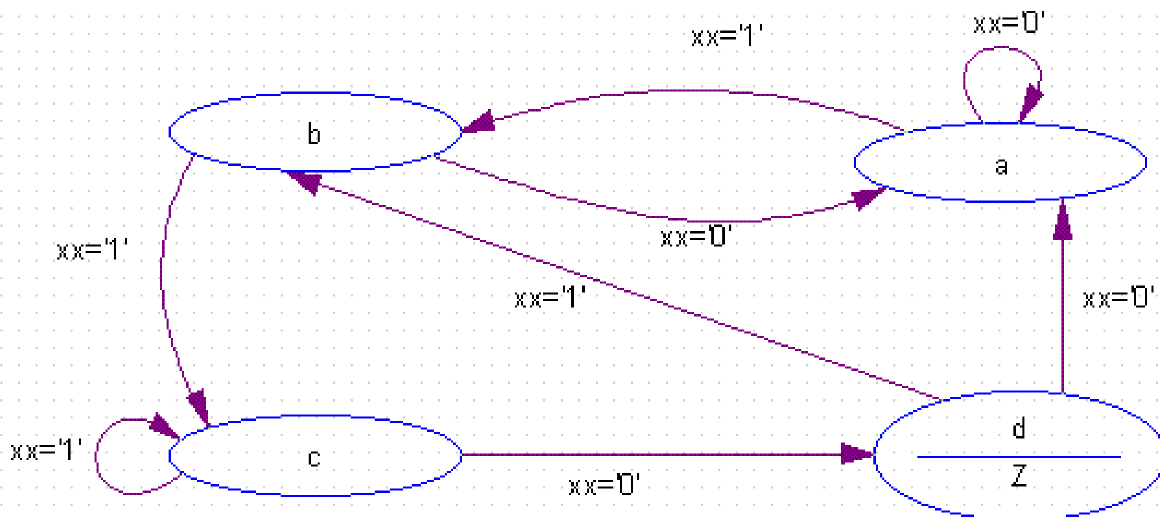
INTRODUCTION TO STATE MACHINES
AND
MORE VHDL
AND
STATECAD SCHEMATIC CAPTURE

This set of notes will deal only with the mechanics of Finite State Machine (FSM) design. The why's and how's will be the subject of other notes. For this reason we will start with the state diagram for on page 519 of the Sandige text (Figure E9-3). This diagram is reproduced, in approximately the same form, below.

This figure was drawn using Statecad Version 3.2[1] (It was captured using a HiJacks image grabber). It differs from the diagram in the machine in several practical aspects. In states a, b, and c, output Z= 0. Modern practice is to list outputs only in states (such as d) where they are asserted. The input, X, has been replaced with XX since X is a reserved symbol in Statecad.



Well, this is a nice diagram but what good is it? The purpose of StateCad is twofold. It can be used to simulate this machine (an initial simulation) AND it can be used to create a VHDL (or Verilog or ABEL or . . . module). The VHDL created by this is shown below. Good

[1]This may be downloaded from my website. It is a demo version. The instructions would leave you to believe it can handle only five states. Actually, it can handle more but you won't be able to save the file so - be sure you've got what you want before you exit the program!

news - you don’t even have to write (much) VHDL in doing state machine design! We shall break the module up into parts for easier discussion.

```
--  F:\FSM01\FIG93.VHD
--  VHDL code created by Visual Software Solution's StateCAD Version 3.2
--  Wed Oct 22 22:34:53 1997
--  This VHDL code (for use with Synario) was generated using:
--  binary encoded state assignment with structured code format.
--  Minimization is enabled,  implied else is disabled,
--  and outputs are manually optimized.
```

The above lines, you may have guessed, are all comments. It seems that every computer language has its own way of doing comments. In VHDL it is two dashes –. Comments extend to the end of the line and do not continue to the next so there is no symbol to close a comment. We shall not discuss the significance of the comments here other than to say that Synario was used because StateCad doesn’t have a WARP selection.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
```

There is nothing more to say here. These are the usual library statements.

```
ENTITY FIG93 IS
     PORT (CLK,xx: IN std_logic;
          Z : OUT std_logic);
END;
```

There is, likewise not much to say here. This is a standard **ENTITY** description with clk and xx as inputs and Z as the output.

