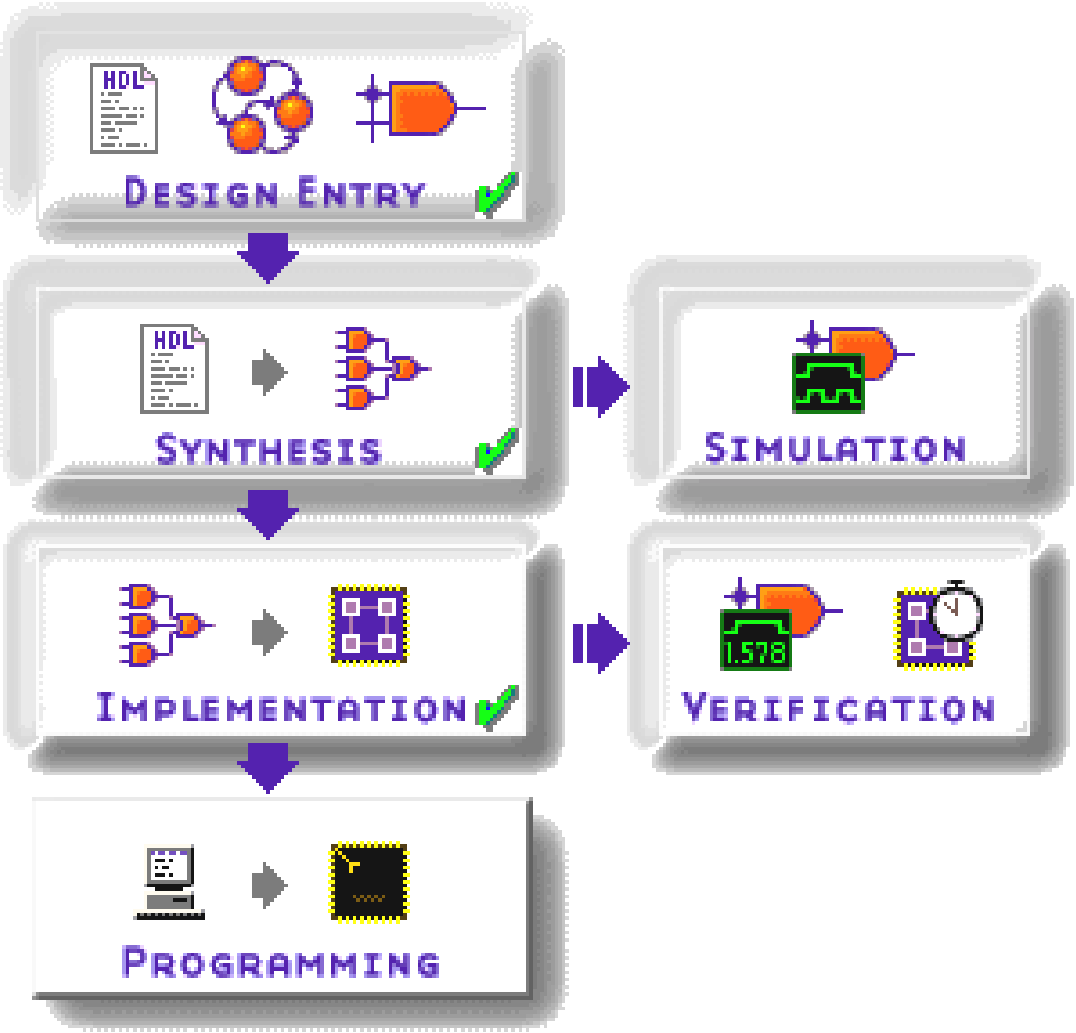
Pragmatic Logic Design

With XILINX Foundation 2.1i



David E. Vanden Bout

XESS Corp

4

State Machine Design

In this chapter you will learn how to:

- Create a finite state machine with the Foundation State Editor;
- Encapsulate the finite state machine in a macro;
- Update a macro when you retarget a project to another device family;
- Interface to a PS/2 keyboard.

Finite State Machines

A simple finite state machine (FSM) uses one or more flip-flops to store its internal state. The pattern of ones and zeroes on the flip-flop outputs are the current state. In a synchronous FSM, the current state is replaced with the next state on the rising edge of a clock signal. The next state is computed by a combinational logic circuit that accepts the current state and possibly some external signals as inputs. So a synchronous FSM is composed basically of a set of flip-flops fed by a combinatorial circuit that accepts feedback from the flip-flops on every clock cycle.

In this chapter we will build an FSM that acts like a combination lock. The requirements for this digital combination lock are:

1. The user enters a combination as a sequence of *n* key presses on a keyboard.
2. The combination lock stores a particular combination as a sequence of *n* key presses.
3. The combination lock will open if the user enters an *n*-key sequence that matches the combination. Otherwise, the lock stays locked.
4. The user must enter an entire *n*-key sequence before the lock either accepts or rejects the sequence.
5. Once the combination lock is unlocked, the user can relock it or enter a new combinations as a sequence of key presses.

6. The lock will require the user to verify any new combination that is entered before it replaces the previous combination.

A hierarchical view of the combination lock and its lower-level modules is shown in Figure 7. The combination lock consists of:

Keyboard interface: This module accepts a serial data stream and clock signal from a standard PS/2 PC keyboard and converts it into a parallel scancode with an associated ready signal that indicates the presence of the scancode.

Lock&key mechanism: This module accepts scancodes from the keyboard interface and determines whether or not the correct combination has been entered and manages the entry of new combinations.

The combination lock accepts the keyboard serial data and clock as inputs along with a main clock that synchronizes the operations of both modules. There is also a reset input to initialize the entire FSM upon startup. The combination lock visually indicates its current status on a seven-segment LED.

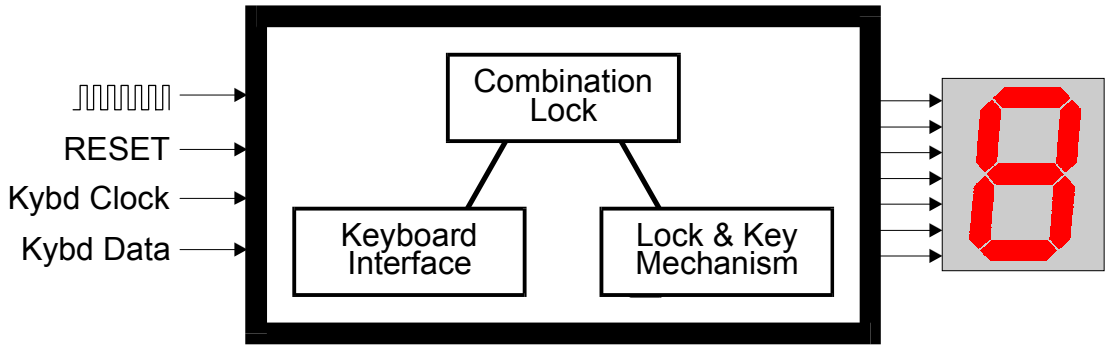
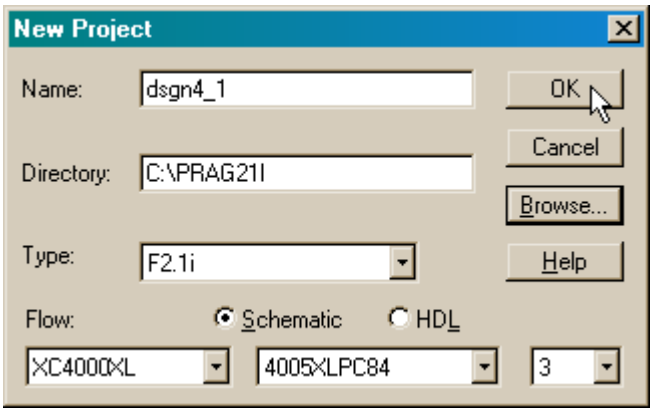


Figure 10: Design hierarchy for a combination lock.

Building the Combination Lock Project

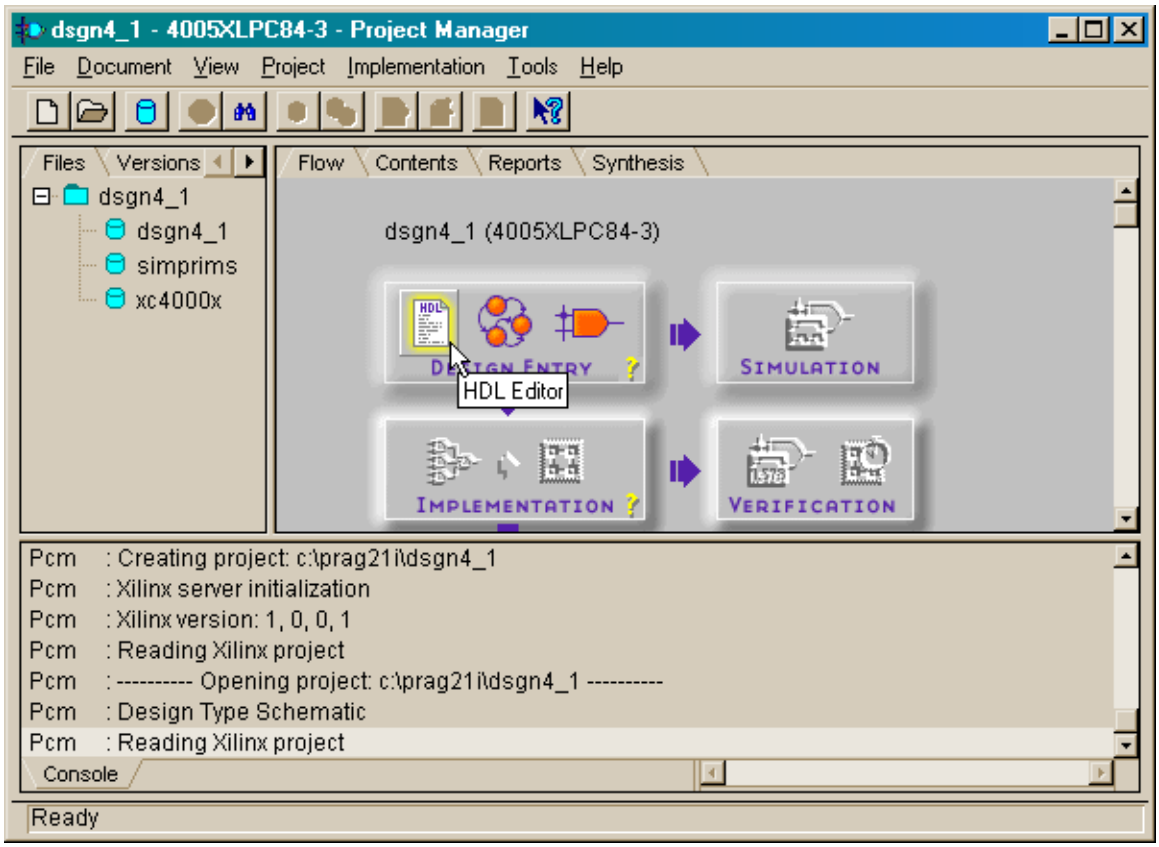
Starting the Project

We will begin the design of the combination lock by creating a schematic-based project for the XC4005XL FPGA as shown below. We will describe the lower-level keyboard interface and the lock&key modules using the HDL Editor and State Editor, respectively, and then tie these modules together with a top-level schematic.



Creating the Keyboard Interface Module

After the Project Manager window appears, we can start the HDL Editor and begin designing the keyboard interface.



A PS/2 keyboard connects to an XS40 or XS95 Board through two signals:

psData: This signal carries the serial data stream as each key is pressed and released. Each key is assigned an eight-bit scancode that is transmitted least-significant bit to most-significant bit with a preceding start bit and a terminating parity bit and stop bit.

psClk: The falling edge of this signal indicates when the psData signal is valid.

The keyboard interface will accept the serial data stream and will output the eight-bit scancode in parallel along with an **rdy** pulse that indicates a valid scancode is available. The **rdy** pulse will be generated when the **psClk** signal goes high and stays there. The timing of the **psData**, **psClk**, and **rdy** signals is shown in Figure 11.

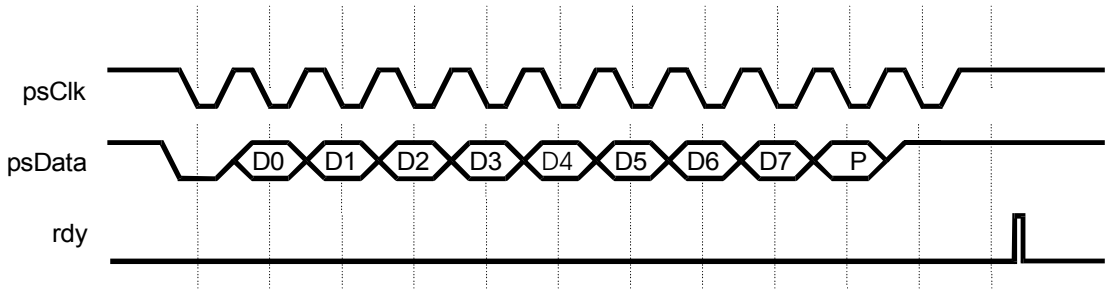


Figure 11: PS/2 keyboard waveforms.

A single scancode is transmitted when a key is pressed. But two scancodes are transmitted when the key is released: an initial scancode of 11110000 to indicate the key release, and then the scancode for the key is sent again. The keyboard interface will be designed such that the **rdy** signal pulses only after the key has been released.

The VHDL code for the keyboard interface is shown in Listing 2. The functions of the code for the **scancodeReg** module are as follows:

Lines 7–12: The module receives the **psData** and **psClk** inputs from the keyboard and outputs the eight-bit scancode and the **rdy** signals that were described above. A master clock is also provided which synchronizes the operations of this module with the lock&key module. A reset signal initializes the module when it first powers up.

Line 17: This line declares a 10-bit shift-register that holds the scancode value as it arrives from the keyboard. The start bit, eight scancode bits, parity bit, and stop bit will enter the most-significant bit of the **sc_r** register and shift towards the least-significant bit. By the end of a scancode transmission the start bit will have shifted completely out of the register and be lost while the scancode will end up in the lower eight bits of **sc_r**. The stop and parity bits will be in the uppermost two bits.

Lines 18–22: These lines define a counter that is used to determine when the **psClk** signal is no longer pulsing. The timeout value (line 20) is determined by dividing the main clock frequency (line 18) by the frequency of the **psClk** (line 19). If the main clock is 50 MHz and the keyboard clock is 10 KHz, then the timeout value is 5000 which means it will take 5000 pulses of the main clock to determine if the **psClk** signal is static. The subtype for a timeout counter is defined on line 21 as a natural number that can take on any value from zero up to the timeout value. Then the timeout counter register is declared on line 22. By defining the counter register in this way, we can change the frequency of the main clock or the keyboard clock and the timeout counter will be automatically resized by the synthesizer with exactly the number of bits needed to store the timeout value.

Lines 30–40: This process parallelizes the serial keyboard data. If the reset input is active, the scancode shift register is cleared to all zeroes. Otherwise, on falling edges of the keyboard clock the value on the keyboard data signal is placed into the most-significant bit of the shift register and the upper nine bits of the register are shifted one bit position downward. By the end of a scancode transmission the start bit will have shifted completely out of the

register and be lost while the scancode will end up in the lower eight bits of **sc_r**. The stop and parity bits will be in the uppermost two bits.

Line 43: The eight lower bits of the **sc_r** register are output as the scancode output of the module.

Lines 47–68: This process detects when the **psClk** signal has stopped pulsing and indicates that a scancode is available. The timeout counter and scancode ready flag are cleared when the module is reset. Then the counter is incremented as long as the **psClk** is at logic 1 and the counter has not reached its timeout value yet. The counter is reset to zero if **psClk** is ever low because that indicates the keyboard clock is still pulsing so the scancode cannot be complete. But if the counter ever reaches the value **timeout**-1, then the scancode ready flag is pulsed high for a single clock cycle.

Lines 72–88: This process checks the **scRdy_r** flag and looks for the scancode that matches the **keyRelease** scancode defined on line 26 (11110000). After seeing the key release scancode, this process looks for the next following scancode. Then it sets the flag that indicates the scancode is ready for output.

Line 90: The ready flag from the previous process is output from the module.

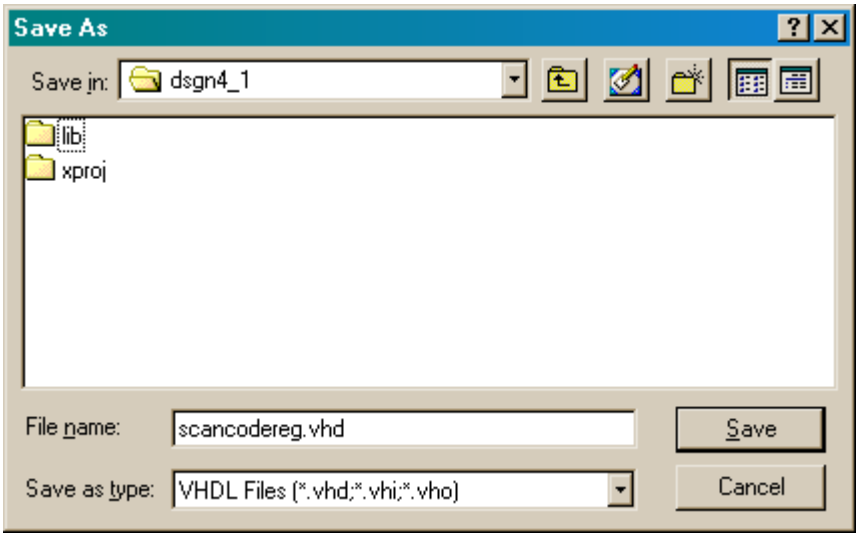
Listing 2: VHDL code for the keyboard interface.

```
1   library IEEE;
2   use IEEE.std_logic_1164.all;
3   use IEEE.numeric_std.all;
4
5   entity scancodeReg is
6   port(
7     clk:      in std_logic;                       -- main clock
8     rst:      in std_logic;                       -- reset
9     psClk:    in std_logic;                       -- keyboard clock
10    psData:   in std_logic;                       -- keyboard data
11    scancode: out std_logic_vector(7 downto 0);   -- key scancode
12    rdy:      out std_logic                       -- scancode ready pulse
13  );
14  end entity;
15
16  architecture arch of scancodeReg is
17  signal   sc_r:       std_logic_vector(9 downto 0); -- scancode shift register
18  constant clkFreq:    natural := 50_000; -- main clock frequency (KHz)
19  constant psClkFreq:  natural := 10;     -- keyboard clock frequency (KHz)
20  constant timeout:    natural := clkFreq / psClkFreq;  -- psClk quiet timeout
21  subtype counter is natural range 0 to timeout;
22  signal   cnt_r:      counter;   -- timeout counter
23  signal   scRdy_r:    std_logic; -- scan code is ready flag
24  signal   rdy_r:      std_logic; -- output scan code is ready flag
25  signal   keyrel_r:   std_logic; -- key has been released flag
26  constant keyRelease: std_logic_vector(7 downto 0) := "11110000";
27  begin
28
29    -- this process places the keyboard scancode into the shift register
```

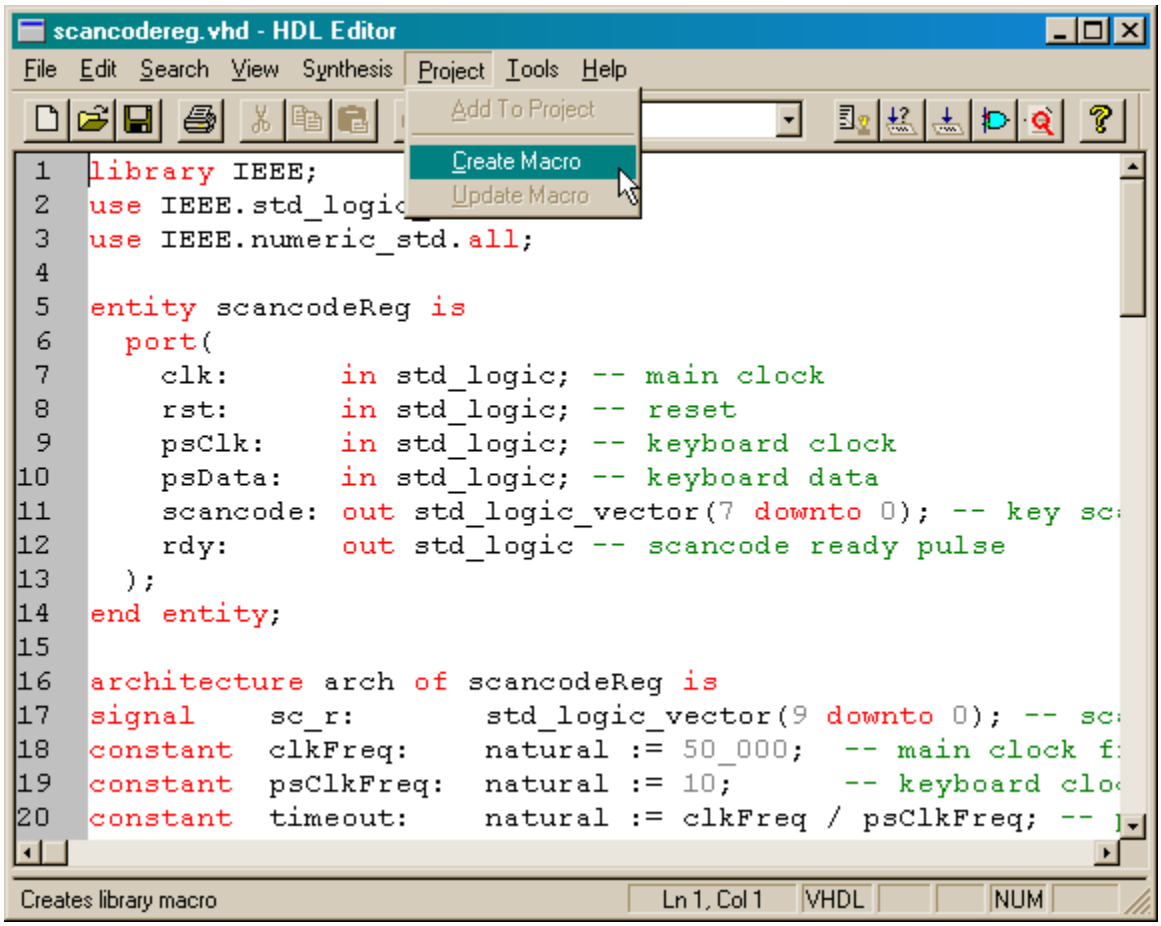
```
30      process(psClk,rst)
31      begin
32        -- async. reset of scancode ready flag
33        if rst = '1' then
34          sc_r <= (others=>'0');
35        -- accept keyboard data on falling edge of keyboard clock
36        elsif psClk'event and psClk='0' then
37          -- key data arrives LSB first so right-shift it into MSB of register
38          sc_r <= psData & sc_r(9 downto 1);
39        end if;
40      end process;
41
42      -- key scancode is in the lower 8-bits of the shift register
43      scancode <= sc_r(scancode'range);  -- output scancode
44
45      -- this process detects the end of the scancode by looking
46      -- for the absence of keyboard clock pulses
47      process(clk,rst)
48      begin
49        if rst = '1' then
50          cnt_r <= 0;     -- clear the timeout counter
51          scRdy_r <= '0'; -- clear the scancode ready flag
52        elsif clk'event and clk = '1' then
53          scRdy_r <= '0';  -- by default, no key scancode is ready for output
54          if psClk = '0' then
55            -- reset the timeout register whenever the keyboard clock pulses low
56            cnt_r <= 0;
57          elsif cnt_r /= timeout then
58            -- increment the timeout counter if the keyvoard clock is high
59            -- and the counter hasn't reached the timeout value yet
60            cnt_r <= cnt_r + 1;
61            if cnt_r = timeout-1 then
62              -- signal that a key scancode is ready when the counter is
63              -- equal to one less than the timeout value
64              scRdy_r <= '1';  -- rdy signal pulses for a single clock period
65            end if;
66          end if;
67        end if;
68      end process;
69
70      -- this process detects when the keyboard key is released and
71      -- signals when the scancode for the released key is ready
72      process(clk)
73      begin
74        if clk'event and clk = '1' then
75          rdy_r <= '0';  -- by default, no key scancode is ready for output
76          if scRdy_r = '1' then
77            -- check the scancode register when a code is ready
78            if sc_r(7 downto 0) = keyRelease then
79              -- set flag if the keyRelease prefix is detected
80              keyrel_r <= '1';
81            elsif keyrel_r = '1' then
82              -- end up here on next scancode received after key release prefix
```

```
83            rdy_r <= '1'; -- released key scancode is in the scancode register
84            keyrel_r <= '0';  -- reset the key release flag
85          end if;
86        end if;
87      end if;
88    end process;
89
90    rdy <= rdy_r;  -- signal that a key scancode is ready
91
92  end architecture;
```

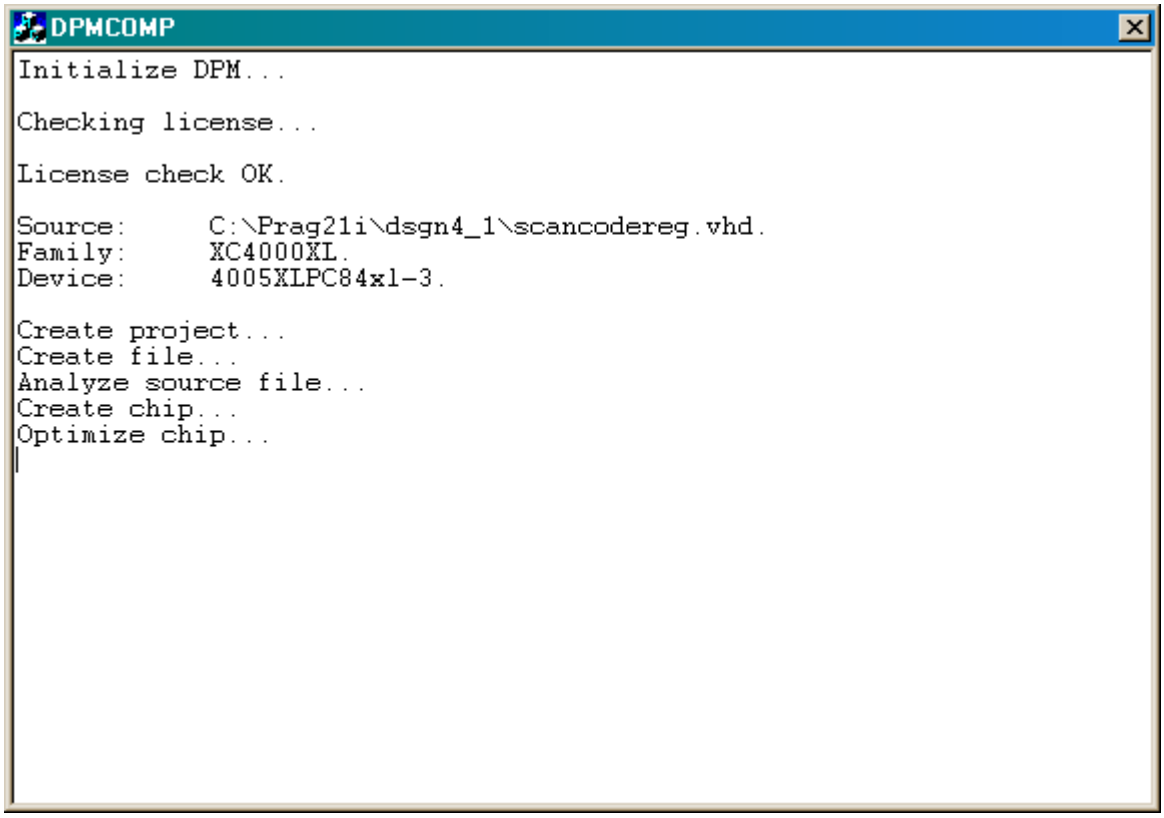
Once the VHDL code from Listing 3 is entered in the **HDL Editor** window, save the code in the scancodereg.vhd file.



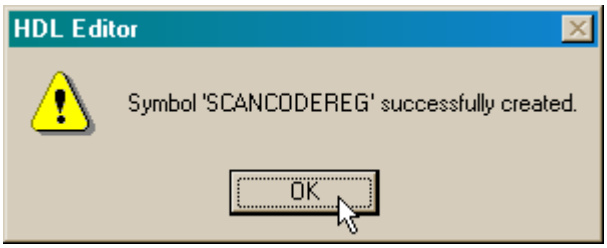
Then encapsulate the keyboard interface into a macro using the Project➔Create Macro command.



The progress as the VHDL synthesizer processes the VHDL will be displayed in the **DPMCOMP** window.



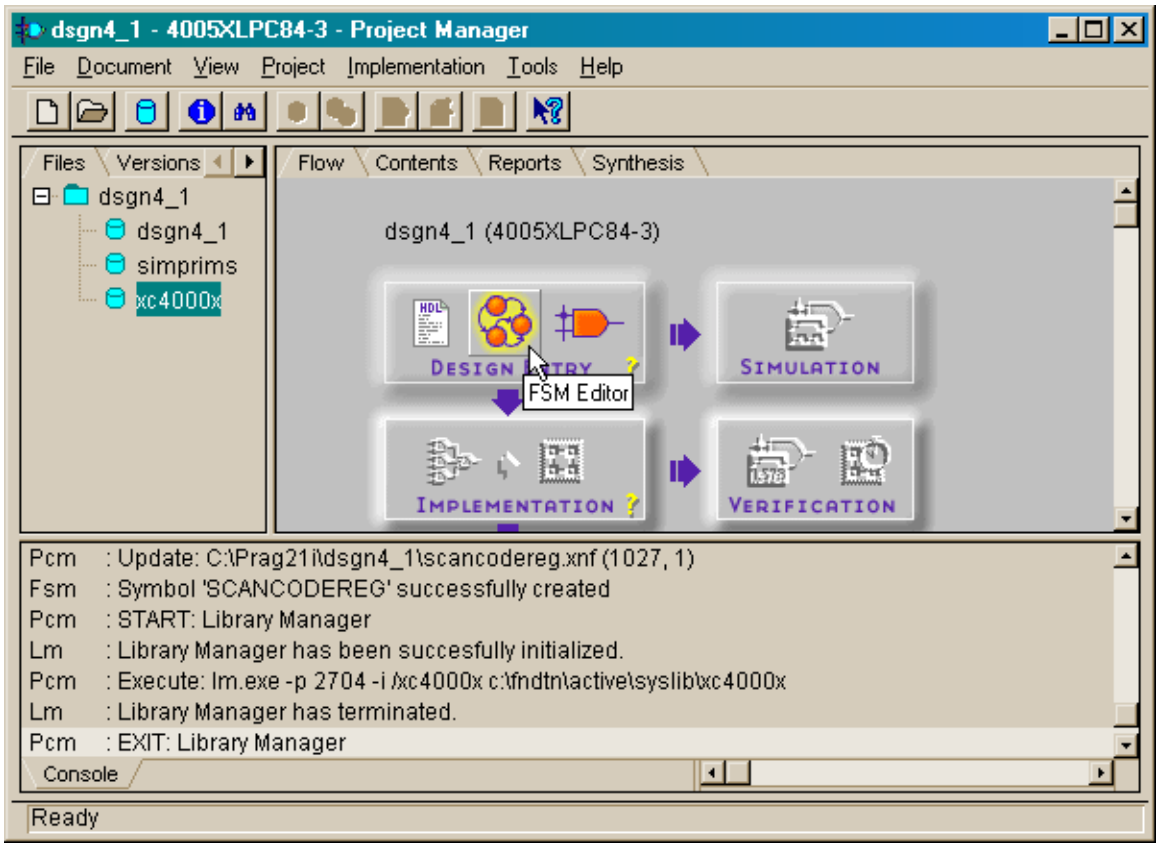
Finally, you should get an indication that the netlist for the keyboard interface macro was successfully synthesized and placed in the library for this project.



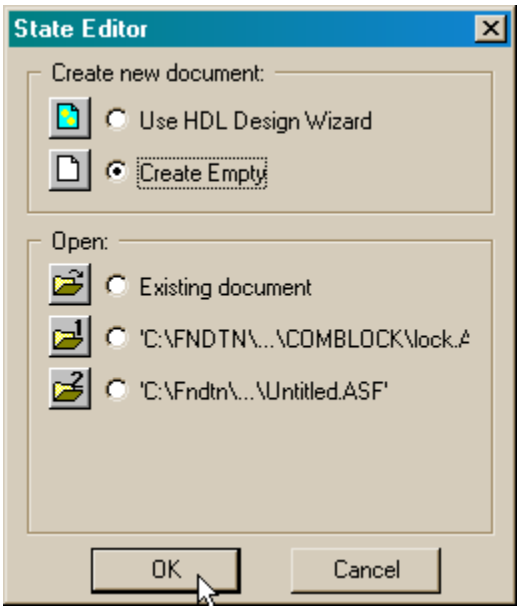
At this point we can exit the **HDL Editor** window and return to the **Project Manager** window.

Creating the Lock&Key Module

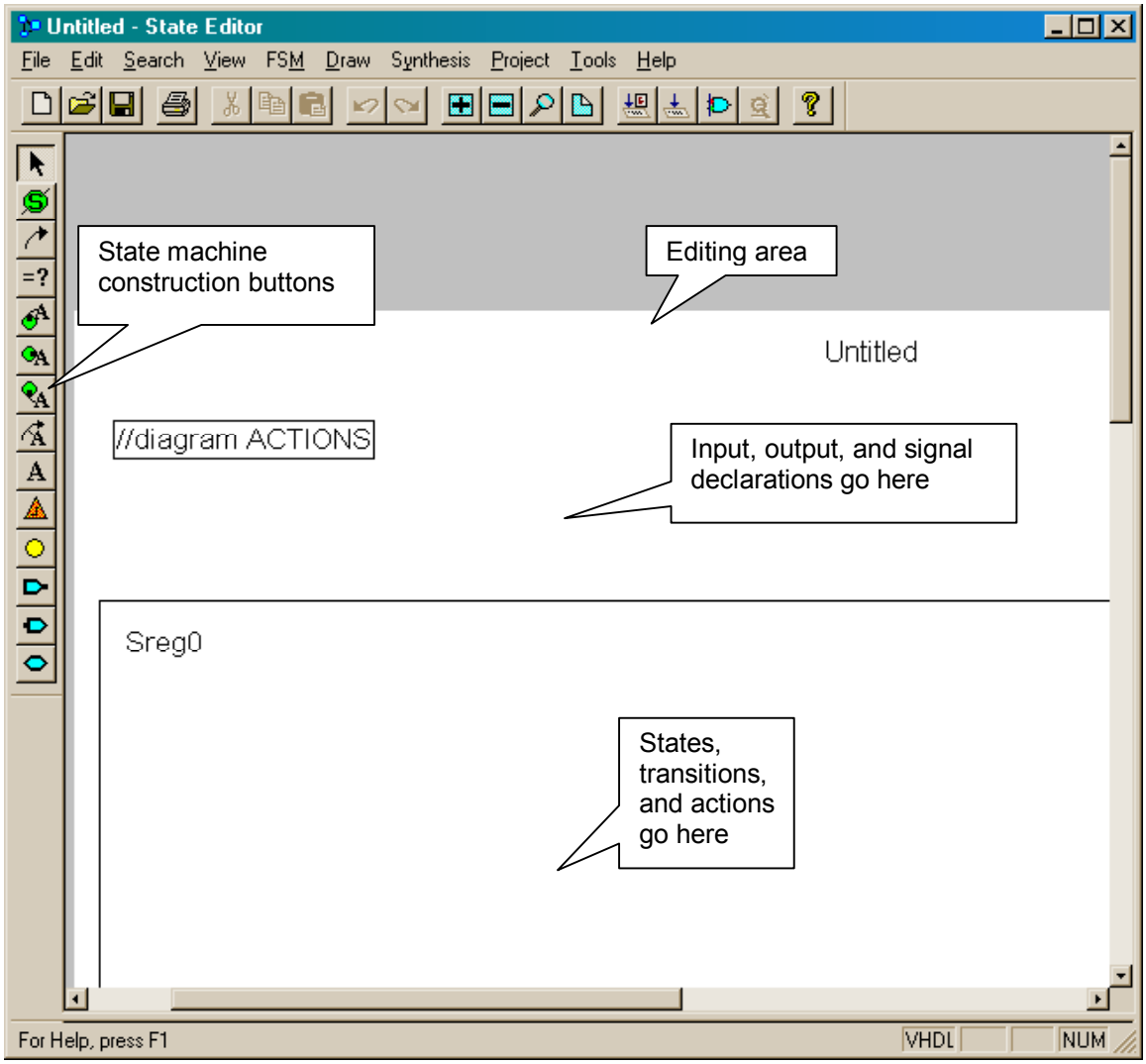
Now we can design the lock&key module using the FSM Editor.



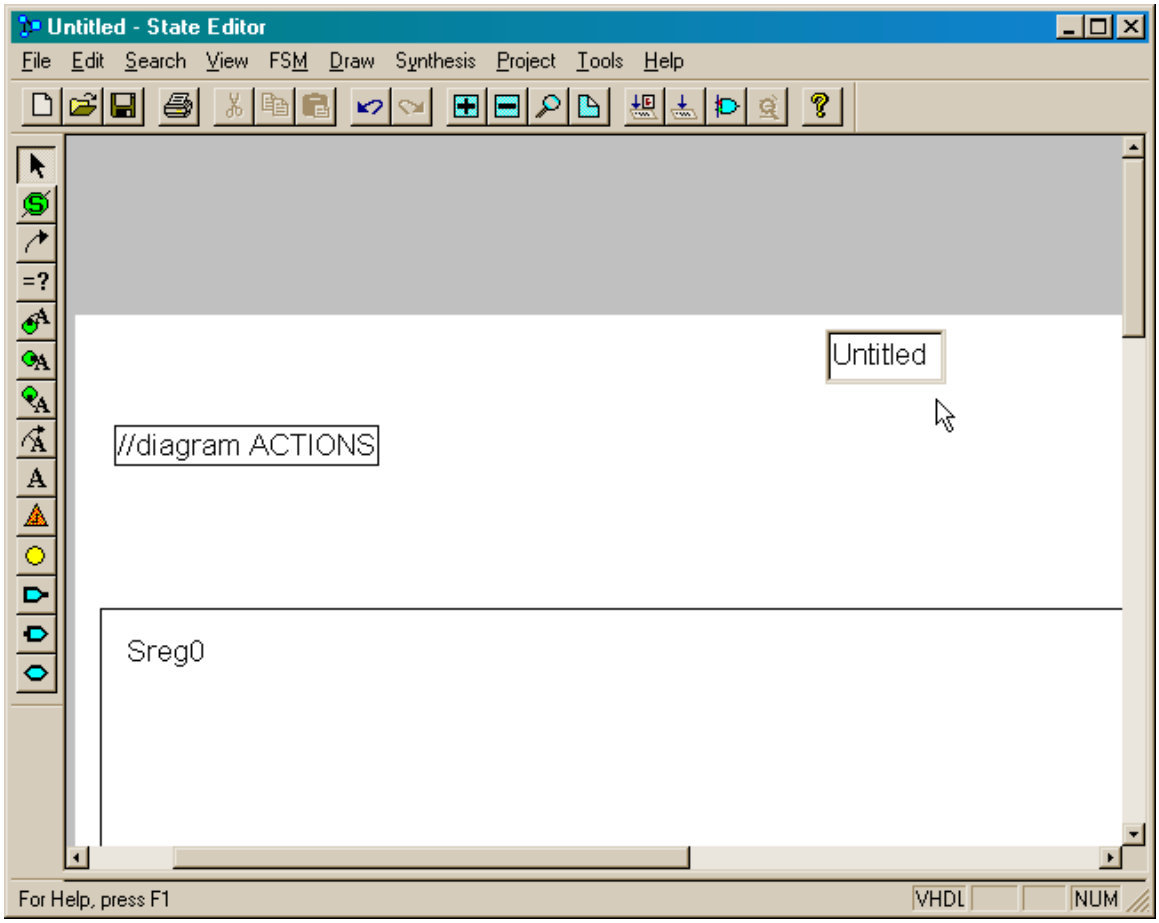
We could begin our state machine design by answering several questions from the HDL Design Wizard and getting an initial template that can be filled-in with the details. But in this example we will start with a blank sheet.



