MICROPROCESSOR LABORATORY

African Regional Course on Advanced VLSI Design Techniques

24 November - 12 December 2003

Kumasi - Ghana

Exercise 6

Design of a Serial Hex Combination Lock Chip

Problem Description

In this exercise a serial combination electronic lock chip is designed starting from the specifications. This design exercise was inspired by the example of a simple combination lock given in the book, The Art of Digital Design, "An Introduction to Top Down Design", by, Franklin P. Prosser & David E. Winkel, Prentice Hall Inc., Chapter 5. In this design example you will learn to:

- Specify the characteristics of the **lock** starting from scratch as an Algorithmic State Machine (**ASM**).
- Describe the behaviour of the lock's ASM in Alliance **fsm** language and generate the behavioural description of the ASM.
- Add the architectural blocks to the generated behavioural description and simulate the design.
- Generate the structural description of the chip.
- Place the necessary pads and re-simulate the structural description.
- Synthesise the layout of the chip.
- Extract the circuit from the layout.
- Extract the behavioural description from the netlist and compare with the original behaviour file we created, to complete formal verification.

In this design example you will:

- Describe the **ASM** using Alliance **fsm** language putting an output for each state so as to debug the machine (elock.fsm).
- Generate the behavioural file using **Syf** (elocks.vbe).
- Write test pattern files for simulation and validation.
- Simulate the behavioural description of the ASM with the pattern file by using **Asimut**.
- Copy the elock.fsm file to the lock.fsm file and remove the outputs for the states.
- Generate the behavioural file using **Syf** (locks.vbe).
- Copy locks.vbe to lock.vbe and add the architectural blocks to the behavioural description (lock.vbe).
- Re-simulate the behavioural description with the architectural blocks using **Asimut**.
- Synthesise the logic and structural descriptions using **Bop** and **Scmap** (lockl.vst).
- Use **Glop** to add buffers to adjust critical paths and fanouts (lockopt.vst).
- Use the **S**tandard **C**ell **R**outer, **Scr** to place and route the core (lockopt.ap).
- Add the necessary pads for the chip and compile using **Genlib** (lockchip.vst).
- Use **Asimut** to simulate the 'lockchip.vst' file with the pattern file developed earlier.
- Place the pads and generate the layout of the chip with pads using **Ring** (lockchip.ap).
- Use **Tas** to perform the static timing analysis.
- Use **Lynx** to extract the netlist from the layout file 'lockchip.ap' (lockchip.al).
- Use **Lvx** to compare the extracted circuit 'lockchip.al' and the original 'lockchip.vst' file created by **Genlib**.
- Use **Yagle** to extract the behaviour, 'lockchip.vbe' from the 'lockchip.al' netlist file.
- Use **Proof** to compare the extracted behaviour file, 'lockchip.vbe' and the behavioural file created in the first phase, 'lock.vbe'.

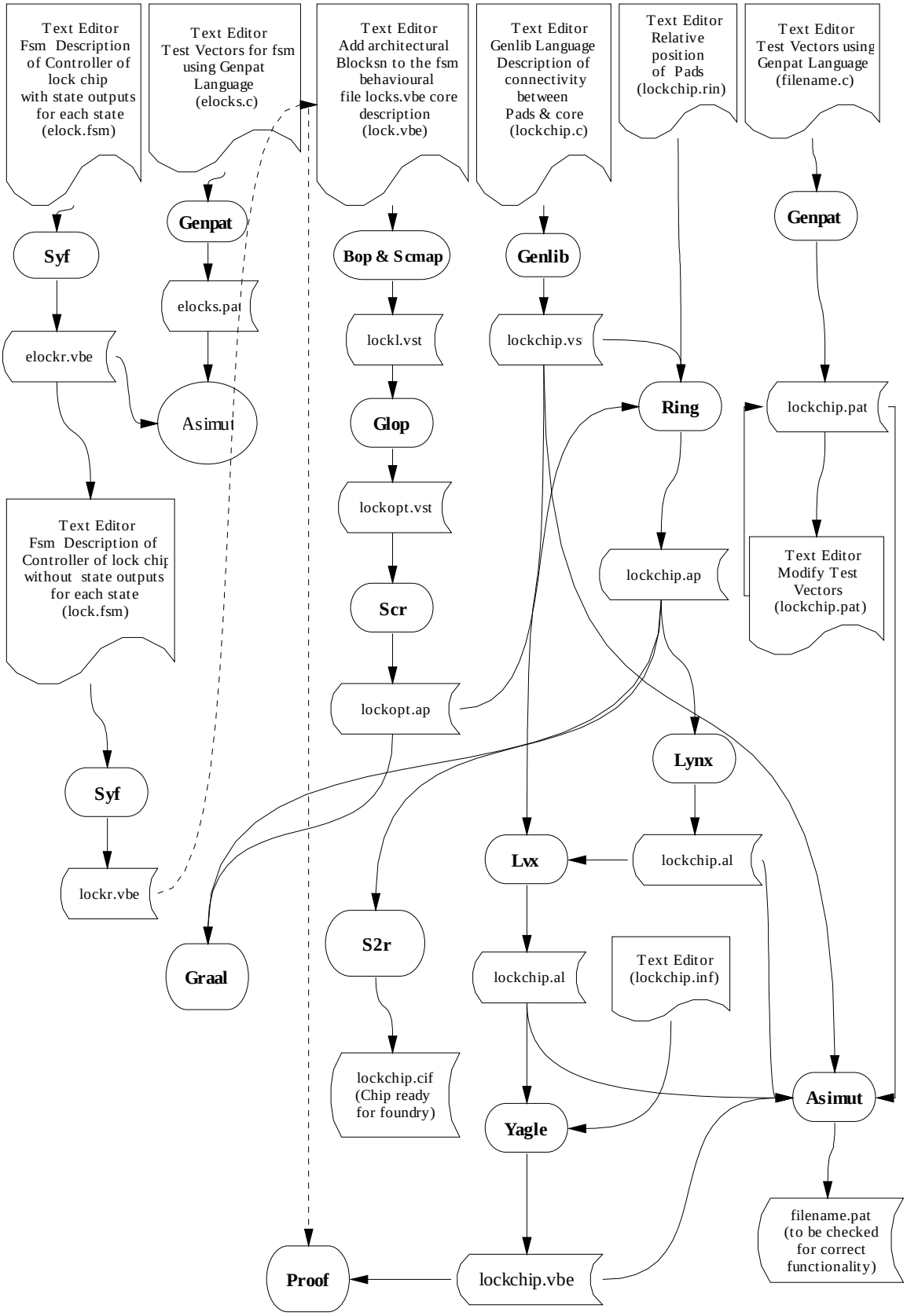


Fig 1. Design flow for the Hex Combination Lock

A Serial Combination Lock

Background:

We build in this exercise an electronic version of a mechanical combination lock that is available in the market.

