united nations
educational, scientific
and cultural
organization

international atomic
energy agency

the
abdus salam
international centre for theoretical physics

301/1352

MCIROPROCESSOR LABORATORY SEVENTH COURSE ON BASIC VLSI DESIGN TECHNIQUES

29 October - 23 November 2001

EXERCISES BOOK

This exercises book is intended only for distribution to participants.

strada costiera, 11 - 34014 trieste italy - tel. +39 0402240111 fax +39 040224163 - sci_info@ictp.trieste.it - www.ictp.trieste.it

SEVENTH COURSE
ON
BASIC VLSI DESIGN TECHNIQUES

MICROPROCESSOR LABORATORY
ICTP-UNESCO

29 October- 23 November, 2001

Trieste, Italy

Exercise 1

Exploring a Chip Layout

Exercise Description

In this exercise you will edit a full chip layout with a specific Alliance tool called **Graal**. This tool allows you to explore the layout of a chip seeing most of the standard layers needed by the designer to implement a CMOS chip.

In this exercise you will

- Edit a CMOS Chip layout.
- Identify the chip core and the ring of Pads.
- Explore the hierarchical structure of the layout.
- Explore the electrical connectivity of different layout elements.
- Edit a standard cell layout.
- Recognise simple CMOS logic gates by inspecting its layout.
- Get used to invoke man pages of standard cells.

(**********)

Exercise guide

Legend

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

Explanation of a topic

A VLSI chip is supposed to process information in a wide sense. To do that a large number of transistors are built in and interconnected in order to handle the binary information. A quantum of information is represented by a bit which can assume two values only. Each of these values is physically represented by an electric signal (*voltage*), which assumes two values: *Low* or *High*. The logic-electrical elementary structures are the *gates*. The physical implementation of a *logic gate* is a circuit that "*maps*" a Boolean operation. Some of the main characteristics of these gates are:

- Small area
- Easy interconnection
- High speed
- Low power consumption

These gates are designed to exchange information among them inside the chip, and then they are not prepared to exchange information with the external world in an efficient way. A typical parasitic capacitance of an external chip connection is about 2 or 3 orders of magnitude larger than a typical internal one. Sometimes there are input signals that are spread inside the core of the chip, and due to this, the large associated parasitic capacitance could degrade the signal at some critical point compromising the correct behavior of the chip. As an example of this consider potential critical signals like "*set*", "*reset*" or "*clock*". A correct handling of input output signals at chip level is mandatory in order to ensure a correct functioning chip.

Alliance has a library (*padlib*) of specific cells to interface the core of the chip with the external world. These cells are *Pad Drivers*. There are different types of them according to its function like:

- *Input*
- *output*
- *input-output*
- *power-supplies*
- *clock*.
- ...

A common characteristic to all pad-drivers is that they have an uncover square piece of metal (the *pad*) for bonding. The area of this square metal is typically ~100us x ~100us.

The output drivers must be able to force fast electrical transitions between logic levels loading big parasitic capacitance. During these transitions a large peak of current is required from the power supplies and consequently the voltage of these power supply lines can vary due to a non-zero resistance connection to the voltage source. Then the voltage of the power supply lines for the drivers become *dirty* and it is not advisable to use it to feed the core where many delicate electrical transitions take effect. Hence for the core there are separate power supplies.

In alliance the distribution of power is facilitated by mean of rings around the core. The same strategy for one privileged signal (typically *clock*, although could be any other). For this signal there is a *pad driver* (*pck_sp*) which takes the external signal to drive the ring and another *pad driver* (*pvsseck_sp*) that takes the signal from the ring and drives it to the core.

There are five concentric metallic rings: *vdde* & *vsse* for pad-drivers, *vddi* & *vssi* for the core, and the *clock* ring. All pad-drivers contain five piece of metal in order to generate these rings when they are disposed all around the core. In a structural description this means that at least five ports must be specified for any Pad-driver.

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

```
mkdir chip-layout
```

Change into this directory:

```
cd chip-layout
```

Copy in this directory the file *chip.ap* (The path of this file will be indicated by the tutors in the Laboratory).

```
cp /<path>/chip.ap .
```

Now we can edit the layout of the chip with **Graal**, the Alliance hierarchical symbolic layout editor. Give the following command at the command line

```
graal &
```

Click on the *file* section of the menu and then click *open* to load *chip.ap*. What is shown is the top-level view of the symbolic layout of a four-bit microprocessor.
Now you can play with the options and some of the tools from the menu. Here is a menu list with some brief description of some of the tool you will need in this exercise.