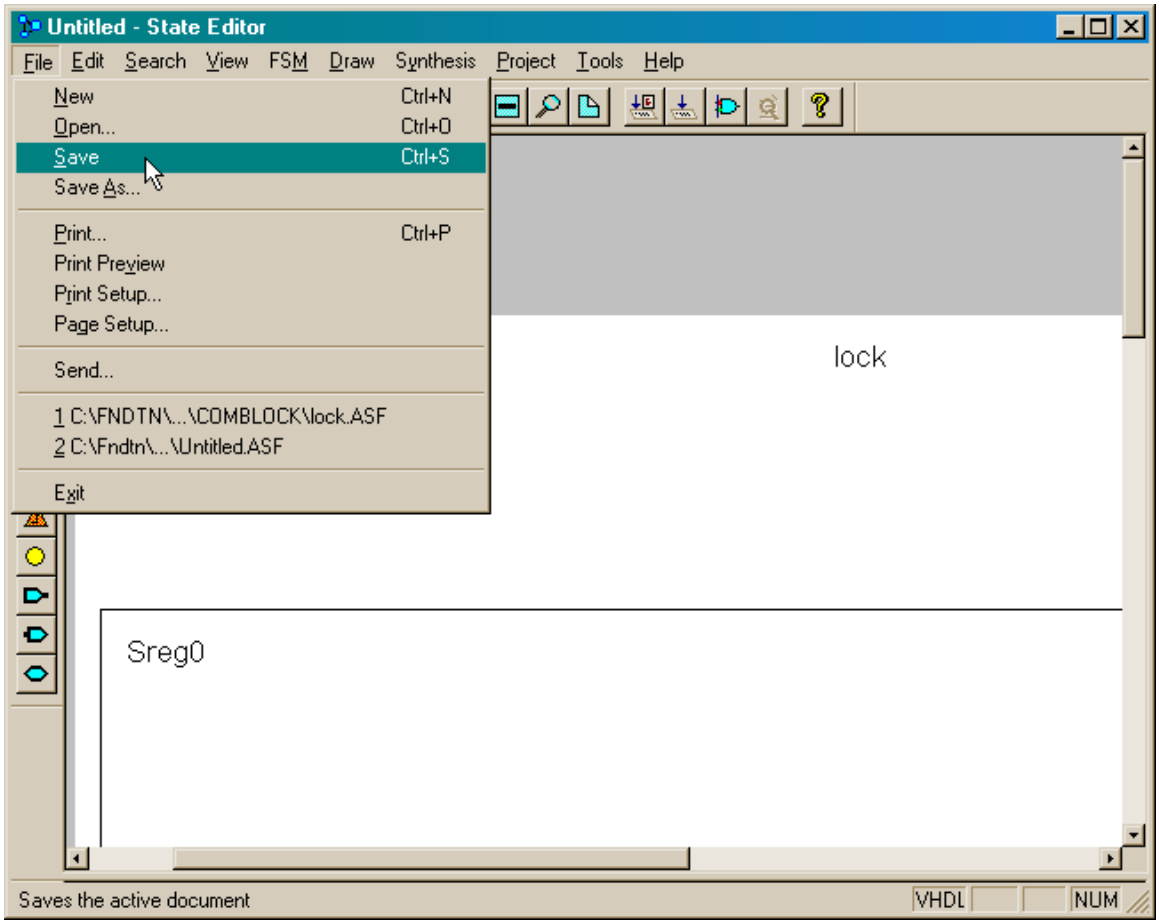
The **State Editor** window that appears has several areas where we can construct the state diagram for our FSM and define the I/O interface to it.



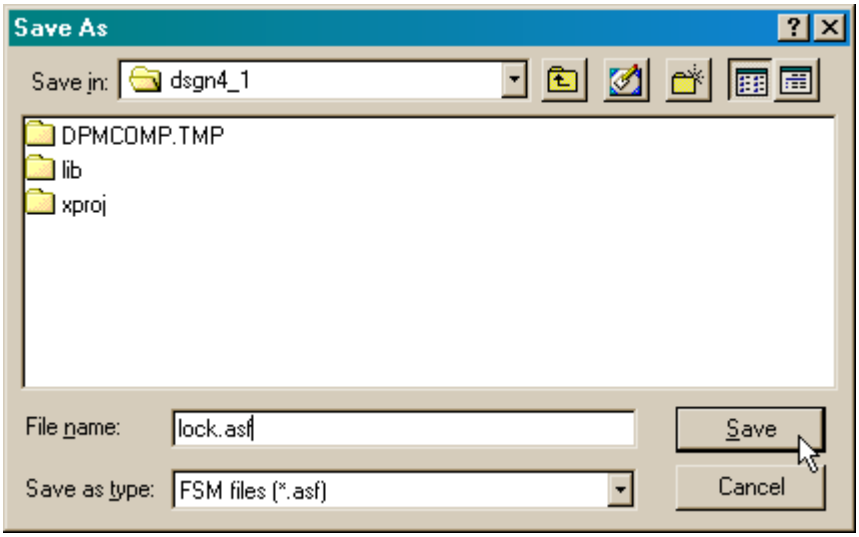
To start, we will name the file that will contain the FSM. Single-click twice on the Untitled text string at the top of the editing area and rename it as lock.



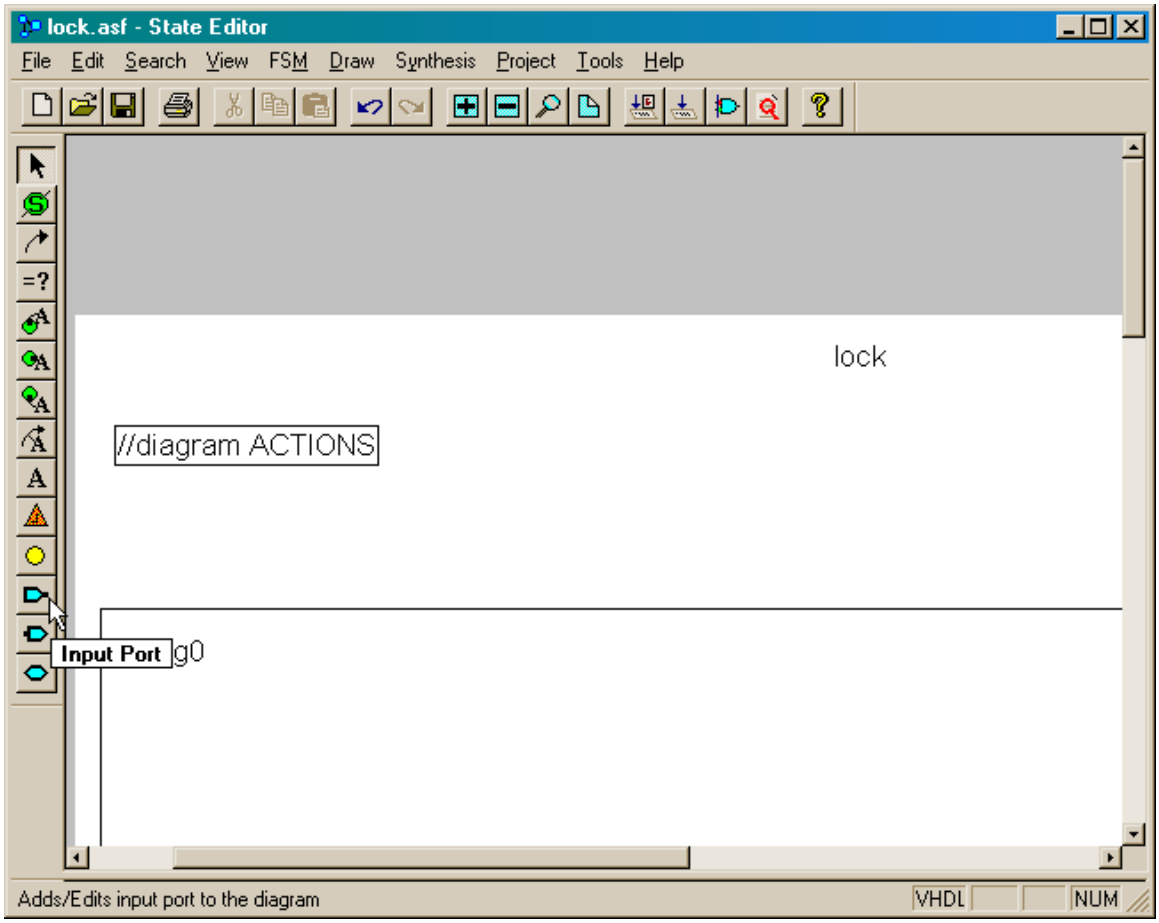
After renaming the editing area, select the File➔Save command.



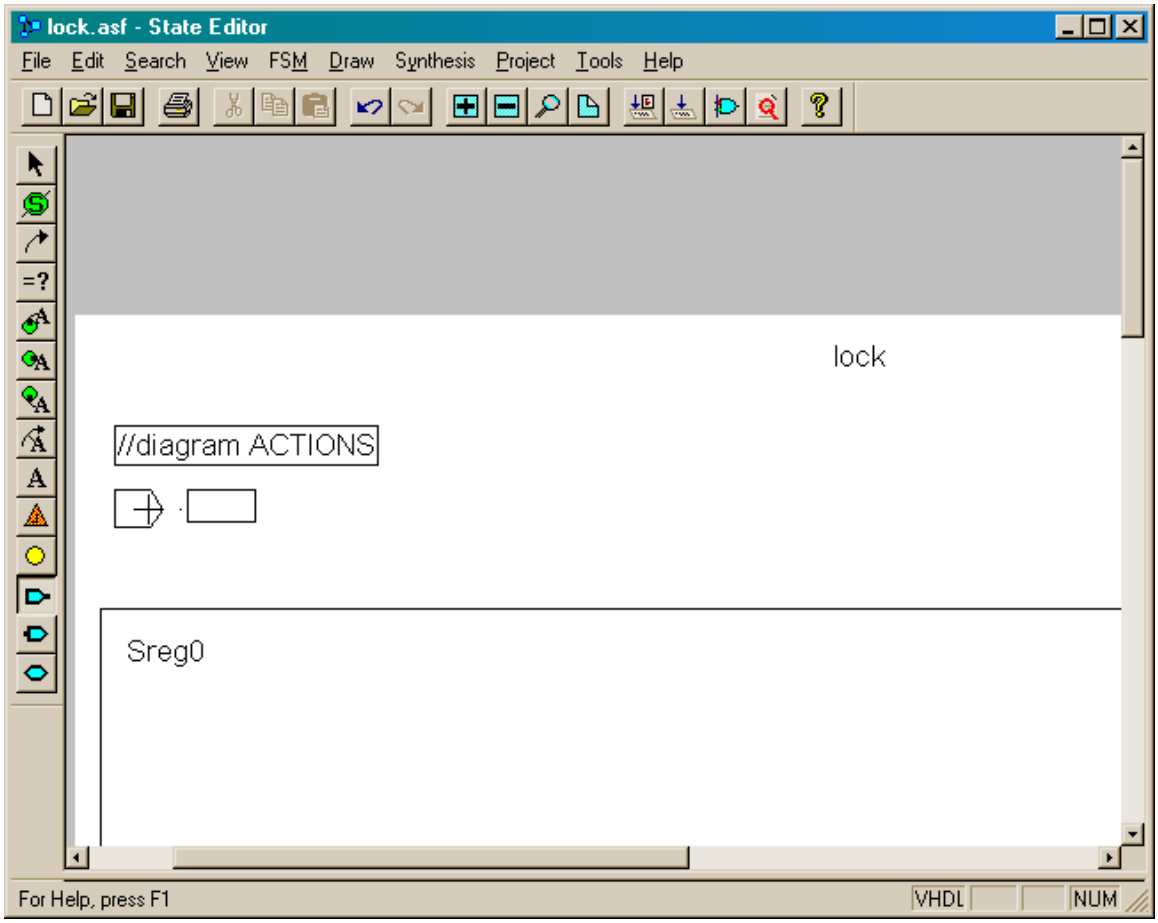
Then change the name of the state machine design file to lock.asf and click on the Save button.



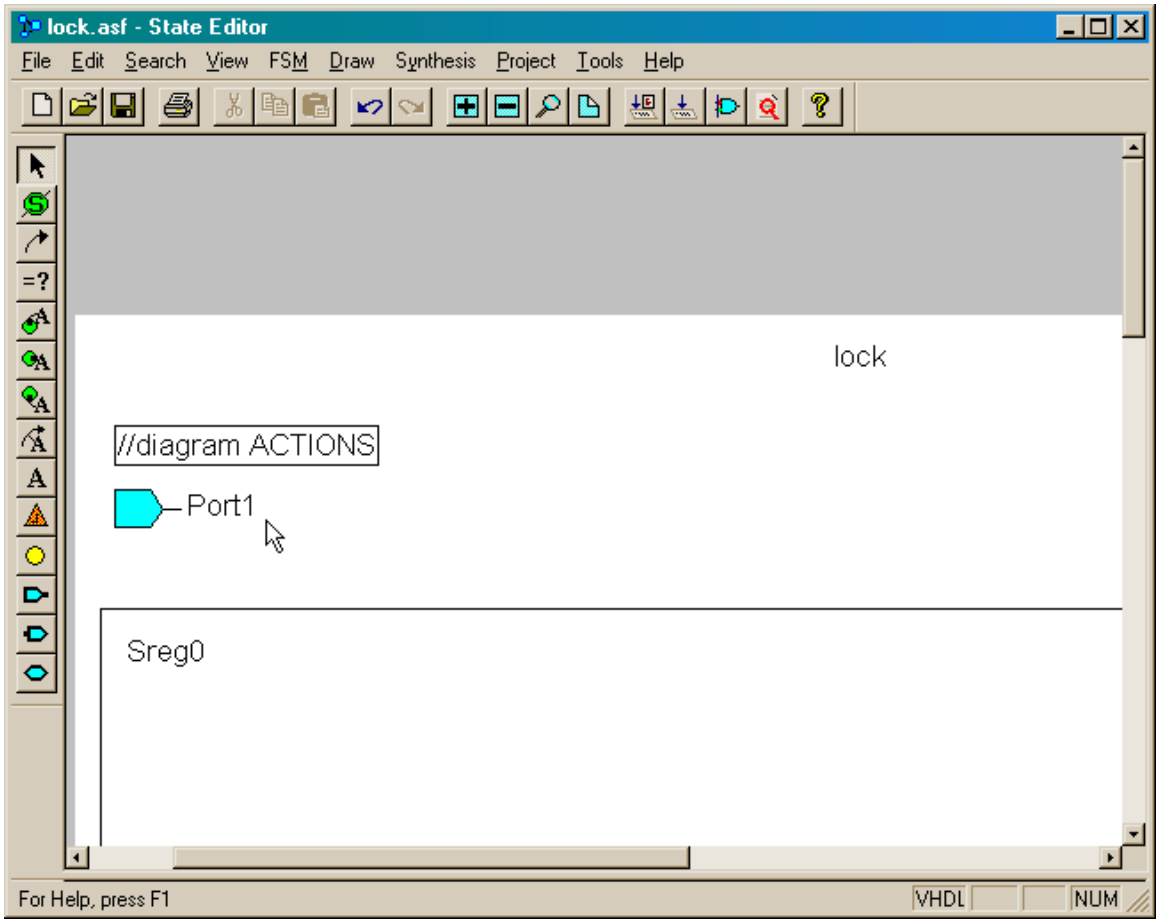
Now we can begin defining the interface to the state machine. We begin by clicking the button that lets us enter input ports.



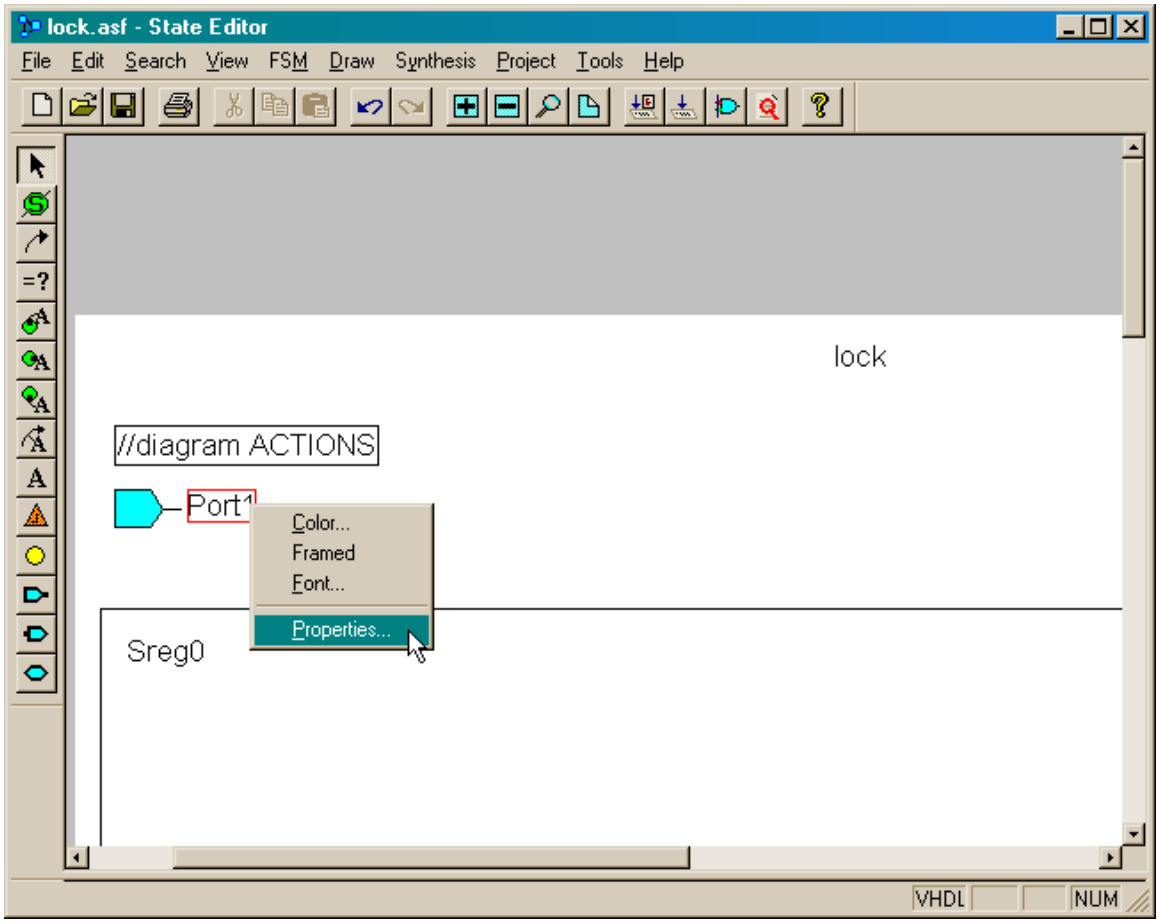
Now move the cursor into the top portion of the editing area. The outline of an input port icon will be attached to the cursor.



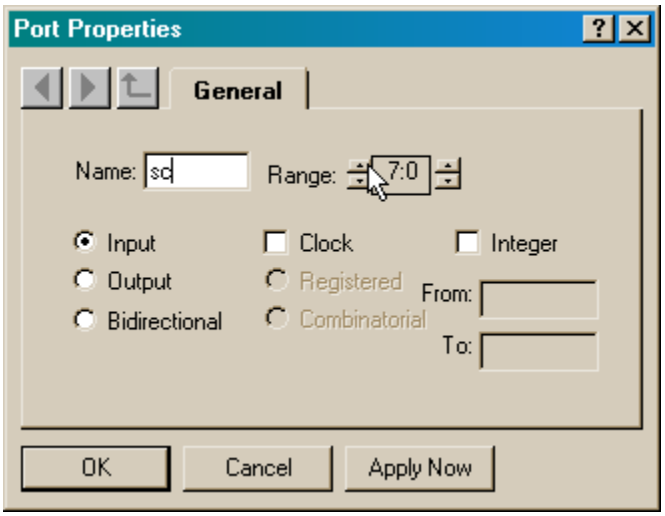
Left-click the mouse in the upper portion of the editing area and an input port icon will appear.



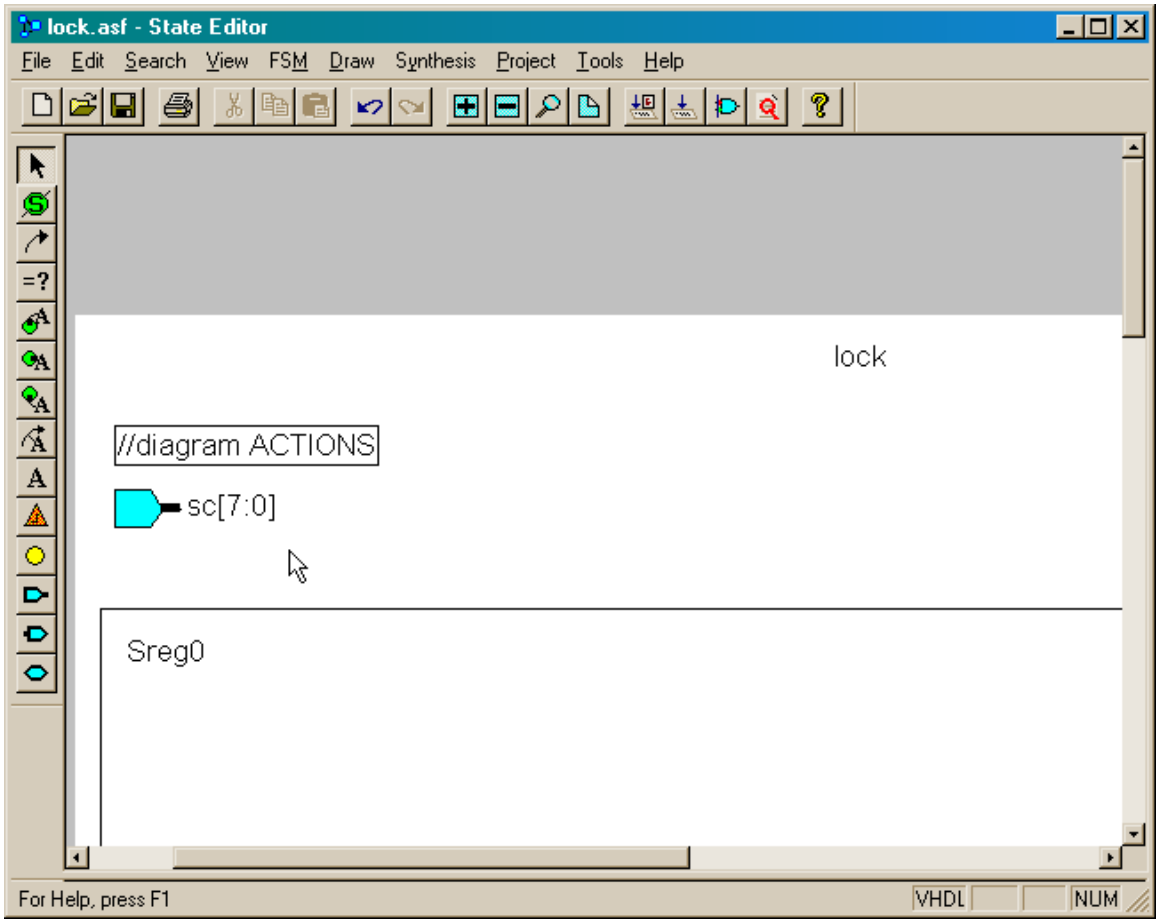
Right-click on the input port icon and select the Properties... entry in the pop-up menu that appears.



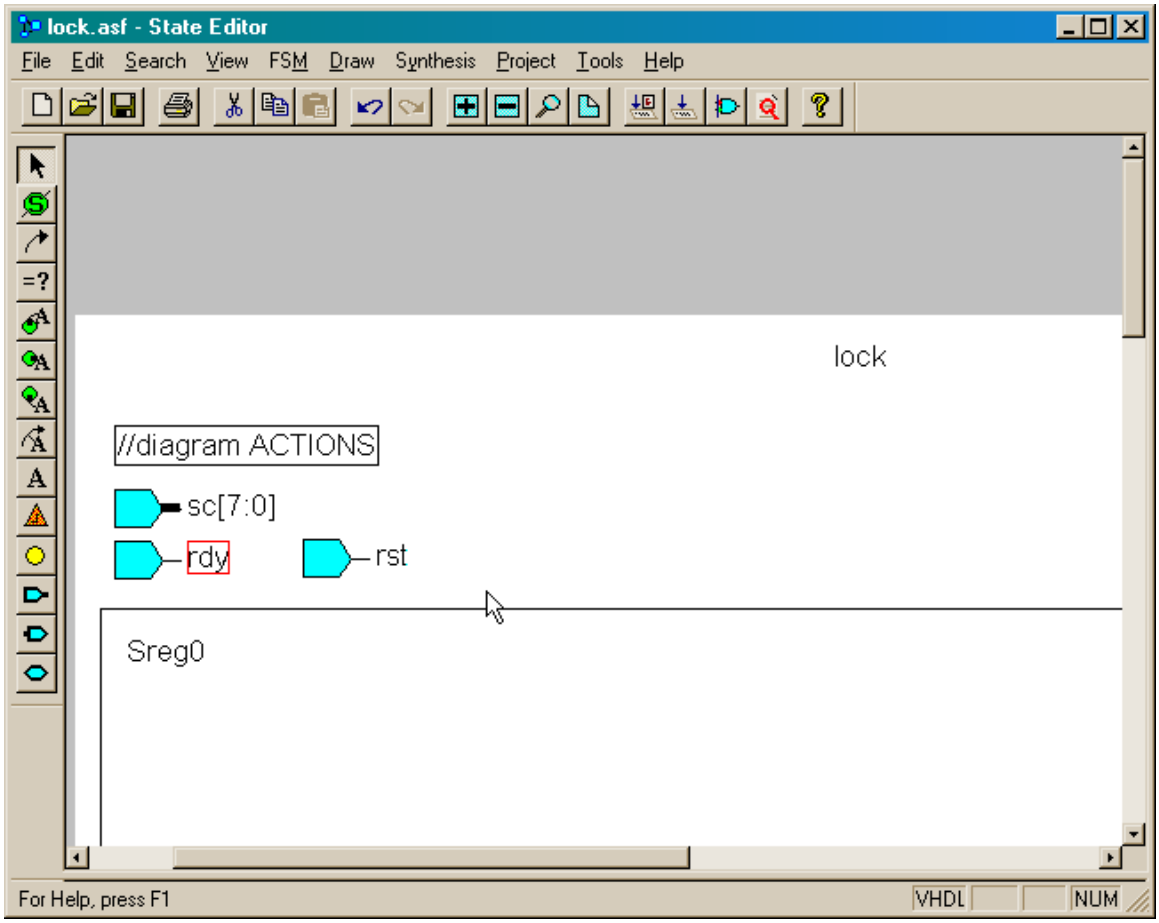
This input will be used to bring the eight-bit scancode into the FSM. In the **Port Properties** window, rename the port to sc and then click on the upper-left button of the Range input until it is eight bits wide. Then click on the OK button.



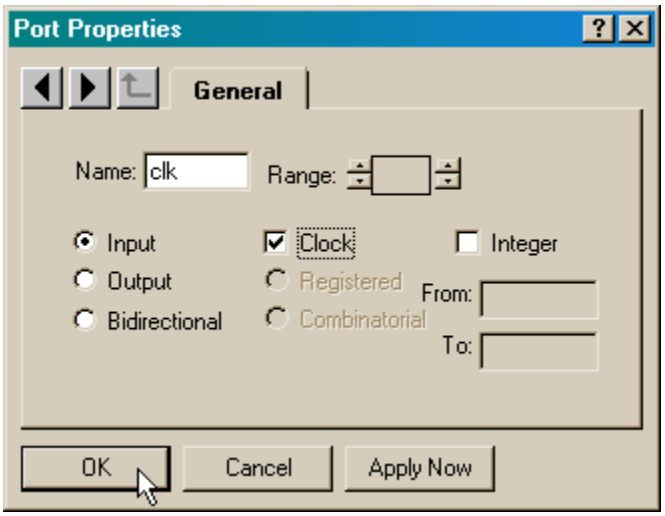
Now the input port in the editing area appears with its new name and the upper and lower indices for its bus width.



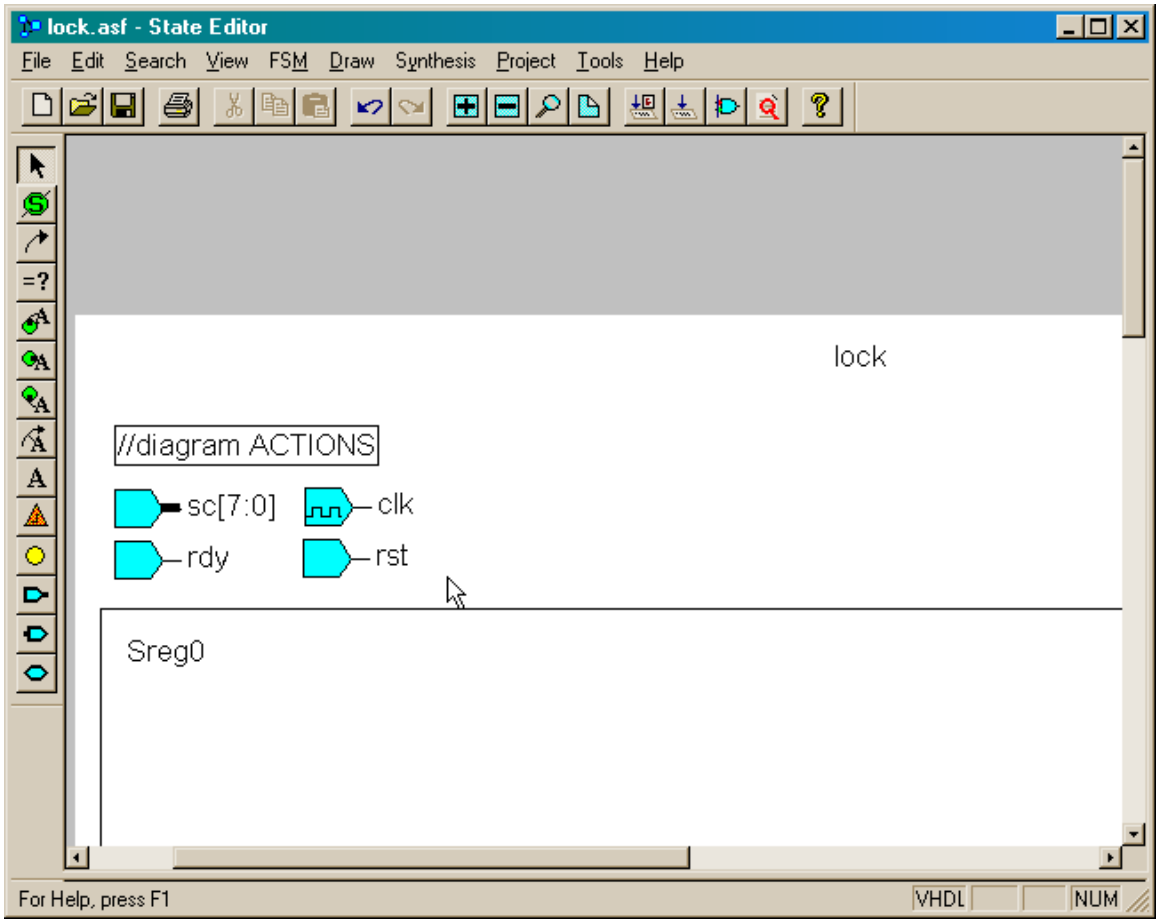
Repeat this process to add single-bit wide input ports for the scancode ready input (rdy), the reset input (rst), and the main clock input (clk).



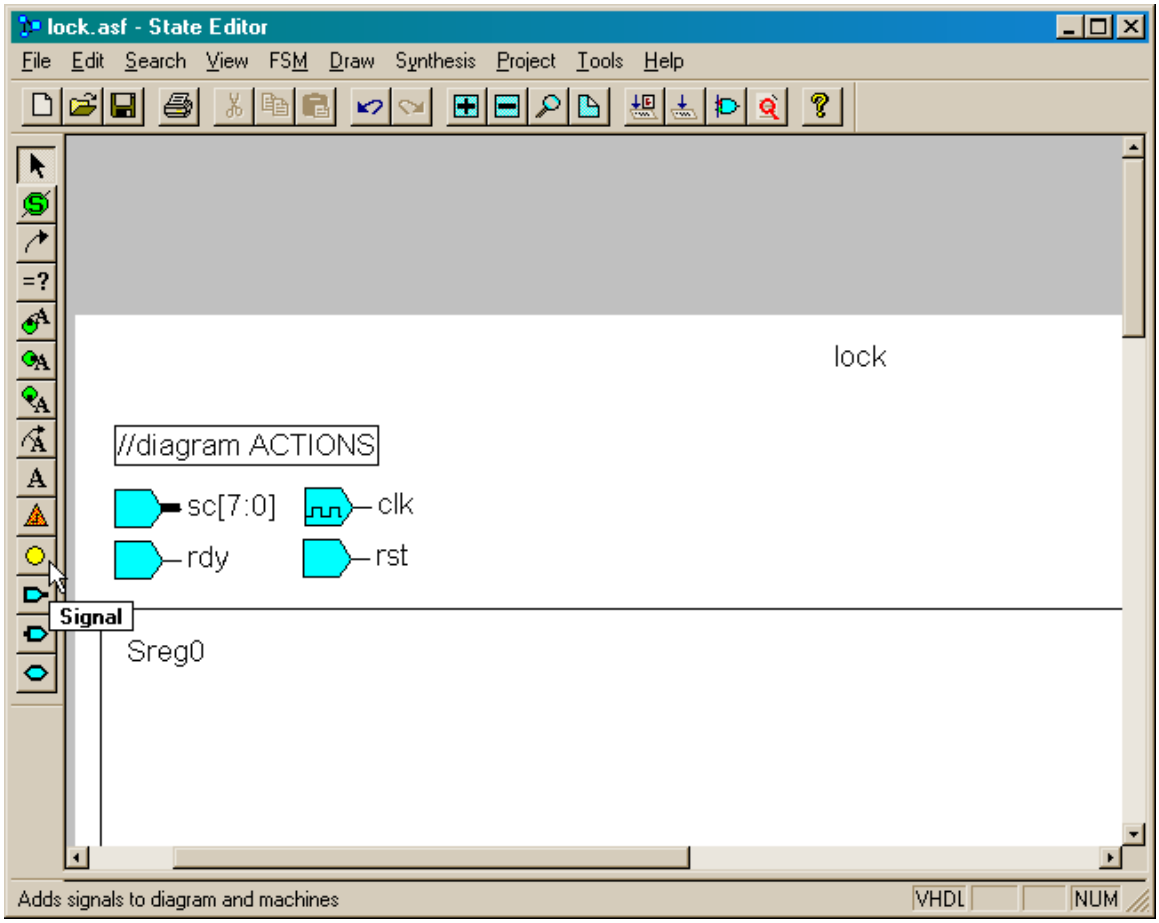
Right-click the clk input and bring up its **Port Properties** window. Click on the Clock checkbox to indicate that this input is a potential clock source for the FSM. Then click on the OK button.



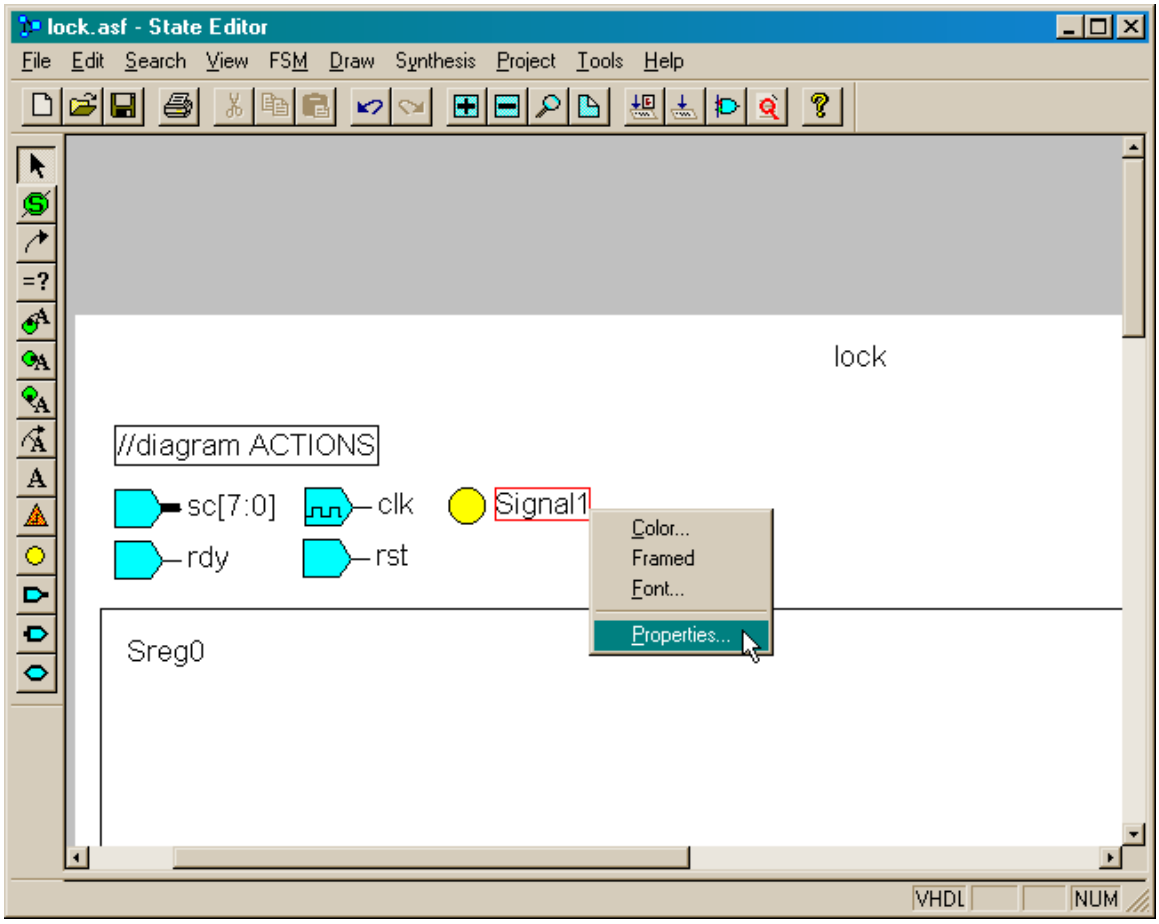
Upon returning to the **State Editor** window, you will notice that the clk input port icon has a clock waveform drawn within it to indicate its added capabilities.



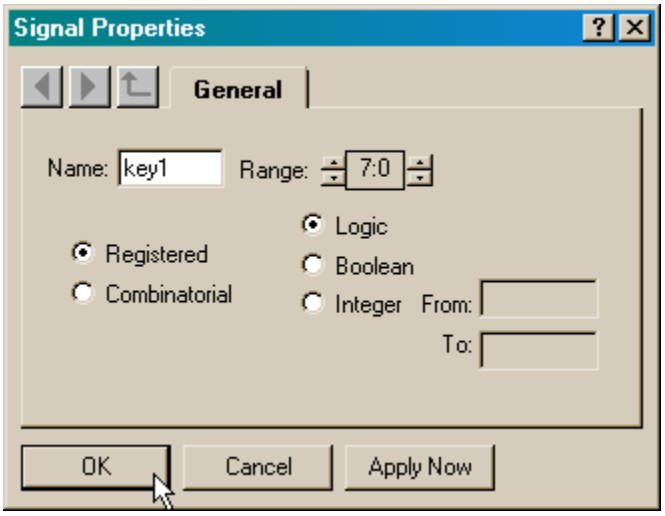
Now we will add signals to the FSM. These signals will store values used internally by the FSM. Click the Signal button and drag a signal icon into the upper portion of the editing area.



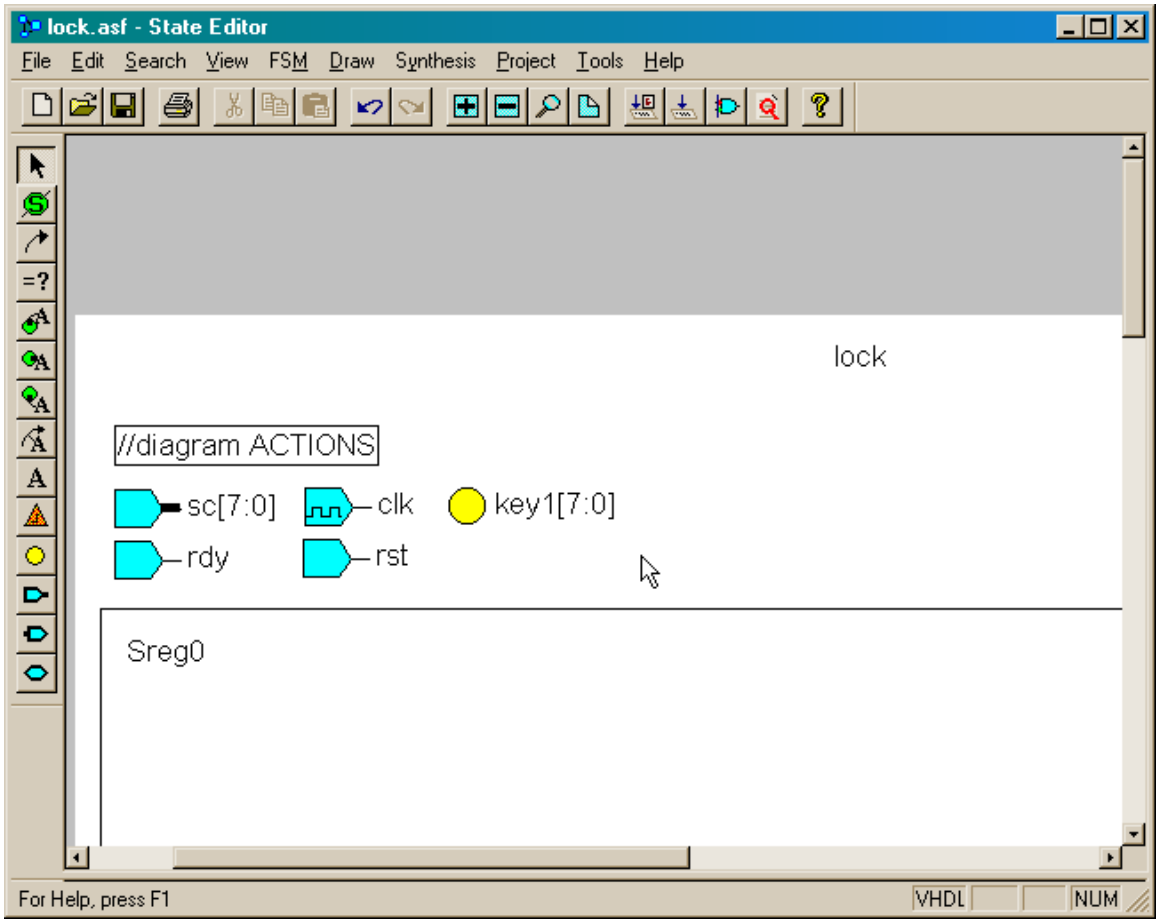
Right-click on the signal icon and select the Properties... entry in the pop-up menu that appears.