Mechanical locks come in two flavours, parallel and serial. A parallel combination lock is a suitcase type of lock, where there are 3 to 4 disks that can be rotated independently to the correct combination. A serial lock is dial type of lock that comes on safety lockers in banks: a single dial is rotated through a sequence of numbers in the correct order. Any wrong number requires that, the procedure of entering the numbers is started all over again.

In this design example we design an electronic version of the serial combination lock. The lock's combination is entered in hexadecimal notation, one digit at a time. Any wrong digit sends the lock to an error state, which requires a reset signal to start all over.

Target System:

A "N" digit serial combination lock that lights a light when the combination is correct. The number of digits "N" for the combination is chosen by the user. The combination is programmed by the user.

Designing the lock's Algorithm:

The following important design decisions are taken before the design of the algorithm.

1. Data is entered through a hexadecimal keypad. The keypad output is a 4 bit bus that is called "keynum[3:0]" which indicates the number that has been punched. The keypad has a strobe signal that is called "keypress" that lasts for one cycle of the system clock that indicates that one of the keys of the key pad has been punched. The keypad is debounced and sends only one "keypress" signal even if any of the keypad buttons is held down. To send another "keypress" signal, the keypad key has to be released and pressed down again.

2. Combination is entered from left to right. A maximum of 8-digit combination is allowed.

3. A "reset" button is provided to start over if a combination error is made. The "reset" button is debounced.

4. A "set" button is provided to allow the user to program the combination. The "set" button is debounced.

5. The user presses a "try" button to indicate the end of sequence entry and the machine should check the sequence and if it matches, to command the lock to open. The "try" signal lasts for only a clock cycle like the "keypress" signal. The "try" button is debounced.

6. A light lights up if the sequence is correct, but does not give any information if the sequence is wrong.

The ASM for the combination lock is shown below.

In each state the ASM checks for the “reset” or the “set” button press. A reset puts the machine in the INIT state. The machine enters the READ_COMB state in the following clock cycle.

In the READ_COMB, a “try” signal sends the ASM to the ERR state, whereas a “keypress” signal compares the number punched in with that stored in the reference. This compared signal is called the “cmpdig” signal. A successful digit comparison allows the comparison of the next number in the sequence, but otherwise puts the machine in the ERR state. How many numbers in the sequence should we check? We have a counter to keep track of the number of digits entered in a sequence. In our lock design we use a 3 bit counter so that we can have a maximum of 8 digit sequence combinations. As each digit is compared successfully we increment the counter, until it reaches the count of “N”. The reference digit is function of the counter’s output, and as each digit is compared successfully, the reference digit is updated to the next digit in the sequence to be compared. The number of digits to be compared “N” is tested and given out as a “cmpnum” signal. This is comparison of the counter’s output and a register that stores the number “N”. When the counter reaches a terminal count equal to “N” after all successful digit comparison operation, the machine goes to the TEST state.

In the TEST state, a “keypress signal” send the machine to the ERR state. The test for the “keypress” signal is included in this state, so that even if someone arrives to the correct combination in the sequence by luck, he does not know the number of digits to be punched in! A “try” signal puts the machine in the state OK.

In the OK state, the “openlock” signal is validated and the lock opens. The lock closes if the “reset” button is pressed and the machine goes back to the state INIT.

The combination sequence is stored in registers. These registers are accessed for a read or write operation by the ASM. The ASM uses the 3 bit counter to present the address to these registers. The reference numbers stored in these registers can be changed by pressing the “set” button that puts the ASM in the SET_INIT state. The number “N” is programmable and is automatically set when the user enters the combination sequence of the lock in the SET_COMB state and then presses a “reset” to indicate the end of the combination setting procedure.