```
ARCHITECTURE BEHAVIOR OF FIG93 IS
     SIGNAL sreg : std_logic_vector (1 DOWNTO 0);
     SIGNAL next_sreg : std_logic_vector (1 DOWNTO 0);
     CONSTANT a : std_logic_vector (1 DOWNTO 0) :="00";
     CONSTANT b : std_logic_vector (1 DOWNTO 0) :="01";
     CONSTANT c : std_logic_vector (1 DOWNTO 0) :="10";
     CONSTANT d : std_logic_vector (1 DOWNTO 0) :="11";
```

Some comments are in order here. First, the declaration **SIGNAL** we have used before. It is used to define logic signals internal to the module. State variables are usually internal signals. For a machine with four states, two state variables are needed (You could have guessed that from Figure E9-3, page 518). Two sets of variables are needed for reasons to be seen below. One is for the PRESENT STATE of the machine and the other is used in determining the NEXT STATE of the machine.

A new declaration, **CONSTANT**, is used here. It means what it says. It simply says that we wish to use the letter “a” to represent the binary number 00. The same use is made for b, c, and d, the names of the states. First, this is a convenience - we can use a, b, c, and d in the rest of the module rather than remembering what the actual state assignments are. More important, if this state assignment does not work out well, we only have to change the above

constant declarations and recompile the design.

```
BEGIN
    PROCESS (CLK, next_sreg)
    BEGIN
        IF CLK='1' AND CLK'event THEN
            sreg <= next_sreg;
        END IF;
    END PROCESS;
```

Above is the first of several processes described in the module. By now you should be able to interpret this process. It says that, on the next rising edge of the clock (CLK), the contents on the state register are to be replaced with the contents of the next state register. In other words, the machine is to advance to the next state. Now, what determines what the next state will be?

```
    PROCESS (sreg, xx)
    BEGIN
        Z <= '0';
        next_sreg<=a;
```

The first part of this process just initializes the next state to state a with the output set to 0. so let's go on and see what happens next. The part that follows next is a CASE statement. Nearly all languages have CASE statements (although they don't all operate in the same manner). ABEL has a "STATE_DIAGRA" statement which is a simplified version of a CASE statement. VHDL is no exception.

```
        CASE sreg IS
            WHEN a =>
                Z<='0';
                IF ( xx='1' ) THEN
                    next_sreg<=b;
                END IF;
                IF ( xx='0' ) THEN
                    next_sreg<=a;
                END IF;
            WHEN b =>
                Z<='0';
                IF ( xx='1' ) THEN
                    next_sreg<=c;
                END IF;
                IF ( xx='0' ) THEN
                    next_sreg<=a;
                END IF;
            WHEN c =>
                Z<='0';
```

```
                IF ( xx='0' ) THEN
                    next_sreg<=d;
                END IF;
                IF ( xx='1' ) THEN
                    next_sreg<=c;
                END IF;
            WHEN d =>
                Z<='1';
                IF ( xx='0' ) THEN
                    next_sreg<=a;
                END IF;
                IF ( xx='1' ) THEN
                    next_sreg<=b;
                END IF;
            WHEN OTHERS =>
        END CASE;
    END PROCESS;
END BEHAVIOR;
```

Lets just take on case and examine it in detail.

```
            WHEN a =>
                Z<='0';
                IF ( xx='1' ) THEN
                    next_sreg<=b;
                END IF;
                IF ( xx='0' ) THEN
                    next_sreg<=a;
                END IF;
```

The first line is the case index - the present state is state a. In state a Z is 0. Depending on the value of the input, xx, the next state is either a or b. This information is placed in the next state register. This, then, is what is loaded into the state register on the rising edge of the clock. In this particular instance, the IF .. THEN .. ELSE form could have have been used but StateCad in order to be more flexible, like most software, uses the slight more cumbersome construction using two IF statements.

```
            WHEN OTHERS =>
        END CASE;
```

The case statement, like most, has a WHEN OTHERS clause to take care of cases not matched by any of the previous statements. Note to you C programmers - this is not like the C SWITCH statement where all of the clauses following the selected clause are also executed.

When this is compiled - exactly as prepared by StateCad, the following results are obtained (in addition to the JEDEC file for the 22V10).