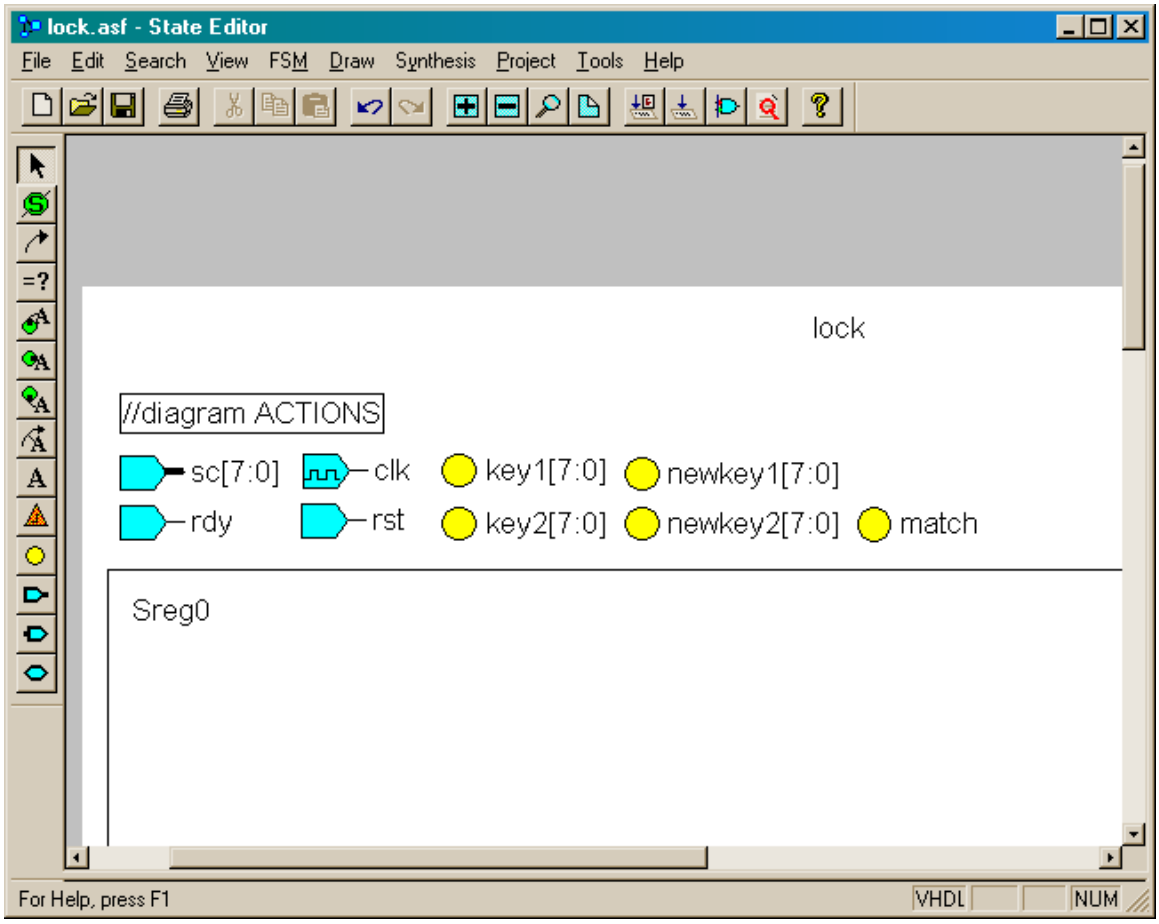
This signal will be used to store the first of the two keys in the combination for the lock. Each key stores a scancode so it must be eight bits wide. In the **Port Properties** window, rename the signal to key1 and then click on the upper-left button of the Range input until it is eight bits wide. Then click on the OK button.



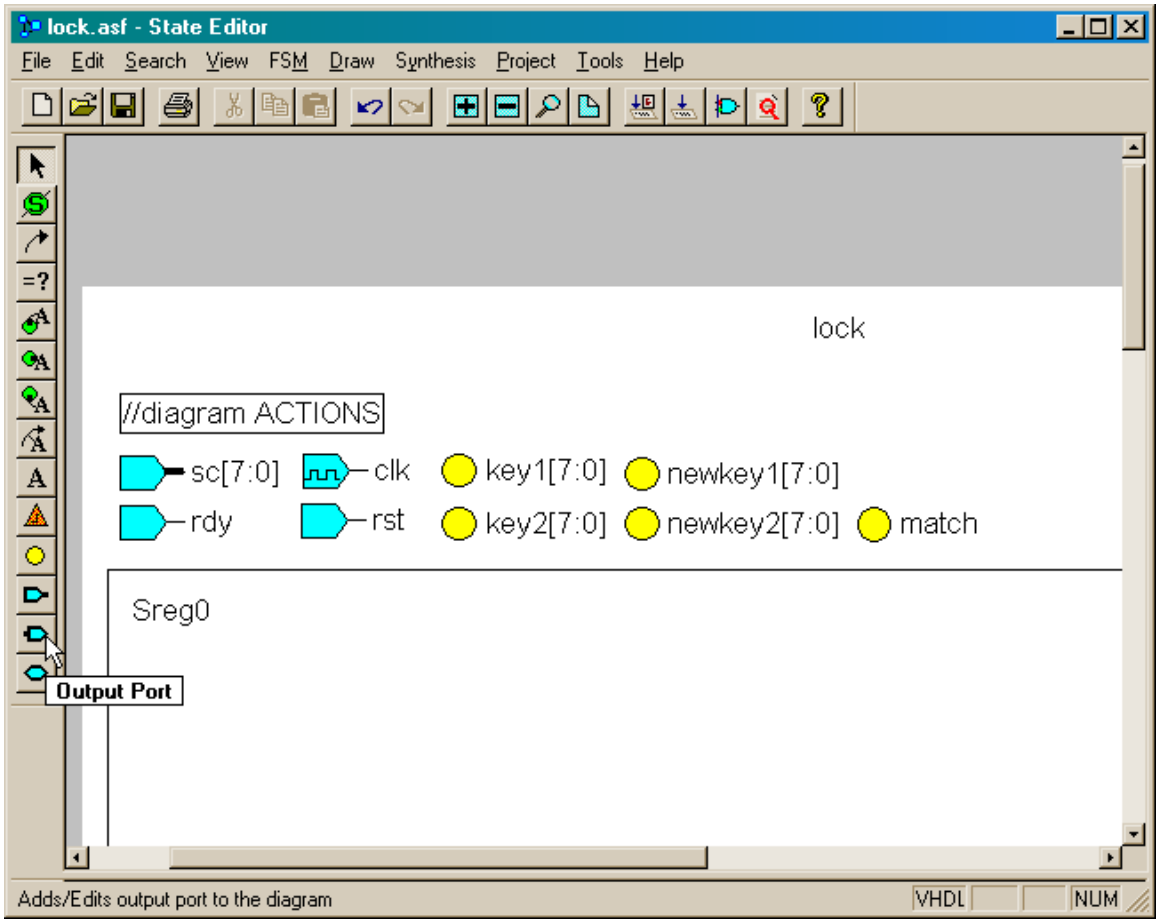
Now the signal in the editing area appears with its new name and the upper and lower indices for its bus width.



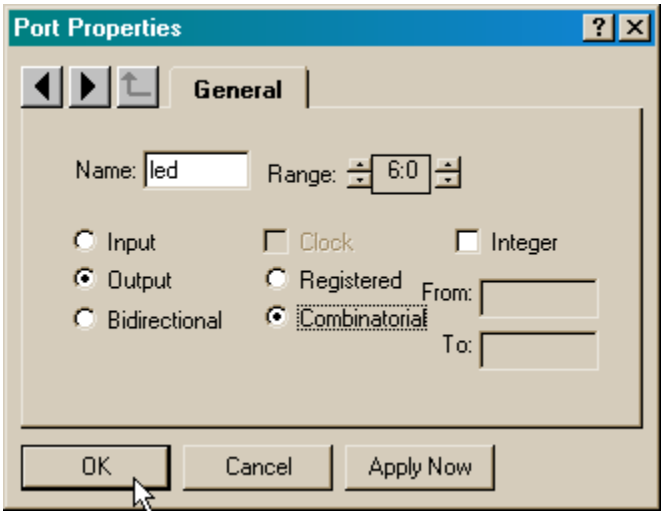
Repeat this process to add the signals for storing the second key (key2). We also add two more byte-wide signals that temporarily store the keys for the new combination as they are entered by the user(newkey1 and newkey2). And there is a single-bit signal (match) that records whether the input keys punched by the user match the keys in the combination.



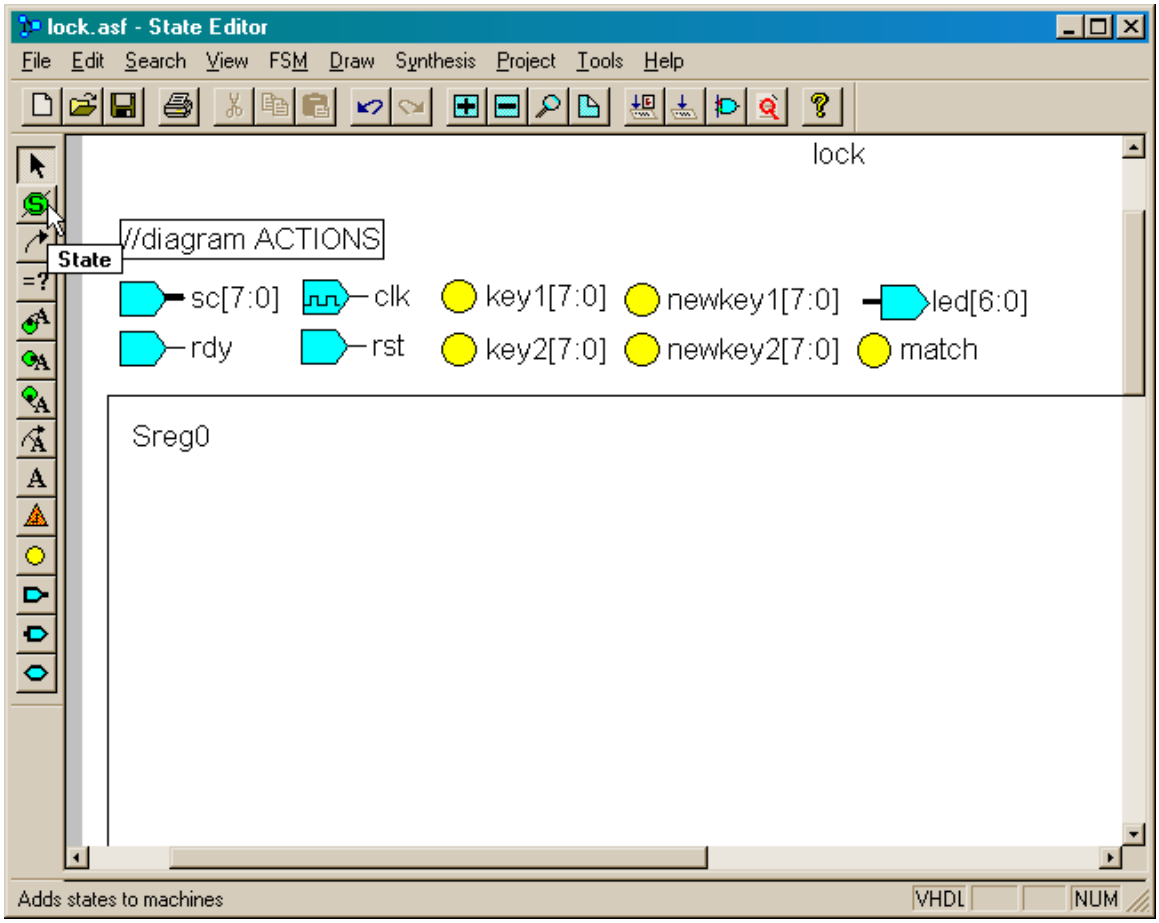
Now we will define the output ports for the FSM. Click the Output Port button and drag the port icon into the editing area. Then right-click on it and select the Properties... command from the pop-up menu.



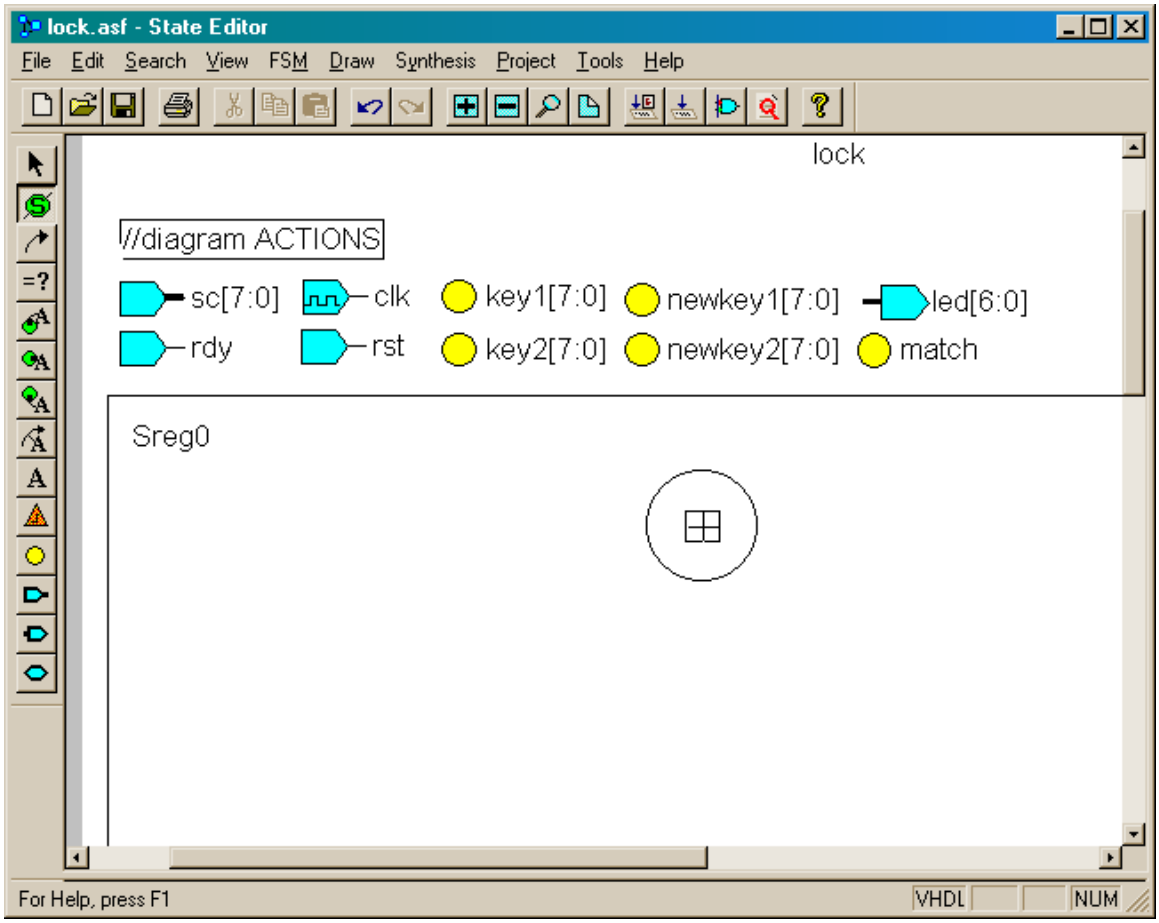
The FSM will have a single, combinatorial output that drives the seven-segment LED. Rename the output to led, set the indices to 6:0, and click on the Combinatorial radio button. Then click on the OK button to close the window.



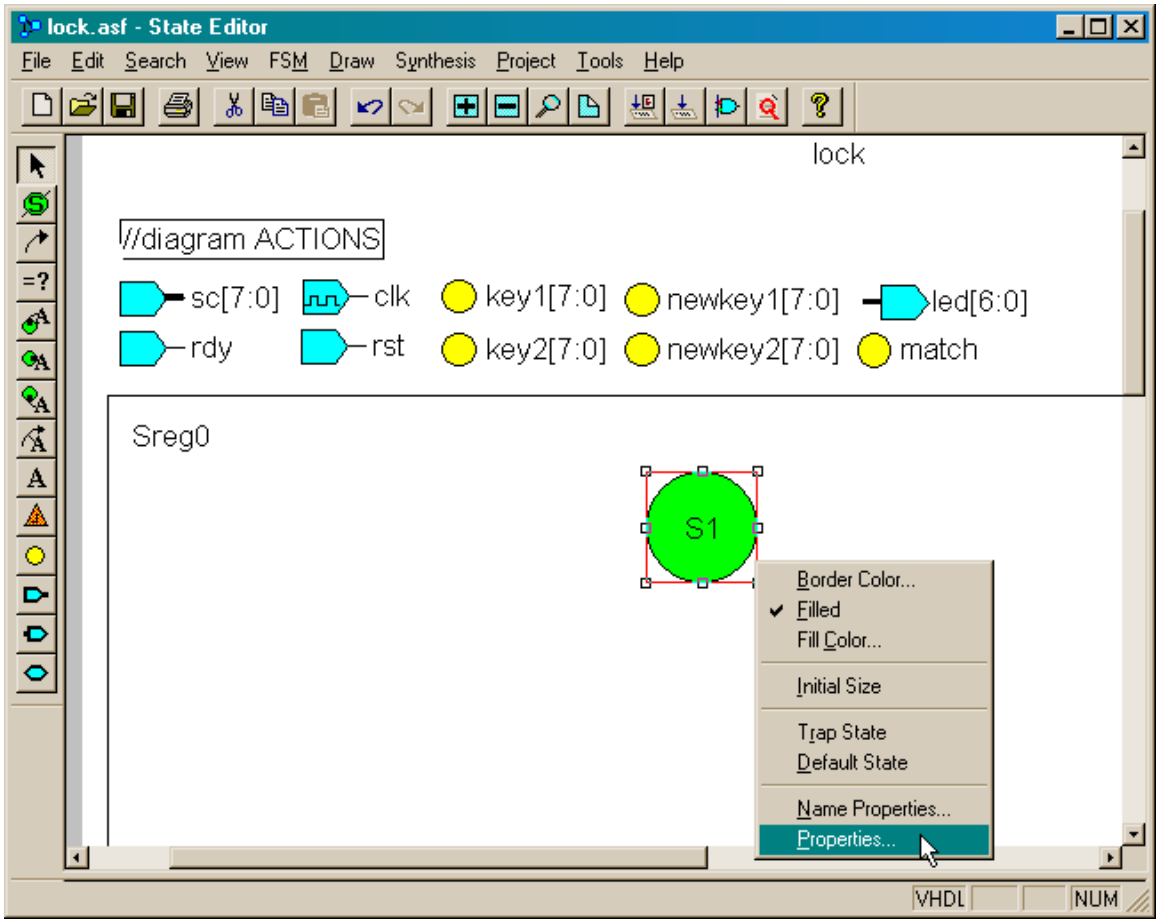
The definition of the FSM interface is complete now that the seven-bit wide led output icon has been added. Now we can click on the State button and begin defining the states for the FSM.



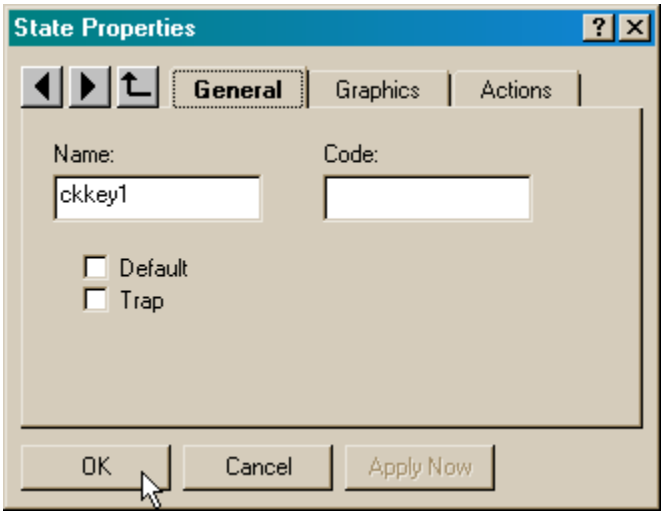
Drag the circular state icon into the lower area of the editing area and left-click with the mouse to drop it.



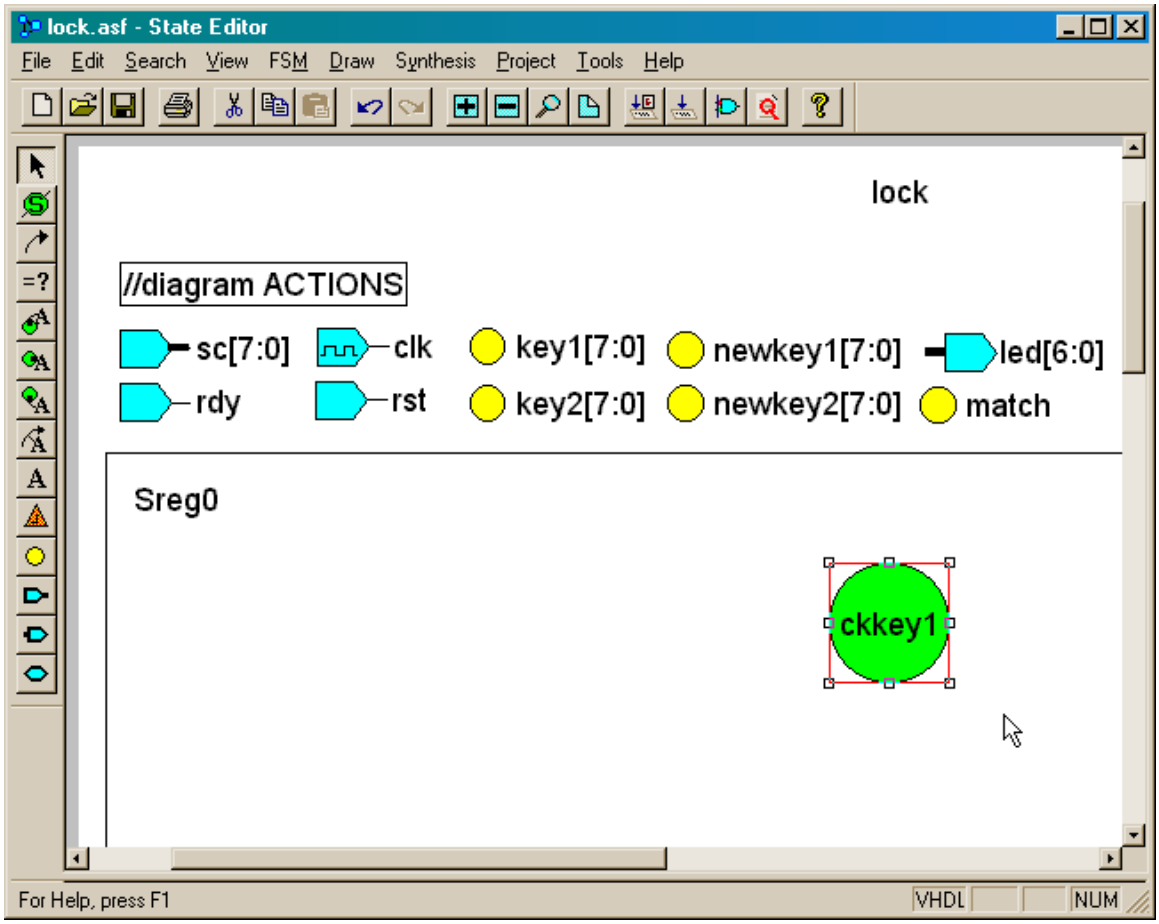
The state icon will appear with a default label of S1. Right-click on the S1 icon and select the Properties... menu item.



In this state, the FSM will compare the scancode entered by the user with the value stored in key1. In the **State Properties** window that appears, rename the state to ckkey1.



The state icon in the editing area is now labeled with its new name.



Repeat this process to add seven more states to the FSM. The names of the states in the FSM are:

ckkey1: Compare the scancode for the key pressed by the user against the scancode value stored in the first key of the combination, key1.

ckkey2: Compare the scancode for the key pressed by the user against the scancode value stored in the second key of the combination, key2.

unlocked: Open the lock if the user pressed the two keys whose scancodes match the two keys in the combination.

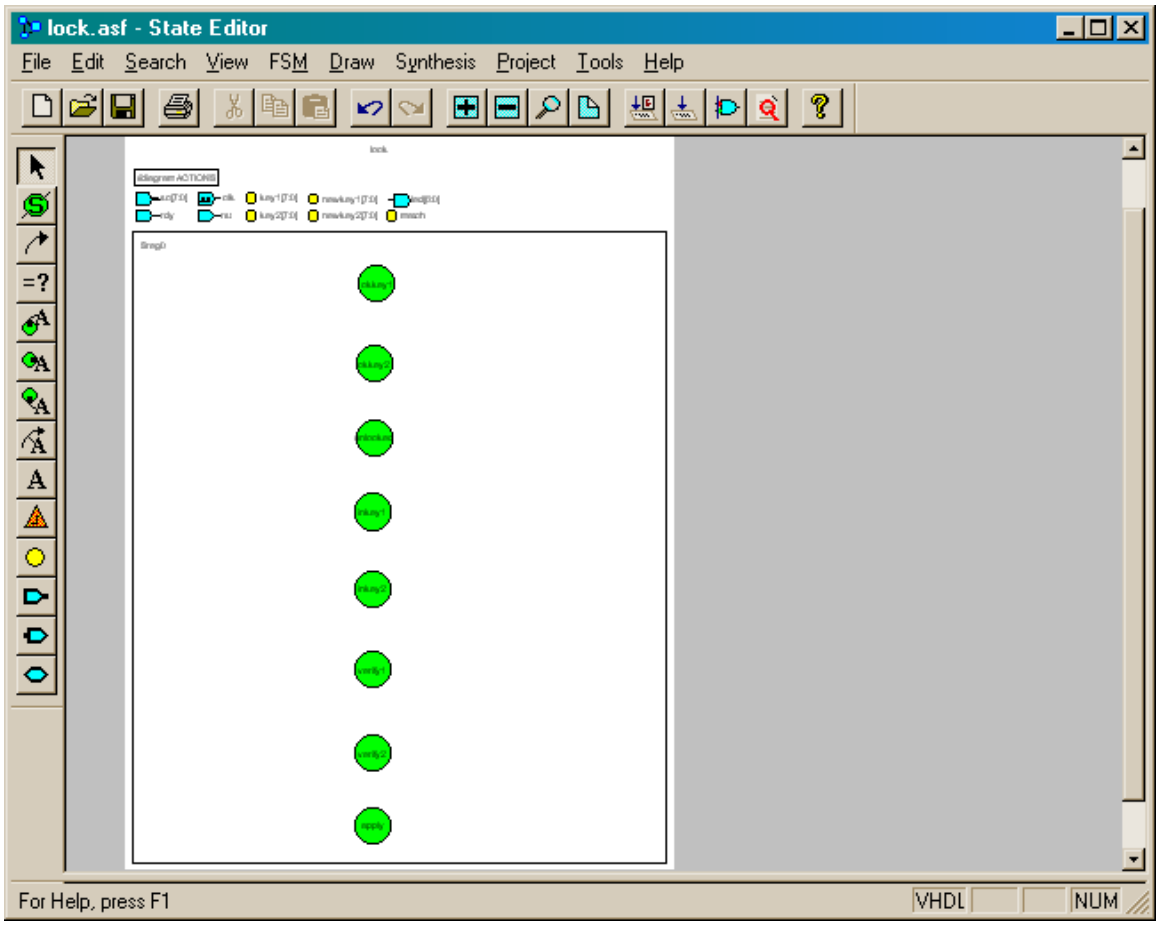
inkey1: Accept a scancode and store it in newkey1.

inkey2: Accept a second scancode and store it in newkey2.

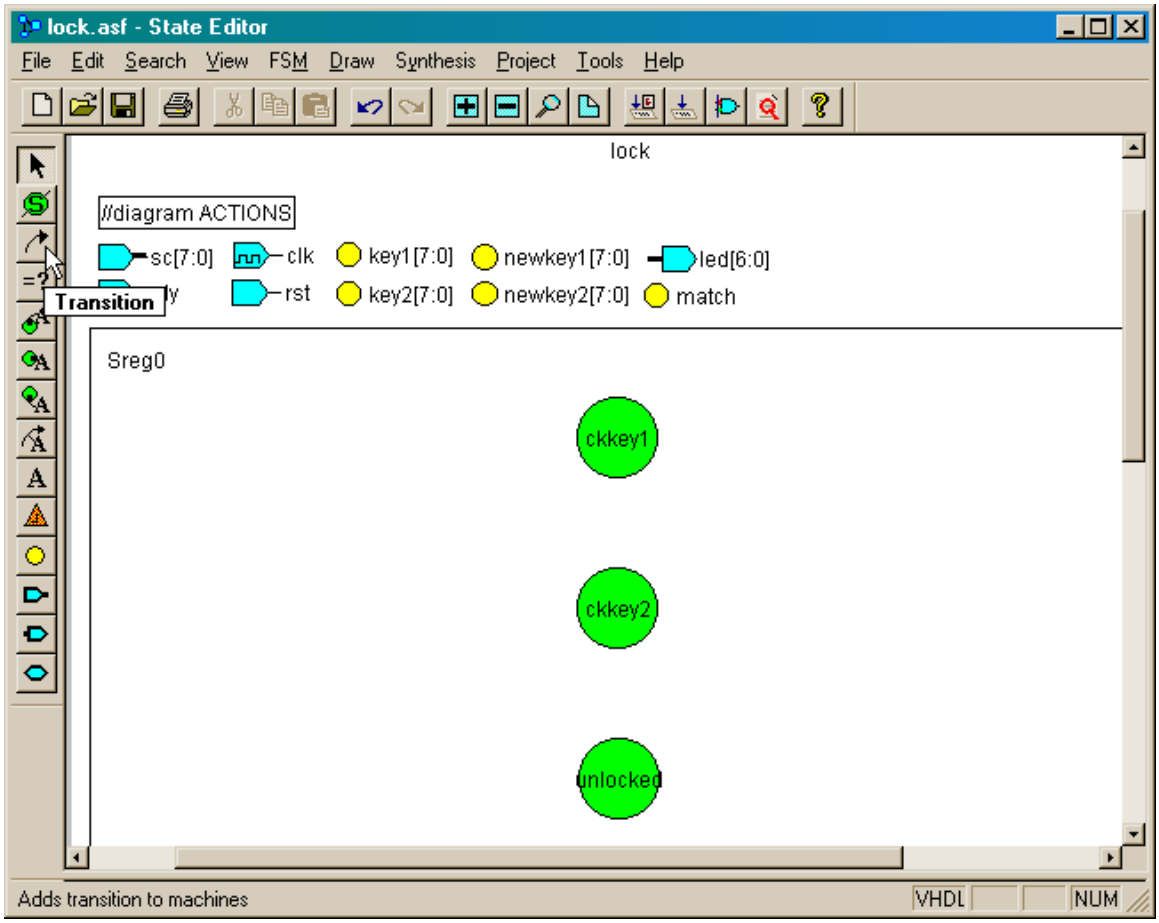
verify1: Accept a scancode and compare it to the scancode stored in newkey1.

verify2: Accept a second scancode and compare it to the scancode stored in newkey2.

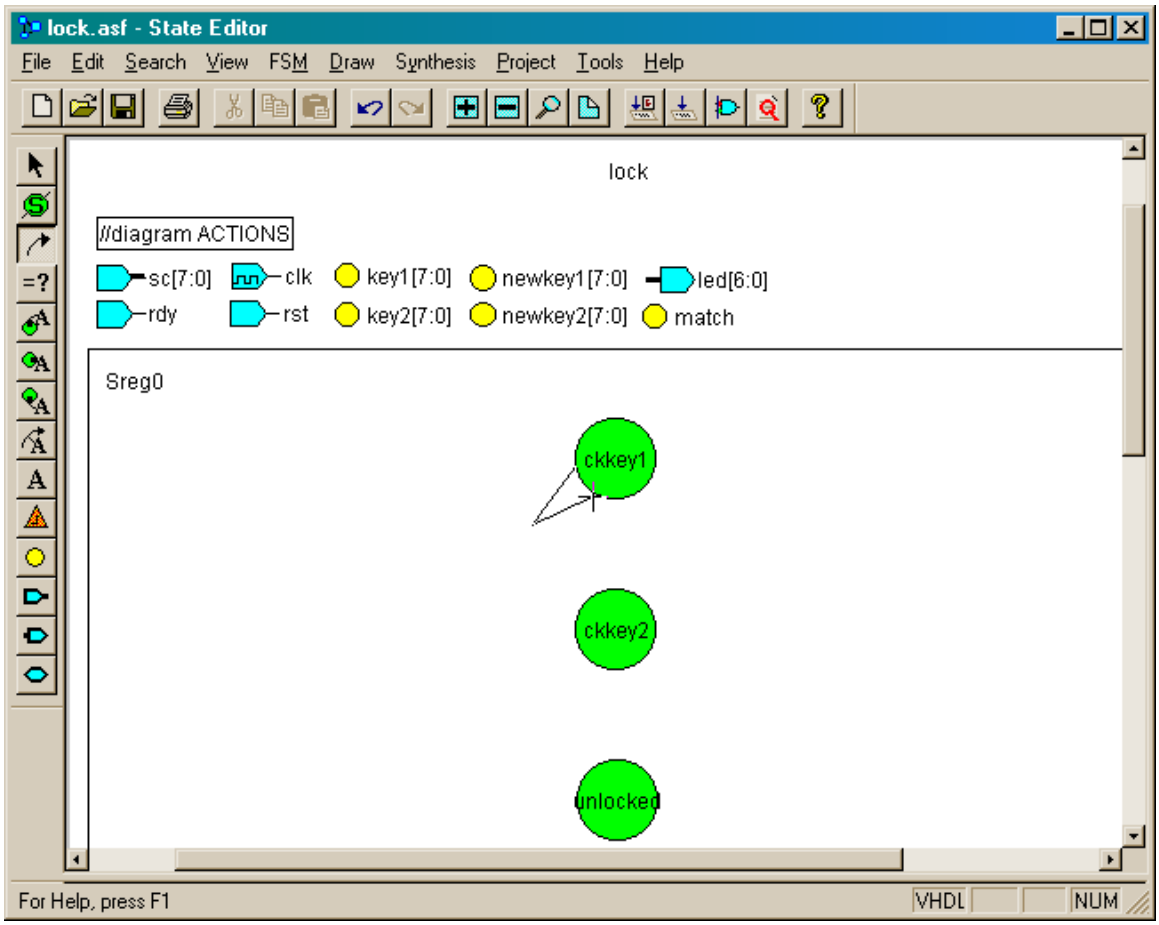
apply: Replace the combination stored in key1 and key2 with the new combination in newkey1 and newkey2.


lock.asf - State Editor
File Edit Search View FSM Draw Synthesis Project Tools Help
For Help, press F1
VHDL
NUM

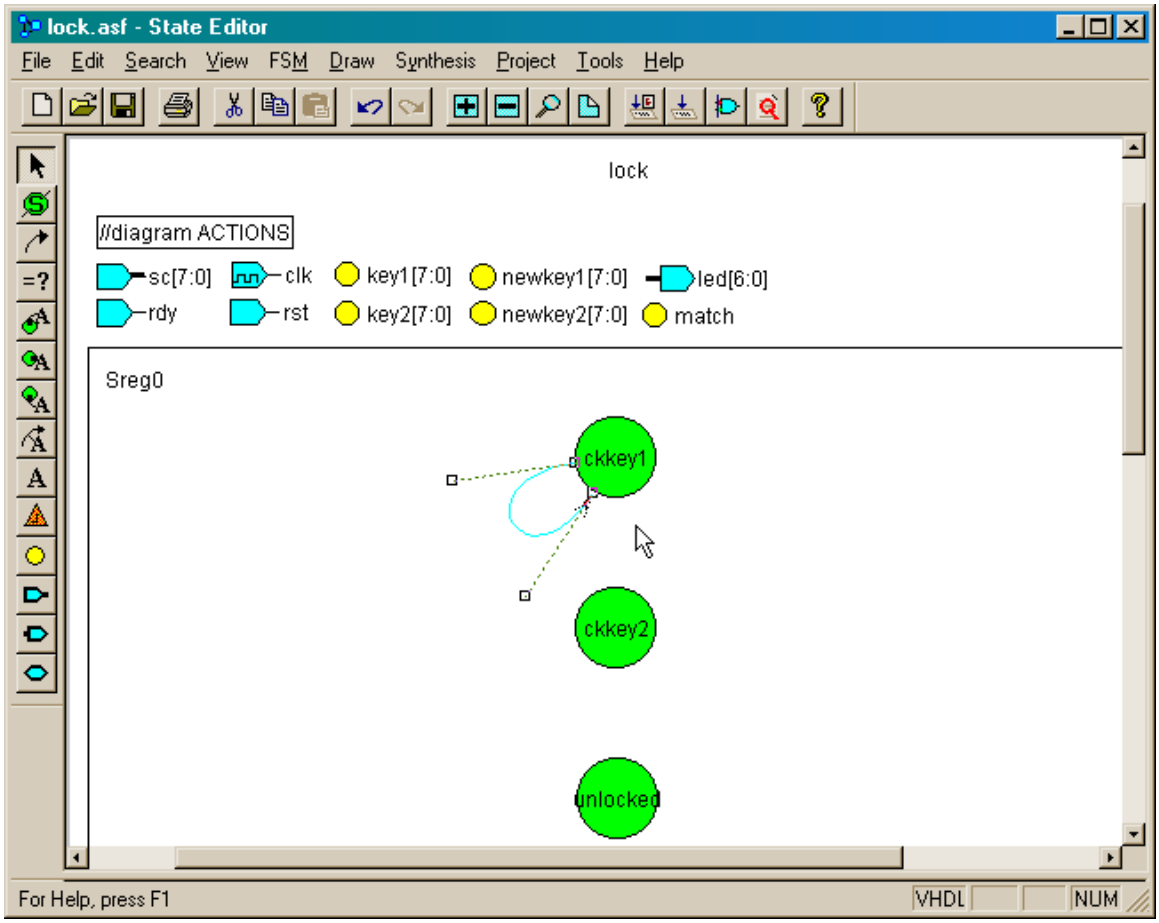
Once the states are entered in the editing area, we can begin drawing the transitions between the states by clicking on the Transition button.



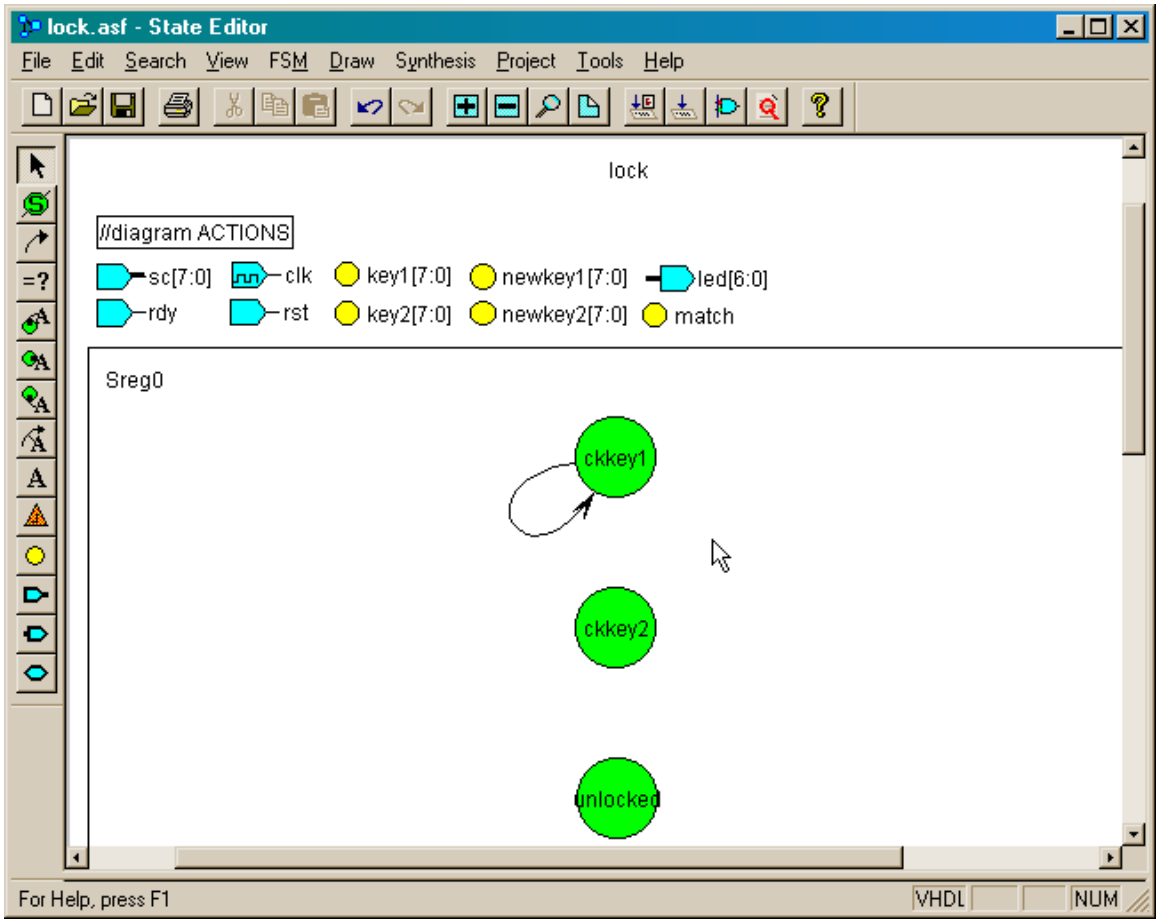
When drawing a state transition, left-click on the state that will be exited, then left-click to place intermediate points in the editing area, and finally left-click on the destination state that will be entered. In the case shown below, the transition exits from state ckkey1 and then returns to the same state.



