acceptable deviation, since these transitions are by definition not valid. If these invalid state transitions must be handled exactly as described by the source code, the FSM compiler can be disabled. However, this may result in a substantial impact on timing and area.

To quantify the impact on timing and area, the three implementations of this state machine were synthesized targeting an Altera Flex10k part and a Xilinx Virtex part. The estimated timing and area results reported by Synplify are displayed in table 1 below.

Target	Altera Flex10k – EPF10K10A-1			Xilinx Virtex – XCV50-4		
	ns	LCs	Regs	ns	LUTs	Regs
Default	4.4	8	5	5.1	2	5
Safe	6.7	14	7	7.2	6	7
Full Others	11.4	18	5	12.7	17	5

Tab 1

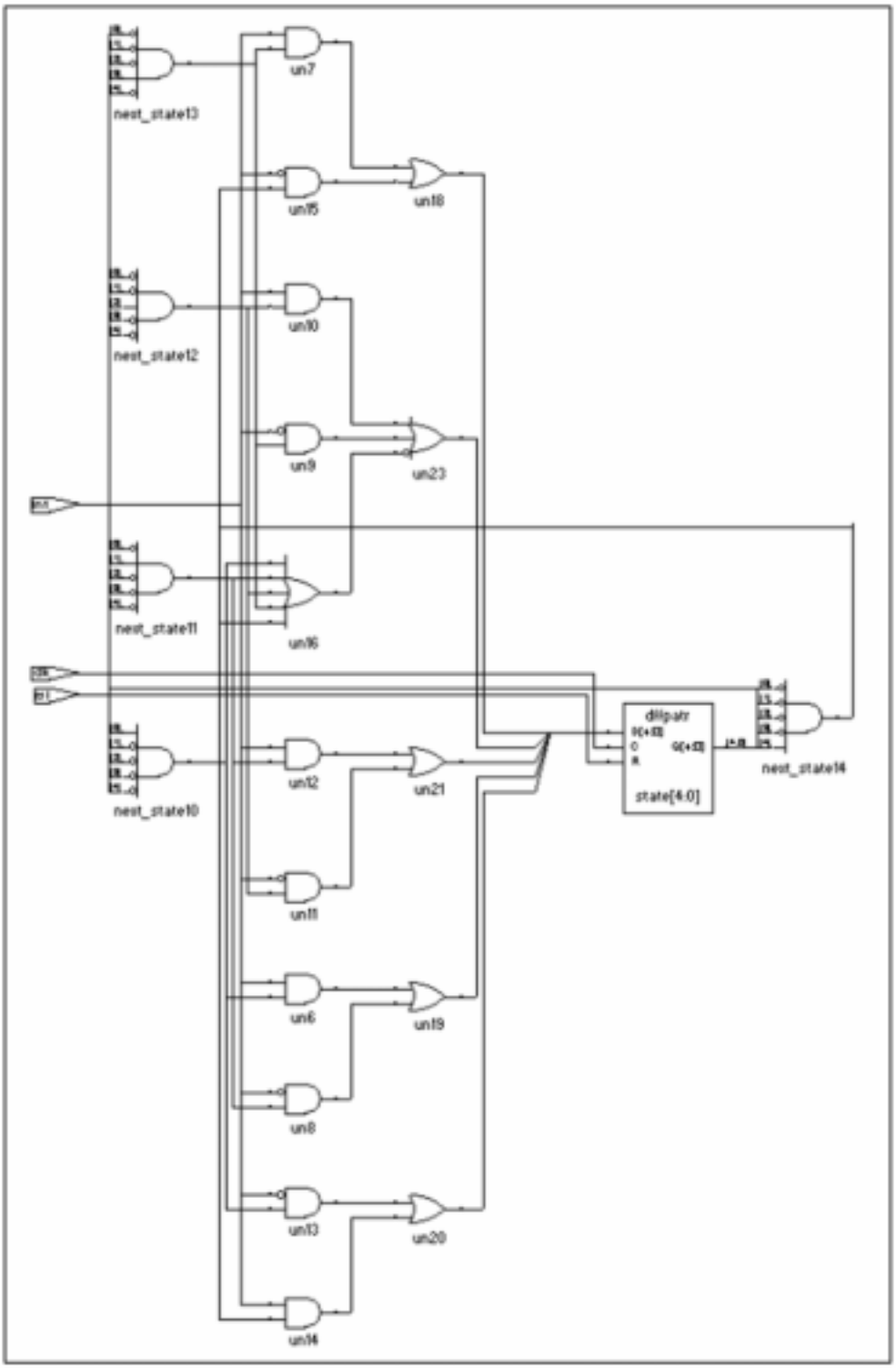
un7
nest_state13
un15
un18
un10
nest_state12
un9
un23
nest_state11
un16
un12
un21
nest_state10
dlip.str
state[4:0]
nest_state14
un11
un6
un19
un8
un13
un20
un14

Fig 5