File: **open:** to load an ".ap" file.

quit: to end the graal session.

Tools: **peek (unpeek):** to show the layers.

Flat (unflat): To eliminate one hierarchical level.

equi (unequi): to render evident the electrical connections.

real flat: to eliminate the hierarchical structure.

View: **fit**: to fit the layout in the windows.

zoom (in, out)- mooz: To zoom in out.

arrows: to move according to the arrow direction.

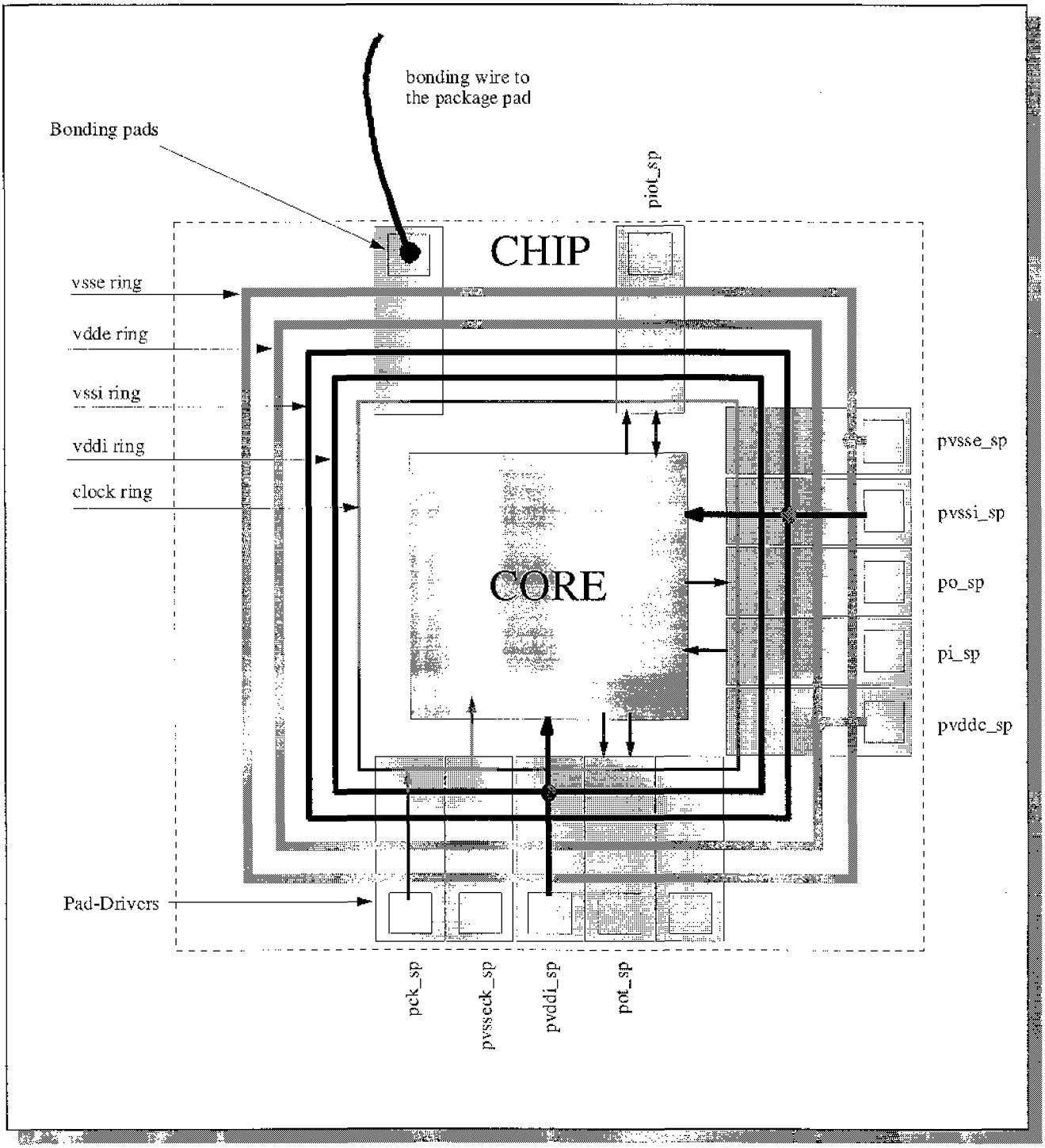
layers: to show/hide specified layers.