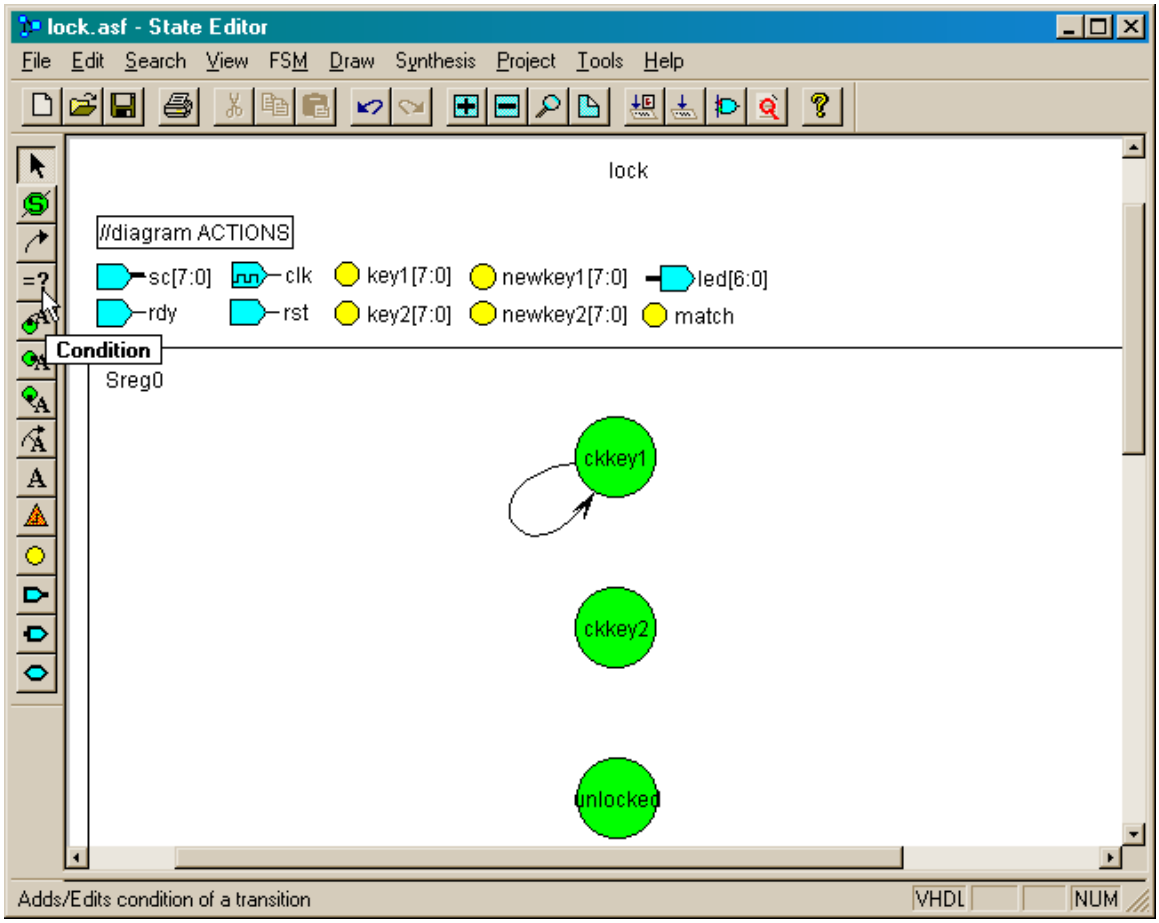
After clicking on the destination state, the transition is drawn with visible control points that you can click-and-drag to change the appearance of the transition edge.



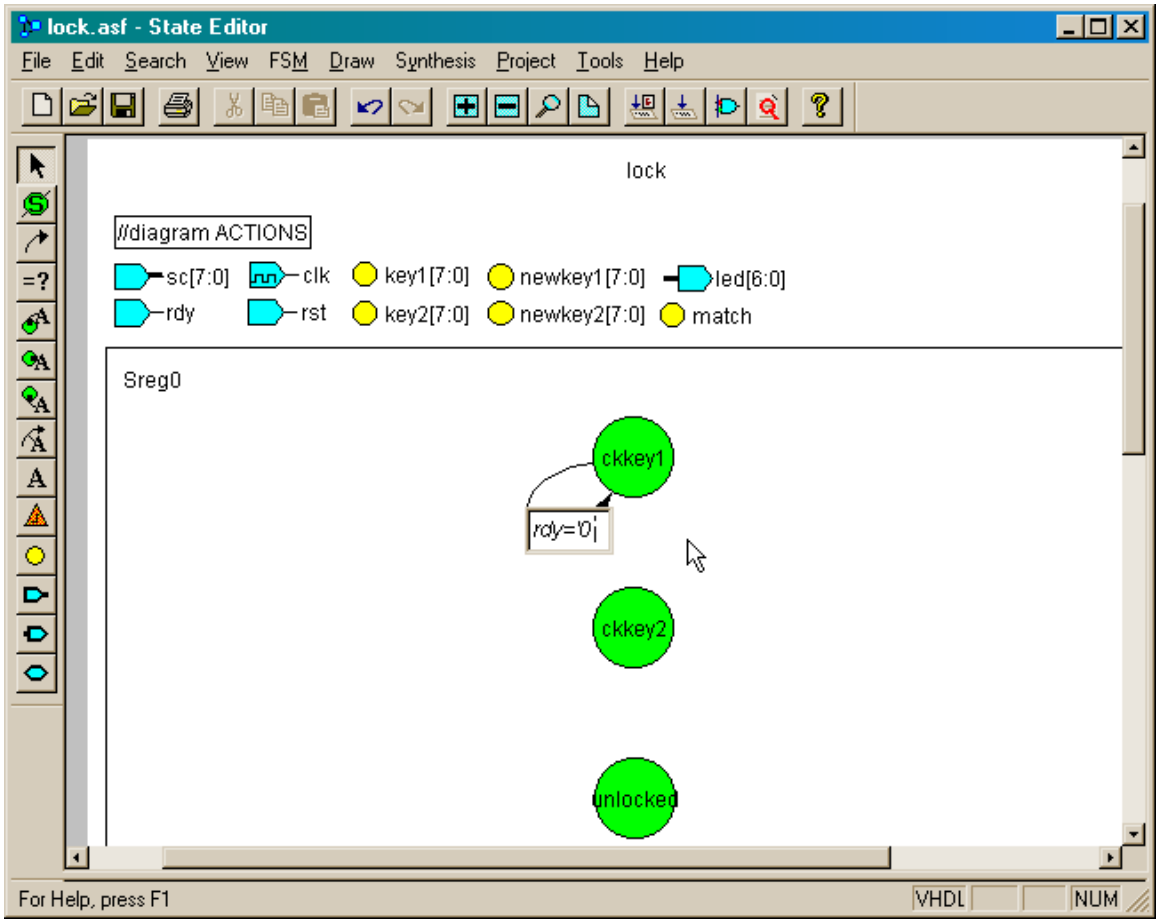
Click in a blank portion of the editing area to end the editing of the transition. The final transition will appear as a directed edge with an arrowhead indicating the direction of the movement from state to state.



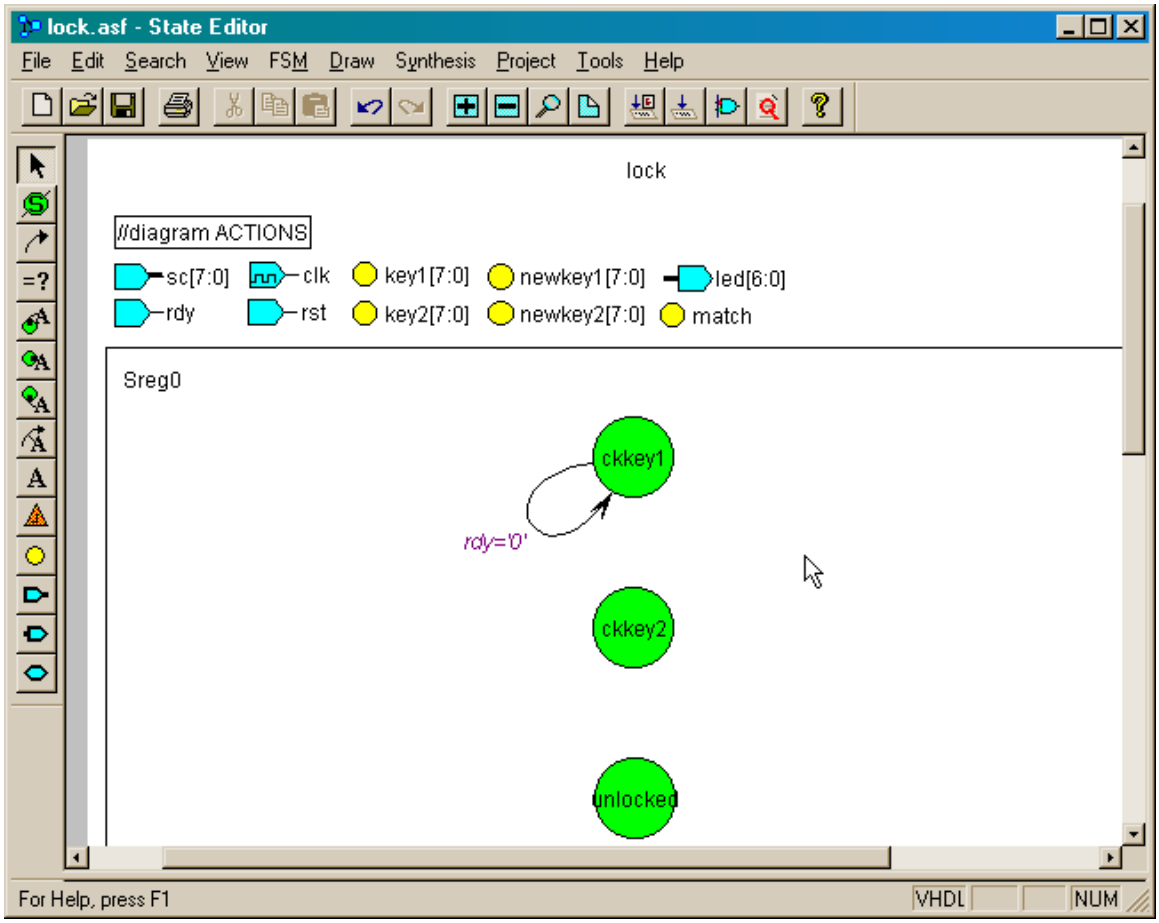
Now we have to specify the conditions under which the state transition will occur. Click on the Condition button to begin this process.



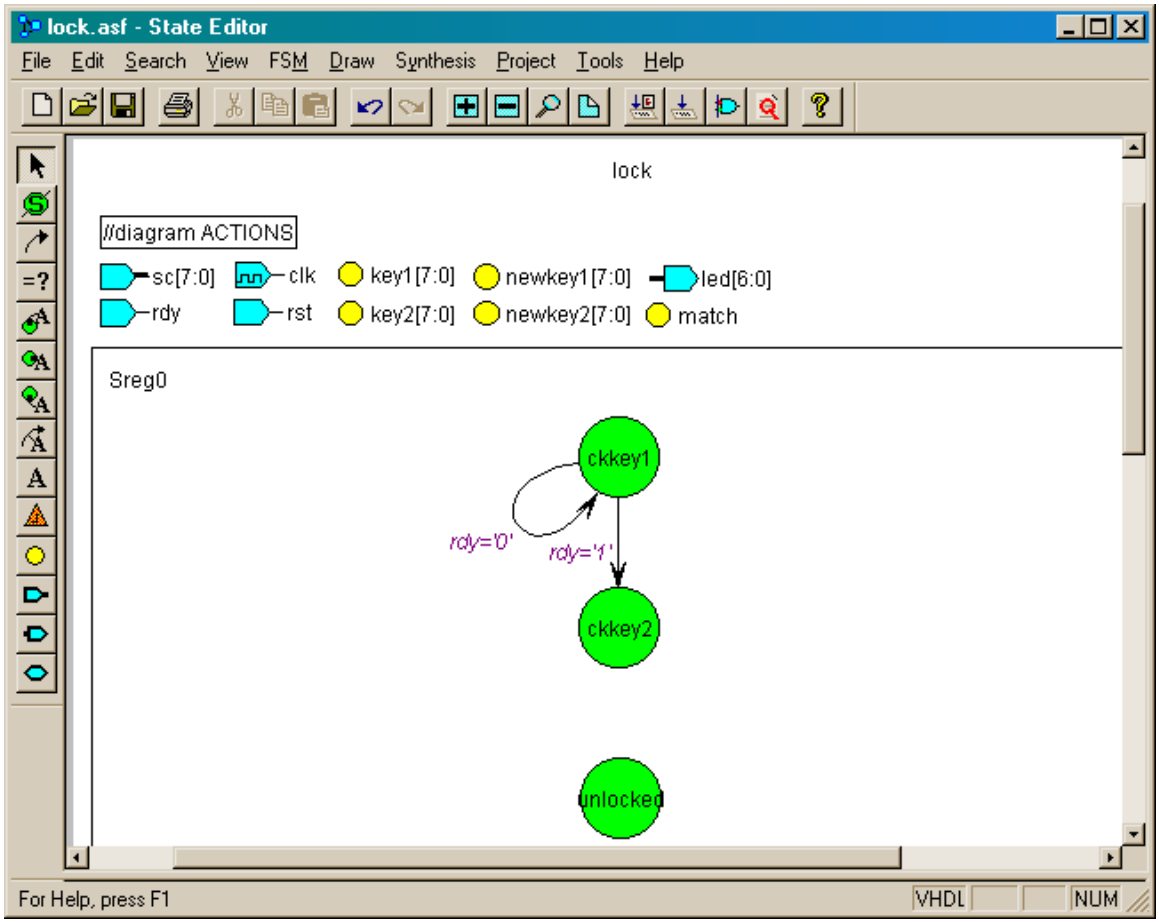
Next, left-click the mouse on the transition edge. An editing box will appear where we can enter the equation for the condition. In this case, the FSM remains in the ckkey1 state as long as the user doesn't press a key on the keyboard. So this transition is taken as long as no new scancode is available from the keyboard interface. The VHDL code for this condition is rdy = '0'.



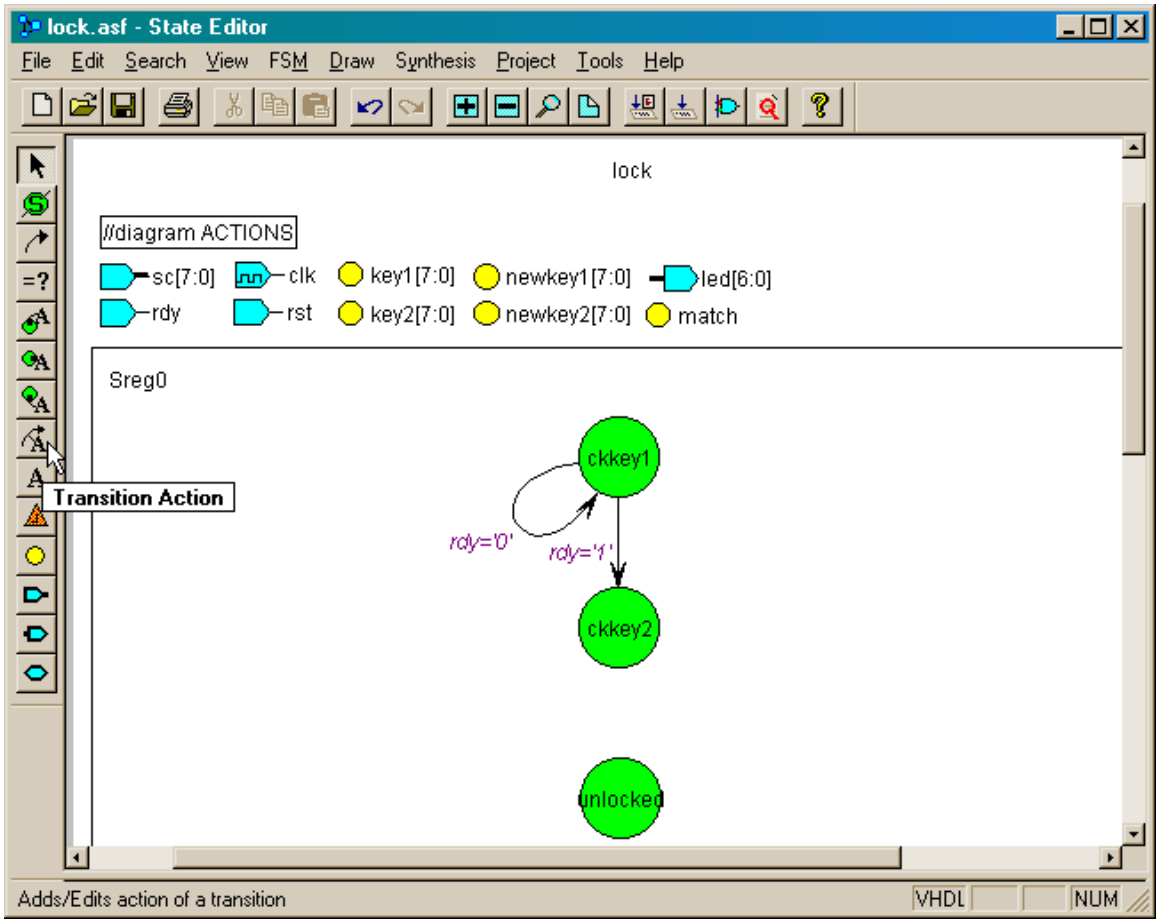
Once the VHDL code is entered, hit the return key or click the mouse outside the editing box and the condition equation will appear next to the transition edge. You can click-and-drag the condition equation to arrange its position.



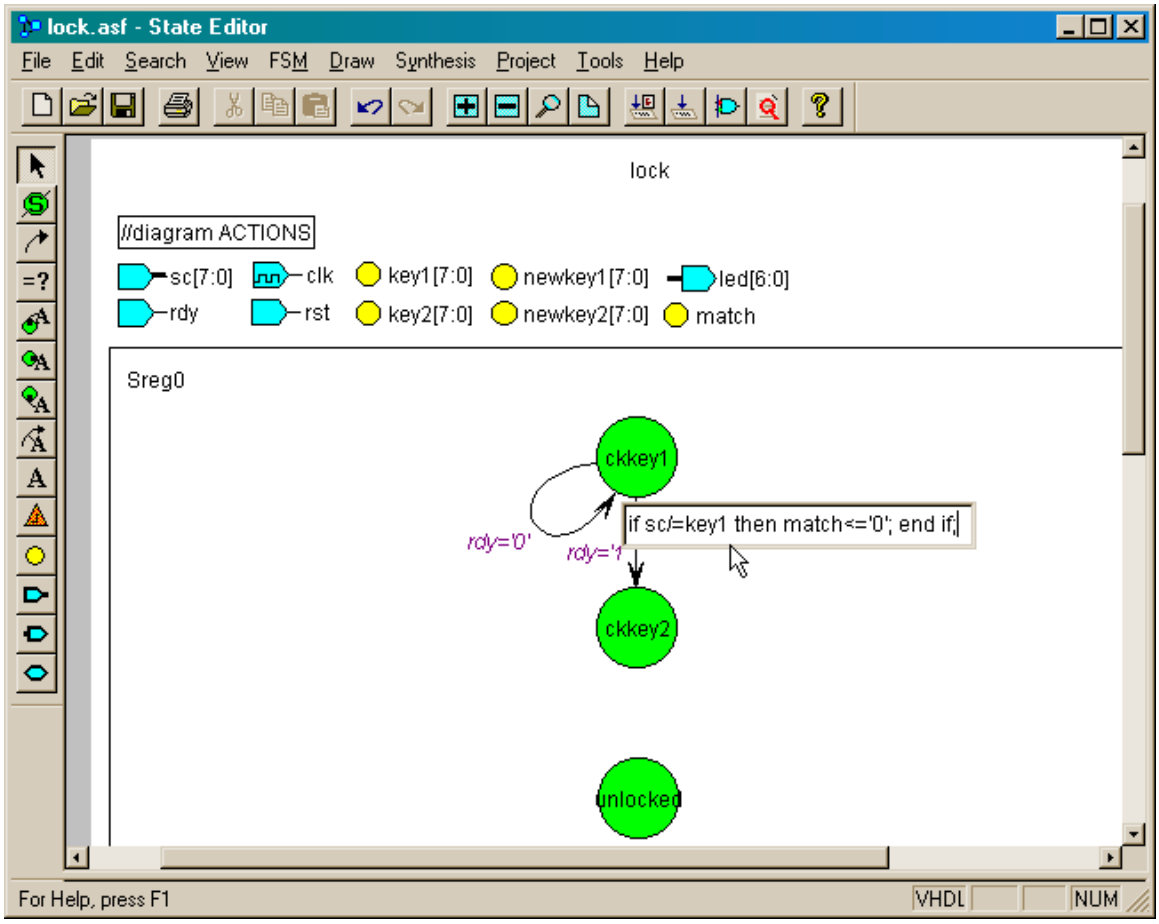
Using the same process, we can add another transition from ckkey1 to ckkey2 that is taken whenever a scancode is available (rdy = '1').



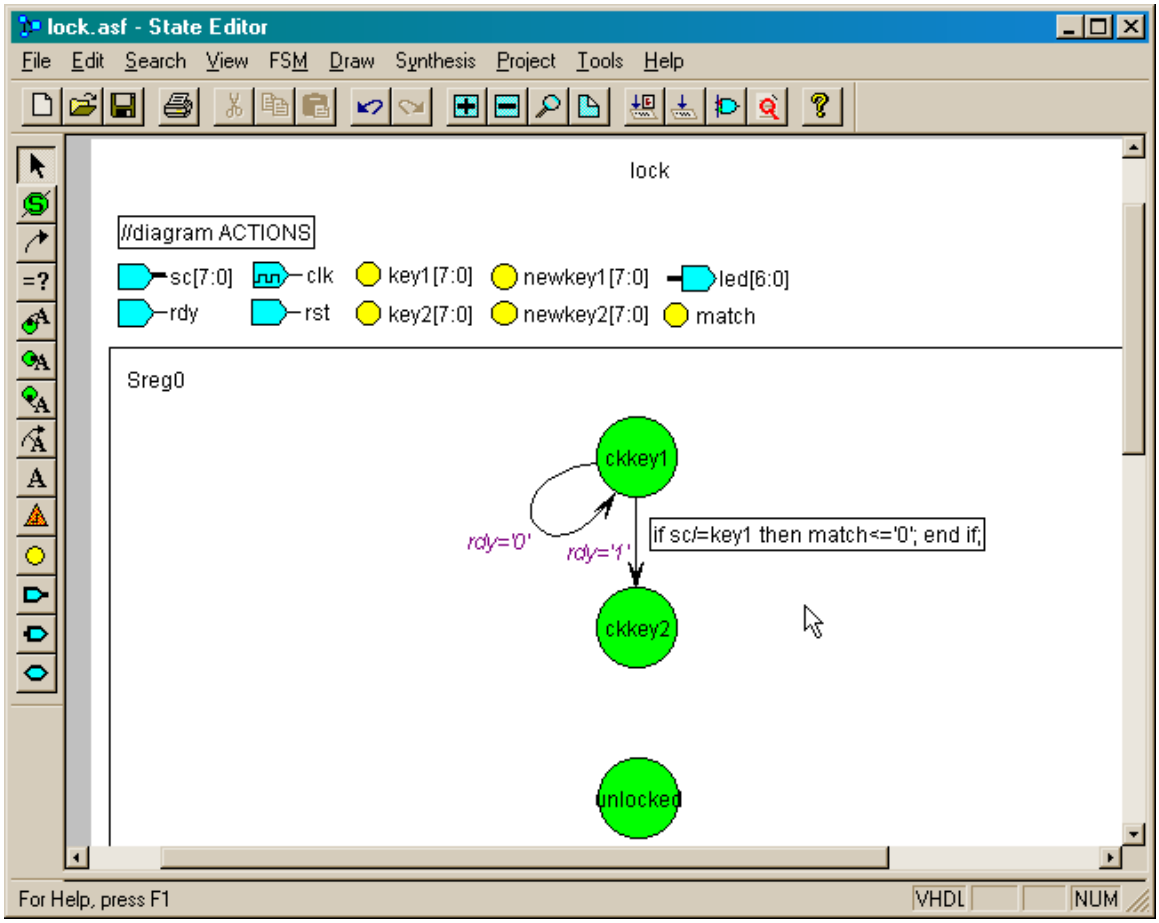
When a scancode enters from the keyboard, the FSM has to check and see if it matches the scancode stored in key1. To do this, we click on the Transition Action button so we can add this checking action to the transition from ckkey1 to ckkey2.



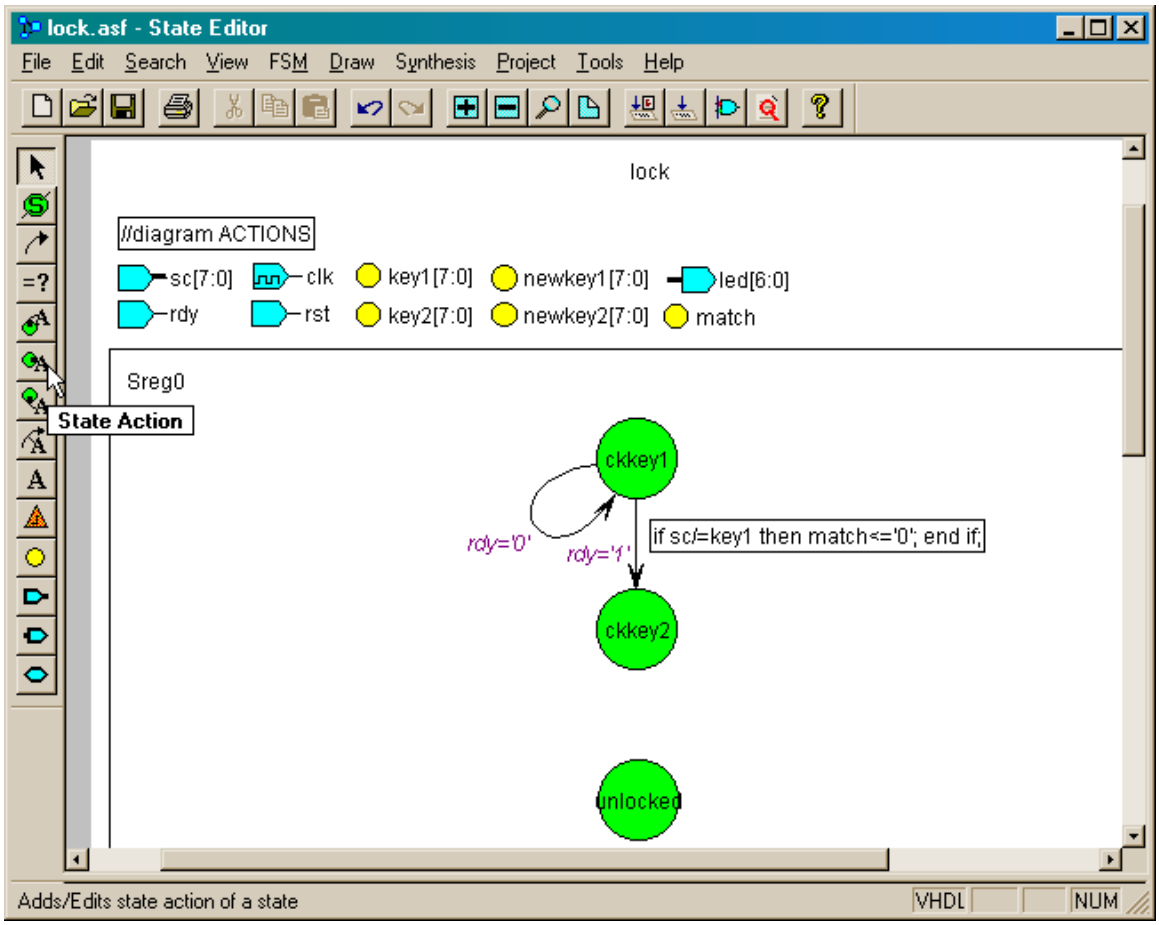
Click on the transition and then enter the transition action in the editing box that appears. In this case, the match flag is cleared if the entering scancode, sc, does not match the scancode in key1. If the match flag is cleared, this indicates one or more of the keys entered by the user did not match the keys in the combination so the lock should not be opened.



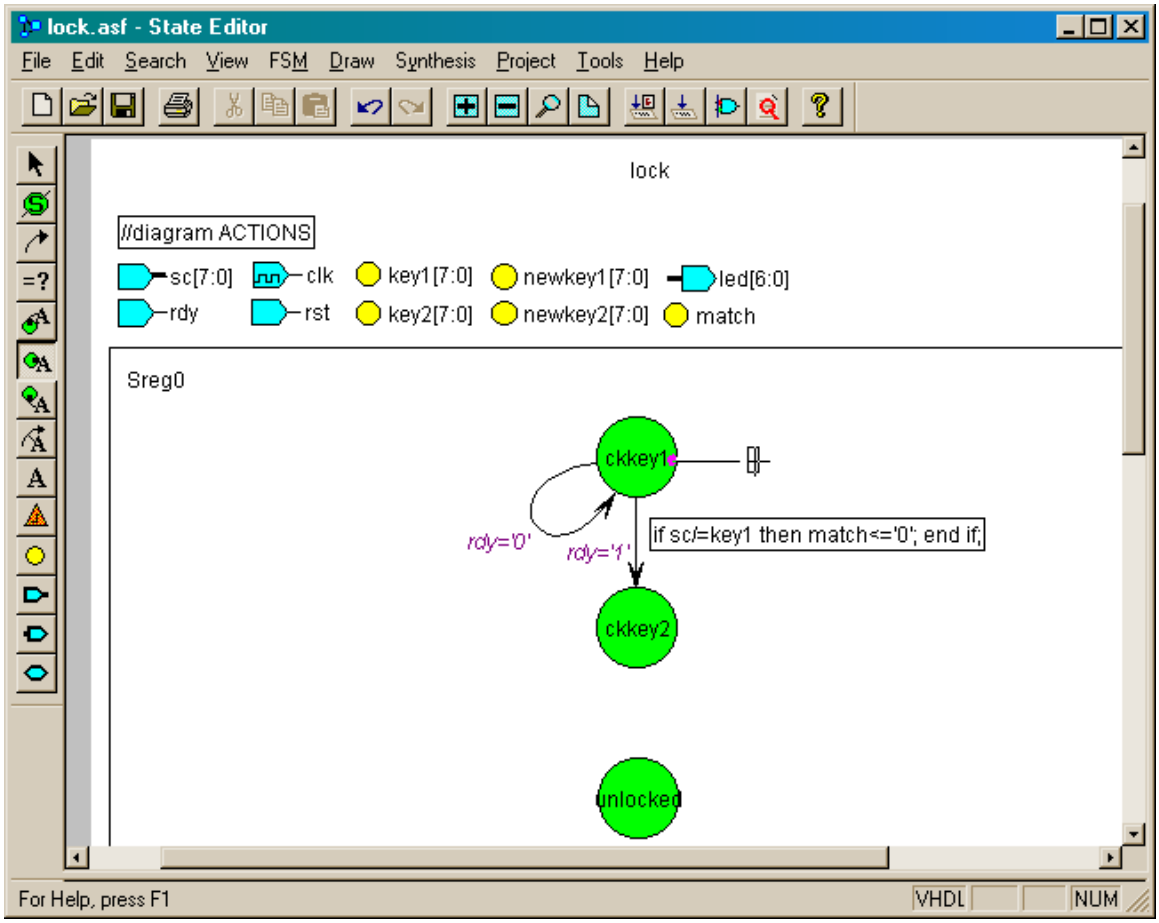
After clicking the mouse outside the editing box, the transition action appears in a rectangle next to the transition edge. The rectangle distinguishes the action from the condition that activates the transition itself.



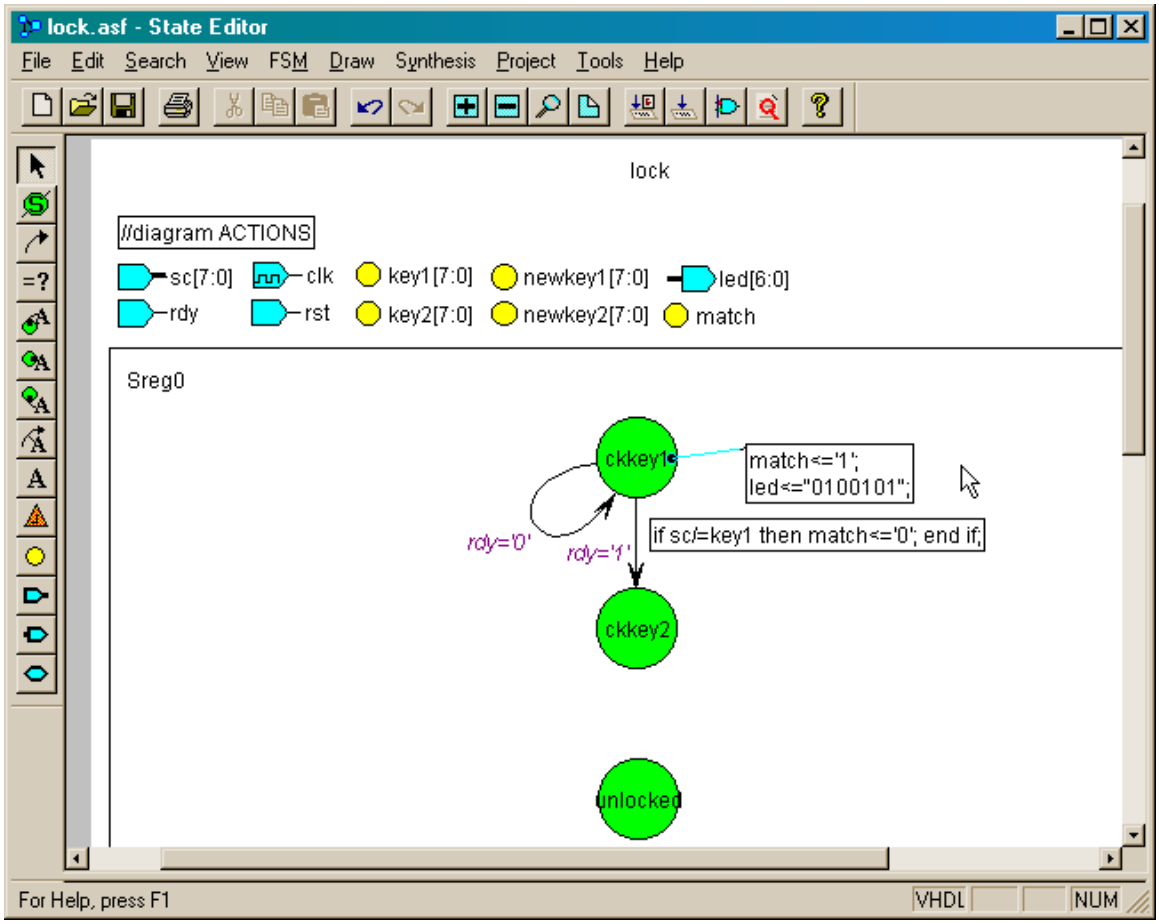
If the FSM resets the match flag to indicate the lock should not open, then we have to initially set the match flag before the comparison to the combination begins. This action occurs when the FSM is in the ckkey1 state. Click on the State Action button to add this action.



Next, move the mouse so the dot on the end of the line attached to the cursor is within the boundary of ckkey1. Then left-click the mouse.

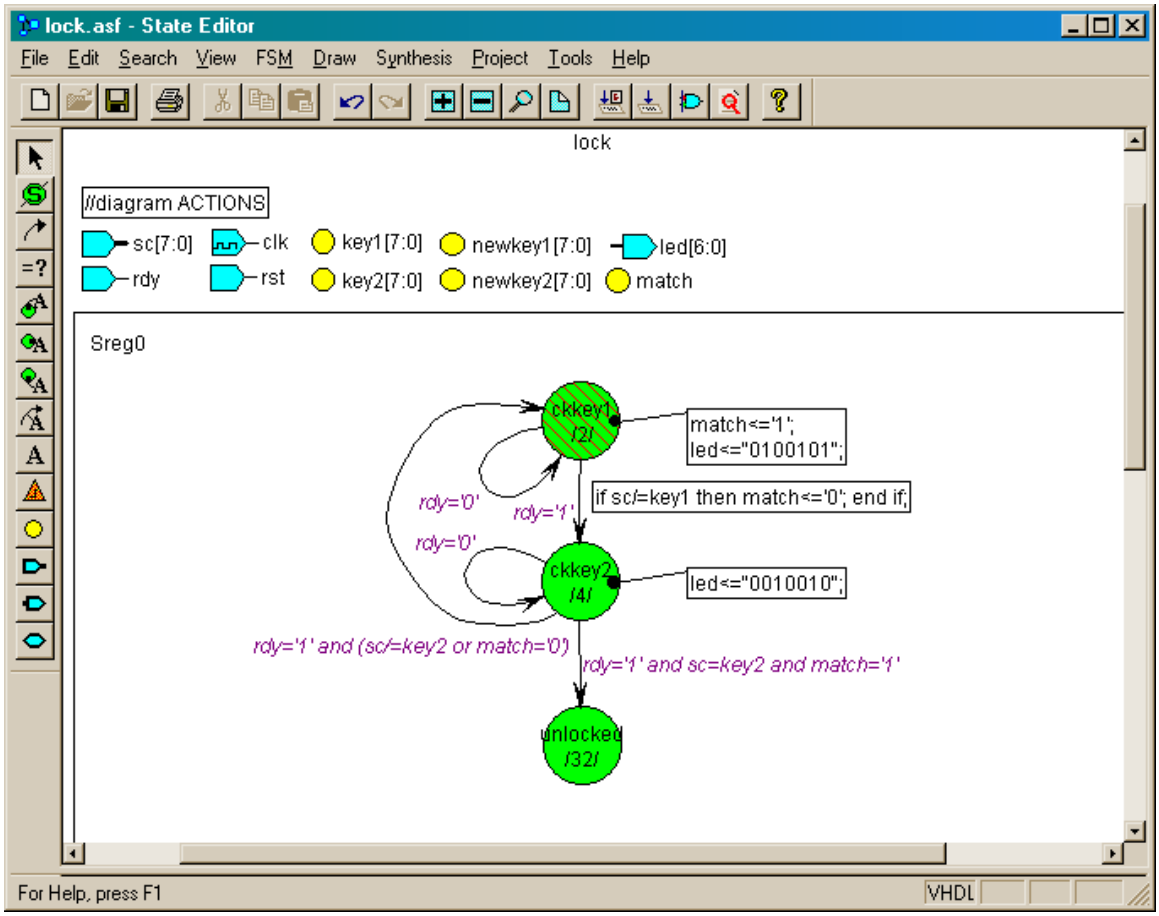


Within the editing box that appears, enter the VHDL code to set the match flag. Also assign the bit pattern “0100101” to the outputs that drive the LED. This will display an L on the LED digit to indicate that the lock is locked. Click outside the editing box to complete the addition of the state actions.