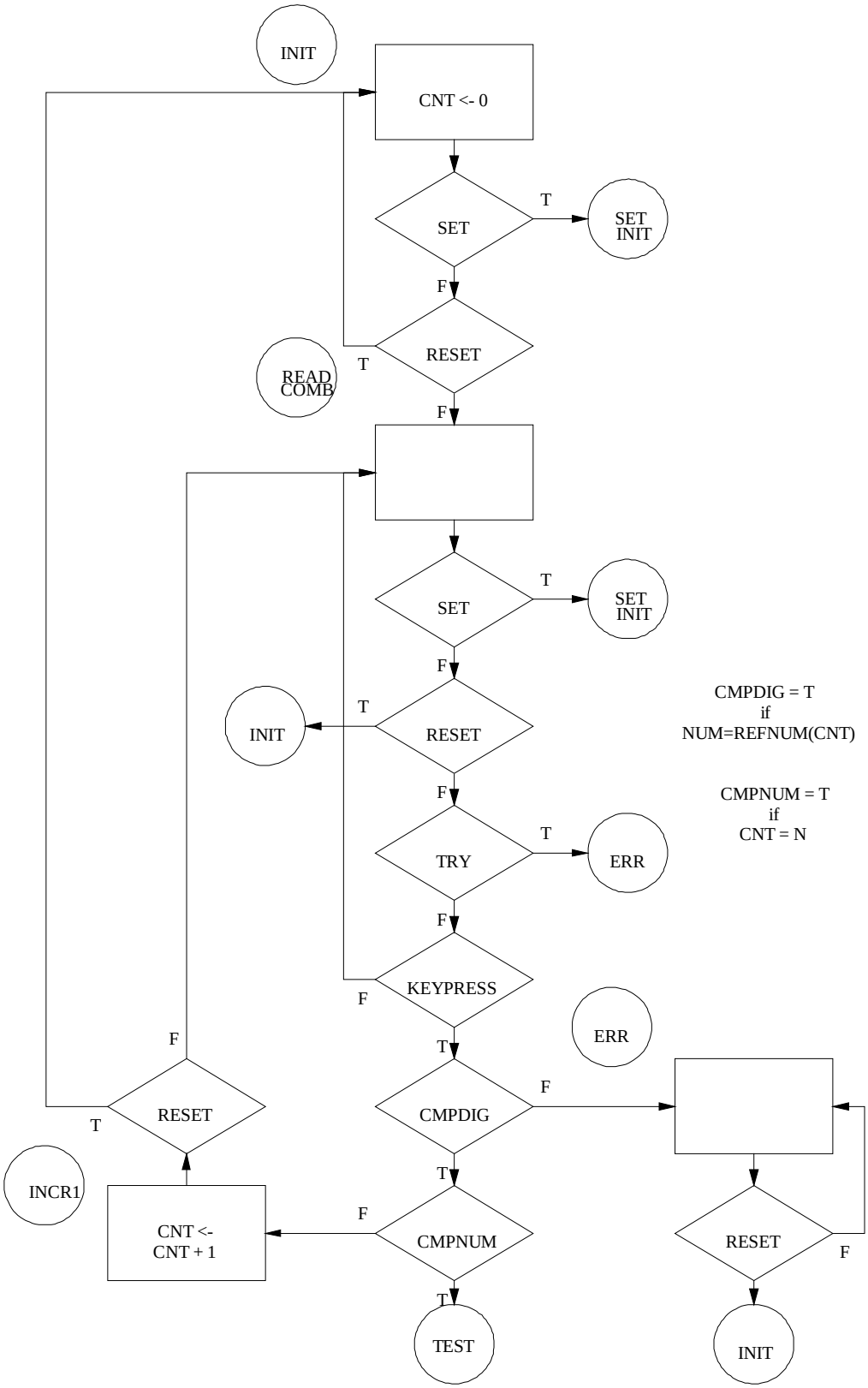


Fig.2 ASM chart of the Serial Combination Lock

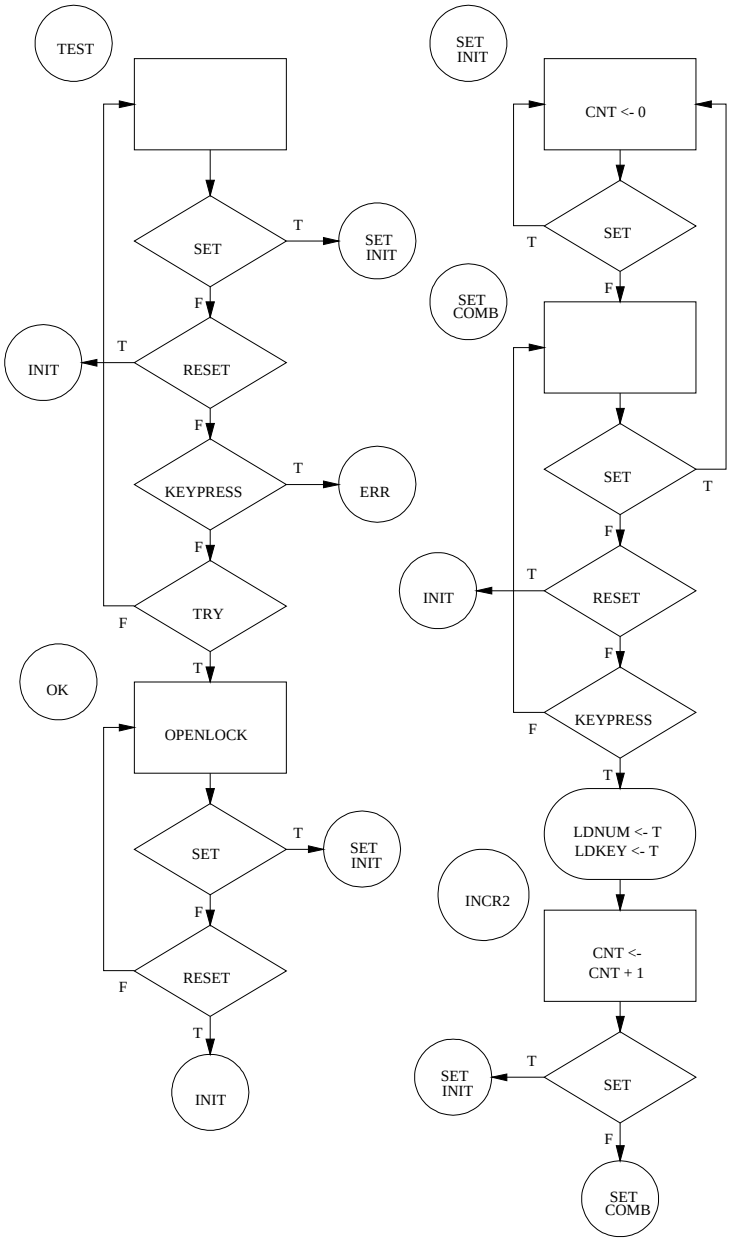


Fig. 2 (cont'd) ASM of the Serial Combination Lock

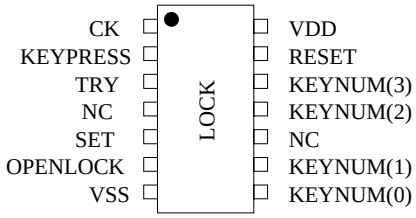


Fig. 3 Lock chip (a possible pinout diagram).

Solution

Legend

Give the command that appears immediately after this symbol, at the command line.

Edit and save into a file, all that appears after this symbol.

Explanation of a topic

Set the environmental variables as shown immediately after this symbol.

Creating the Design

Begin by creating a design directory, at a convenient position in your work space:

```
mkdir lock
```