Windows: **Identify:** to identify cells, model, instance name, layers, etc.

For more information about **Graal** menu type

```
man graal
```

By inspecting the layout identify the core and the pads. Compare the pads trying to recognize its functionality by mean of its connections with the metallic rings.
To do this see figure 1 and use the functions: *flat*, *zoom-in*, *equi*, *layers*, etc.



Now you can load standard cells from the Alliance *Standard Cell Library*. Copy some physical description files (<filename>.ap) from *"/alliance/share/cells/sclib"* in to your working directory. For example we can copy a simple *inverter* gate:

```
cp /alliance/share/cells/sclib/n1_y.ap .
```

Copy also from the previous location the following files:

`no2_y.ap`	two input *nor* gate
`na2_y.ap`	two input *nand* gate
`a2_y.ap`	two input *and* gate
`o2_y.ap`	two input *or* gate
`xr2_y.ap`	*exclusive-or* gate
`msx_y.ap`	*D-flip-flop* with complementary outputs.

You can have more information about the previous cells by invoking the corresponding man pages. Just type for example

```
man no2_y
```

Once you have edited the cell with **Graal**, use the "*layer*" tool to inspect the different layers: *diffusions, polysilicon gates, metals, vias*, etc. Identify the P-MOS and N-MOS transistors and how they are connected in order to perform its logic function. Try several standard cells till becoming familiar with them, and **don't forget** to make extensive use of the Alliance man pages.

SEVENTH COURSE
ON
BASIC VLSI DESIGN TECHNIQUES

MICROPROCESSOR LABORATORY
ICTP-UNESCO

29 October- 23 November, 2001

Trieste, Italy

Exercise 2

Design of a 2 to 4 Decoder chip

Introduction

This first design example is a two to four line decoder. It is presented to familiarize you with the Alliance design environment and facilities. In doing this example you will:

- Describe the decoder behavior using **VHDL** language.
- Check syntax errors by passing it through **Asimut**.
- Write the test pattern file to test the decoder.
- Simulate the behavioral description using the pattern file by using **Asimut**.
- Synthesize the logic and structural descriptions using **Bop** and **Scmap**.
- Optimize the netlist using **Glop**.
- Use the standard cell router called **Scr** to place and route the core.
- Use **Graal** to see or edit the core layout.
- Add the necessary pads for the chip and compile using **Genlib**.
- Place the pads and generate the layout file using **Ring**.
- Analyze timing delays using **Tas**.
- Extract back the behavioral description from the layout using **Lynx**, **Lvx** and **Yagle**.
- Use **Proof** to compare the original behavioral description and the extracted one to complete formal verification.
- Use **S2r** to generate the "cif " file ready for the foundry from the symbolic layout.

The design example consists of two phases. The first phase is to describe the behavior of the chip as is seen at the pins of the chip. The second phase is to describe the functions of the core of the chip, and then connect it to the pads.

In the first phase you will:
- Describe the decoder's behavior using VHDL (dec2to4.vbe).
- Write a test pattern file (dec2to4.pat).
- Simulate the behavioral description using the pattern file by using **Asimut**.

In the second phase you will:
- Describe the behavior of the core in VHDL as is seen inside the chip by the pads (dec0core.vbe).
- Synthesize the logic and structural descriptions using **Bop** and **Scmap** (dec0core.vst).
- Use **Glop** to optimize for critical path and fanout (dec0opt.vst).
- Use the standard cell router called **Scr** to place and route the core (dec0opt.ap).
- Add the necessary pads for the chip and compile using **Genlib** (dec0chip.vst).
- Use **Asimut** to simulate the 'dec0chip.vst' file using the pattern file 'dec2to4.pat'.
- Place the pads and generate the layout of the chip with pads using **Ring** (dec0chip.ap).
- Use **Lynx** to extract the netlist from the layout file 'dec0chip.ap' (dec0chip.al).
- Use **Tas** to obtain static timing information.
- Use **Lvx** to compare the extracted circuit 'dec0chip.al' and the original 'dec0chip.vst' file created by **Genlib**.
- Use **Yagle** to extract the behavior, 'dec0chip.vbe' from the 'dec0chip.al' netlist file.
- Use **Proof** to compare the extracted behavior file, 'dec0chip.vbe' and the behavioral file created in the first phase, 'dec2to4.vbe'.
- Use **S2r** to generate the "cif " file ready for the foundry from the symbolic layout.