```
DESIGN EQUATIONS          (22:41:54)

   z =  sreg_1.Q * sreg_0.Q

   sreg_1.D =  /sreg_1.Q * sreg_0.Q * xx + sreg_1.Q * /sreg_0.Q
   sreg_1.AR = GND
   sreg_1.SP = GND
   sreg_1.C =  clk

   sreg_0.D =  sreg_1.Q * /sreg_0.Q * /xx   /sreg_1.Q * /sreg_0.Q * xx
        + sreg_1.Q * sreg_0.Q * xx
   sreg_0.AR = GND
   sreg_0.SP = GND
   sreg_0.C =  clk

Completed Successfully

                              C22V10

                 ------------------------------------------
        clk =| 1|                                   |24|* not used
         xx =| 2|                                   |23|= (sreg_1)
   not used *| 3|                                   |22|* not used
   not used *| 4|                                   |21|* not used
   not used *| 5|                                   |20|* not used
   not used *| 6|                                   |19|* not used
   not used *| 7|                                   |18|* not used
   not used *| 8|                                   |17|* not used
   not used *| 9|                                   |16|* not used
   not used *|10|                                   |15|= z
   not used *|11|                                   |14|= (sreg_0)
   not used *|12|                                   |13|* not used
                 ------------------------------------------
```

These equations are not what are found in the Sandige text since he used a different state assignment and JK flip-flops. Note that, since the 22V10 has not internal nodes, the two state variables appear on output pins.

Had the state assignment of the text been used, the following results would be obtained (by simply editing the constant declarations and recompiling).

```
DESIGN EQUATIONS          (10:03:34)
z =   sreg_1.Q * /sreg_0.Q
sreg_0.D =   xx
sreg_0.AR = GND
sreg_0.SP =  GND
sreg_0.C =    clk
sreg_1.D =    sreg_0.Q * xx + sreg_1.Q * sreg_0.Q
sreg_1.AR =  GND
sreg_1.SP =   GND
sreg_1.C =     clk
```

These equations are more consistant with what we see on page 518.

Well, let's do this design once more. We will do it in a manner more typical of CPLD design. We need not redraw the diagram but we will run StateCad again. This time we will make a

minor change (specific generin IEEE design rather than Scenario) and a major change - the ONE-HOT design. This is not covered in Sandige's text. It is covered in Section 5.4.4 of the VHDL book. In this assignment, there is one state variable for each state. In the past, state variables (flip-flops) were relatively expensive and so this type of assignment was not desirable. Today, especially in CPLD's, the cost of the logic in the NEXT STATE DECODER and the OUTPUT decoder is the important factor and so the one-hot assignment is often desirable. One other sometimes desirable feature on the one-hot assignment is the elimination (or minimization) of possible glitches in the output.

The VHDL generated for this assignment is as below. Again, we will break it up for discussion.

```
--   F:\FSM01\FIG93.VHD
--   VHDL code created by Visual Software Solution's StateCAD Version 3.2
--   Thu Oct 23 10:18:47 1997
--   This VHDL code (for use with IEEE compliant tools) was generated using:
--   one-hot state assignment with boolean code format.
--   Minimization is enabled,  implied else is disabled,
--   and outputs are area optimized.
```

Note that the comments indicate "IEEE Compliant" rather than Scenario and that the one-hot assignment has been used.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY FIG93 IS
      PORT (CLK,xx: IN std_logic;
            Z : OUT std_logic);
END;
```

No changes here.

```
ARCHITECTURE BEHAVIOR OF FIG93 IS
      -- State variables for machine sreg
      SIGNAL a, next_a, b, next_b, c, next_c, d, next_d : std_logic;
BEGIN
      PROCESS (CLK, next_a, next_b, next_c, next_d)
      BEGIN
            IF CLK='1' AND CLK'event THEN
                  a <= next_a;
                  b <= next_b;
                  c <= next_c;
                  d <= next_d;
            END IF;
      END PROCESS;
```