Change into this directory:

```
cd lock
```

Create with the text editor a file called “elock.fsm”. Enter the following and save the file.

```
Entity elock is

   port(
ck : in bit ;
reset: in bit;
try : in bit;
keypress : in bit;
set : in bit;
cmpnum : in bit;
cmpdig : in bit;
openlock : out bit;
        inccnt : out bit;
        rescnt : out bit;
        ldkey : out bit;
        ldnum : out bit;
        testflag, initflag, okflag, errflag, readflag, inc1flag,
                inc2flag,      setinitflag, setcombflag : out bit
);
End elock;

architecture auto of elock is
type STATE_TYPE is
 (INIT,READ_COMB,INC1,ERR,SET_INIT,SET_COMB,INC2,TEST,OK);

-- pragma CLOCK ck
-- pragma CUR_STATE CURRENT_STATE
-- pragma NEX_STATE NEXT_STATE

signal CURRENT_STATE, NEXT_STATE : STATE_TYPE;

begin

PROCESS(CURRENT_STATE,reset,try,keypress,set,cmpnum,cmpdig)
     begin
     case CURRENT_STATE is
                WHEN INIT => initflag <= ’1’;
                if (set=’1’) then
```

```
        NEXT_STATE <= SET_INIT;
        rescnt <= '1';
else if (reset='0') then
        NEXT_STATE <= INIT;
        rescnt <= '1';
else
        NEXT_STATE <= READ_COMB;
end if;
end if;

WHEN READ_COMB => readflag <= '1';
if (set='1') then
        NEXT_STATE <= SET_INIT;
        rescnt <= '1';
else if (reset='0') then
        NEXT_STATE <= INIT;
        rescnt <='1';
else if
        (try='1') then
        NEXT_STATE <= ERR;
else if
        (keypress='0') then
        NEXT_STATE <= READ_COMB;
else if
        (cmpdig='0') then
        NEXT_STATE <= ERR;
else if
        (cmpnum='1') then
        NEXT_STATE <= TEST;
else
        NEXT_STATE<= INC1;
        inccnt <= '1';
end if;
end if;
end if;
end if;
end if;
end if;

WHEN ERR => errflag <= '1';
if (set='1') then
        NEXT_STATE <= SET_INIT;
        rescnt <= '1';
else if (reset='0') then
        NEXT_STATE <= INIT;
        rescnt <= '1';
else
        NEXT_STATE <= ERR;
end if;
end if;
WHEN INC1 => inc1flag <= '1';
if (set='1') then
        NEXT_STATE <= SET_INIT;
else if (reset='0') then
        NEXT_STATE <= INIT;
        rescnt <= '1';
else
        NEXT_STATE <= READ_COMB;
end if;
end if;

WHEN TEST => testflag <= '1';
if (set = '1') then
        NEXT_STATE <= SET_INIT;
        rescnt <= '1';
else if (reset='0') then
        NEXT_STATE <= INIT;
        rescnt <= '1';
else if
        (keypress='1') then
        NEXT_STATE <= ERR;
else if
        (try = '0') then
        NEXT_STATE <= TEST;
else
        NEXT_STATE <= OK;
end if;
end if;
end if;
end if;
```

```
                    WHEN OK => okflag <= '1';
                    openlock <= '1';
                    if (set = '1') then
                           NEXT_STATE <= SET_INIT;
                           rescnt <= '1';
                    else if (reset = '0') then
                           NEXT_STATE <= INIT;
                           rescnt <= '1';
                    else
                           NEXT_STATE <= OK;
                    end if;
                    end if;

                    WHEN SET_INIT => setinitflag <= '1';
                    if (set = '1') then
                           NEXT_STATE <= SET_INIT;
                           rescnt <= '1';
                    else
                           NEXT_STATE <= SET_COMB;
                    end if;

                    WHEN SET_COMB => setcombflag <= '1';
                    if (set = '1') then
                           NEXT_STATE <= SET_INIT;
                           rescnt <= '1';
                    else if
                           (reset = '0') then
                           NEXT_STATE <= INIT;
                           rescnt <= '1';
                    else if
                           (keypress = '0') then
                           NEXT_STATE <= SET_COMB;
                    else
                           NEXT_STATE <= INC2;
                           ldnum <= '1';
                           ldkey <= '1';
                           inccnt <= '1';
                    end if;
                    end if;
                    end if;

                    WHEN INC2 => inc2flag <= '1';
                    if (set = '1') then
                           NEXT_STATE <= SET_INIT;
                           rescnt <= '1';
                    else
                           NEXT_STATE <= SET_COMB;
                    end if;

                    WHEN others =>
                            assert ('1')
                           report "illegal state";

             end case;
end process;

process(ck)
     begin
          if(ck = '1' and not ck' stable) then
               CURRENT_STATE <= NEXT_STATE;
          end if;
end process;

end auto;
```

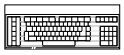
Compare the state assignments and the conditions under which the state, changes with that shown in the ASM chart. Notice the similarity between the ASM chart and the description given in the **fsm**. We want to debug the state machine before we do anything else with it. Therefore we have assigned a output flag to each of the state, which become '1' if the machine is in that state. Thus we can follow the transition of states during a simulation.

Give the following command at the command line

```
syf -rV elock

r       -       Random encoding
V       -       verbose mode
```

This command produces a file "elockr.vbe", which is the behavioural description of the fsm description. This behavioural description can be simulated using asimut.

Test pattern file and simulation of the state machine



Write a pattern file for simulation. (You can write a C file that when treated with **Genpat** will generate the pattern file for you. See exercise 3).
Modify the pattern file by editing it and simulate using **Asimut** with the **-b** option and check if the state machine performs satisfactorily.

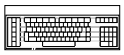
Adding Architectural Blocks

The behavioural file "elockr.vbe" contains only the description of the ASM. Now we will have to add the architectural blocks, like the register that stores the combination, the register that stores the number of digits to be compared, the counter and, implement the various comparison operations. The ASM controls the architectural blocks and some of the signals that appear in the "Entity" declaration become internal signals that control these blocks.

Once we are sure that the state machine changes state as it should under the specified conditions, the various flag signals that we put in the "elock.fsm" file to debug the state machine, can be removed.
We start by copying the "elock.fsm" file to "lock.fsm" and editing this file to remove the state flag signals from the description.

```
cp elock.fsm lock.fsm
```

Edit the file "lock.fsm" to remove the state flag signals to produce a description as shown below.



```
Entity lock is

   port(
ck : in bit ;
reset: in bit;
try : in bit;
keypress : in bit;
set : in bit;
cmpnum : in bit;
cmpdig : in bit;
openlock : out bit;
       inccnt : out bit;
       rescnt : out bit;
       ldkey : out bit;
       ldnum : out bit
);
End lock;

architecture auto of lock is

type STATE_TYPE is
 (INIT,READ_COMB,INC1,ERR,SET_INIT,SET_COMB,INC2,TEST,OK);

-- pragma CLOCK ck
-- pragma CUR_STATE CURRENT_STATE
-- pragma NEX_STATE NEXT_STATE

signal CURRENT_STATE, NEXT_STATE : STATE_TYPE;


begin

PROCESS(CURRENT_STATE,reset,try,keypress,set,cmpnum,cmpdig)
    begin
    case CURRENT_STATE is
               WHEN INIT =>
               if (set='1') then
                      NEXT_STATE <= SET_INIT;
               else if (reset='0') then
                      NEXT_STATE <= INIT;
                      rescnt <= '1';
               else
                      NEXT_STATE <= READ_COMB;
               end if;
               end if;

               WHEN READ_COMB =>
               if (set='1') then
                      NEXT_STATE <= SET_INIT;
               else if (reset='0') then
                      NEXT_STATE <= INIT;
                      rescnt <='1';
               else if
                      (try='1') then
                      NEXT_STATE <= ERR;
               else if
                      (keypress='0') then
                      NEXT_STATE <= READ_COMB;
               else if
                      (cmpdig='0') then
                      NEXT_STATE <= ERR;
               else if
                      (cmpnum='1') then
                      NEXT_STATE <= TEST;
               else
                      NEXT_STATE<= INC1;
                      inccnt <= '1';
               end if;
               end if;
               end if;
               end if;
               end if;
               end if;

               WHEN ERR =>
               if (set='1') then
                      NEXT_STATE <= SET_INIT;
               else if (reset='0') then
                      NEXT_STATE <= INIT;
                      rescnt <= '1';
               else
                      NEXT_STATE <= ERR;
               end if;
               end if;
```

```
                        WHEN INC1 =>
                        if (set='1') then
                                NEXT_STATE <= SET_INIT;
                        else if (reset='0') then
                                NEXT_STATE <= INIT;
                                rescnt <= '1';
                        else
                                NEXT_STATE <= READ_COMB;
                        end if;
                        end if;

                        WHEN TEST =>
                        if (set = '1') then
                                NEXT_STATE <= SET_INIT;
                        else if (reset='0') then
                                NEXT_STATE <= INIT;
                                rescnt <= '1';
                        else if
                                (keypress='1') then
                                NEXT_STATE <= ERR;
                        else if
                                (try = '0') then
                                NEXT_STATE <= TEST;
                        else
                                NEXT_STATE <= OK;
                        end if;
                        end if;
                        end if;
                        end if;

                        WHEN OK =>
                        openlock <= '1';
                        if (set = '1') then
                                NEXT_STATE <= SET_INIT;
                        else if (reset = '0') then
                                NEXT_STATE <= INIT;
                                rescnt <= '1';
                        else
                                NEXT_STATE <= OK;
                        end if;
                        end if;

                        WHEN SET_INIT =>
                        if (set = '1') then
                                NEXT_STATE <= SET_INIT;
                                rescnt <= '1';
                        else
                                NEXT_STATE <= SET_COMB;
                        end if;

                        WHEN SET_COMB =>
                        if (set = '1') then
                                NEXT_STATE <= SET_INIT;
                                rescnt <= '1';
                        else if
                                (reset = '0') then
                                NEXT_STATE <= INIT;
                                rescnt <= '1';
                        else if
                                (keypress = '0') then
                                NEXT_STATE <= SET_COMB;
                        else
                                NEXT_STATE <= INC2;
                                ldnum <= '1';
                                ldkey <= '1';
                                inccnt <= '1';
                        end if;
                        end if;
                        end if;

                        WHEN INC2 =>
                        if (set = '1') then
                                NEXT_STATE <= SET_INIT;
                                rescnt <= '1';
                        else
                                NEXT_STATE <= SET_COMB;
                        end if;

                        WHEN others =>
                                 assert ('1')
                                report "illegal state";

                end case;
        end process;

        process(ck)
             begin
                if(ck = '1' and not ck' stable) then
```

```
                    CURRENT_STATE <= NEXT_STATE;
                end if;
        end process;

        end auto;
```