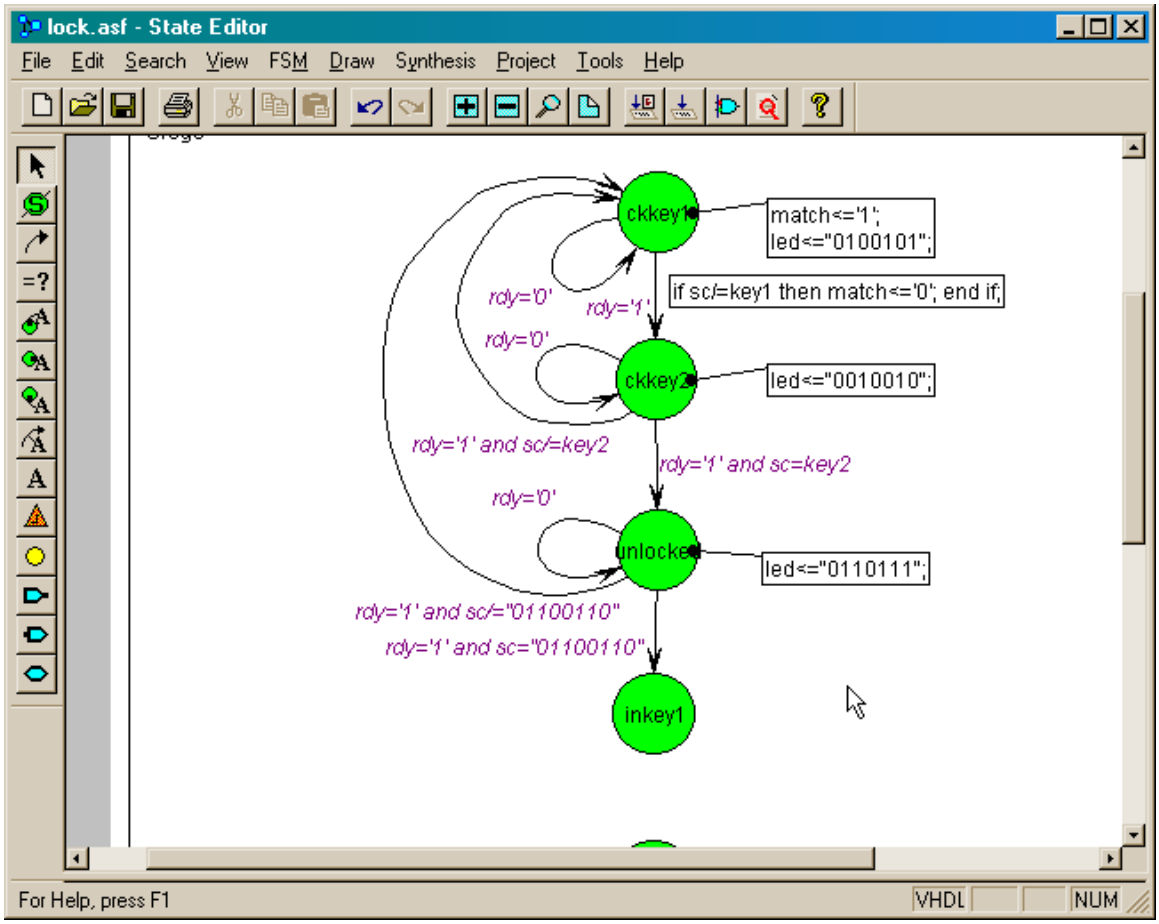


Now we can repeat the preceding steps to define the transitions and actions for the remaining states. When the FSM is in the ckkey2 state, a 1 is displayed on the LED digit to indicate that one key has already been entered by the user. A transition will be made back to the ckkey1 state if the user hits a key whose scancode does not match key2 in the combination or if the match flag is already zero (indicating a key mismatch during the previous state). But the FSM transitions to the unlocked state if the current scancode agrees with key2 and the match flag is still set.

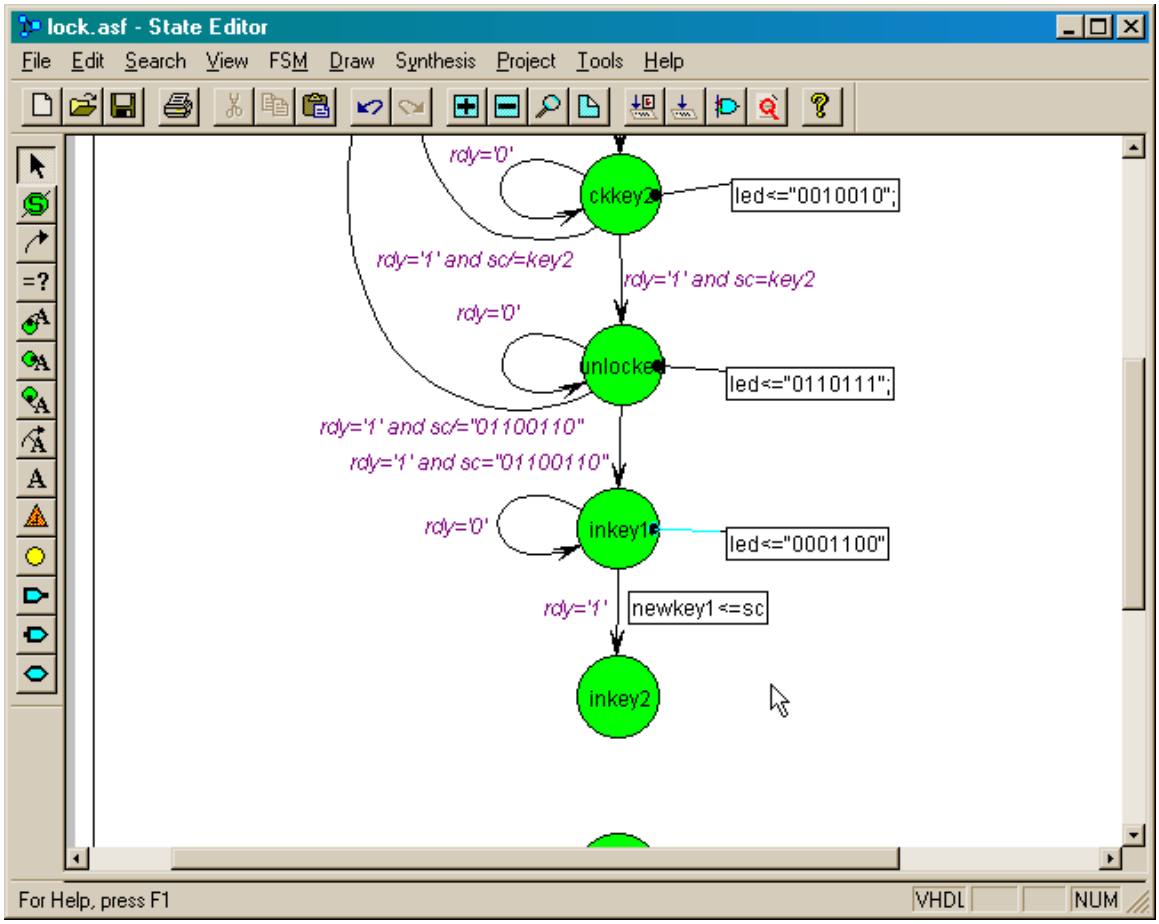
Note that this FSM requires a single clock cycle duration for the scancode ready signal. If the rdy output from the keyboard interface module stayed high for more than a single clock cycle on each key press, this would cause a transition between multiple states of the FSM.



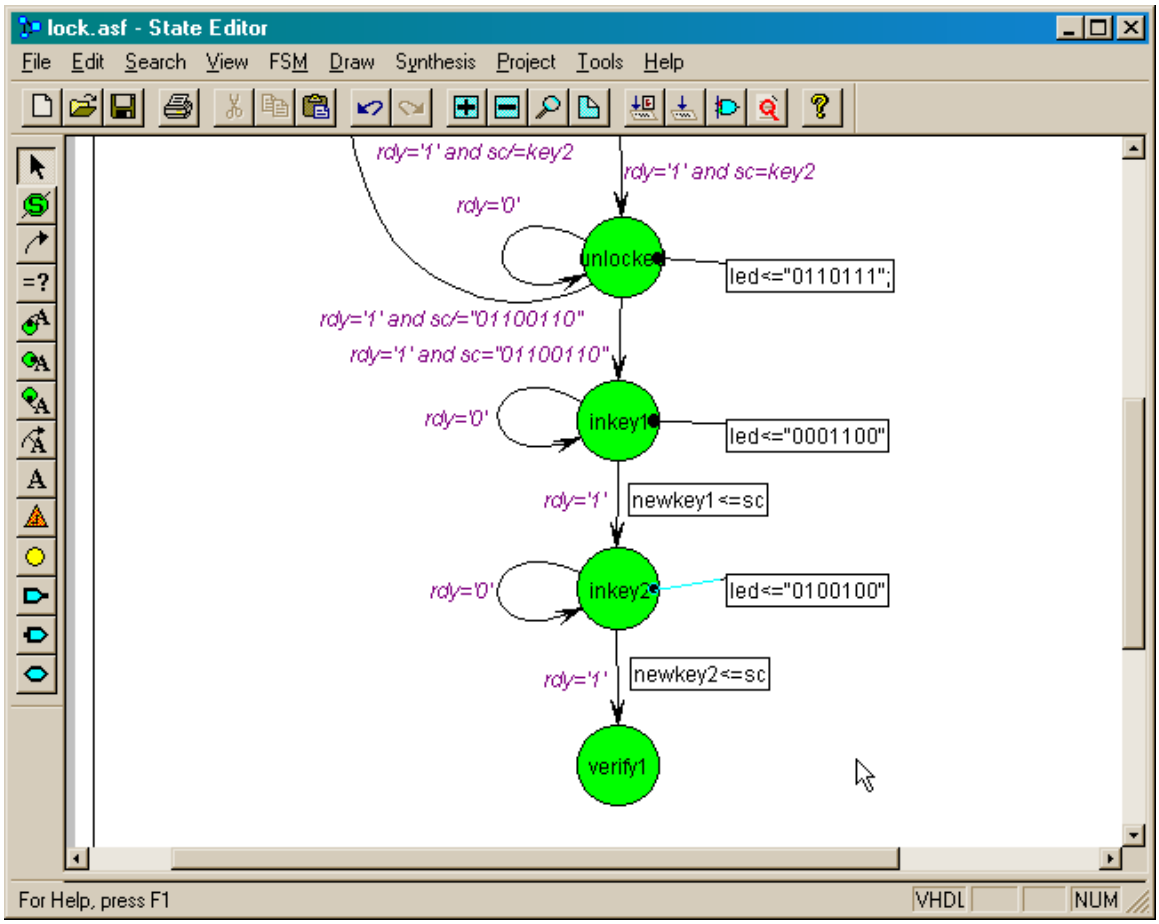
In the unlocked state, the LED digit displays a U. If the user presses the backspace key (with a scancode of 01100110), the FSM will move to the inkey1 state where they can enter a new combination. Any other key press forces the FSM back to the ckkey1 state where the lock is locked.



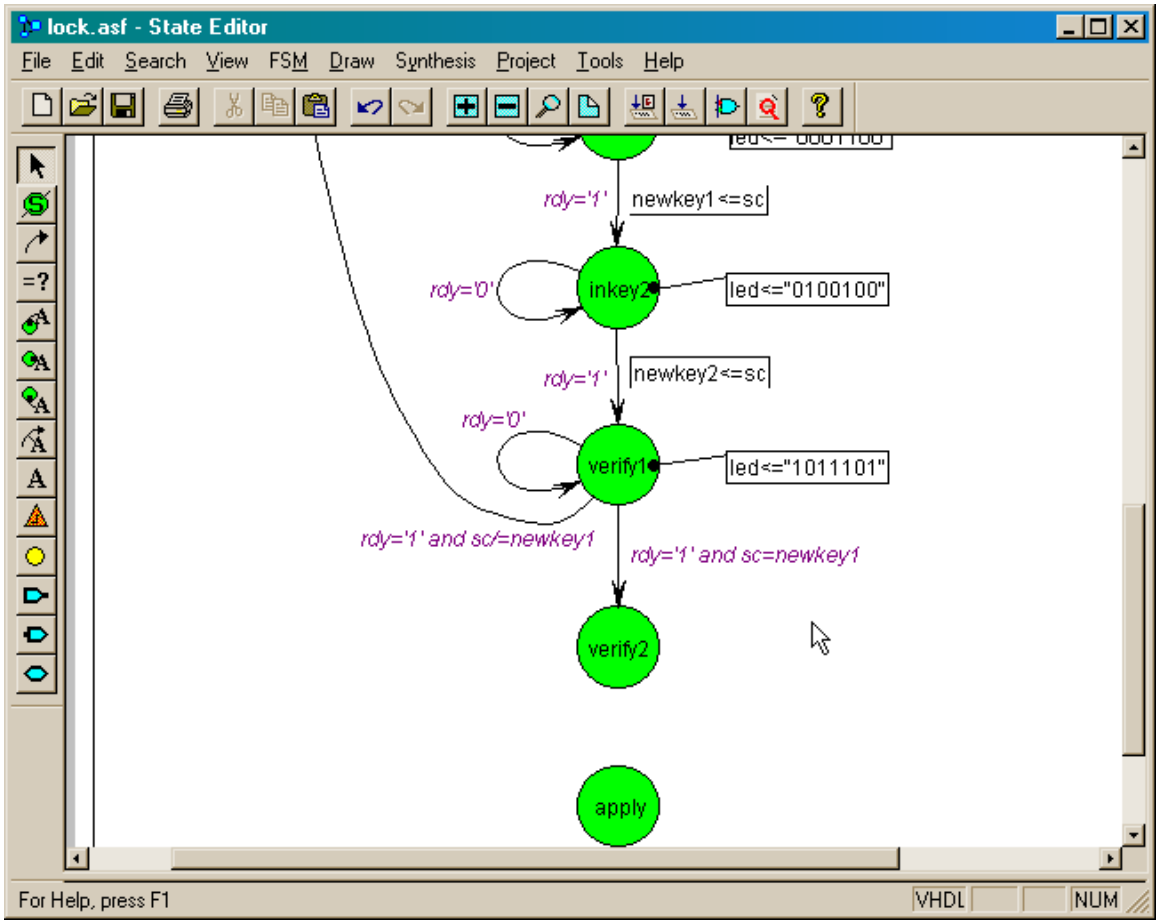
A lower-case r is displayed on the LED when the FSM is in the inkey1 state to indicate that the combination is to be replaced by the next two key presses from the user. When a scancode from the keyboard arrives, it is stored in newkey1 and the FSM moves into state inkey2.



A 1 is displayed on the LED in state inkey2 to indicate that one key of the new combination has been entered. When a second scancode from the keyboard arrives, it is stored in newkey2 and the FSM moves into state verify1.

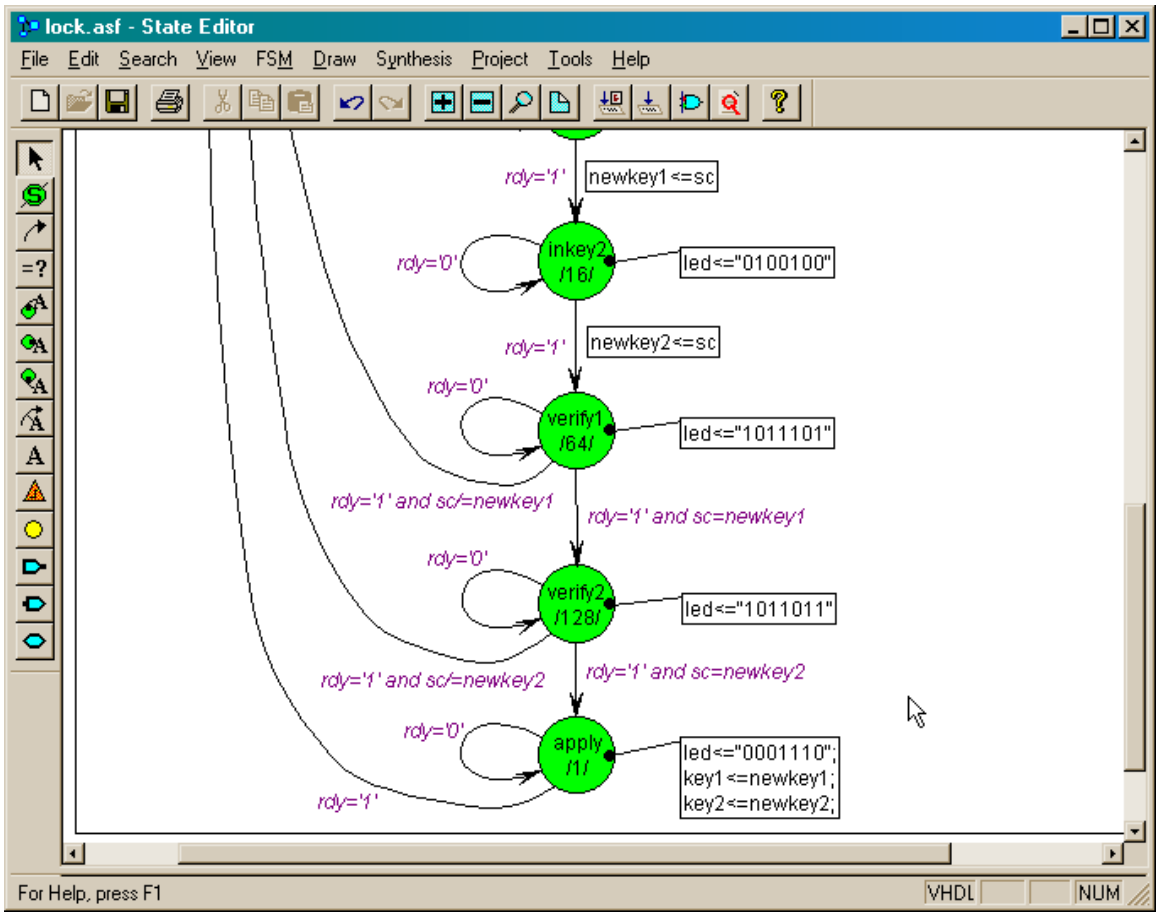


In the verify1 state, the LED displays 2 to show that both keys in the combination have been entered. The user is now required to repeat the combination entered in the previous two states. This prevents the user from erroneously entering a combination that he can't repeat and permanently locking the lock. If the user presses a key whose scancode does not match the scancode in newkey1, then the FSM transitions back to the ckkey1 state and the new combination is discarded. But if the key scancode matches the value in newkey1, then the FSM transitions to state verify2.

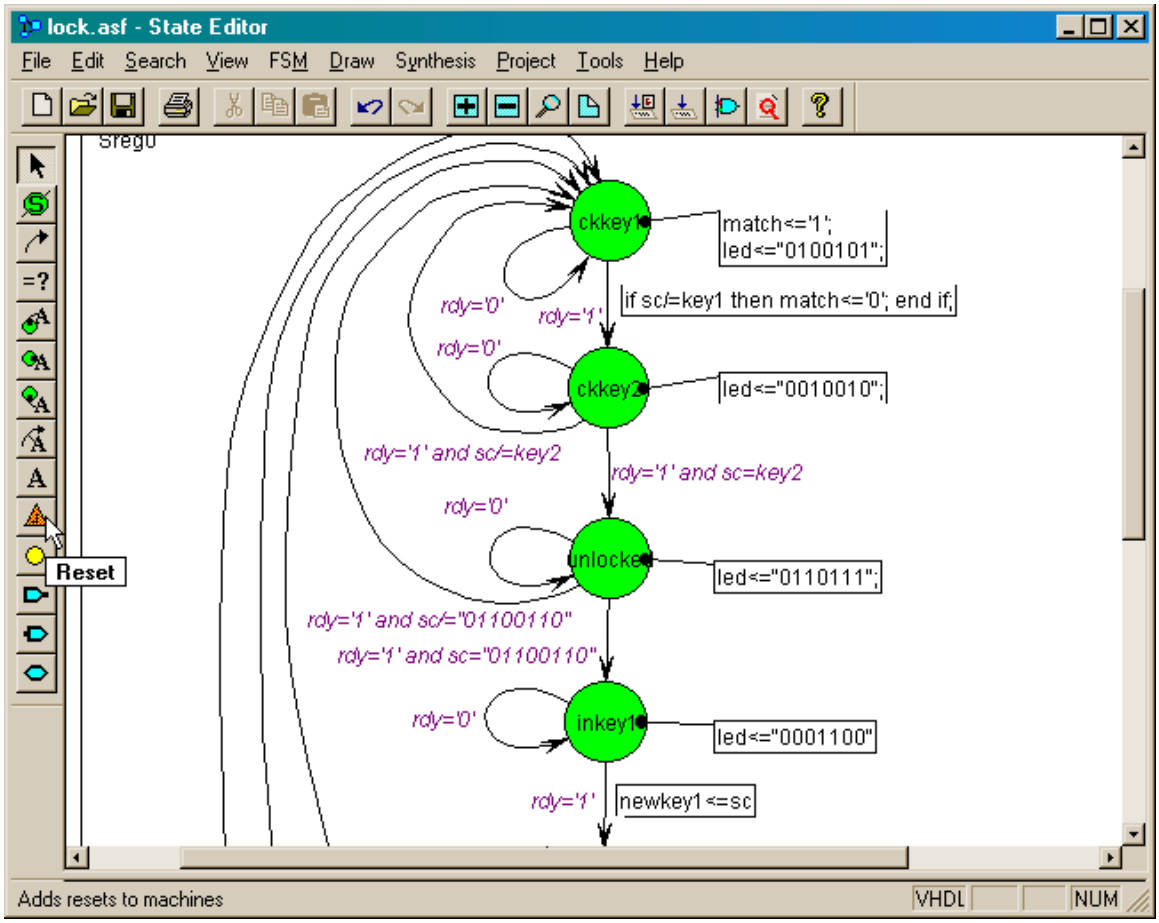


In the verify2 state, the LED displays 3 to show that both keys in the combination have been entered and one has been verified. The FSM transitions back to the ckkey1 state and the new combination is discarded if the user presses a key whose scancode does not match the scancode in newkey2. But if the key scancode matches the value in newkey2, then the FSM transitions to state apply.

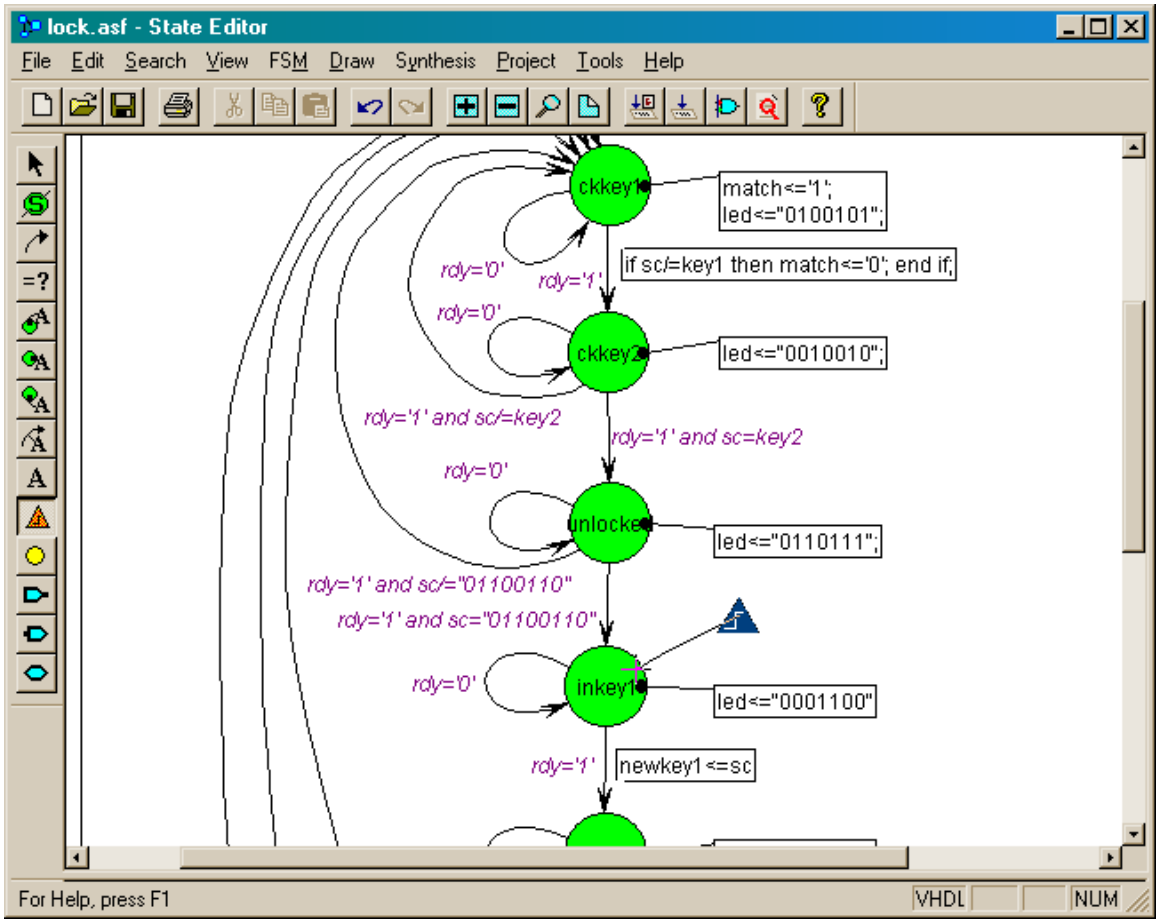
In the apply state, a lower-case n is displayed on the LED to indicate that a new combination has been accepted. The scancodes in newkey1 and newkey2 are transferred to key1 and key2, respectively. Then any key press by the user will move the FSM back to the ckkey1 state.



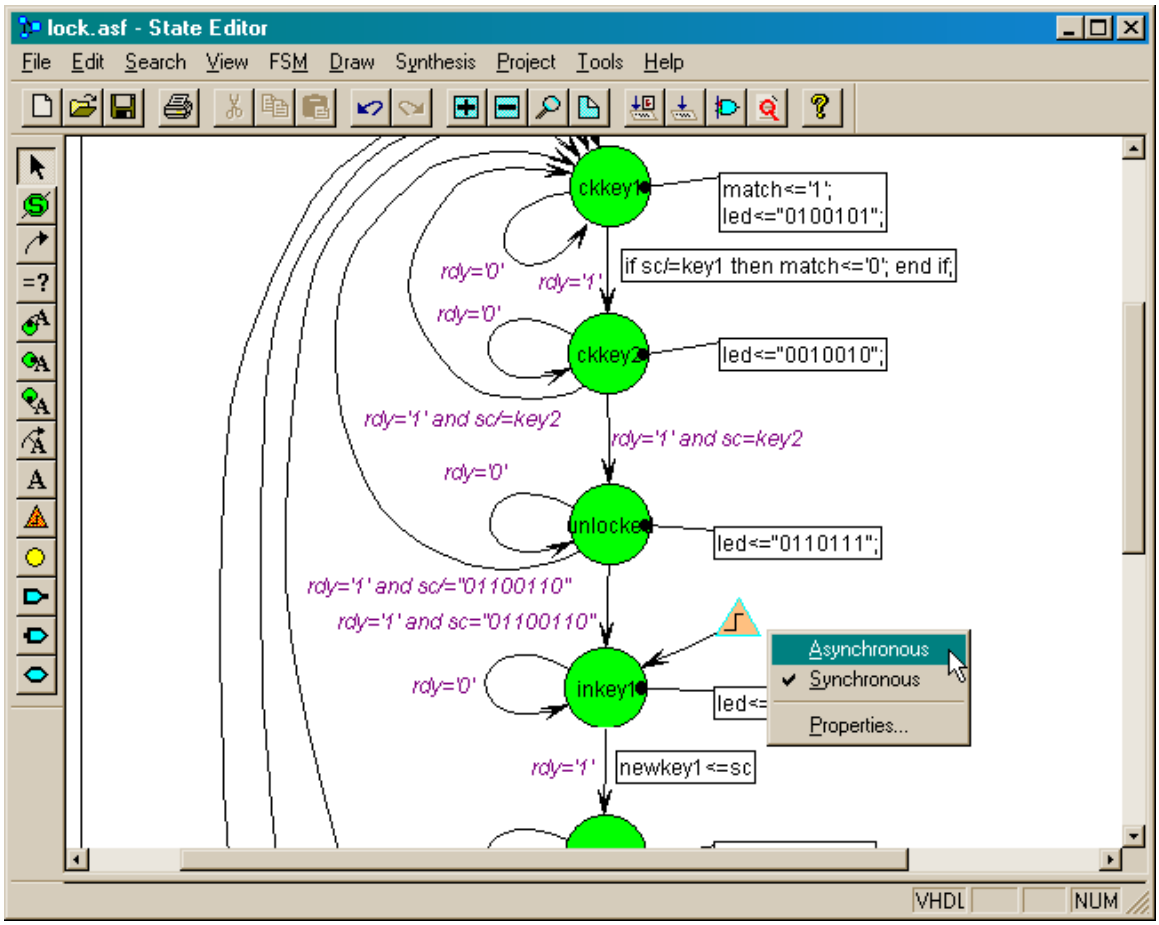
In addition to the normal operations of the FSM, we have to initialize its behavior upon start-up. The Reset button is used to specify the initial state of the FSM and the conditions upon which it is entered.



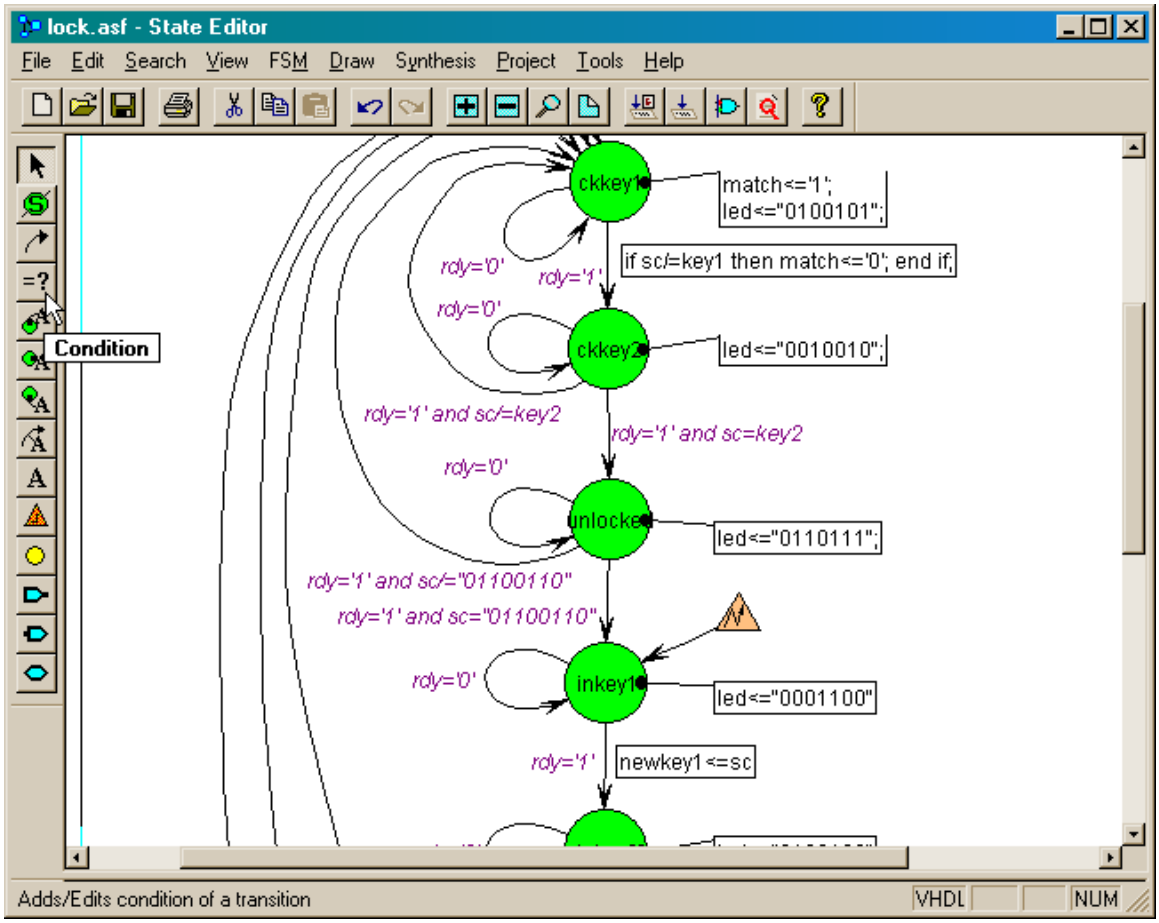
Drag the triangular reset icon into the editing area and then left-click to drop it. At this point a line will connect the cursor to the reset icon. Move the cursor over the inkey1 state circle and click again. This denotes that the FSM will move into the inkey1 state whenever a reset condition occurs. This makes sense because a reset should be a very infrequent event and it should allow the user to gain control of the lock by entering a new combination that overrides the old one.



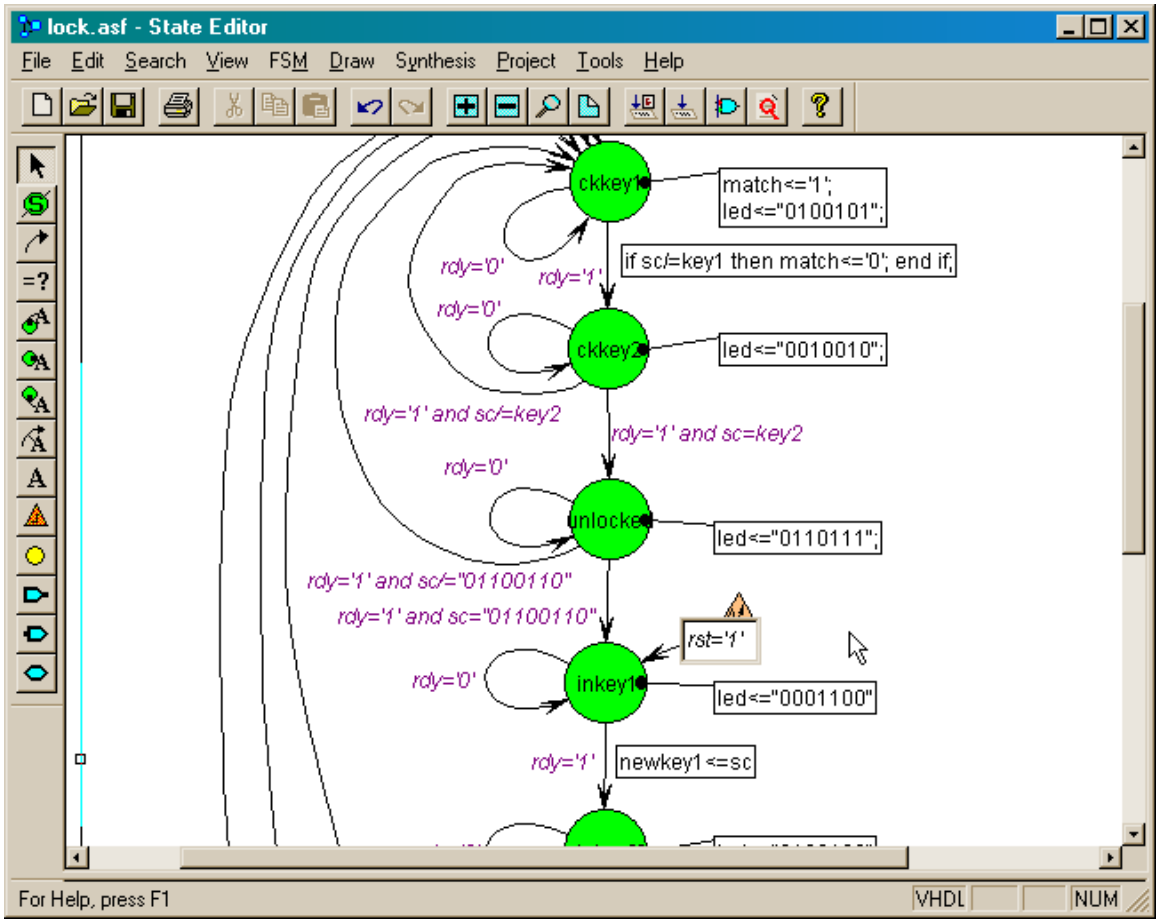
The reset action can occur on a clock edge (synchronous) or whenever the reset condition is satisfied (asynchronous). Right-click on the reset icon and select either Asynchronous or Synchronous from the pop-up menu. I have picked Asynchronous in this example, but either one will work.



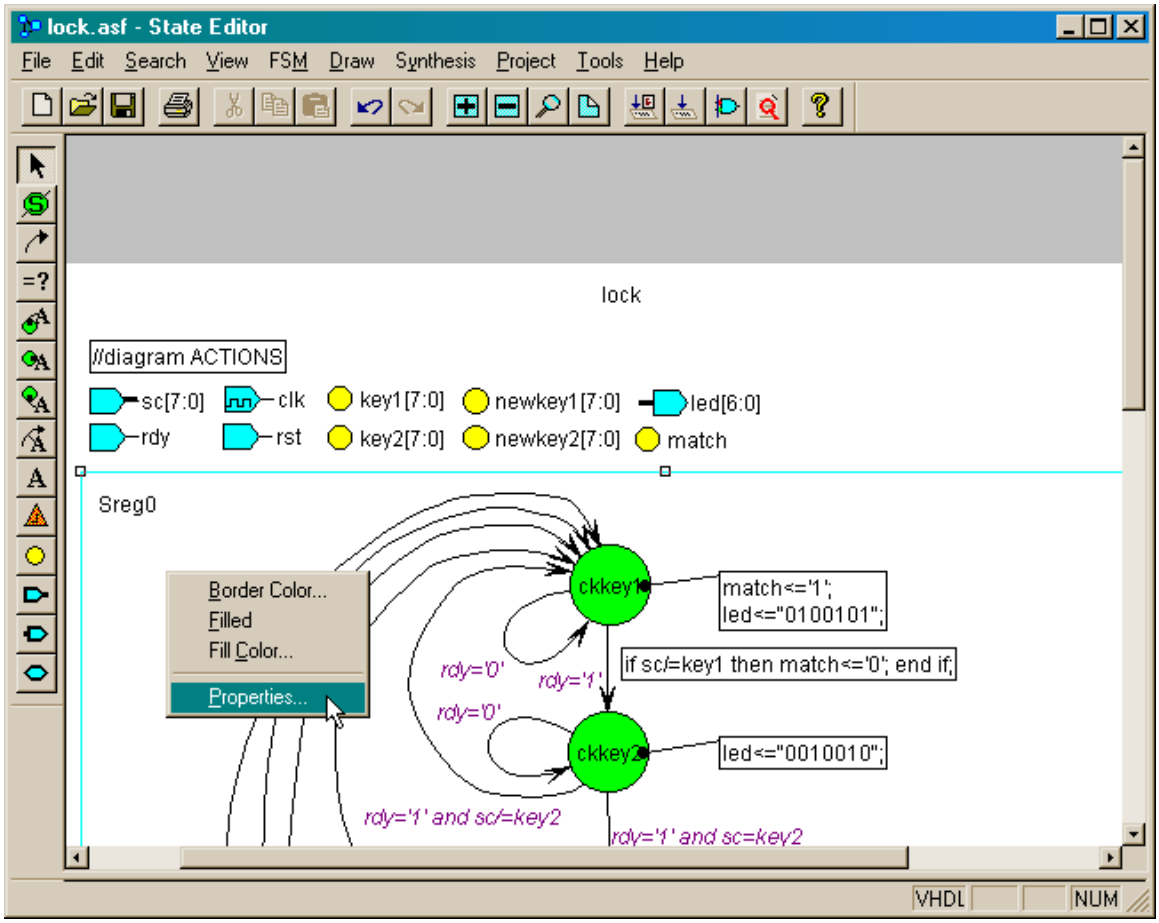
To specify the reset condition, just click on the Condition button and then click on the edge connecting the reset icon to the inkey1 state.



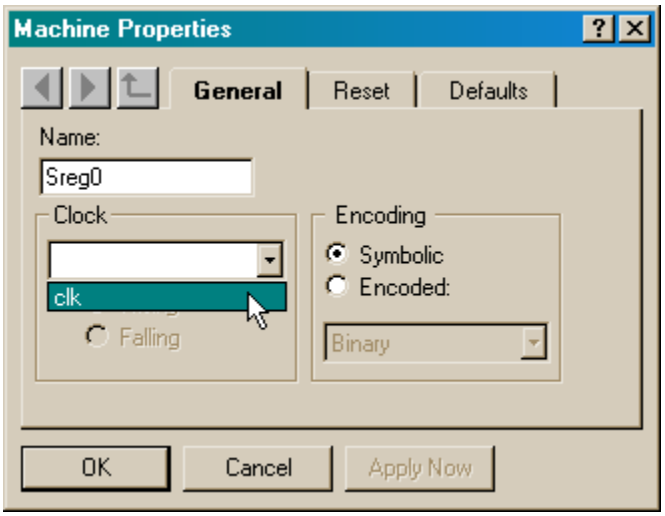
Then type the VHDL code into the editing box that directs the FSM into the inkey1 state whenever the reset input is at a logic 1 level.



Now we can set some global options that affect the entire FSM. Right-click in an empty section of the editing area and select Properties... from the pop-up menu.

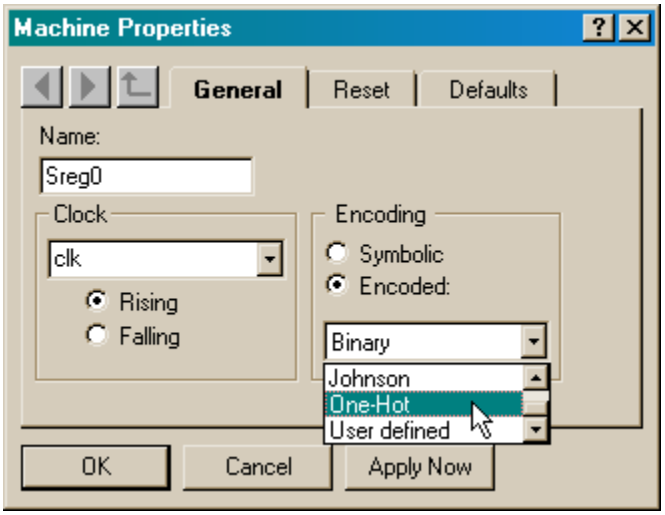


The **Machine Properties** window for the FSM will appear. In the General tab, select the clk signal from the drop-down list attached to the Clock field. This directs the FSM to change states on the rising edge of the clk input.