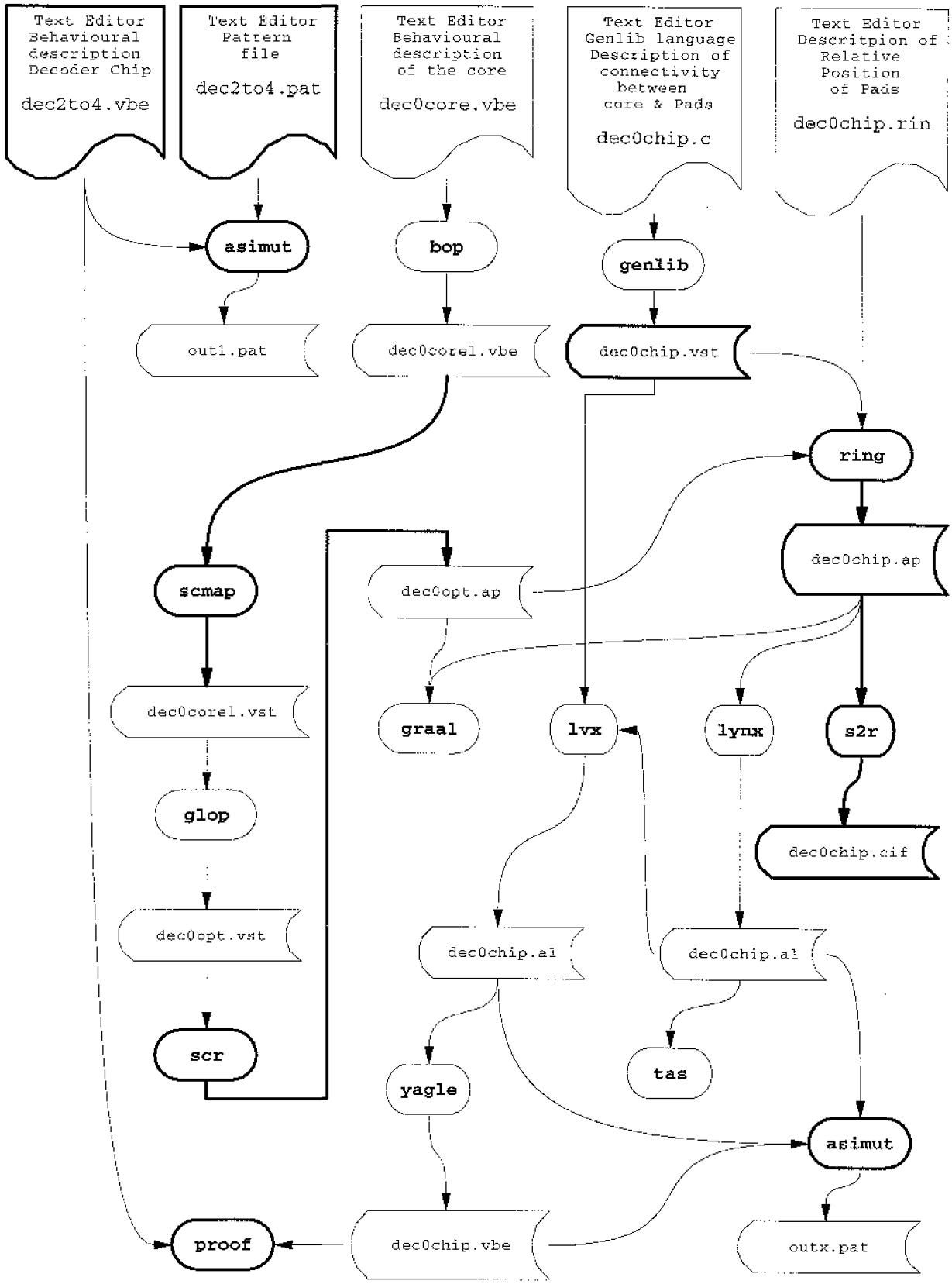


Fig 1. Design Flow for the Decoder Chip

Decoder Chip

The example is a 2 to 4 decoder. The decoder's function is summarized by the truth table shown in Table 1. below.

a	b	enable	y0	y1	y2	y3
X	X	0	0	0	0	0
0	0	1	1	0	0	0
0	1	1	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	1

Table 1. Truth Table for the 2 to 4 decoder.