Give the command to synthesise the ".vbe" file.

```
syf -rV lock
```

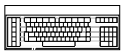
This produces a "lockr.vbe" file as output. This file contains only the controller. The "Entity" statement here contains the output signals that control the architectural blocks and the input signals that decide the next state of the state machine. The architectural blocks are:

1. the 3-bit counter that counts the number of digits punched in,
2. the comparator that gives the "cmpdig" signal to the state machine,
3. the comparator that compares the reference number with the one that is punched in through the key board. and gives the "cmpnum" signal,
4. the decoder that brings in the correct reference number from the memory and,
5. the memory that holds the reference numbers.

To add the architectural blocks, to this file we edit the state machine behavioural description. We convert the signals that control the architectural blocks, the signals that are input to the state machine, (and are not required outside) as internal "Signals". Then the block's behaviour are described while keeping the interface signals between the blocks and the state machine the same.

Copy the file "lockr.vbe" to the file named "lock.vbe". Edit this file and add the architectural block description to the behavioural description as shown below.

Read the comments that have been given under the special comment line marked by --**, to understand the changes that have been made to the file.



```
-- VHDL data flow description generated from 'locks'

-- Entity Declaration

ENTITY lock IS
  PORT (
  vdd, vss: in BIT;
  ck : in BIT; -- ck
  reset : in BIT;      -- reset
  try : in BIT;        -- try
  keypress : in BIT;   -- keypress
  set : in BIT;        -- set
  openlock : out BIT;  -- openlock
  keynum: in BIT_VECTOR (3 downto 0)
  );
END lock;


-- Architecture Declaration

ARCHITECTURE behaviour_data_flow OF lock IS
  --** All the signals that control the architectural blocks and that
  --** are not required outside the chip become internal signals.
  SIGNAL cmpdig, cmpnum, inccnt, rescnt, ldkey, ldnum : BIT;
  --** The memory that stores the combination is declared
  SIGNAL mem0, mem1, mem2, mem3, mem4, mem5, mem6, mem7: REG_VECTOR (3
downto 0) REGISTER;
  --** The counter and the register that stores the number of digits to
  --** compare in a sequence is declared
  SIGNAL counter, num : REG_VECTOR (2 downto 0) REGISTER;
```

```
  --** These are signals declared by syf
  SIGNAL current_state_0 : REG_BIT REGISTER; -- current_state_0
  SIGNAL current_state_1 : REG_BIT REGISTER; -- current_state_1
  SIGNAL current_state_2 : REG_BIT REGISTER; -- current_state_2
  SIGNAL current_state_3 : REG_BIT REGISTER; -- current_state_3
  SIGNAL init_s : BIT;        -- init_s
  SIGNAL init_m : BIT;        -- init_m
  SIGNAL read_comb_s : BIT;          -- read_comb_s
  SIGNAL read_comb_m : BIT;          -- read_comb_m
  SIGNAL inc1_s : BIT;        -- inc1_s
  SIGNAL inc1_m : BIT;        -- inc1_m
  SIGNAL err_s : BIT;         -- err_s
  SIGNAL err_m : BIT;         -- err_m
  SIGNAL set_init_s : BIT;           -- set_init_s
  SIGNAL set_init_m : BIT;           -- set_init_m
  SIGNAL set_comb_s : BIT;           -- set_comb_s
  SIGNAL set_comb_m : BIT;           -- set_comb_m
  SIGNAL inc2_s : BIT;        -- inc2_s
  SIGNAL inc2_m : BIT;        -- inc2_m
  SIGNAL test_s : BIT;        -- test_s
  SIGNAL test_m : BIT;        -- test_m
  SIGNAL ok_s : BIT;          -- ok_s
  SIGNAL ok_m : BIT;          -- ok_m

BEGIN
--** counter description
count: BLOCK (ck = '1' and not ck'STABLE)
       BEGIN
       counter <= GUARDED B"000" when (rescnt='1') else
                       B"001" when ((inccnt='1') and (counter = B"000"))
else
                       B"010" when ((inccnt='1') and (counter = B"001"))
else
                       B"011" when ((inccnt='1') and (counter = B"010"))
else
                       B"100" when ((inccnt='1') and (counter = B"011"))
else
                       B"101" when ((inccnt='1') and (counter = B"100"))
else
                       B"110" when ((inccnt='1') and (counter = B"101"))
else
                       B"111" when ((inccnt='1') and (counter = B"110"))
else
                       B"000" when ((inccnt='1') and (counter = B"111"))
else
                       counter;
end BLOCK count;

--** Generation of the cmpdig signal
cmpdig <= ((counter=B"000") and (mem0 = keynum)) or
          ((counter=B"001") and (mem1 = keynum)) or
          ((counter=B"010") and (mem2 = keynum)) or
          ((counter=B"011") and (mem3 = keynum)) or
          ((counter=B"100") and (mem4 = keynum)) or
          ((counter=B"101") and (mem5 = keynum)) or
          ((counter=B"110") and (mem6 = keynum)) or
          ((counter=B"111") and (mem7 = keynum));

--** Generation of the cmpnum signal
cmpnum <= (counter=num);

--** condition under which the num register is loaded
loadnum: BLOCK (ck='1' and not ck'STABLE)
          BEGIN
          num <= GUARDED counter WHEN (ldnum='1') else
                       num;
end BLOCK loadnum;

--** condition under which the sequence is loaded into the registers.