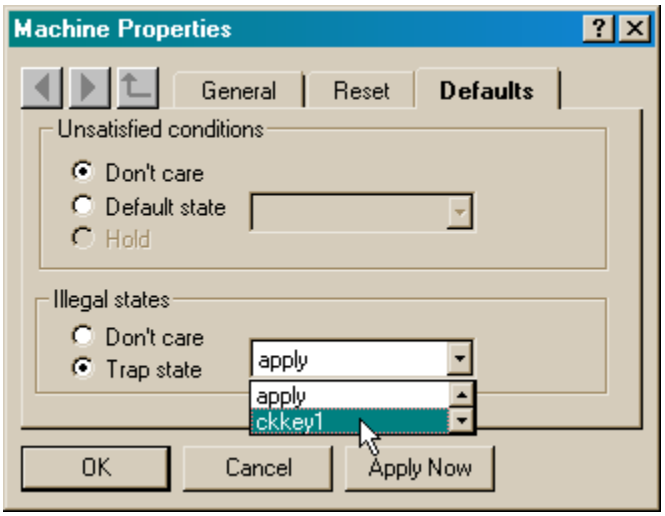
Next, click on the Encoded radio button so we can specify how the states are stored in the circuitry of the FPGA. Select One-Hot from the drop-down list. One-hot encoding uses a

flip-flop for each state with the flip-flop for the active state being set while all the others are cleared. Eight flip-flops are needed by this FSM which is no problem since the XC4005XL FPGA has 384 of them in the CLB array. For a CPLD, which typically has fewer flip-flops, we might select binary encoding which uses three flip-flops to store the binary code of the active state among the eight total states.

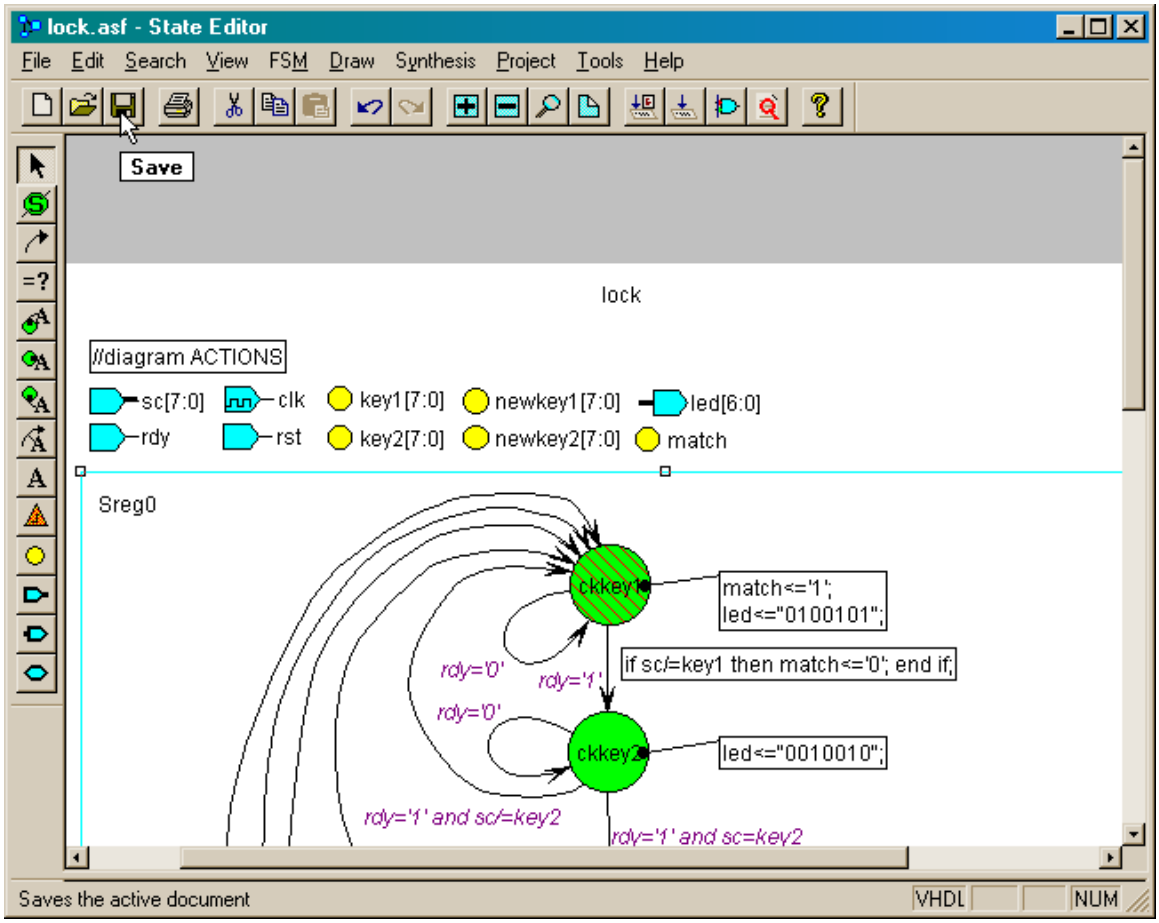


Next, click on the Defaults tab so we can stipulate the actions of the FSM when illegal states occur. If an illegal state occurs we would like the lock to stay closed and not spring open, so it should transition into the ckkey1 state. Click on the Trap state radio button in the Illegal states section and then select the ckkey1 state from the drop-down list.

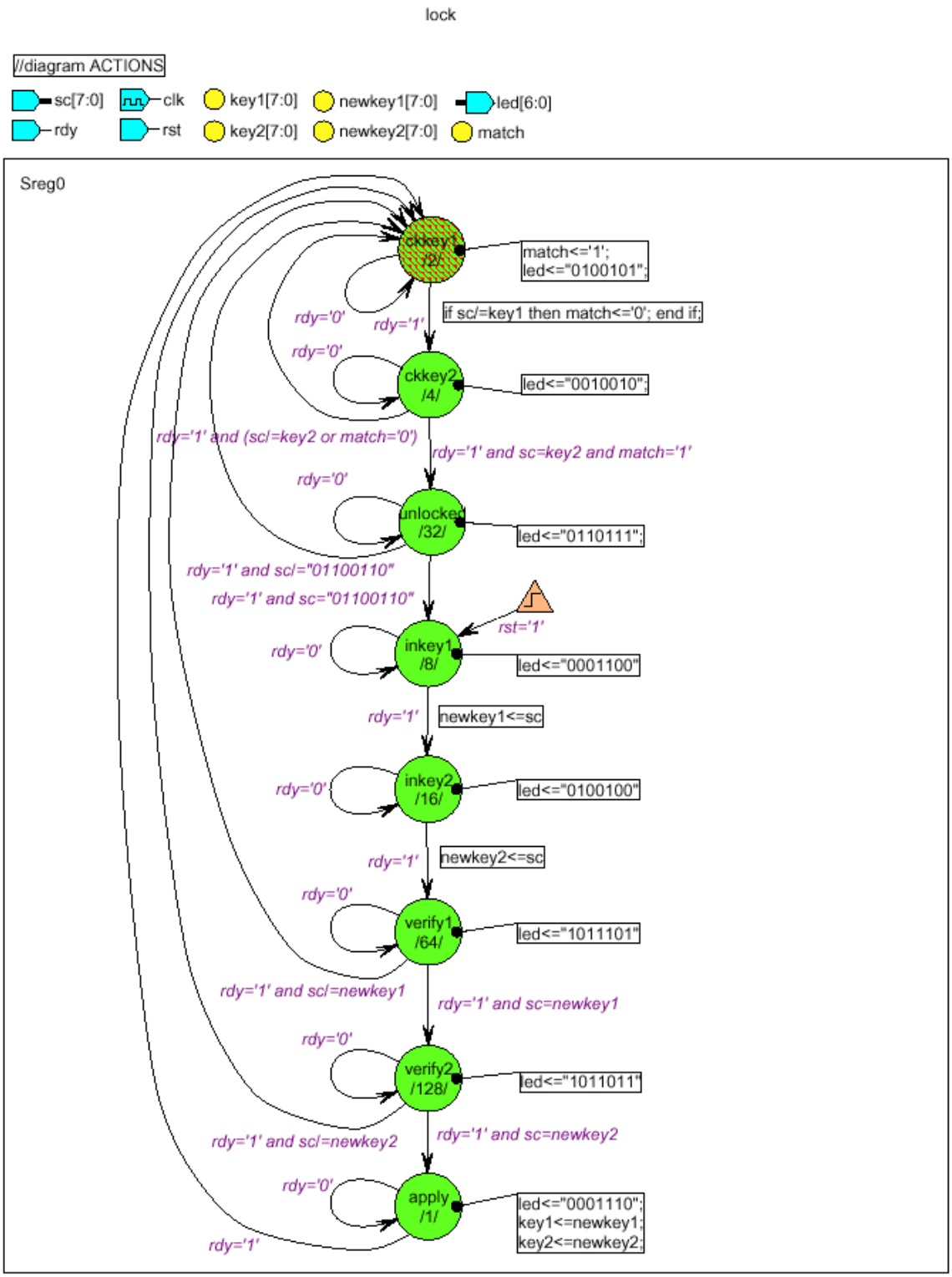
We have specified all the possible conditions that control transitions between states so we can keep the Don't care option in the Unsatisfied conditions section. Then click on OK to close the **Machine Properties** window.



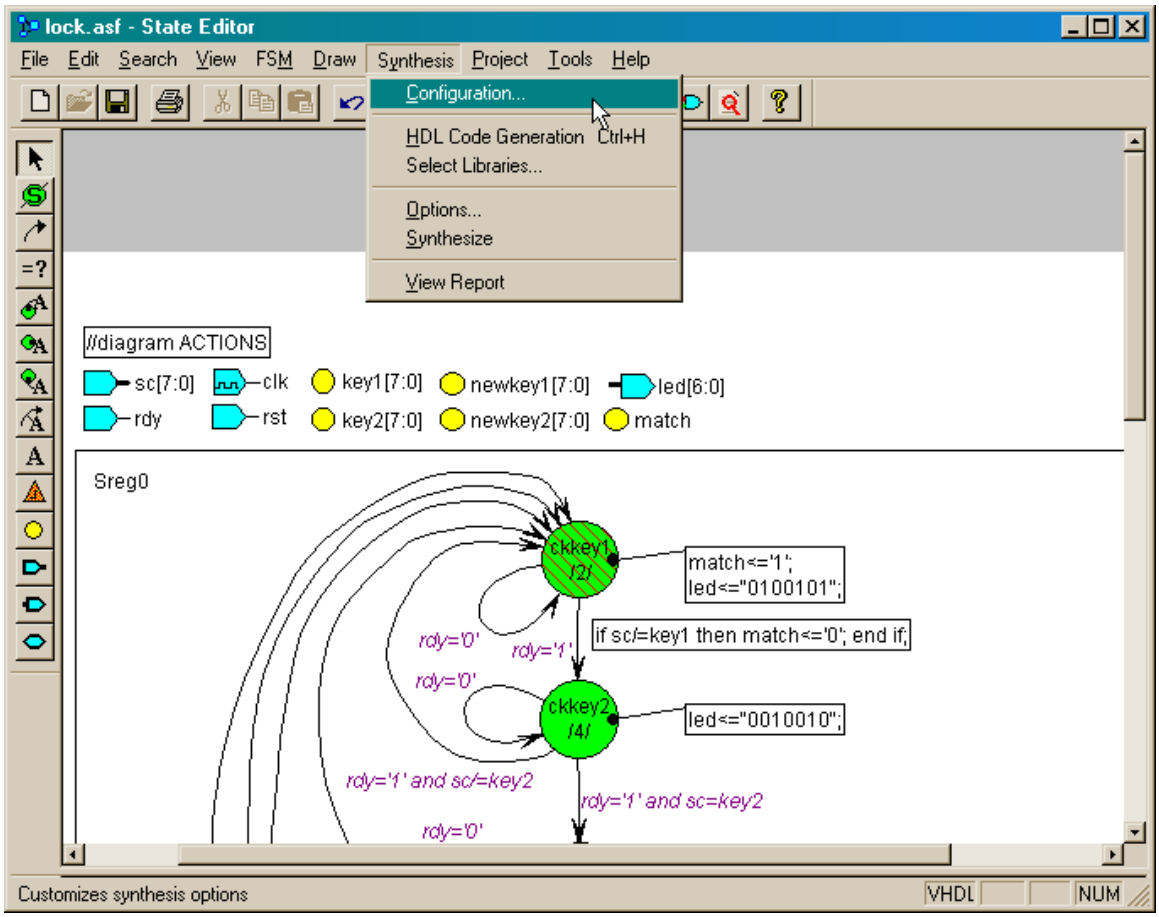
Upon returning to the **State Editor** window, note that the ckkey1 state is now drawn with a cross-hatched pattern to indicate it is the designated trap state. Click on the Save button to store the FSM description.



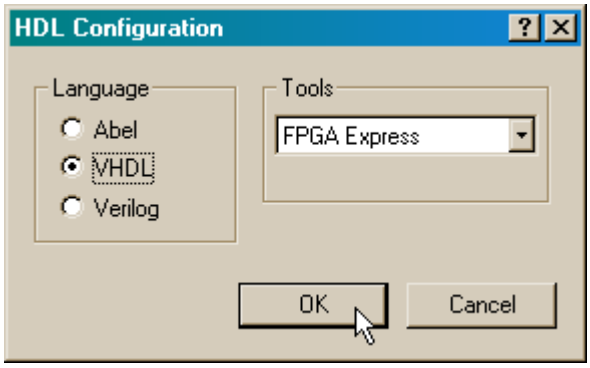
The complete FSM for the lock&key module is shown below.



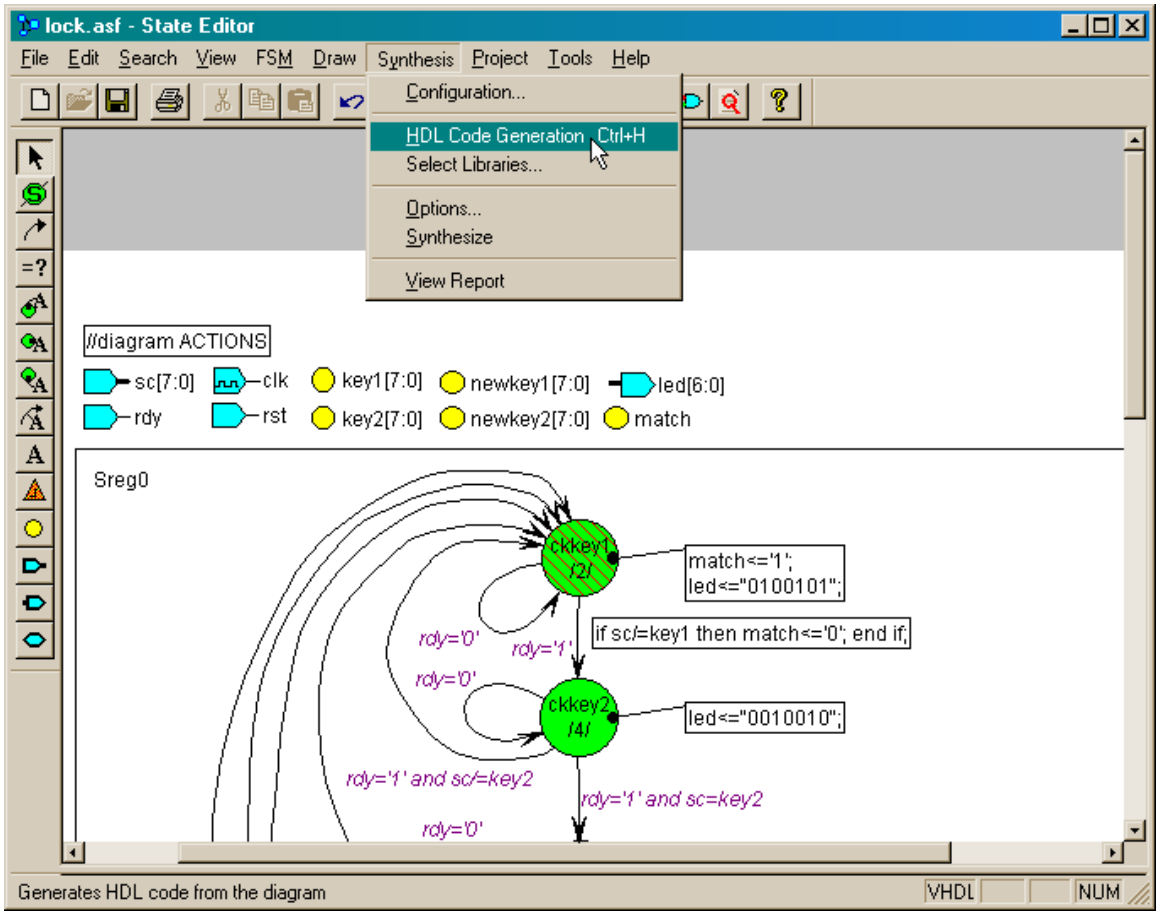
Now that the FSM description is complete, we can generate a VHDL description of it. (This is not necessary in order to use the FSM in our project, but is done for illustrative purposes.) Select the Synthesis➔Configuration... menu item.



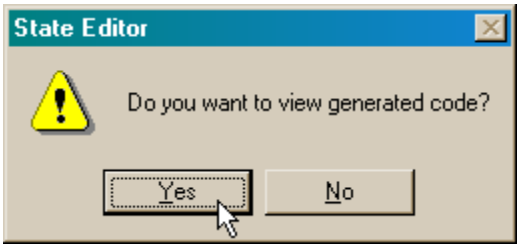
In the **HDL Configuration** window that appears, click on the VHDL radio-button to select it as the language used for an HDL description of the FSM. Then click on the OK button.



Next, activate the Synthesis➔HDL Code Generation command.



Indicate that you wish to see the VHDL code that is generated for the state machine we built.



Within a few seconds, the code in Listing 3 will appear in an **HDL Editor** window. We can correlate pieces of the VHDL code with objects we have placed in the editing area of the **State Editor** window:

Lines 16–20: The input and output ports placed at the top of the FSM editing area are declared in the entity section.

Lines 26–30: The signals placed at the top of the FSM editing area are declared in the architecture section.

Lines 33–45: Here is the state encoding and the signal, Sreg0, that holds the state.

Line 55: The FSM changes states on the rising edge of the clk signal as we specified in the **Machine Properties** window.

Lines 56–57: Upon a reset, the FSM moves into the inkey1 state.

Lines 60–122: The transitions from the current state to the next state and any actions associated with these transitions are described here.

Lines 123–124: Here is the specification of ckkey1 as the trap state that is entered if the FSM ever gets into an illegal state.

Lines 132–139: The seven-segment LED activation pattern associated with each state is listed here.

The VHDL description of the FSM can be useful for two reasons:

1. The editing area of the State Editor window gets very cluttered for complicated FSMs. You can use the State Editor to draw an initial, simplified version of your FSM and then add the rest of your description directly to the VHDL file. You cannot automatically back-annotate the additions to the VHDL file back into the State Editor, so the VHDL file must be used as the master design file for the FSM after you do this.

2. If you are unsure how to write FSM descriptions using VHDL, you can create simple FSMs in the State Editor and export them as VHDL to view the basic language constructs that are used.

Listing 3: Generated VHDL code for the lock & key module.

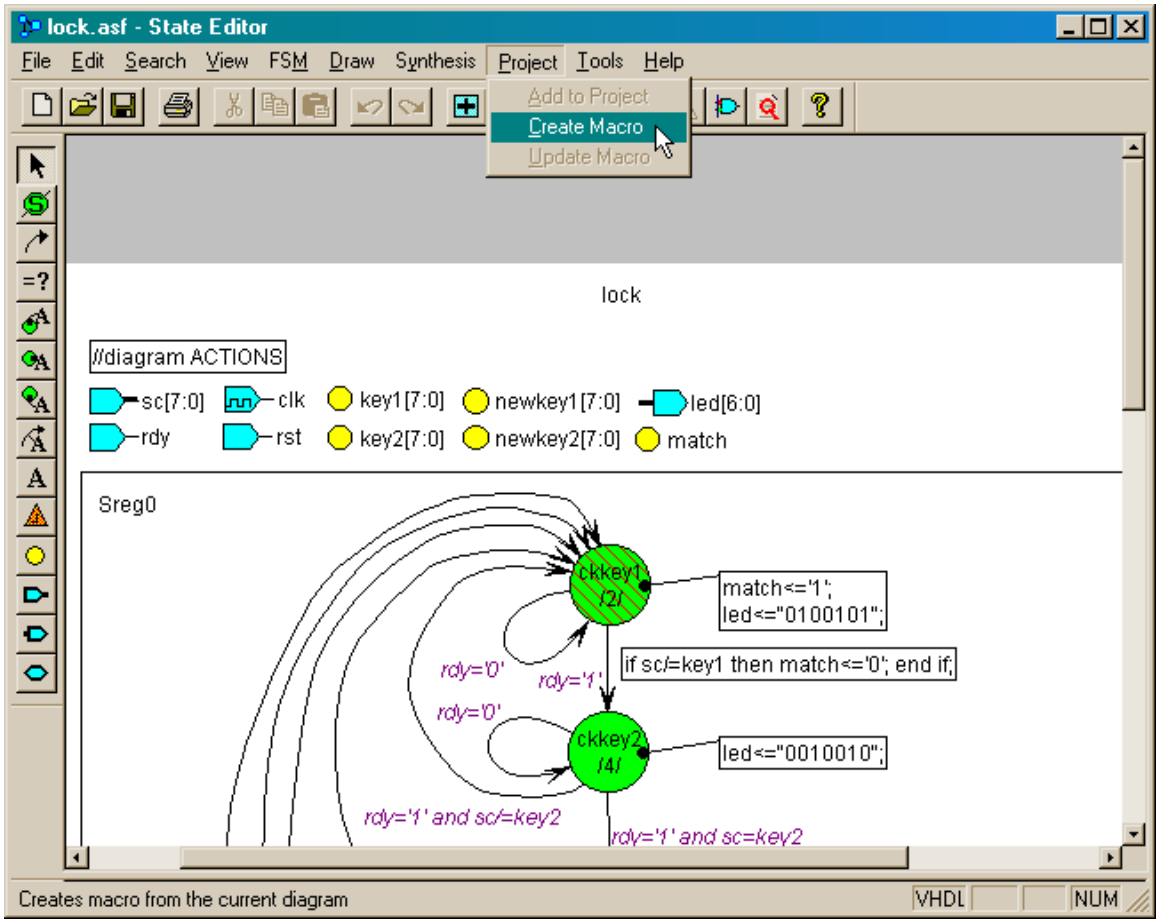
```
1   --  File: C:\PRAG21I\DSGN4_1\lock.vhd
2   -- created: 04/13/01 12:32:29
3   --  from: 'C:\PRAG21I\DSGN4_1\lock.asf'
4   --  by fsm2hdl - version: 2.0.1.53
5   --
6   library IEEE;
7   use IEEE.std_logic_1164.all;
8
9   use IEEE.std_logic_arith.all;
10  use IEEE.std_logic_unsigned.all;
11
12  library SYNOPSYS;
13  use SYNOPSYS.attributes.all;
14
15  entity lock is
16    port (clk: in STD_LOGIC;
17          rdy: in STD_LOGIC;
18          rst: in STD_LOGIC;
19          sc: in STD_LOGIC_VECTOR (7 downto 0);
20          led: out STD_LOGIC_VECTOR (6 downto 0));
21  end;
22
23  architecture lock_arch of lock is
24
25  --diagram signal declarations
```

```
26  signal key1: STD_LOGIC_VECTOR (7 downto 0);
27  signal key2: STD_LOGIC_VECTOR (7 downto 0);
28  signal match: STD_LOGIC;
29  signal newkey1: STD_LOGIC_VECTOR (7 downto 0);
30  signal newkey2: STD_LOGIC_VECTOR (7 downto 0);
31
32  -- ONE HOT ENCODED state machine: Sreg0
33  type Sreg0_type is (apply, ckkey1, ckkey2, inkey1, inkey2, unlocked, ve
34  attribute enum_encoding of Sreg0_type: type is
35    "00000001 " &    -- apply
36    "00000010 " &    -- ckkey1
37    "00000100 " &    -- ckkey2
38    "00001000 " &    -- inkey1
39    "00010000 " &    -- inkey2
40    "00100000 " &    -- unlocked
41    "01000000 " &    -- verify1
42    "10000000";    -- verify2
43
44  signal Sreg0: Sreg0_type;
45
46  begin
47  --concurrent signal assignments
48
49
50  Sreg0_machine: process (clk)
51
52  begin
53
54  if clk'event and clk = '1' then
55    if rst='1' then
56      Sreg0 <= inkey1;
57    else
58    case Sreg0 is
59      when apply =>
60        key1<=newkey1;
61        key2<=newkey2;
62        if rdy='1' then
63          Sreg0 <= ckkey1;
64        elsif rdy='0' then
65          Sreg0 <= apply;
66        end if;
67      when ckkey1 =>
68        match<='1';
69        if rdy='1' then
70          Sreg0 <= ckkey2;
71          if sc/=key1 then match<='0';
72          end if;
73        elsif rdy='0' then
74          Sreg0 <= ckkey1;
75        end if;
76      when ckkey2 =>
77        if rdy='1' and sc=key2 then
78          Sreg0 <= unlocked;
```

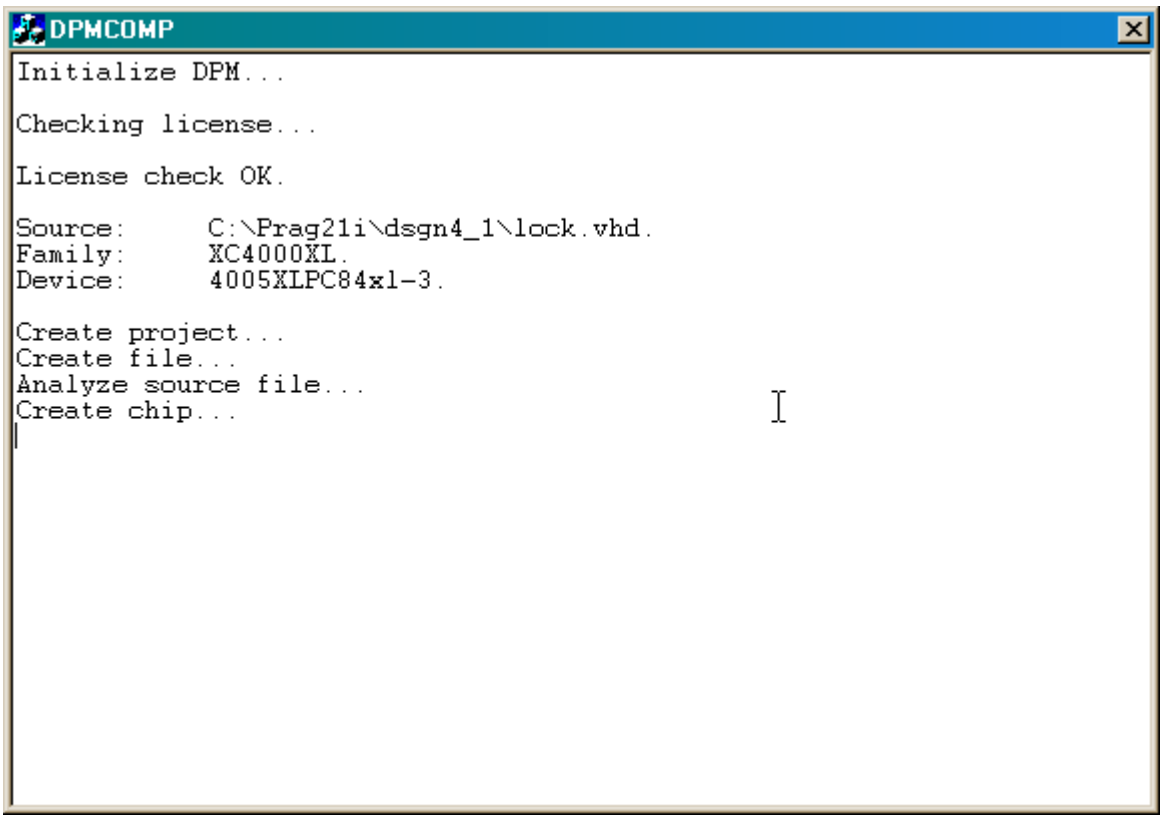
```
79          elsif rdy='1' and sc/=key2 then
80            Sreg0 <= ckkey1;
81          elsif rdy='0' then
82            Sreg0 <= ckkey2;
83          end if;
84        when inkey1 =>
85          if rdy='1' then
86            Sreg0 <= inkey2;
87            newkey1<=sc;
88          elsif rdy='0' then
89            Sreg0 <= inkey1;
90          end if;
91        when inkey2 =>
92          if rdy='1' then
93            Sreg0 <= verify1;
94            newkey2<=sc;
95          elsif rdy='0' then
96            Sreg0 <= inkey2;
97          end if;
98        when unlocked =>
99          if rdy='1' and sc/="01100110" then
100           Sreg0 <= ckkey1;
101         elsif rdy='1' and sc="01100110" then
102           Sreg0 <= inkey1;
103         elsif rdy='0' then
104           Sreg0 <= unlocked;
105         end if;
106       when verify1 =>
107         if rdy='1' and sc=newkey1 then
108           Sreg0 <= verify2;
109         elsif rdy='1' and sc/=newkey1 then
110           Sreg0 <= ckkey1;
111         elsif rdy='0' then
112           Sreg0 <= verify1;
113         end if;
114       when verify2 =>
115         if rdy='0' then
116           Sreg0 <= verify2;
117         elsif rdy='1' and sc/=newkey2 then
118           Sreg0 <= ckkey1;
119         elsif rdy='1' and sc=newkey2 then
120           Sreg0 <= apply;
121         end if;
122       when others =>   -- trap state
123         Sreg0 <= ckkey1;
124     end case;
125     end if;
126   end if;
127   end process;
128
129   -- signal assignment statements for combinatorial outputs
130   led_assignment:
131   led <= "0001110" when (Sreg0 = apply) else
```

```
132         "0100101" when (Sreg0 = ckkey1) else
133         "0010010" when (Sreg0 = ckkey2) else
134         "0100100" when (Sreg0 = inkey2) else
135         "0110111" when (Sreg0 = unlocked) else
136         "1011101" when (Sreg0 = verify1) else
137         "1011011" when (Sreg0 = verify2) else
138         "0001100";
139
140   end lock_arch;
```

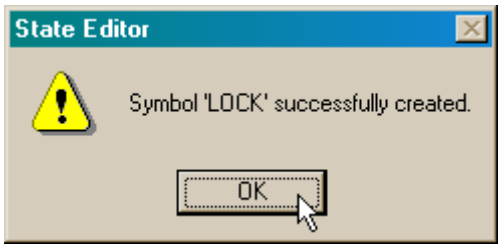
Now it is time to make the FSM available for use as a building block of the combination lock. Select the Project➔Create Macro... command to initiate this process.



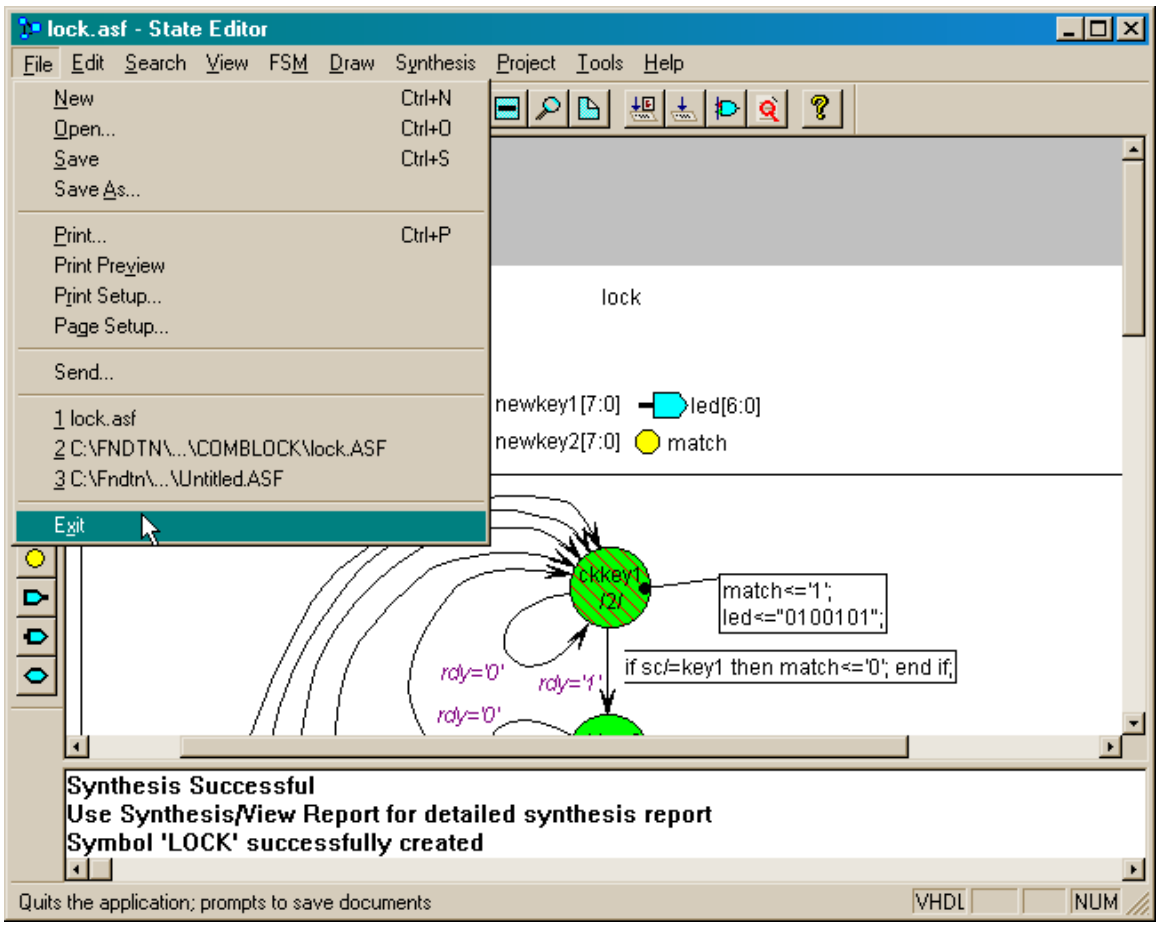
The progress of the macro creation is displayed in the **DPMCOMP** window.



Upon successful completion of the macro generation process, click OK in the confirmation window that appears.

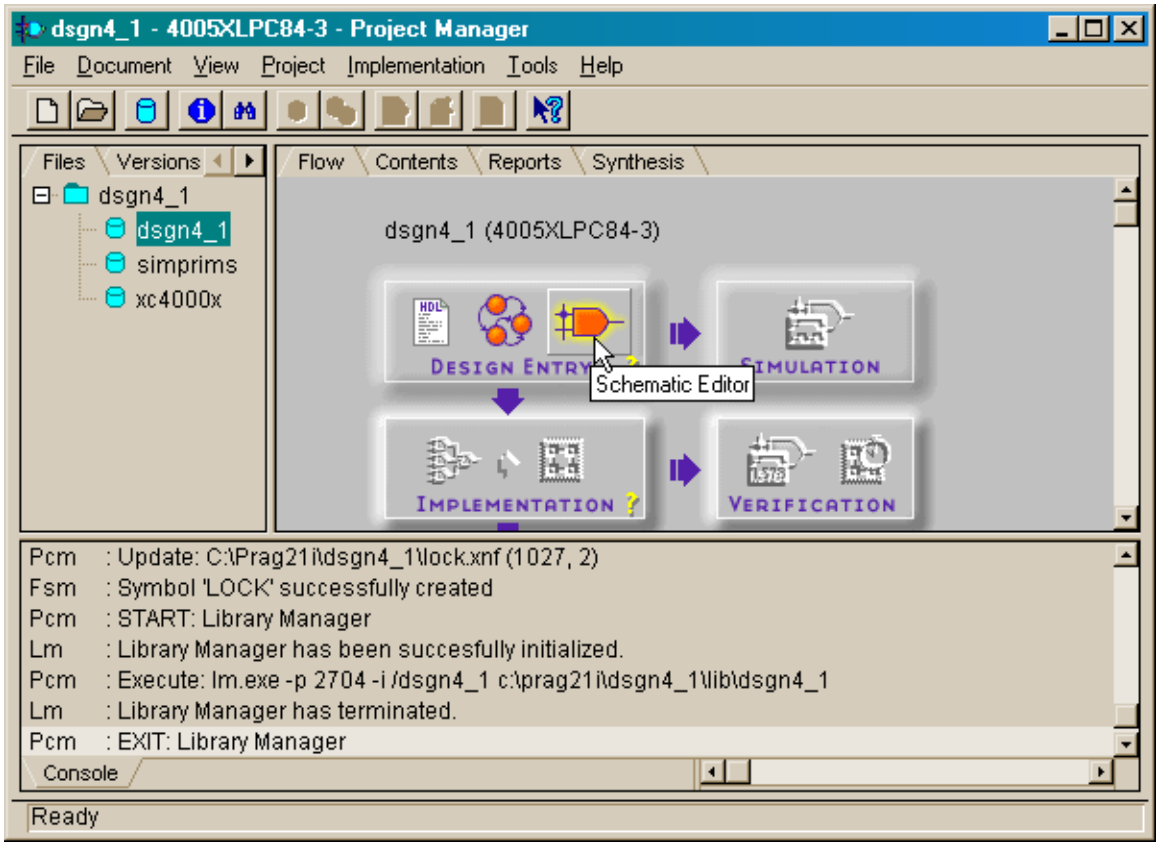


Finally, close the **State Editor** window.

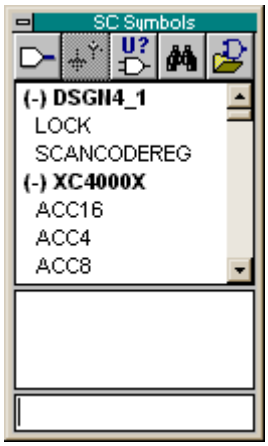


Creating the Top-Level Module

The top-level of the combination lock will be built by connecting the keyboard interface and the lock&key modules together in a schematic. Click on the Schematic Editor button to begin this phase.



Within the **Schematic Editor** window, bring up the list of library symbols and you will see the keyboard interface macro (SCANCODEREG) and the lock& key macro (LOCK) at the top of the list. Select each macro and drop it into the drawing area of the **Schematic Editor** window.



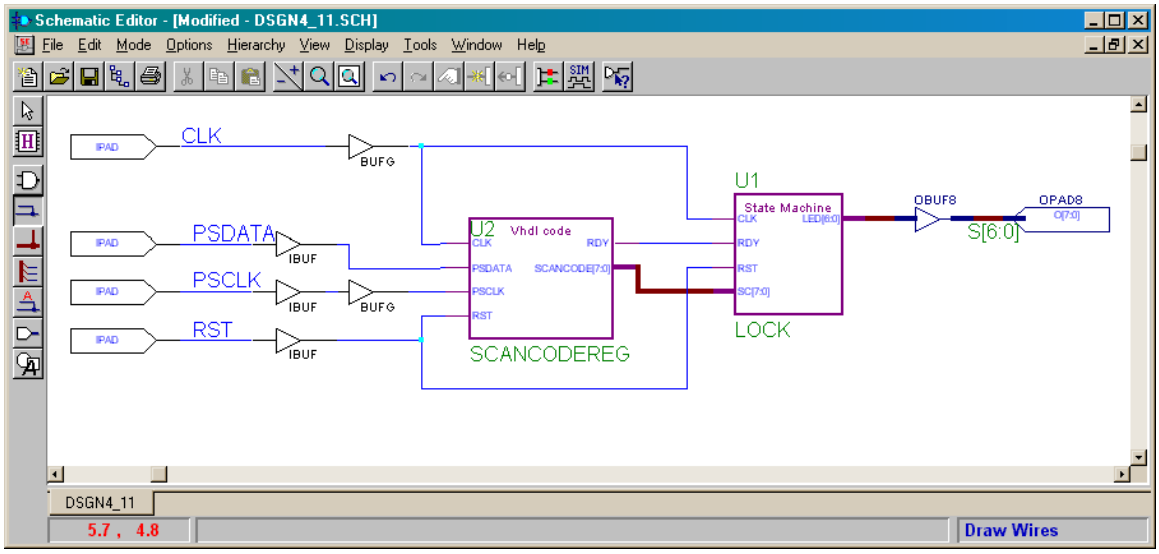
Connect the macros to I/O buffers and pads as shown below. Note the following:

1. The main clock (CLK) will enter the XC4000 FPGA on a dedicated clock pin (because that is the way it is connected on the XS40 Board) so the input pad (IPAD) can connect directly to a general clock buffer (BUFG). Using the BUFG ensures that the clock signal reaches all the flip-flops in the design with minimal skew so they all change state at the same time.