There is no significant change here except that the state variables are no longer vectors. We need no constant declarations with one-hot assignment.

```
PROCESS (a,b,c,d,xx)
BEGIN
```

```
  IF((xx='0'AND (a='1')) OR (xx='0' AND (b='1')) OR (xx='0' AND (d='1')))
THEN next_a<='1';
    ELSE next_a<='0';
    END IF;

  IF (( xx='1' AND  (a='1')) OR ( xx='1' AND  (d='1')))
    THEN next_b<='1';
    ELSE next_b<='0';
    END IF;

  IF (( xx='1' AND  (b='1')) OR ( xx='1' AND  (c='1')))
    THEN next_c<='1';
    ELSE next_c<='0';
    END IF;

  IF (( xx='0' AND  (c='1')))
    THEN next_d<='1';
    ELSE next_d<='0';
    END IF;

  IF ((  (d='1')))
    THEN Z<='1';
    ELSE Z<='0';
    END IF;
END PROCESS;
END BEHAVIOR;
```

The CASE statement is not especially useful for the one-hot assignment. for this reason, StateCad uses if the IF statement for each state variable and output.

```
DESIGN EQUATIONS            (10:20:38)
z.D  = /xx * c.Q
/a.D = /z.Q * /a.Q * /b.Q + xx
 b.D = xx * a.Q + xx * z.Q
 c.D = xx * c.Q + xx * b.Q

z.AR = GND
z.SP = GND
z.C  = clk
a.AR = GND
a.SP = GND
a.C  = clk
b.AR = GND
b.SP = GND
b.C  = clk
c.AR = GND
c.SP = GND
c.C  = clk
```

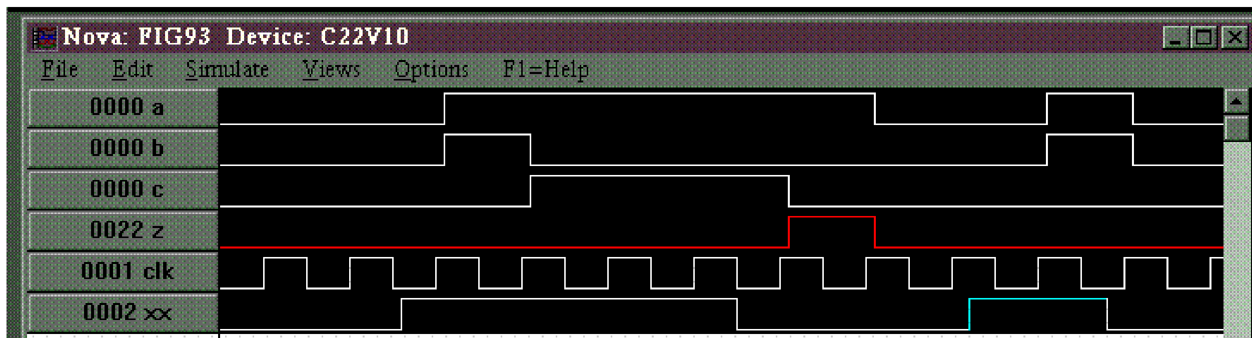
One might observe something missing in the above equations. Where are the equations for state variable d? Well, we glossed over some of the information in the report file. Let's go back and take a look at it.

```
OPTIMIZATION OPTIONS        (10:20:35)
```

```
Messages:
  Information: Process virtual 'next_d' ... expanded.
  Information: Process virtual 'next_c' ... expanded.
  Information: Process virtual 'c' ... converted to NODE.
  Information: Process virtual 'next_b' ... expanded.
  Information: Process virtual 'b' ... converted to NODE.
  Information: Process virtual 'next_a' ... expanded.
  Information: Process virtual 'a' ... converted to NODE.
```

Variables for states a, b, and c were converted to NODES (i.e. output pins). Variable d was not because it was discovered that Z= d. The flip-flop for variable (state) d is the one which provides the Z output. In this case, we used one more pin on the 22V10 than in the previous example. This is not necessarily a practical way to implement a FSM in a small PAL. In a CPLD, however, this would likely be the better design.

Well, what more is there to do? We still haven't verified that this machine works. Let's use NOVA again.



While the above timing diagram is not a complete test, it does show most of the allowable transitions. Wait! Something is wrong with state a! No there isn't - look at the equations again! To minimize the logic, state a was implemented as an active low signal and so is inverted in the simulation. It is correct.

Well, this has been a relative short introduction to state machines and StateCad. We shall certainly get a lot further into it in future notes.