loadkey: BLOCK (ck='1' and not ck'STABLE)
BEGIN
mem0 <= GUARDED keynum WHEN ((counter=B"000") and (ldkey='1')) else
                    mem0;
mem1 <= GUARDED keynum WHEN ((counter=B"001") and (ldkey='1')) else
                    mem1;
mem2 <= GUARDED keynum WHEN ((counter=B"010") and (ldkey='1')) else
                    mem2;
mem3 <= GUARDED keynum WHEN ((counter=B"011") and (ldkey='1')) else
                    mem3;
mem4 <= GUARDED keynum WHEN ((counter=B"100") and (ldkey='1')) else
                    mem4;
mem5 <= GUARDED keynum WHEN ((counter=B"101") and (ldkey='1')) else
                    mem5;
mem6 <= GUARDED keynum WHEN ((counter=B"110") and (ldkey='1')) else
                    mem6;
mem7 <= GUARDED keynum WHEN ((counter=B"111") and (ldkey='1')) else
                    mem7;
end BLOCK loadkey;
```

```
--** This is the .vbe description synthesised by syf for the state
--** machine description made in lock.fsm
  ok_m <= ((try and not (keypress ) and reset and not (set ) and
test_s)
or (reset and not (set ) and ok_s));
  ok_s <= (not (current_state_0 ) and current_state_1 and not
(current_state_2) and current_state_3);
  test_m <= ((not (try ) and not (keypress ) and reset and not (set )
and test_s) or (cmpnum and cmpdig and keypress and not (try ) and
reset and not (set ) and read_comb_s));
  test_s <= (not (current_state_0 ) and not (current_state_1 ) and
current_state_2 and not (current_state_3 ));
  inc2_m <= (keypress and reset and not (set ) and set_comb_s);
  inc2_s <= (not (current_state_0 ) and current_state_1 and
current_state_2and not (current_state_3 ));
  set_comb_m <= ((not (set ) and inc2_s) or (not (keypress ) and reset
and not(set ) and set_comb_s) or (not (set ) and set_init_s));
  set_comb_s <= (not (current_state_0 ) and current_state_1 and
current_state_2 and current_state_3);
  set_init_m <= ((set and test_s) or (set and inc2_s) or (set and
err_s) or (set and set_comb_s) or (set and ok_s) or (set and
set_init_s) or
(set and inc1_s) or (set and read_comb_s) or (set and init_s));
  set_init_s <= (current_state_0 and not (current_state_1 ) and not
(current_state_2) and not (current_state_3 ));
  err_m <= ((keypress and reset and not (set ) and test_s) or (reset
and
not (set ) and err_s) or (not (cmpdig ) and keypress and not
(try ) and reset and not (set ) and read_comb_s) or (try and
reset and not (set ) and read_comb_s));
  err_s <= (not (current_state_0 ) and not (current_state_1 ) and not
(current_state_2) and current_state_3);
  inc1_m <= (not (cmpnum ) and cmpdig and keypress and not (try ) and
reset and not (set ) and read_comb_s);
  inc1_s <= (not (current_state_0 ) and current_state_1 and not
(current_state_2) and not (current_state_3 ));
  read_comb_m <= ((reset and not (set ) and inc1_s) or (not (keypress )
and not(try ) and reset and not (set ) and read_comb_s) or (reset and
not (set ) and init_s));
  read_comb_s <= (not (current_state_0 ) and not (current_state_1 ) and
not (current_state_2) and not (current_state_3 ));
  init_m <= ((not (reset ) and not (set ) and test_s) or (not (reset )
and not (set ) and err_s) or (not (reset ) and not (set ) and
set_comb_s) or (not (reset ) and not (set ) and ok_s) or (not (reset )
and not (set ) and inc1_s) or (not (reset ) and not (set ) and
read_comb_s) or (not (reset ) and not (set ) and init_s));
  init_s <= (not (current_state_0 ) and not (current_state_1 ) and
current_state_2 and current_state_3);
  label0 : BLOCK ((ck and not (ck'STABLE )) = '1')
  BEGIN
    current_state_3 <= GUARDED (init_m or err_m or set_comb_m or ok_m);
  END BLOCK label0;
  label1 : BLOCK ((ck and not (ck'STABLE )) = '1')
  BEGIN
    current_state_2 <= GUARDED (init_m or set_comb_m or inc2_m or
test_m);
  END BLOCK label1;
  label2 : BLOCK ((ck and not (ck'STABLE )) = '1')
  BEGIN
    current_state_1 <= GUARDED (inc1_m or set_comb_m or inc2_m or
ok_m);
  END BLOCK label2;
  label3 : BLOCK ((ck and not (ck'STABLE )) = '1')
  BEGIN
    current_state_0 <= GUARDED set_init_m;
  END BLOCK label3;

openlock <= not ok_s;

inccnt <= ((not (cmpnum ) and cmpdig and keypress and not (try ) and
reset and not (set ) and read_comb_s) or (keypress and reset and not
(set ) and set_comb_s));

rescnt <= ((not (reset ) and not (set ) and init_s) or (not (reset )
and
not (set ) and read_comb_s) or (not (reset ) and not (set )
and inc1_s) or (not (reset ) and not (set ) and err_s) or (set
and set_init_s) or (not (reset ) and not (set ) and set_comb_s)
or (set and set_comb_s) or (set and inc2_s) or (not (reset )
and not (set ) and test_s) or (not (reset ) and not (set ) and
ok_s));

ldkey <= (keypress and reset and not (set ) and set_comb_s);

ldnum <= (keypress and reset and not (set ) and set_comb_s);
END;
```

Test Pattern Generation and Simulation of the Complete Behavioural Description

Write a pattern file to test the “lock.vbe” file.
Modify the pattern file by editing it and simulate using **Asimut** with the **-b** option and check if the behavioural description performs satisfactorily.

Logic and Structural Synthesis of the Core

Now Logic can be used to optimise and synthesise the core of the chip from the above behavioural description.