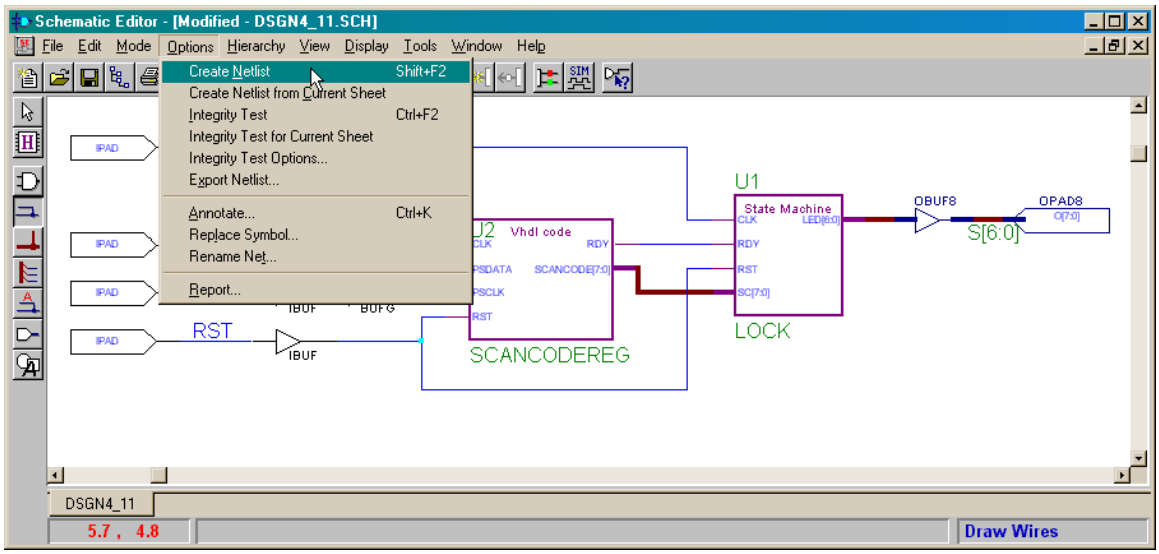
2. The clock from the PS/2 keyboard (PSCLK) enters on a generic I/O pin so it must go through an input buffer (IBUF) before going through a BUFG.

3. The keyboard serial data signal (PSDATA) and the reset signal (RST) are standard, non-clock inputs so they just connect to IBUFs.

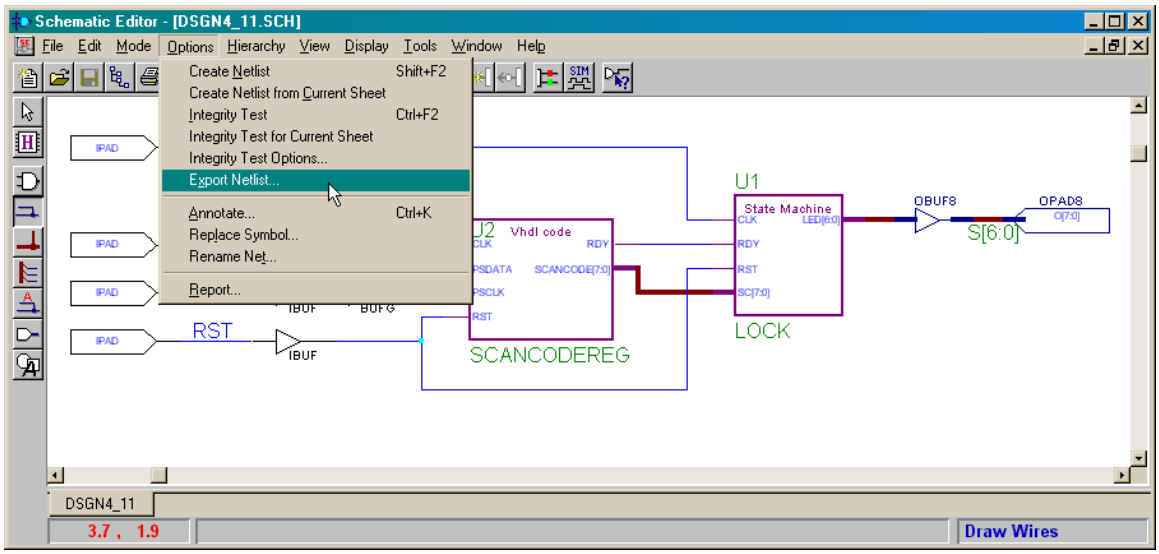
4. The seven LED outputs of the LOCK macro connect to a set of eight output buffers (OBUF8). The eight buffers connect to a set of eight output pads (OPAD8). The bus connecting the OBUF8 to the OPAD8 is named S[6:0] so it is only has a width of seven. This disconnects the eighth buffer and output pad so only the lower seven buffers and pads are used as actual outputs.



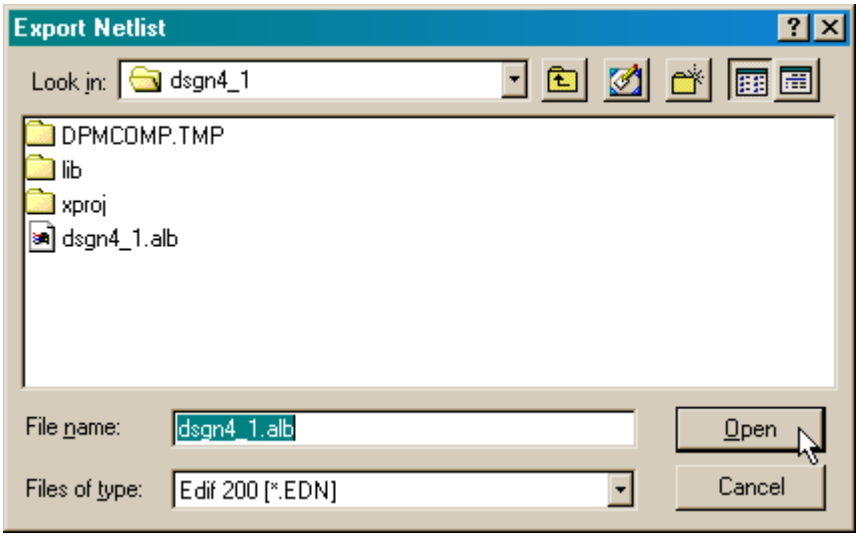
Once the macros are connected to each other and the I/O, select the Options➔Create Netlist menu item.



After the netlist is created, export the netlist to the other Foundation tools using the Options➔Export Netlist... command.



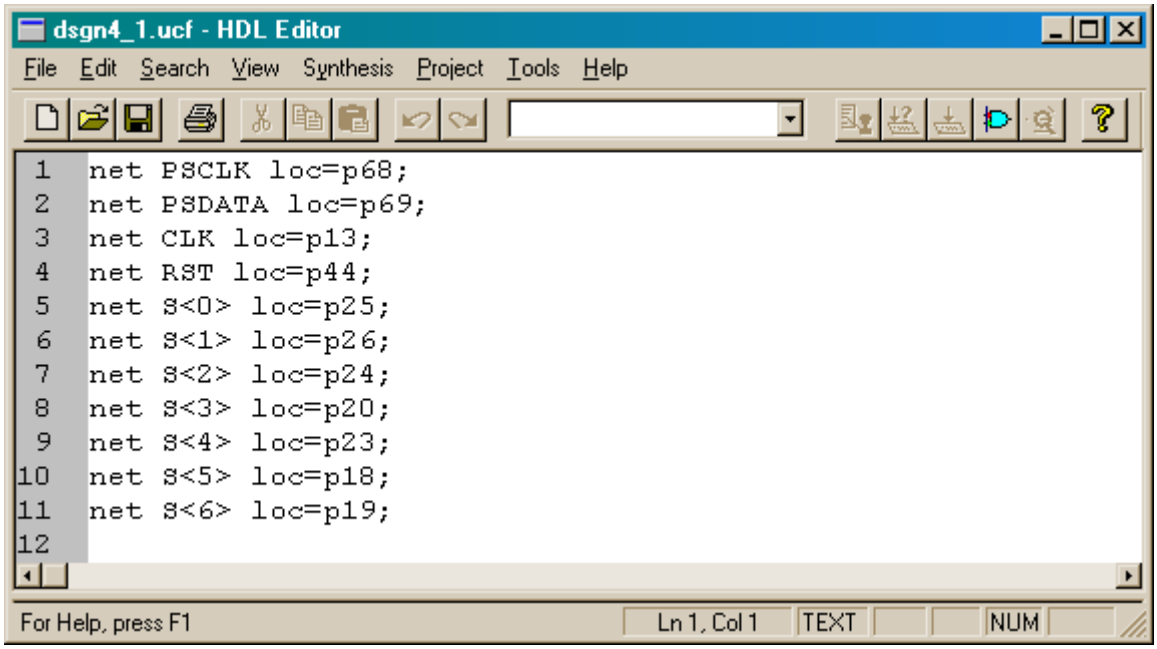
Accept the default name shown for the file in the **Export Netlist** window and click on the Open button.



Now that the top-level netlist has been generated and exported, close the **Schematic Editor** window.

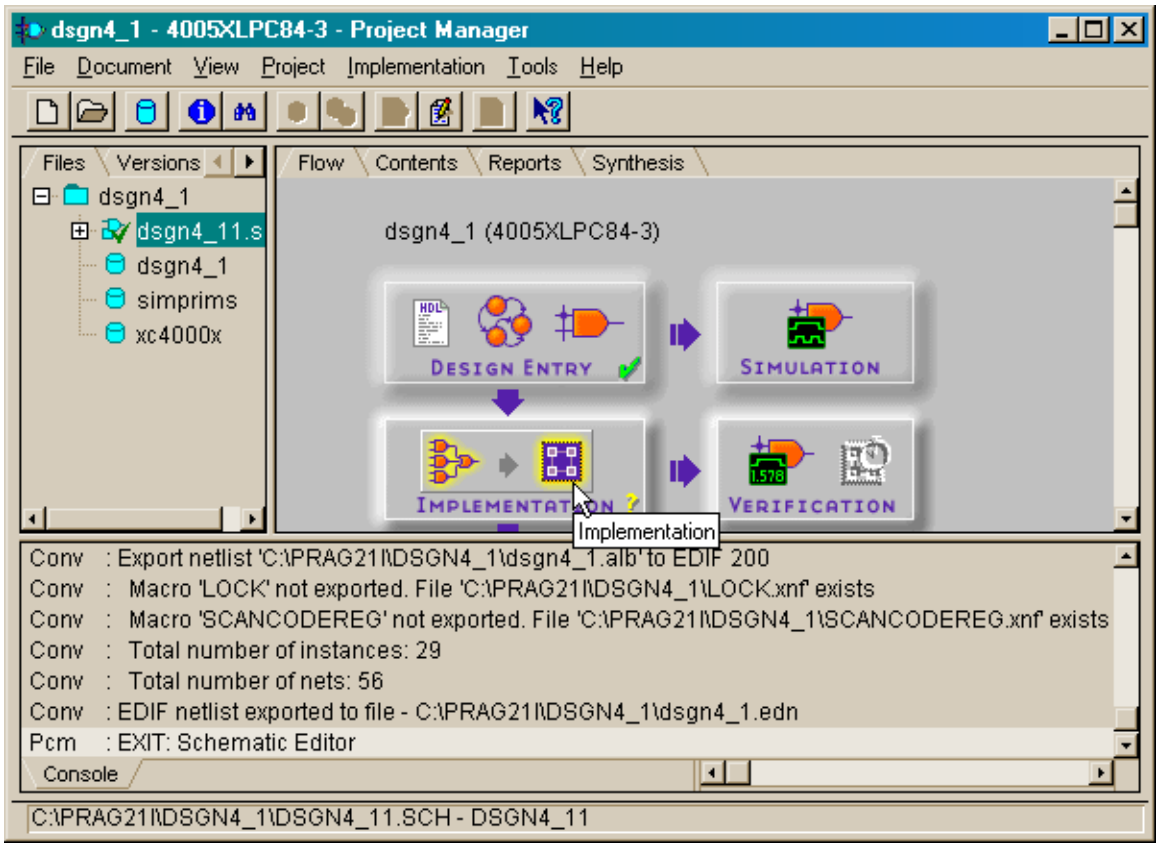
Entering the Pin Assignments for the XS40 Board

Open the dsgn4_1.ucf constraints file and enter the following pin assignments that map the I/O signals of the combination lock to the appropriate pins of the XS40 Board.

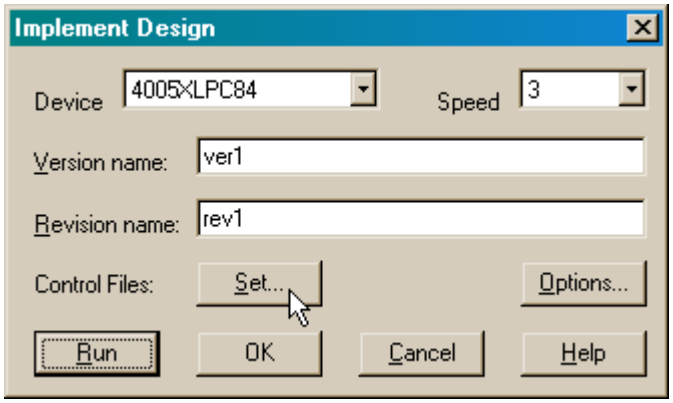


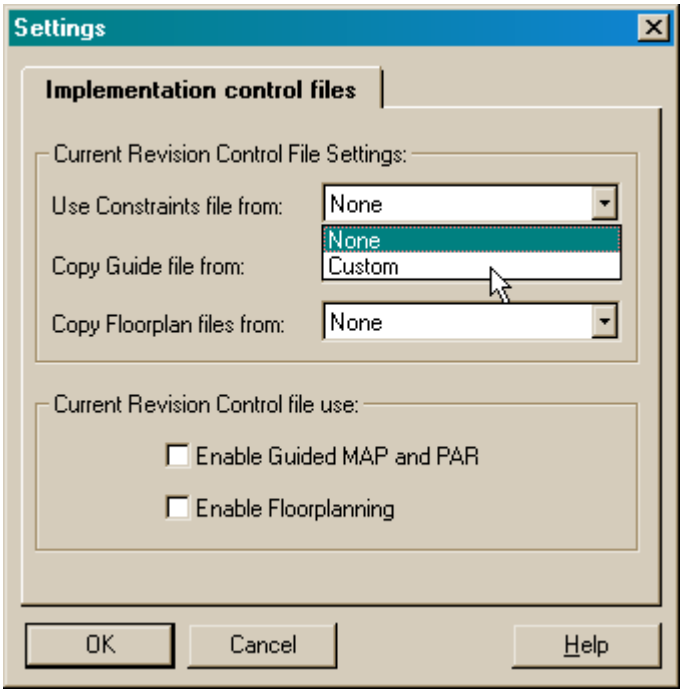
Implementing the Design for the XC4005XL FPGA

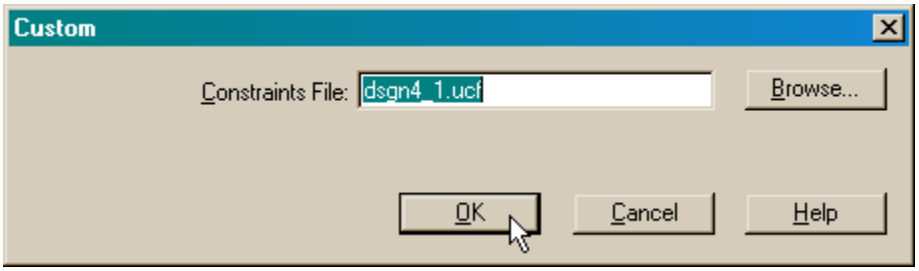
Now run the implementation tools.


dsgn4_1 - 4005XLPC84-3 - Project Manager
File Document View Project Implementation Tools Help
Files Versions
dsgn4_1
dsgn4_11.s
dsgn4_1
simprims
xc4000x
Flow Contents Reports Synthesis
dsgn4_1 (4005XLPC84-3)
DESIGN ENTRY
SIMULATION
IMPLEMENTATION
VERIFICATION
Implementation
Conv : Export netlist 'C:\PRAG21I\DSGN4_1\dsgn4_1.alb' to EDIF 200
Conv : Macro 'LOCK' not exported. File 'C:\PRAG21I\DSGN4_1\LOCK.xnf' exists
Conv : Macro 'SCANCODEREG' not exported. File 'C:\PRAG21I\DSGN4_1\SCANCODEREG.xnf' exists
Conv : Total number of instances: 29
Conv : Total number of nets: 56
Conv : EDIF netlist exported to file - C:\PRAG21I\DSGN4_1\dsgn4_1.edn
Pcm : EXIT: Schematic Editor
Console
C:\PRAG21I\DSGN4_1\DSGN4_11.SCH - DSGN4_11

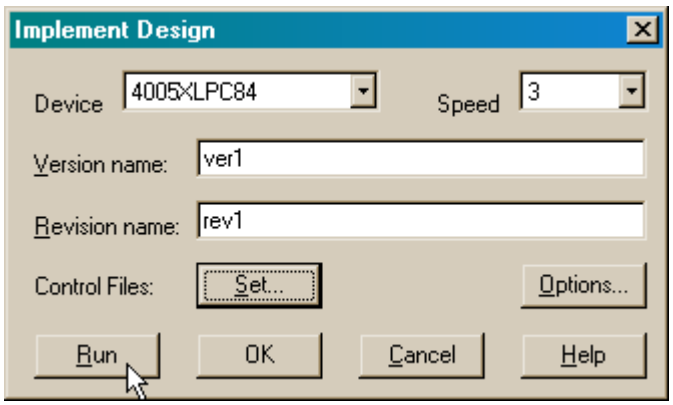
Go through the following sequence of windows to specify the dsgn4_1.ucf file as the constraints file for this design.



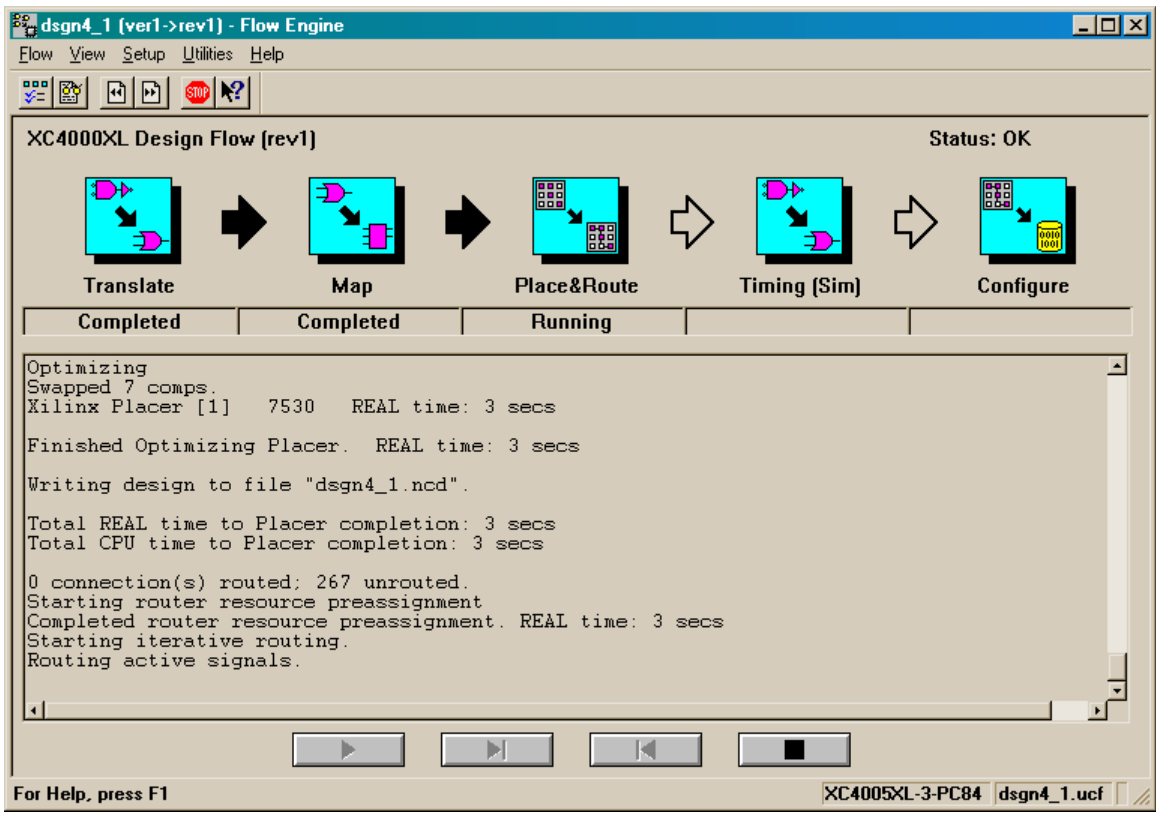




Then click on the Run button to implement the netlist in the XC4005XL FPGA.

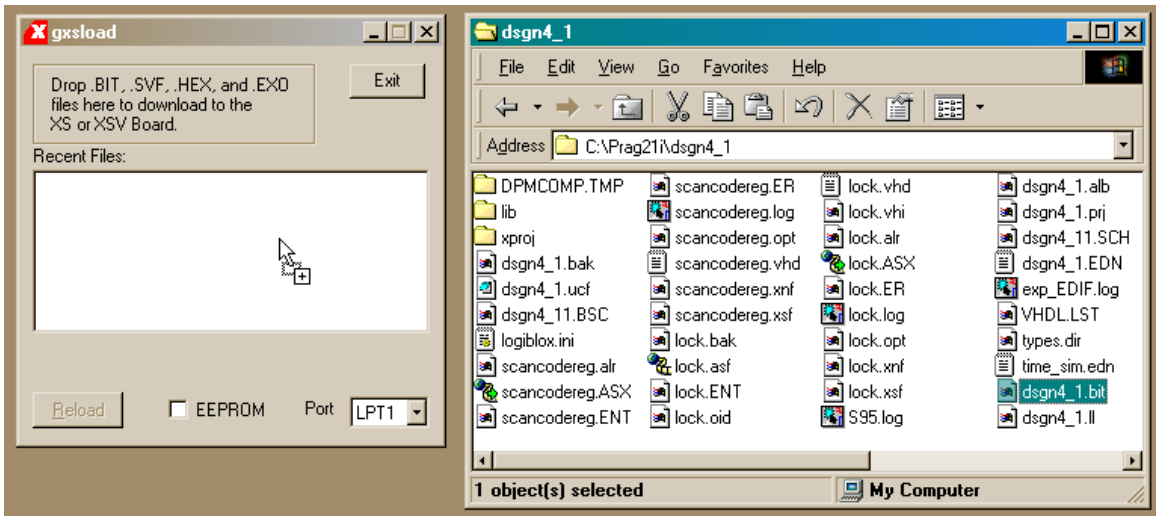


The implementation tools should run through all five phases without any problems.



Downloading the Bitstream to the XS40 Board

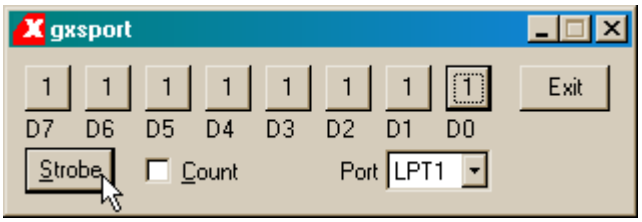
After the implementation tools finish, drag-and-drop the dsgn4_1.bit file into the **GXSLOAD** window to download the bitstream into the XS40-005XL Board.



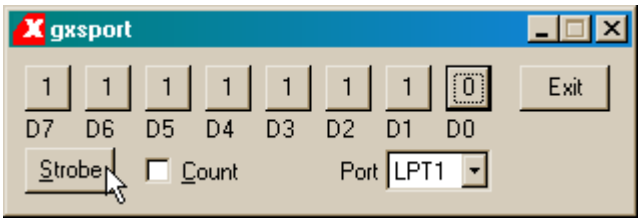
Testing the Combination Lock

After downloading the bitstream to the XS40 Board, attach a PS/2 keyboard to the six-pin mini-DIN socket at the bottom of the board.

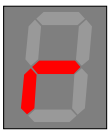
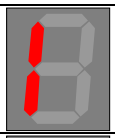
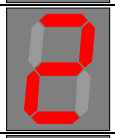
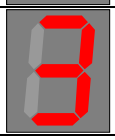
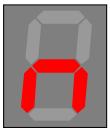
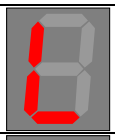
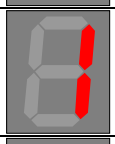
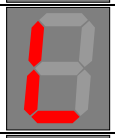
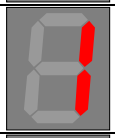
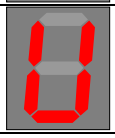
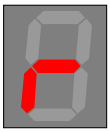
If the LED on the XS40 Board does not display a lower-case R upon startup, then you may have to manually reset the combination lock. The reset input for the lock (pin 44 on the FPGA) is connected to data bit D0 of the parallel port. Start the GXSPORT utility and apply a logic 1 to the reset input.



Then apply a logic 0 to release the reset.



Now the combination lock should be ready to respond to key presses. A sequence of key presses and the results are shown below:

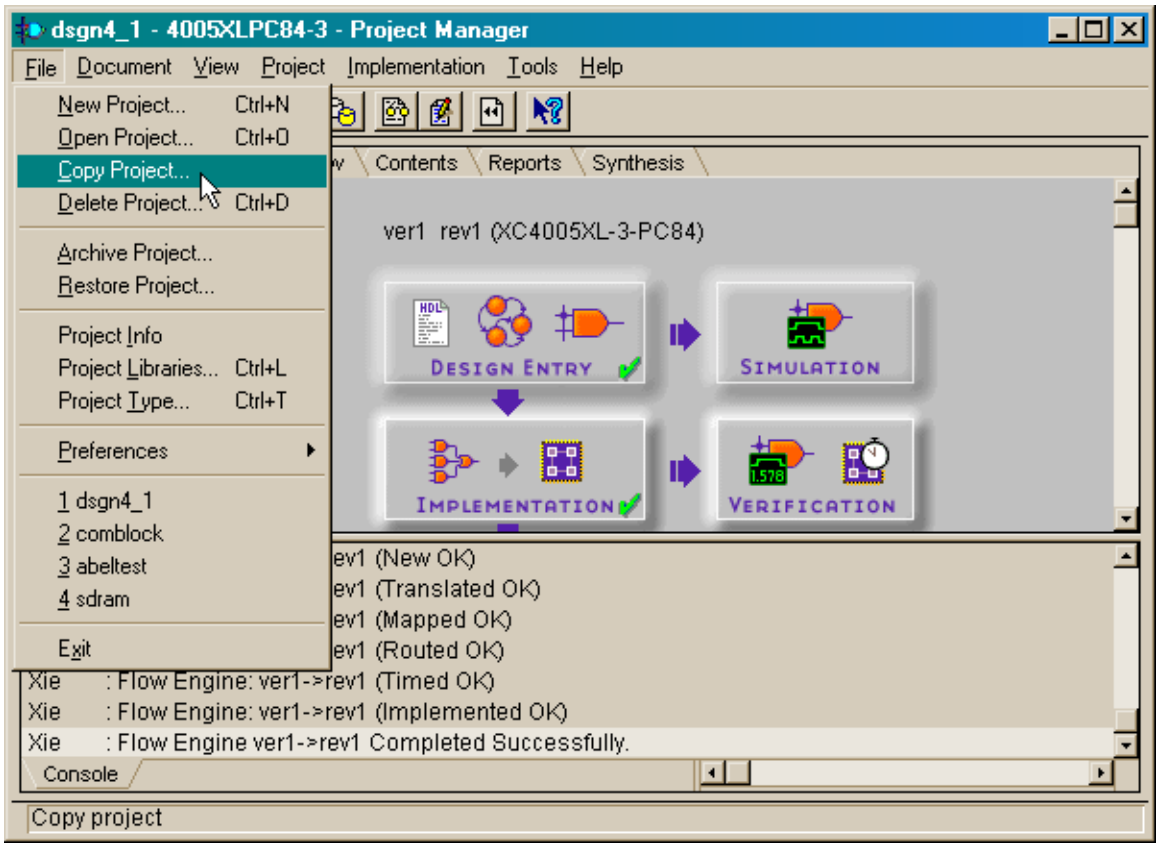
Press key…	LED displays…	New State…	This means…
None		inkey1	The combination lock is ready to begin replacing its current combination with a new combination entered from the keyboard.
a		inkey2	The scancode for ‘a’ has been stored in newkey1.
b		verify1	The scancode for ‘b’ has been stored in newkey2.
a		verify2	The first key of the new combination has been verified.
b		apply	The second key has been verified and the new combination in newkey1 and newkey2 has been moved into key1 and key2, respectively.
return		ckkey1	The lock is locked and is waiting for the combination to be entered.
a		ckkey2	The first key of the combination has been entered.
c		ckkey1	The key sequence did not match the combination so the lock stays locked and waits for the combination to be entered.
a		ckkey2	The first key of the combination has been entered.
b		unlocked	The key sequence matched the combination so the lock opened.
backspace		inkey1	The backspace key indicates the user wants to replace the current combination with a new combination. Any other key would have returned the state machine to the ckkey1 state and locked the lock.

Retargeting the Project to the XS95 Board

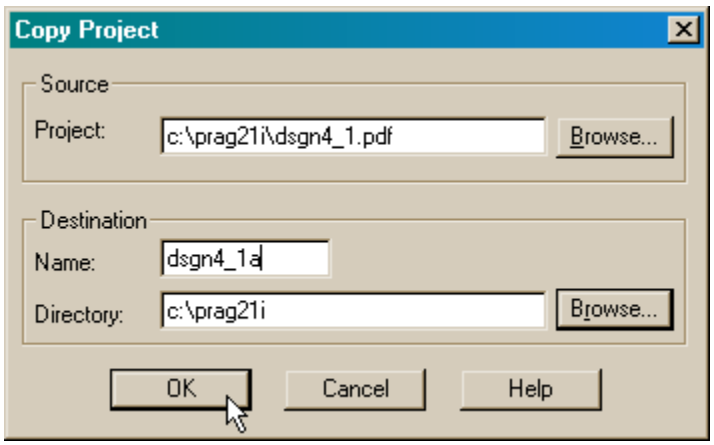
Now we will retarget the combination lock to an XC95108 CPLD on an XS95 Board.

Copying the XS40-Based Combination Lock Project

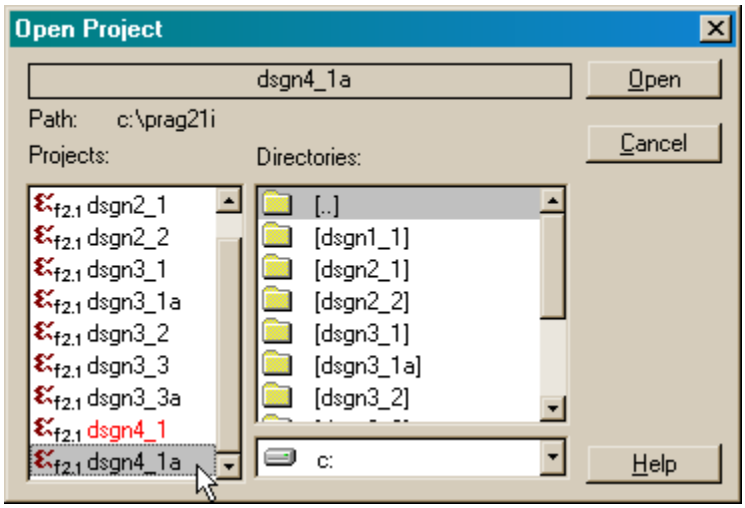
Create a copy of the previous project using the File➔Copy Project... command.



Name the new copy of the project dsgn4_1a.

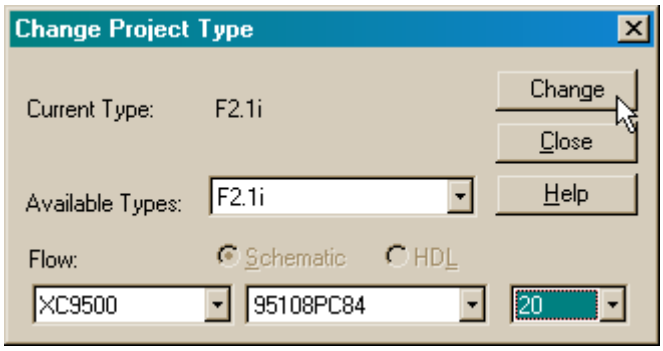


Next, use the File➔Open Project... command to bring up the **Open Project** window. Highlight dsgn4_1a in the list of projects and click on the Open button.

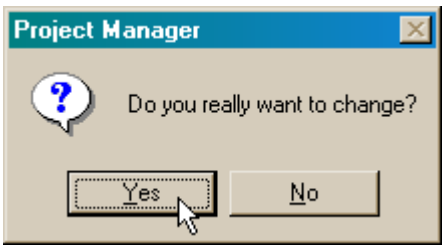


Selecting a New Target Device

Once the new project is opened, use the File➔Project Type... command to change the target device for the project. Select an XC95108 CPLD with a –20 speed grade as the target device.

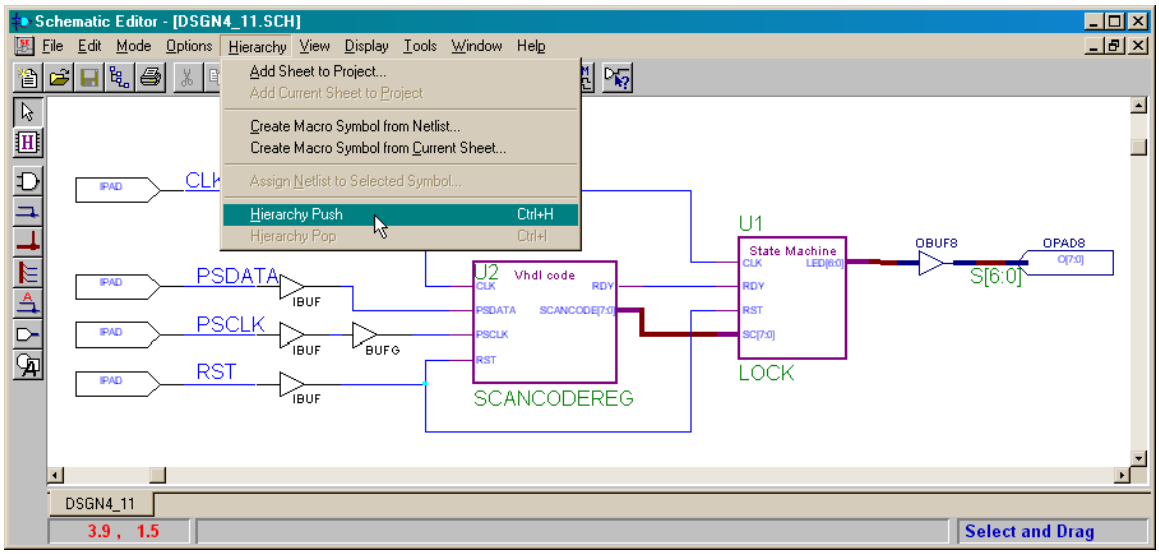


Click the Yes button when asked to verify the change in the target device.

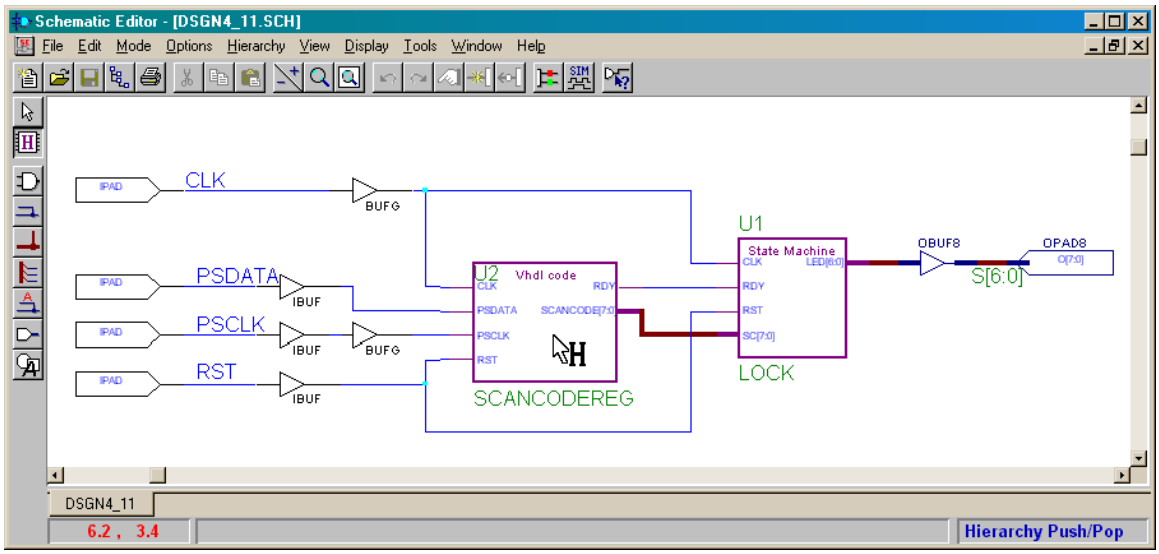


Updating the Modules to Account for the New Target Device

Now double-click the dsgn4_11.sch entry in the **Hierarchy pane** of the **Project Manager** window. We need to re-synthesize the macro netlists so they utilize the features of the XC9500 CPLD instead of the XC4000 FPGA. To start this process, first select the Hierarchy➔Hierarchy Push command in the **Schematic Editor** window.

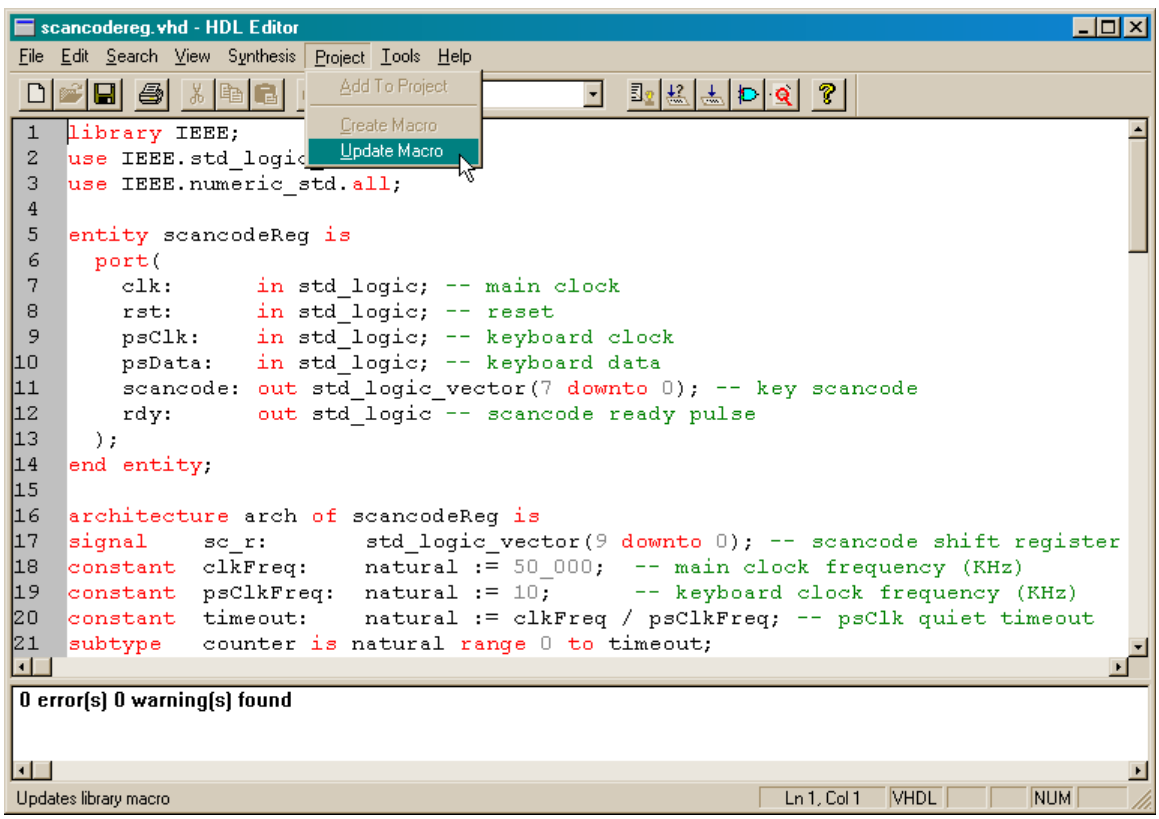


Then double-click on the SCANCODEREG keyboard interface macro. This causes the VHDL source code for the macro to appear in an **HDL Editor** window.