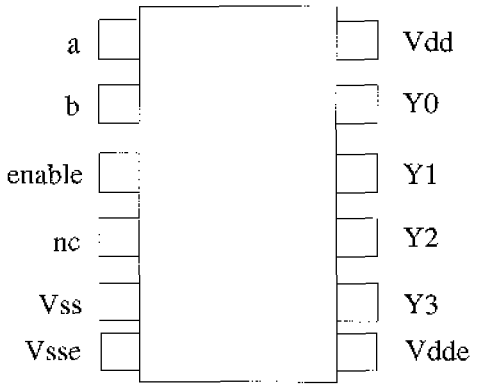


Fig 2. Possible pinout of the decoder chip.

Legend

Give the command that appear 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.

The text or picture following this symbol appears on the monitor.

Creating the design

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

```
mkdir dec2to4
```

Change into this directory:

```
cd dec2to4
```

Before starting the design you will have to set the environmental variables as shown below so that you will not run into problems later.
The MBK_CATA_LIB environmental variable tells the Alliance software, the paths through which it has to search for the cells that are instantiated in the design. More details are available in the man pages (man MBK_CATA_LIB).
The MBK_IN_LO environmental variable sets the logical input format of the mbk database. Details of the valid formats that can be used are available in the man pages (man MBK_IN_LO).
The MBK_OUT_LO environmental variable sets the logical output format of the mbk data structure. Details of the valid formats that can be used are available in the man pages (man MBK_OUT_LO).
The MBK_IN_PH environmental variable sets the physical input format of the mbk data structure. Details of the valid formats that can be used are available in the man pages (man MBK_IN_PH).
The MBK_OUT_PH environmental variable sets the physical output format of the mbk data structure. Details of the valid formats that can be use are available in the man pages (man MBK_OUT_PH).

```
setenv MBK_CATA_LIB .:/alliance/archi/Linux_elf/cells/sclib:
                      /alliance/archi/Linux_elf/cells/padlib
setenv MBK_IN_LO vst
setenv MBK_OUT_LO vst
setenv MBK_IN_PH ap
setenv MBK_OUT_PH ap
setenv MBK_WORK_LIB .
```

Many of the environmental variables needed during the Alliance practical exercises are set automatically when you start your computer working session. A specific ".login" file does it for you. Nevertheless we insist in setting all necessary environmental variables each time it is required by the tools in order to have a better control on the design flow.

Describing the chip behavior

The behavioral description is done using a VHDL subset. Only concurrent statements are supported. No sequential statements are allowed. More details are available in the man pages (man vbe). We begin our design by describing the behavior of the signals at the decoder chip pins, as is described by the truth table above in Table 1.
Create with the "pico" editor (or any other text editor you prefer) a file called dec2to4.vbe, enter the following data, and save the file.

```
-- Port declaration of the simple 2 to 4 decoder

ENTITY dec2to4 IS

PORT ( A, B, enable, nc, vdd, vss, vdde, vsse : in BIT ;
        Y : out bit_vector (0 to 3));

end dec2to4;

ARCHITECTURE dEc2to4_data_flow OF dec2to4 IS

signal A_bar, B_bar : bit ;
signal a1, a2, a3, a4 : bit;

 begin

     A_bar <= not A ;
     B_bar <= not B ;

     a1 <= A_bar and B_bar and enable ;
     a2 <= A_bar and B and enable ;
     a3 <= A and B_bar and enable ;
     a4 <= A and B and enable ;

     Y(0) <= a1;
     Y(1) <= a2;
     Y(2) <= a3;
     Y(3) <= a4;

end DeC2to4_data_flow;
```

Asimut is a logical simulation tool for hardware descriptions. It compiles and loads a **VHDL** description, which may be behavioral or structural. Only the VHDL subset discussed above is supported. Information on Asimut's command line parameters, options, environmental variables required are available in the man pages (man asimut).