Give the command:

```
bop –o lock lockl
```

This takes as input the “lock.vbe” description and creates an optimised Boolean behavioural description file “lockl.vbe”.

To synthesise the structural description give the command:

```
scmap lockl lockl
```

This takes as input the optimised behavioural description “lockl.vbe” and creates the structural description file “lockl.vst” using the components from the standard cell library.

The structural description created above has been created without worrying about the standard cells fanout limits and critical path signals. Alliance **Glop** can analyse the structural description and create a new description by adding buffers to the appropriate nets.

Give the command:

```
glop -g lockl lockopt –i -t
```

This command takes “lockl.vst” structural description and generates a “lockopt.vst” file after buffers have been added to the critical paths.

Give the command:

```
glop -f lockopt lockopt
```

-f - fanout optimization.

This command should add buffers to the appropriate nets to resolve fanout problems and write over the “lockopt.vst” file created above.

Placement and Routing of the core

The core can now be routed using **Scr**. Give the following command at the command line:

```
scr -p -r lockopt
```

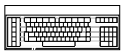
-p	-	placement option
-r	-	routing option

A “lockopt.ap” file is created which can be viewed with Graal.

Describing the Pads and Core using the Procedural Design Language

The procedural description language is actually a set of **C** functions that allows you to describe circuit objects like pads and the core and their connectivity.

Create and edit and save into the file “lockchip.c” the following:



```
#include<genlib.h>
main()
{
  DEF_LOFIG("lockchip");

  LOCON("VDD", 'I',"VDD");
  LOCON("VSS", 'I',"VSS");
  LOCON("VSSE",'I',"VSSE");
  LOCON("VDDE",'I',"VDDE");
  LOCON("CK",'I',"CK");
  LOCON("RESET",'I',"RESET");
  LOCON("TRY",'I',"TRY");
  LOCON("KEYPRESS",'I',"KEYPRESS");
  LOCON("SET",'I',"SET");
  LOCON("OPENLOCK",'O',"OPENLOCK");
  LOCON("keynum[0:3]",'I',"keynum[0:3]");

/* Instance of pads of the chip. The instance_name of the pads is the
one that is to be */
/* given to the Ring tool for it to understand the names for pad
placement on the chip */
/* On passing this file through Genlib, a .vst file is generated. This
file has the output*/
/* input and IO pins as specified in the above list. Asimut understands
only these as the */
/* pins for simulation */

  LOINS("pvsse_sp","vss","cki","vdde","vdd","vsse","vss",0);
  LOINS("pvdde_sp","vdd","cki","vdde","vdd","vsse","vss",0);
  LOINS("pvddi_sp","ivdd","cki","vdde","vdd","vsse","vss",0);
  LOINS("pvssi_sp","ivss","cki","vdde","vdd","vsse","vss",0);

  LOINS("pck_sp","RINGCLK","CK","CKI","VDDE","VDD","VSSE","VSS",0);
  LOINS("pvsseck_sp","CLOCK","PCK","CKI","VDDE","VDD","VSSE","VSS",0);


LOINS("pi_sp","RESET","RESET","PRESET","cki","VDDE","VDD","VSSE","VSS",
0);
  LOINS("pi_sp","TRY","TRY","PTRY","cki","VDDE","VDD","VSSE","VSS",0);

LOINS("pi_sp","KEYPRESS","KEYPRESS","PKEYPRESS","cki","VDDE","VDD","VSS
E","VSS",0);
  LOINS("pi_sp","SET","SET","PSET","cki","VDDE","VDD","VSSE","VSS",0);

LOINS("pi_sp","KEYNUM0","KEYNUM[0]","PKEYNUM[0]","cki","VDDE","VDD","VS
SE","VSS",0);

LOINS("pi_sp","KEYNUM1","KEYNUM[1]","PKEYNUM[1]","cki","VDDE","VDD","VS
SE","VSS",0);
```

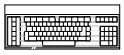
```
LOINS("pi_sp","KEYNUM2","KEYNUM[2]","PKEYNUM[2]","cki","VDDE","VDD","VS
SE","VSS",0);

LOINS("pi_sp","KEYNUM3","KEYNUM[3]","PKEYNUM[3]","cki","VDDE","VDD","VS
SE","VSS",0);


LOINS("po_sp","OPENLOCK","POPENLOCK","OPENLOCK","cki","VDDE","VDD","VSS
E","VSS",0);

/* The first name is the name of the .vst file that is to be used for
reference */
/* The second name is the instance_name and can be anything */
/* the names that follow can be anything  except that they should be in
the same */
/* order as in the .vst file. Bus signals should have the same
dimensions. Names given */
/* should be the inputs or outputs of other instances which means that
the block is */
/* physically connected to other blocks in the description and is not
left hanging */

LOINS("lockopt","lock","vdd","vss","pck","preset",
        "ptry","pkeypress","pset","popenlock","pkeynum[3:0]",0);

SAVE_LOFIG();

exit(0);
}
```

Give the command at the command line:

```
genlib lockchip
```

This creates a “lockchip.vst” structural description file with pads.

Simulating the Structural Description

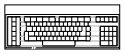


You can now simulate this structural description with the test vector file that you developed for “lock.vbe”. Simulate the structural description and confirm the functioning of the structural description.

Placing and routing the pads

Now the chip’s pads and the core has to be connected together physically in a layout. This is done by using **Ring**.

Edit and save the following in the file “lockchip.rin”:



```
# File used by RING tool
# Placement of pads for the lock chip
north  (clock vdd reset)
east (set ivdd try keypress)
south (openlock vss keynum0)
west (keynum1 ivss keynum2 ringclk keynum3)
```

This file describes the relative position of the pads on the four sides of the chip.

Give the command at the command at the command line:

```
ring lockchip lockchip
```

A “lockchip.ap” file is created that can be examined by using **Graal**.

Examine the layout using **Graal**.

Static Timing Analysis

The “lockchip.ap” contains the layout information. However we do not know if the physical description produced reflect the desired behaviour. Therefore to check the layout we use two tools, **Lynx** and **Tas**.
Lynx is a netlist extractor. It extracts a netlist representation of the circuit from the layout. The file created by **Lynx** will be the input file for **Tas**.
Tas is a switch level timing analyser for CMOS circuits.
Give the following command at the command line:

```
setenv MBK_OUT_LO  al
```

This tells that the output file should be in the “.al” (Alliance) format.

```
lynx -v -t lockchip lockchip
```

-v	-	verbose
-t	-	build the netlist to the transistor level.
first lockchip	-	take the “lockchip.ap” layout file as input.
second lockchip	-	generate the “lockchip.al” netlist file.