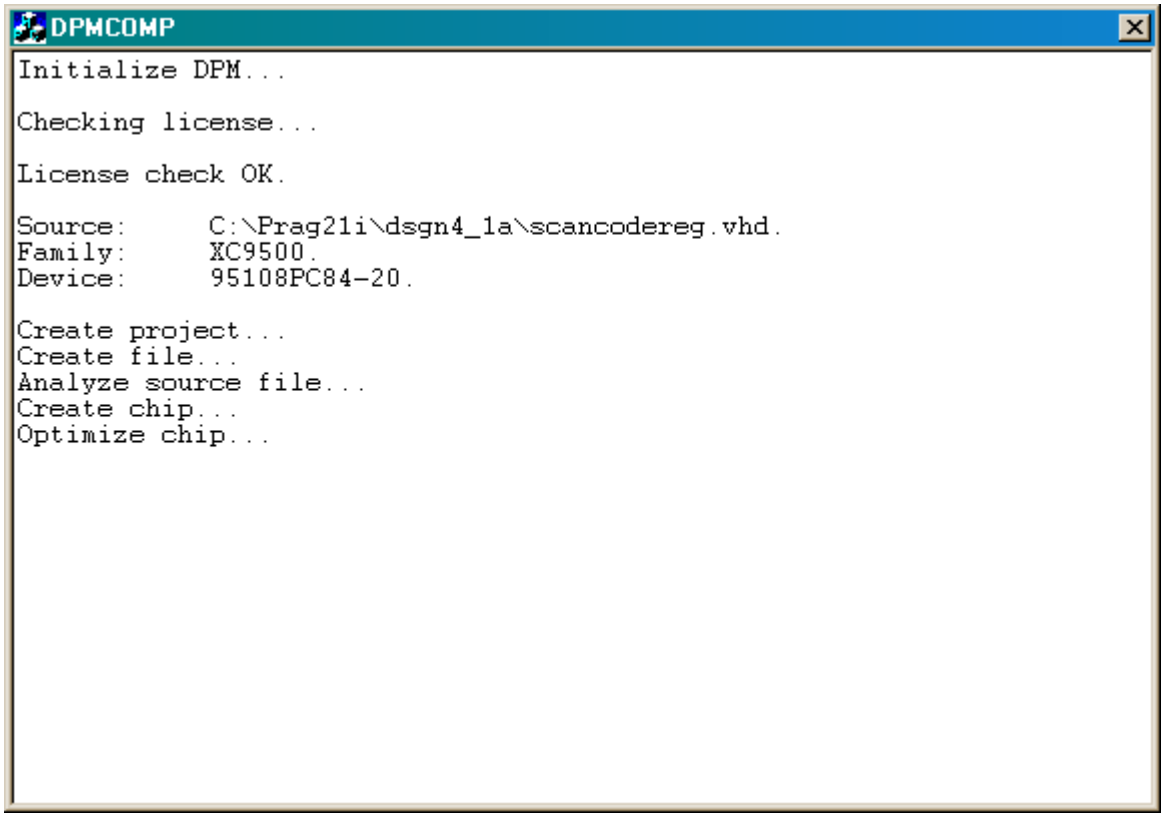


Updating the Keyboard Interface Module

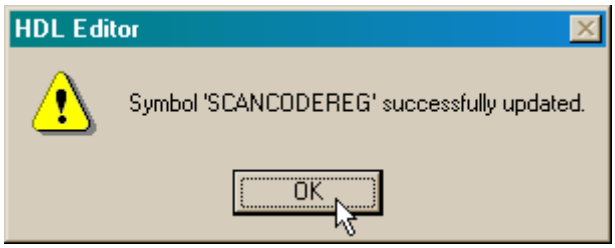
In the **HDL Editor** window, activate the Project➔Update Macro command.



The netlist for the keyboard interface macro will be re-synthesized for the XC95108 CPLD.

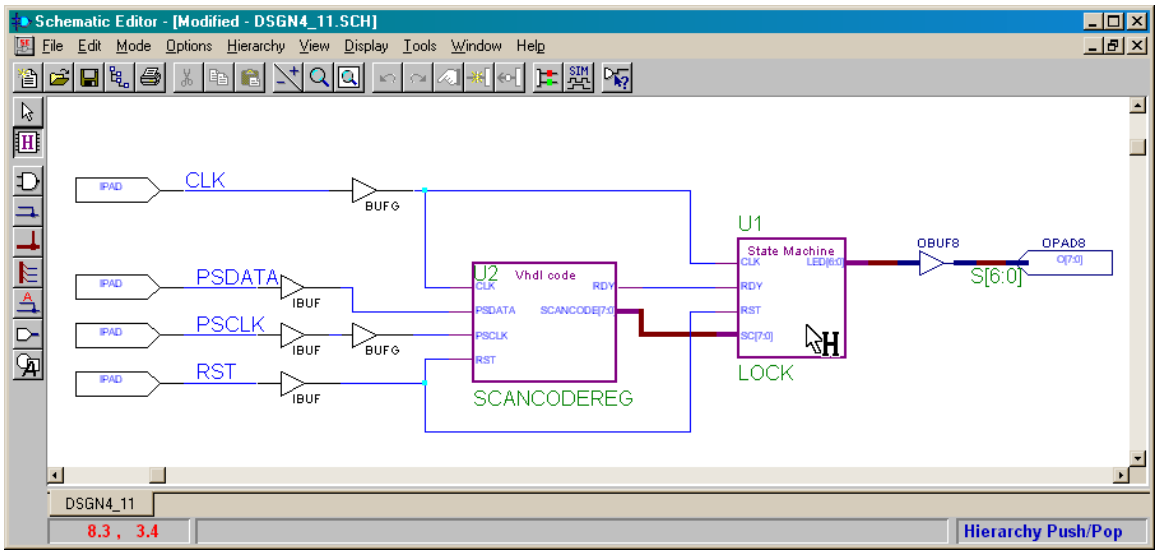


Click the OK button on the acknowledgement of the successful macro update, then close the **HDL Editor** window.

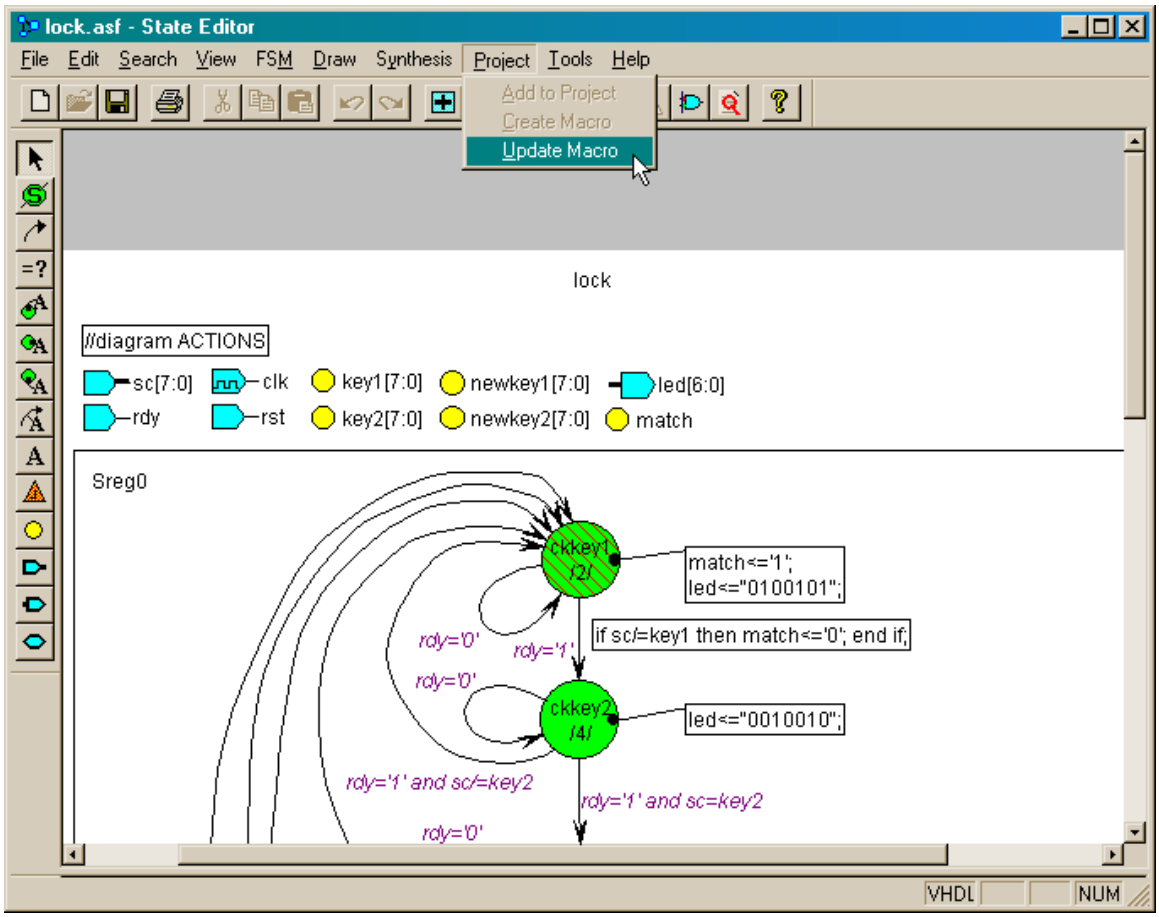


Updating the Lock&Key Module

Once you return to the **Schematic Editor** window, double-click on the lock&key macro (LOCK) to bring up the state diagram in the **State Editor** window.



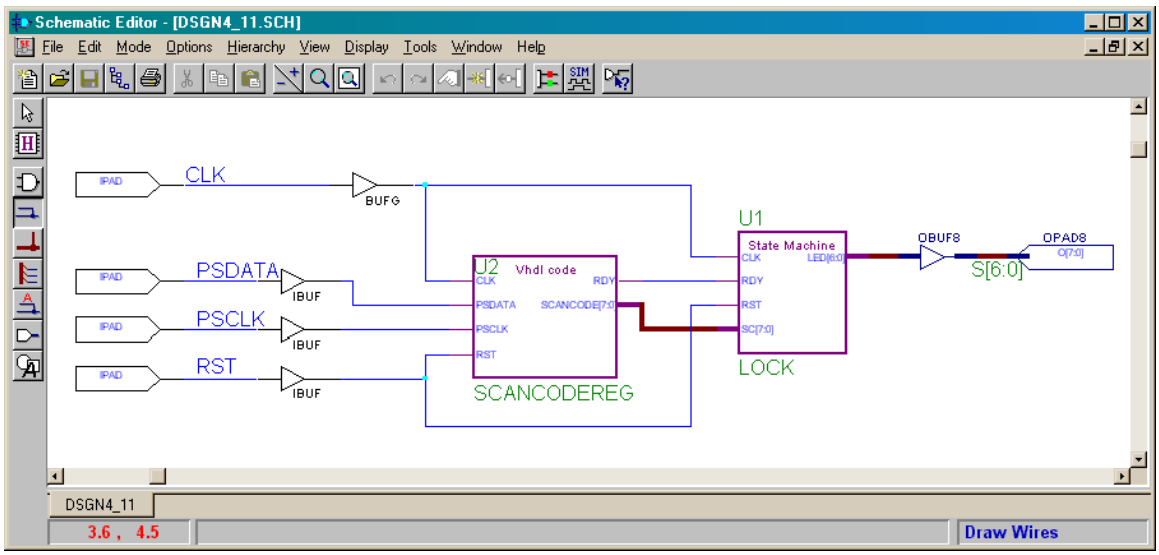
Once again, use the Project➔Update Macro command to re-synthesize the FSM for the XC95108 CPLD.



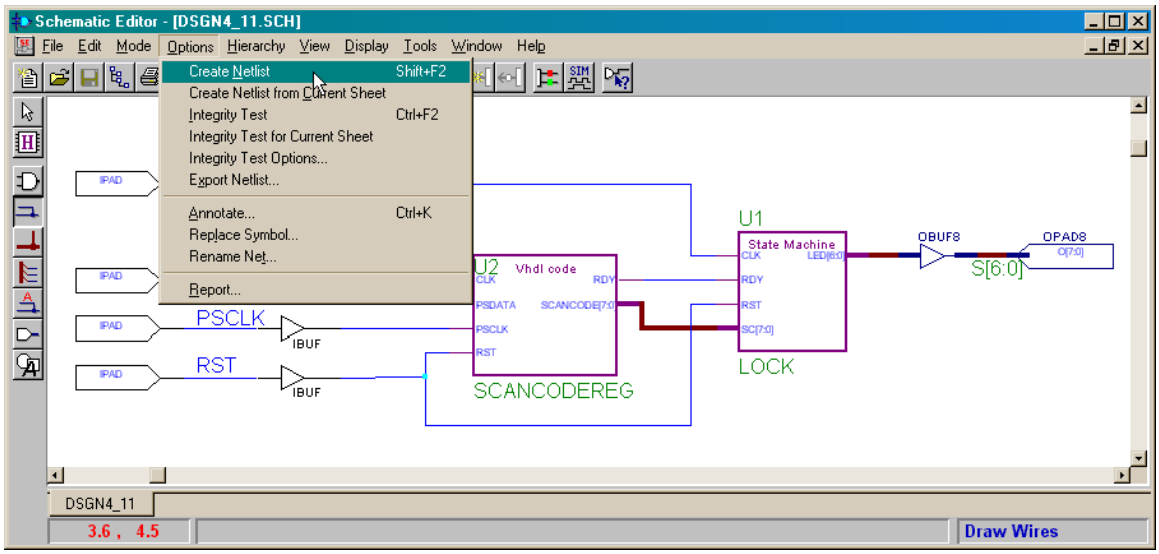
After the lock&key FSM netlist is re-synthesized, close the **State Editor** window and return to the **Schematic Editor** window.

Updating the Top-Level Module

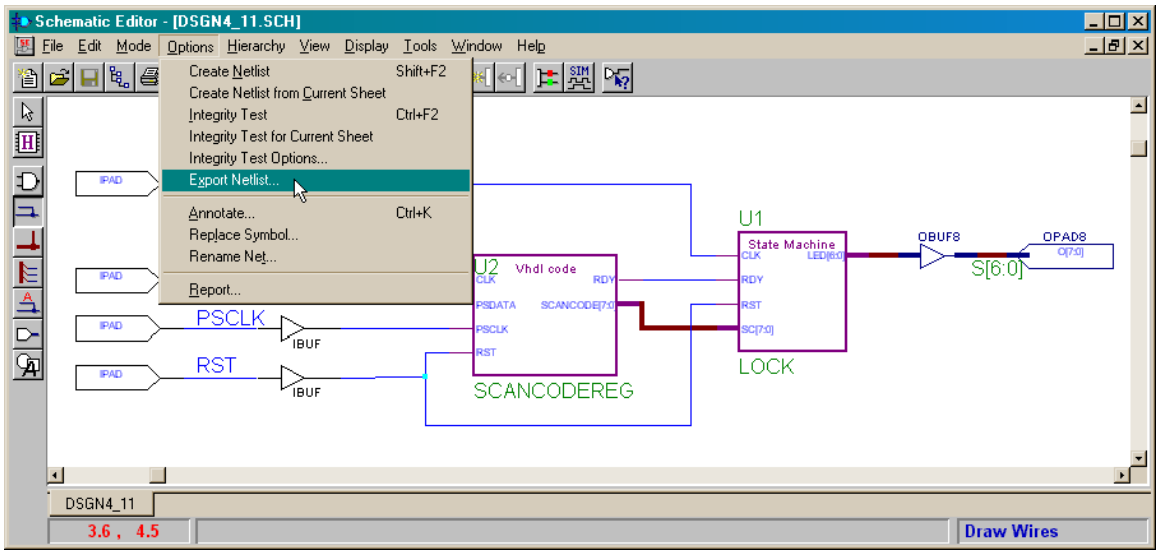
Change the top-level circuit slightly by removing the BUFG from the PSCLK net since that component cannot be connected to a general-purpose I/O in an XC9500 CPLD.



Now execute the Options➔Create Netlist command…

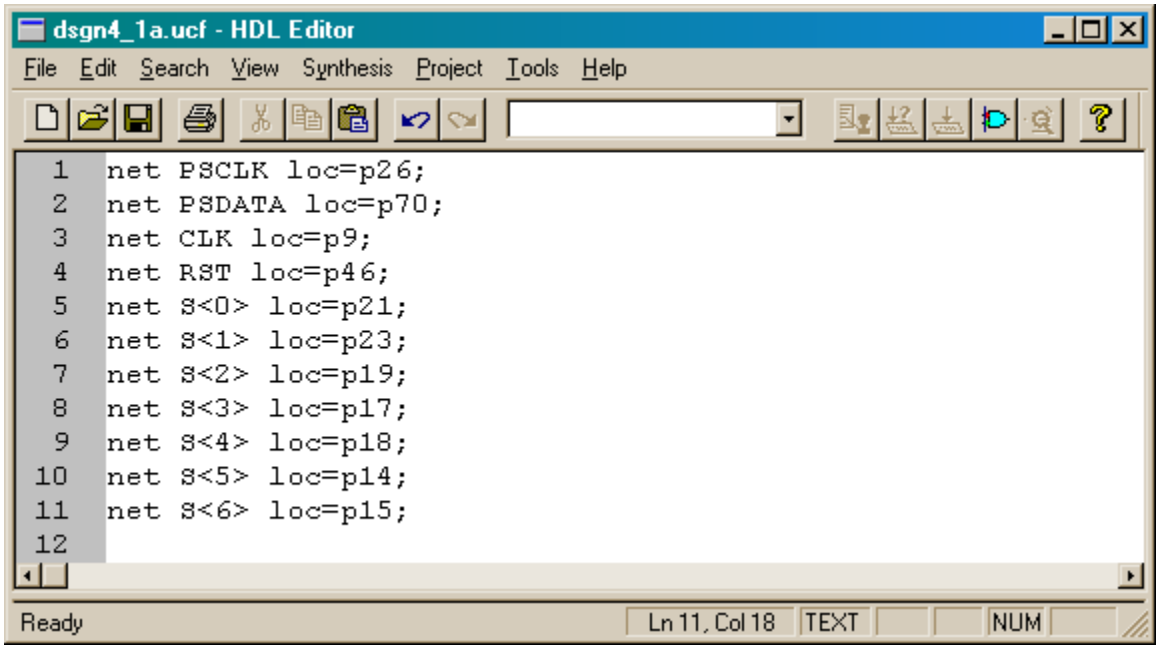


And then activate the Options➔Export Netlist... command. Close the **Schematic Editor** window after the top-level netlist is exported,.



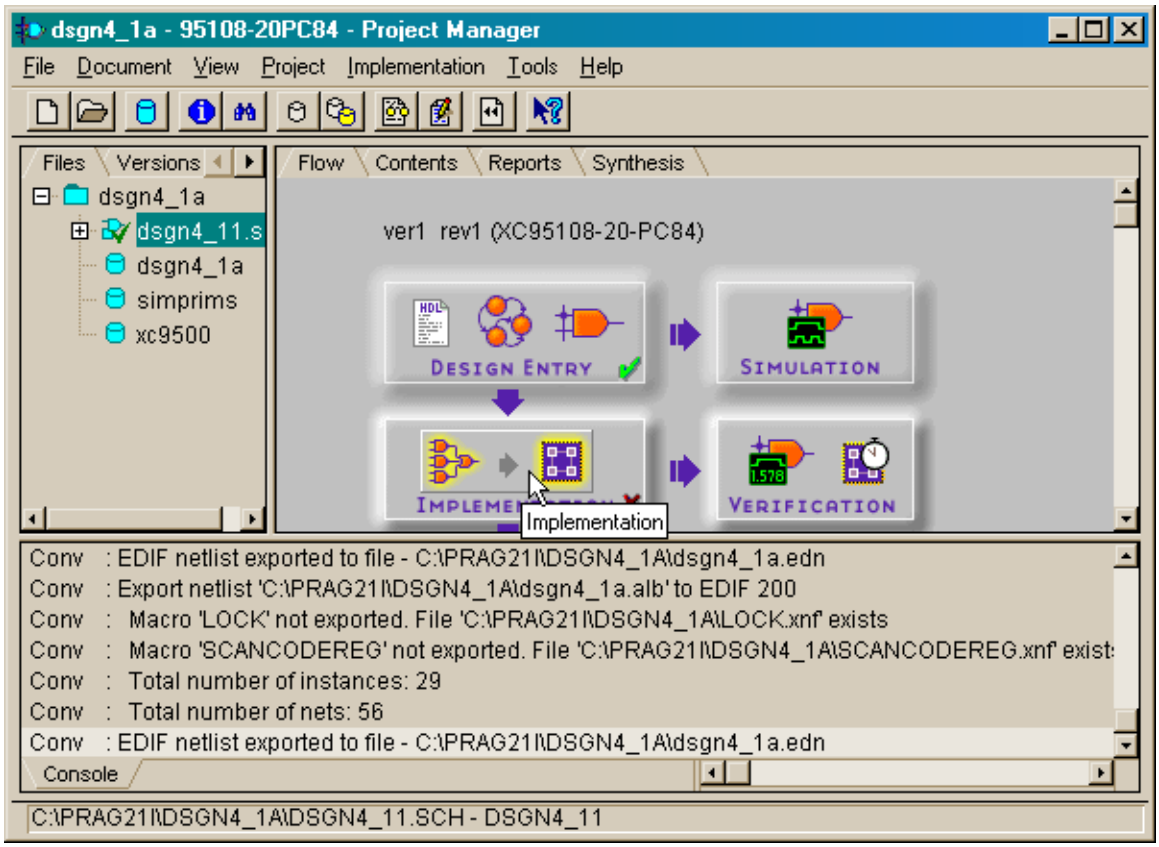
Entering the Pin Assignments for the XS95 Board

Place the following pin assignments for the XS95-108 Board into the dsgn4_1a.ucf constraints file.



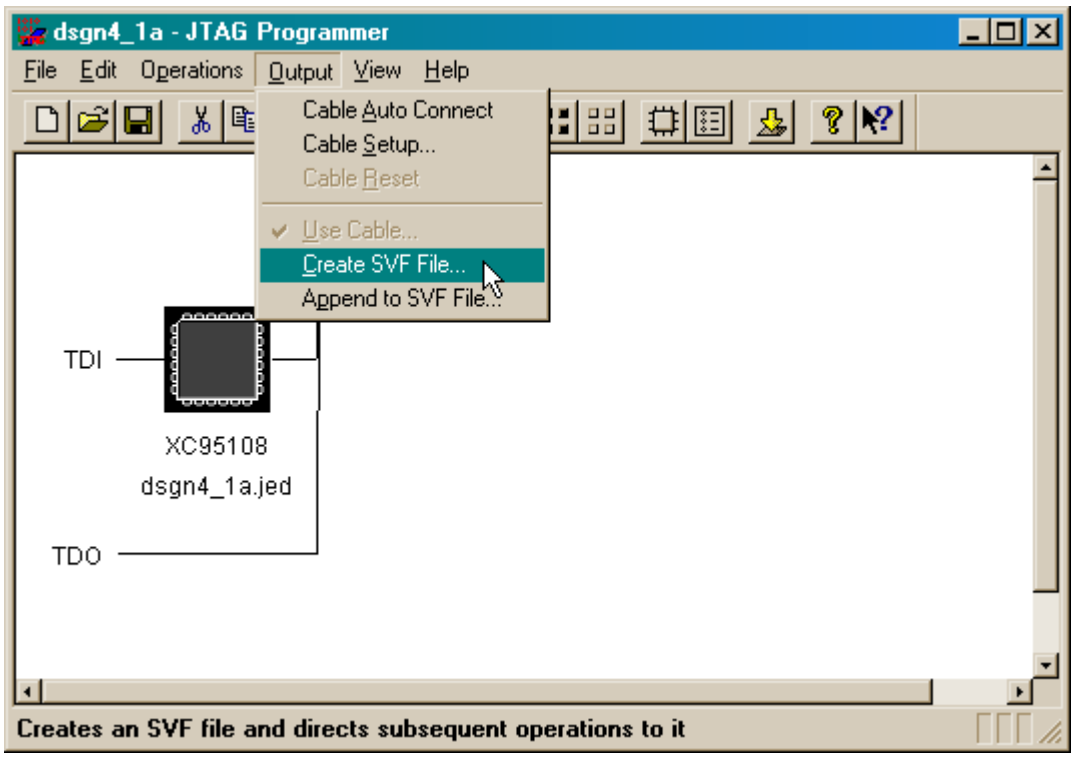
Implementing the Design for the XC95108 CPLD

Then use the implementation tools to map the netlist for the combination lock to the XC95108 CPLD.

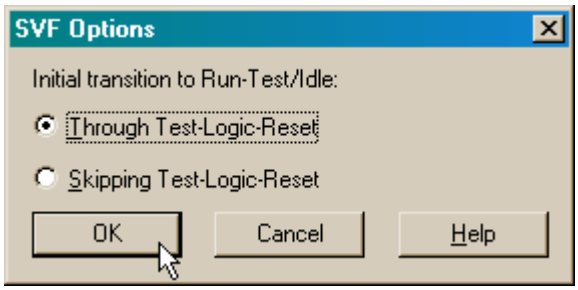


Creating the SVF Bitstream for the XC95108 CPLD

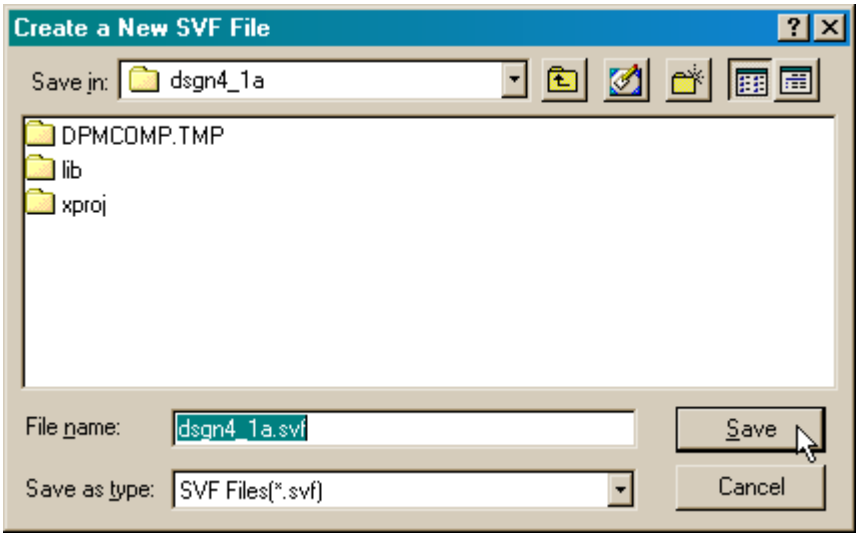
Once the implementation tools complete their tasks, click on the Programming button in the **Flow** pane of the **Project Manager** window. Select the Output➔Create SVF File... in the **JTAG Programmer** window that appears.



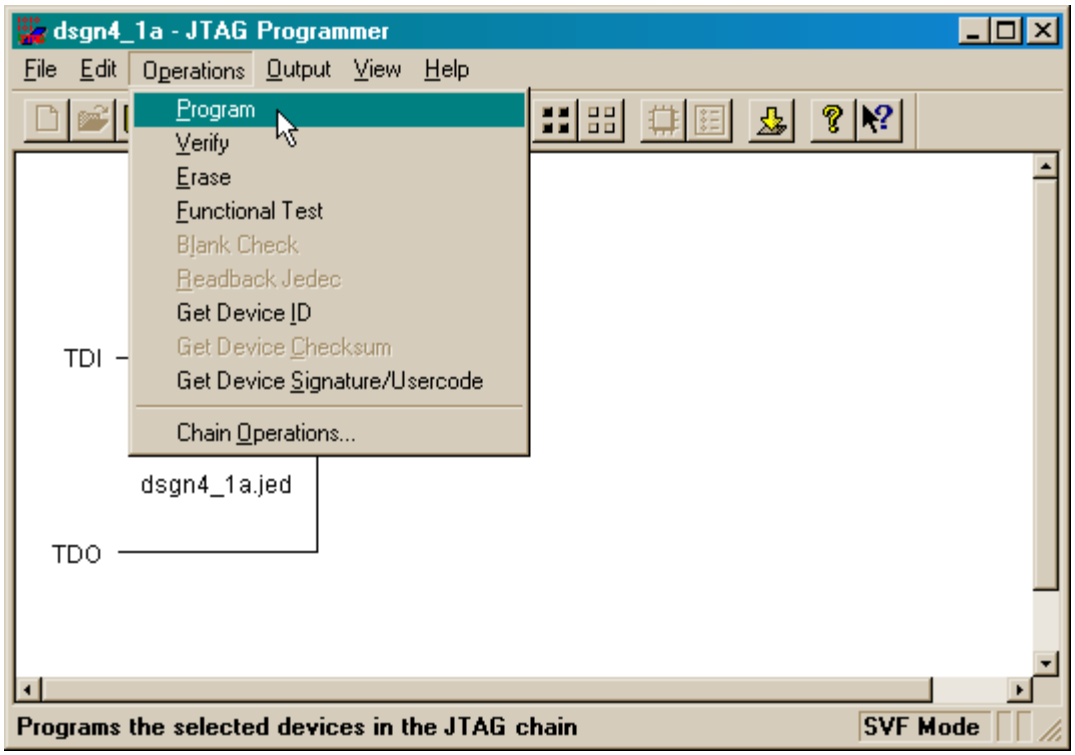
Accept the default SVF option that transitions the XC9500 JTAG downloading circuitry through Test-Logic-Reset before entering the Run-Test/Idle state.



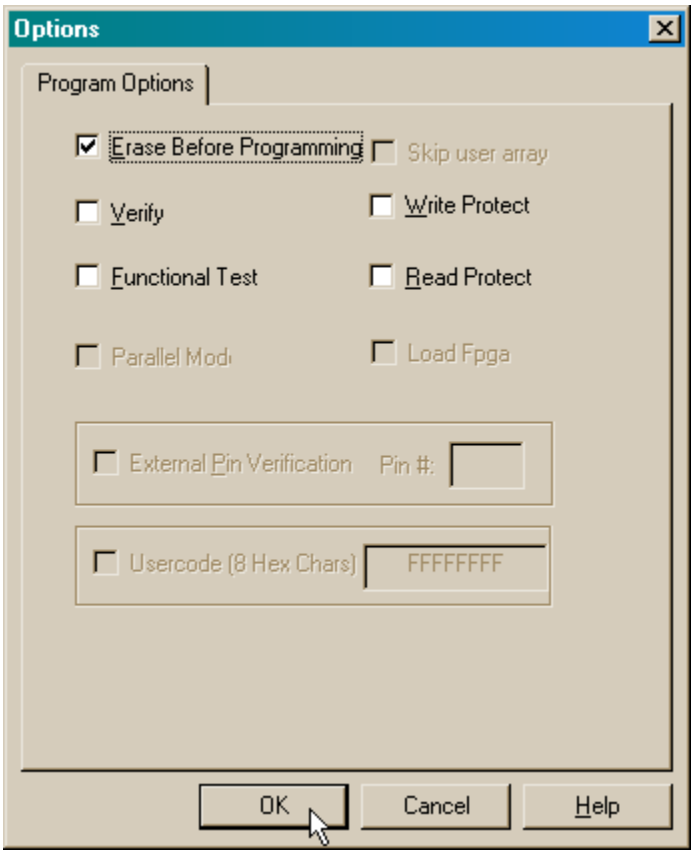
In the **Create a New SVF File** window, move up the directory tree to the top-level of the ***dsgn4_1a*** project and specify the filename for the XC95108 bitstream.



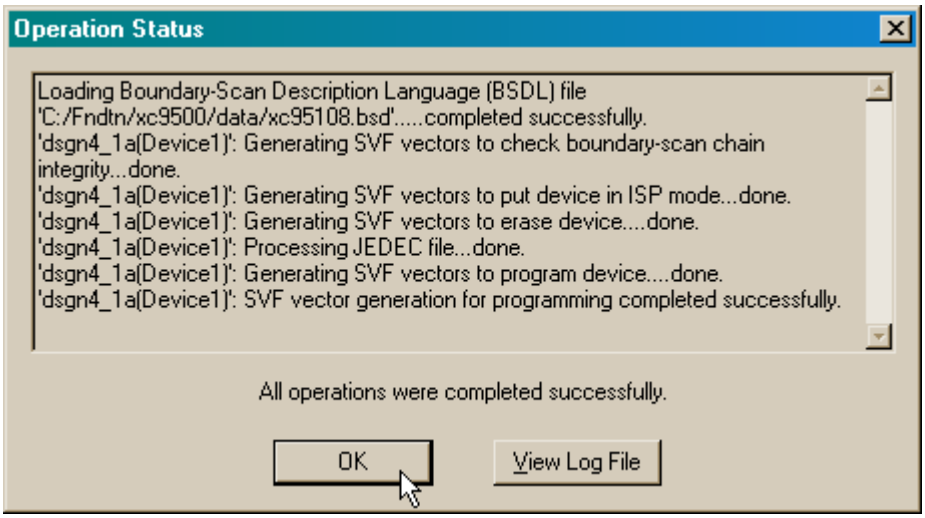
Generate the bitstream using the Operations➔Program command.



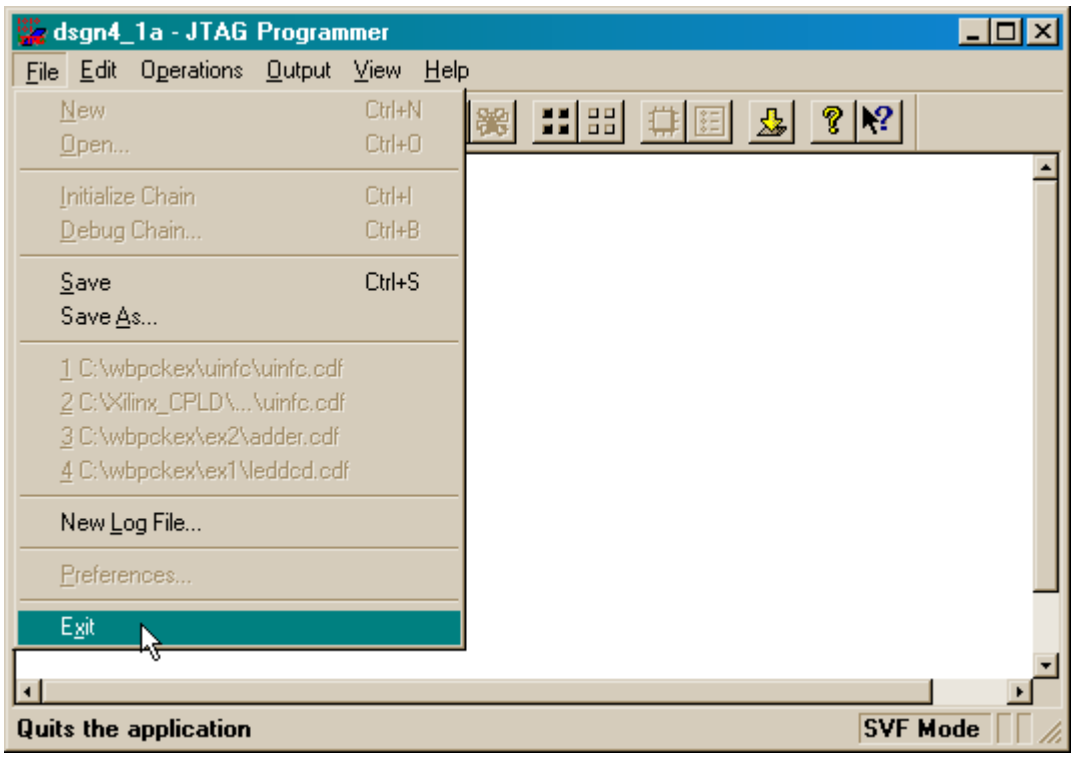
In the **Options** window, check the option that erases the Flash memory in the XC95108 CPLD before programming it with the new bitstream.



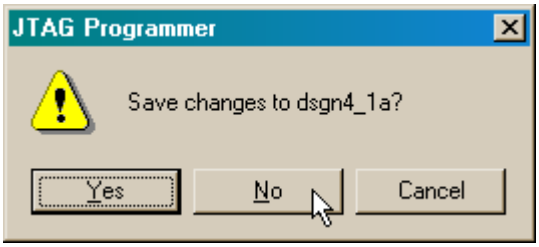
The SVF bitstream should be generated without incident.



Once the SVF file is generated, exit from the **JTAG Programmer** window.

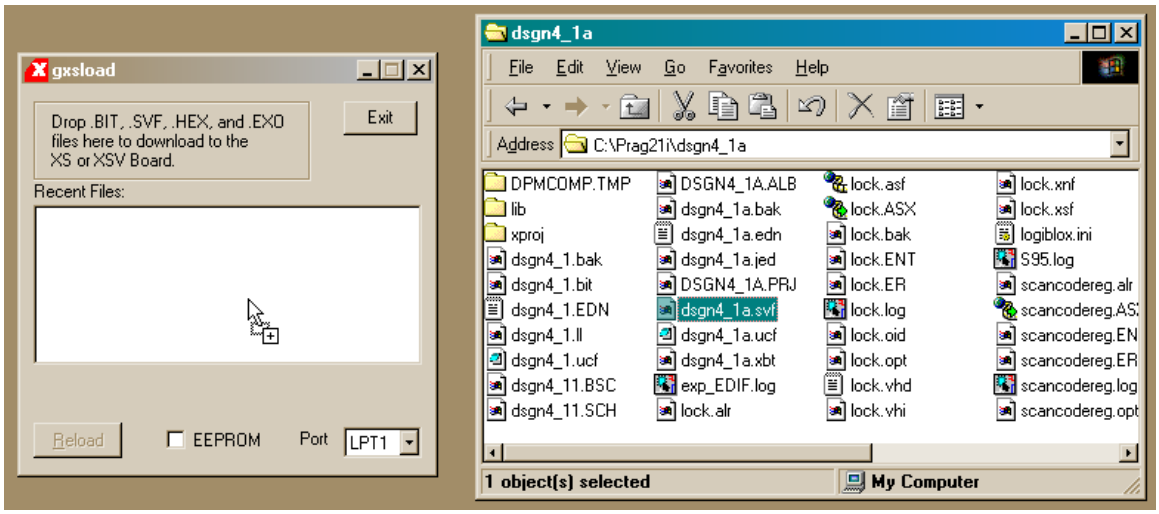


Discard any changes you made to the programming setup for this project. This will not affect the bitstream that you stored in the SVF file.



Downloading the Bitstream to the XS95 Board

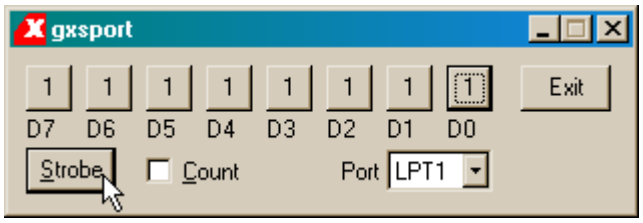
After the programming tools finish, drag-and-drop the dsgn4_1a.svf file into the **GXSLOAD** window to download the bitstream into the XS95-108 Board.



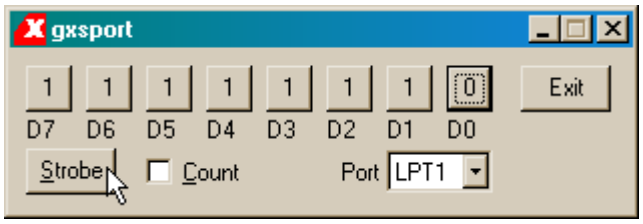
Testing the Combination Lock

After downloading the bitstream to the XS95 Board, attach a PS/2 keyboard to the six-pin mini-DIN socket at the bottom of the board.

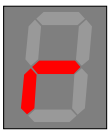
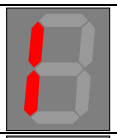
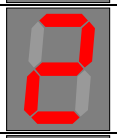
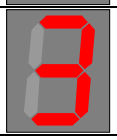
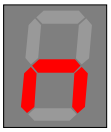
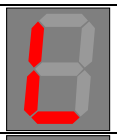
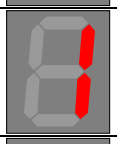
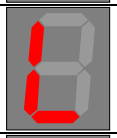
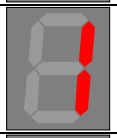
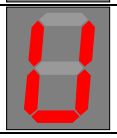
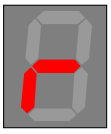
If the LED on the XS95 Board does not display a lower-case R, then you may have to manually reset the combination lock. The reset input for the lock (pin 46 on the CPLD) is connected to data bit D0 of the parallel port. Start the GXSPORT utility and apply a logic 1 to the reset input.



Then apply a logic 0 to release the reset.



Now the combination lock should be ready to respond to key presses. A sequence of key presses and the results are shown below:

Press key…	LED displays…	New State…	This means…
None		inkey1	The combination lock is ready to begin replacing its current combination with a new combination entered from the keyboard.
5		inkey2	The scancode for ‘a’ has been stored in newkey1.
6		verify1	The scancode for ‘b’ has been stored in newkey2.
5		verify2	The first key of the new combination has been verified.
6		apply	The second key has been verified and the new combination in newkey1 and newkey2 has been moved into key1 and key2, respectively.
return		ckkey1	The lock is locked and is waiting for the combination to be entered.
4		ckkey2	The first key of the combination has been entered.
6		ckkey1	The key sequence did not match the combination so the lock stays locked and waits for the combination to be entered.
5		ckkey2	The first key of the combination has been entered.
6		unlocked	The key sequence matched the combination so the lock opened.
backspace		inkey1	The backspace key indicates the user wants to replace the current combination with a new combination. Any other key would have returned the state machine to the ckkey1 state and locked the lock.