Give the following command at the command line:

```
setenv MBK_IN_LO  al
```

This tells that the input file for **Tas** must be in the “.al” (Alliance) format.

```
tas  -tec=/alliance/archi/Linux_elf/etc/prol10.elp lockchip
```

-tec - selects the technology file prol10.elp.

Layout Extraction and Netlist Comparison

The “lockchip.ap” contains the layout information. However we do not know if the physical description produced reflect the behavioural description. Therefore to check the layout we use two tools, **Lynx** and **Lvx**.

Lynx is a netlist extractor. It extracts a netlist representation of the circuit from the layout.

For this you have to set some environmental variables. You have to specify the format in which the extracted netlist is generated.

Give the following command at the command line:

```
setenv MBK_OUT_LO  al
```

This tells that the output file should be in the ".al" (Alliance) format.

Give the command at the command line:

```
lynx -v -f lockchip lockchip
```

-v	-	verbose
-f	-	asks **Lynx** to generate the netlist from the Standard-cells level.
first lockchip	-	Take the "lockchip.ap" layout file as input.
second lockchip	-	Generate the "lockchip.al" netlist file.

Lvx is a netlist comparison software that compares two netlists. Along with the comparison it re-orders the interface terminals to produce a consistent netlist interface.

Give the command at the command line

```
lvx vst al lockchip lockchip –f –o
```

-f	-	build the netlist to the standard cell level.
vst	-	take the first file in .vst format.
al	-	take the second file in .al format.
first lockchip	-	"lockchip.vst" file.
second lockchip	-	"lockchip.al" file.

The comparison should not produce any errors. If errors are produced by the program, then there is some problem with the layout. The router has done something funny and corrective action is to be taken at the layout level by studying the error messages.

The **Lvx** tool has also re-ordered and built the netlist in the ".al" to the standard cell format. This file can be simulated using **Asimut**.

Simulating the Extracted netlist file

The netlist file "lockchip.al" can be simulated using **Asimut** and the test vector file that has been created to test "lock.vbe".

Give the following command at the command line:

```
setenv MBK_IN_LO al
```

to set the input file format for **Asimut** for the ".al" format, before doing the simulation. Any error during simulation means that you will have to retrace your steps back to find out the source of the error.

Functional Abstraction

Yagle is a program that extracts from a standard cell level, the behaviour of the circuit. Essentially a VHDL file is created from a standard cell connectivity description! This VHDL file can be simulated in turn to verify the function of the chip.

Give the command at the command line:

```
yagle -v lockchip
```

-v	-	vectorized
lockchip	-	Takes the “lockchip.al” as input.

The extracted VHDL description is put in the file “lockchip.vbe”.
Simulate the extracted behavioural description to verify the extracted behavioural description.

Alliance has a program that compares the extracted behavioural file with the original behavioural file to formally prove the functional congruence of the described and the extracted circuit. However this step requires that the registers in the two behavioural descriptions have the same names. This can be done automatically by **Yagle** by giving it a list of registers to be renamed in an *information file* “lockchip.inf”.

Edit and save a file “lockchip.inf” with the following:

```
rename
lock.mem6_3.dff_s : mem6_3 ;
lock.mem5_0.dff_s : mem5_0 ;
lock.mem5_3.dff_s : mem5_3 ;
lock.mem7_1.dff_s : mem7_1 ;
lock.num_1.dff_s : num_1 ;
lock.current_state_1.dff_s : current_state_1 ;
lock.mem0_2.dff_s : mem0_2 ;
lock.mem1_1.dff_s : mem1_1 ;
lock.mem5_2.dff_s : mem5_2 ;
lock.mem2_1.dff_s : mem2_1 ;
lock.mem7_3.dff_s : mem7_3 ;
lock.counter_1.dff_s : counter_1 ;
lock.num_0.dff_s : num_0 ;
lock.current_state_2.dff_s : current_state_2 ;
lock.mem6_0.dff_s : mem6_0 ;
lock.mem6_1.dff_s : mem6_1 ;
lock.mem7_2.dff_s : mem7_2 ;
lock.mem3_1.dff_s : mem3_1 ;
lock.counter_2.dff_s : counter_2 ;
lock.mem6_2.dff_s : mem6_2 ;
lock.mem5_1.dff_s : mem5_1 ;
lock.mem7_0.dff_s : mem7_0 ;
lock.num_2.dff_s : num_2 ;
lock.mem3_2.dff_s : mem3_2 ;
lock.current_state_0.dff_s : current_state_0 ;
lock.mem1_2.dff_s : mem1_2 ;
lock.mem4_1.dff_s : mem4_1 ;
lock.mem4_2.dff_s : mem4_2 ;
lock.current_state_3.dff_s : current_state_3 ;
lock.mem4_0.dff_s : mem4_0 ;
lock.mem1_0.dff_s : mem1_0 ;
lock.mem3_3.dff_s : mem3_3 ;
lock.mem2_0.dff_s : mem2_0 ;
lock.counter_0.dff_s : counter_0 ;
lock.mem0_0.dff_s : mem0_0 ;
lock.mem3_0.dff_s : mem3_0 ;
lock.mem1_3.dff_s : mem1_3 ;
lock.mem2_2.dff_s : mem2_2 ;
lock.mem4_3.dff_s : mem4_3 ;
lock.mem2_3.dff_s : mem2_3 ;
lock.mem0_1.dff_s : mem0_1 ;
lock.mem0_3.dff_s : mem0_3 ;
end
```

Give the command:

```
yagle -i -v lockchip
```

-i - asks **yagle** to read the “lockchip.inf” file and rename the registers in the “lockchip.vbe” file as given in the list.

Give the command:

```
proof -p -d lockchip lock
```

-p - negates the input and output signal expressions of the registers.
-d - display errors to screen.

If no errors are reported, then the two behavioural descriptions concur. It is possible to have errors due to the missing signals vdde and vsse in the lock.vbe file; If this is the case just add these signal in the port declaration of lock.vbe and run again **proof**.

Real Technology Conversion

Up till now all the files describe the circuit only as symbolic cells. The foundry requires the layout of the chip, described in terms of rectangles and layers in the **gds** or the **cif** format. This can be done in Alliance, by using **S2r**.

```
setenv RDS_TECHNO_NAME /alliance/archi/Linux_elf/etc/prol10_7.rds
setenv RDS_OUT cif
setenv RDS_IN cif
```

This chooses the 1.0µm CMOS process, chooses the output form of the chip in **cif** format and, replaces the symbolic pads with their real equivalent.
Give the command:

```
s2r -cv lockchip lockchip
```

-c - deletes connectors at the highest hierarchy. (Use man to see full description)
-v - verbose mode on

first lockchip - “lockchip.ap” file as input
second lockchip - “lockchip.cif” file as output.

This completes the design of the lock chip.