Any typographical or syntax error in a behavioral description can be found when the file is passed through **Asimut**.

Give the following command at the command line:

```
asimut -b -c dec2to4
```

-b - behavioral option
-c - compile

The following is typically displayed.

```
[cicuttin@mlab-42]$ asimut -b -c dec2to4

                A SIMUlation Tool

          Alliance CAD System 3.2b,      asimut v2.01
          Copyright (c) 1991-1999,       ASIM/LIP6/UPMC
          E-mail support: alliance-support@asim.lip6.fr

     Paris, France, Europe, Earth, Solar system, Milky Way, ...
Initializing ...
Searching `dec2to4` ...
BEH : Compiling `dec2to4.vbe` (Behaviour) ...
Making GEX ...

[cicuttin@mlab-42]$
```

Creating the test pattern for simulation

If the above step functions with out giving syntax errors then the behavioral description is ready for simulation.

A file with the test patterns in the **pat** format is required for the simulation. The **pat** format file has a declaration part and a description part of the signals. The declaration part consists of a list of inputs, outputs, internal signals and registers of the design. Inputs are forced to a particular value while the outputs are observed during a simulation.

Create a test pattern file called dec2to4.pat, editing the following as is:

```
-- description generated by Pat driver v104

--                              date : Sep 12 12:55:24 1999

--                        sequence : dec2to4

-- input / output list :
in      vdd B;
in      vss B;;;;;;
in      a B;;
in      b B;;;
in      enable B;;;
out     y (3 downto 0) B;;;

begin

-- Pattern description :

--              vv   a b e    y
--              ds        n
--              ds        a
--                        b
--                        l
--                        e


              : 10    0 0  0  ?0000 ;
              : 10    0 1  0  ?0000 ;
              : 10    1 0  0  ?0000 ;
              : 10    1 1  0  ?0000 ;
              : 10    1 0  0  ?0000 ;
               : 10   0 0  1  ?0001  ;
              : 10    0 1  1  ?0010 ;
              : 10    1 0  1  ?0100 ;
              : 10    1 1  1  ?1000 ;
              : 10    1 1  0  ?0000 ;


end;
```

In the previous file, lines starting with "--" are comments. In next exercises we will see a more powerful procedure to generate larger pattern files.

Simulating the behavioral description

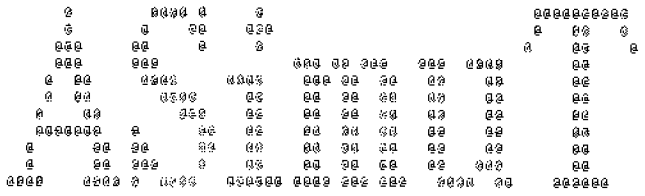
Give the following command at the prompt to start simulating

```
asimut -b dec2to4 dec2to4 out1
```

-b	-	chooses the behavioral simulation option
first dec2to4	-	dec2to4.vbe
second dec2to4	-	dec2to4.pat
out1	-	simulation result in out1.pat

The following screen is typically displayed.

[cicuttin@mlab-42]$ asimut -b dec2to4 dec2to4 out1



A SIMUlation Tool

Alliance CAD System 3.2b, asimut v2.01
Copyright (c) 1991-1999, ASIM/LIP6/UPMC
E-mail support: alliance-support@asim.lip6.fr

Paris, France, Europe, Earth, Solar system, Milky Way, ...
Initializing ...
Searching 'dec2to4' ...
BEH : Compiling 'dec2to4.vbe' (Behaviour) ...
Making GEX ...

Searching pattern file : 'dec2to4' ...
Restoring ...

Linking ...
###----- processing pattern 0 -----###
###----- processing pattern 1 -----###
###----- processing pattern 2 -----###
###----- processing pattern 3 -----###
###----- processing pattern 4 -----###
###----- processing pattern 5 -----###
###----- processing pattern 6 -----###
###----- processing pattern 7 -----###
###----- processing pattern 8 -----###
###----- processing pattern 9 -----###
[cicuttin@mlab-42]$

You can see the simulation result in the file out1.pat. To see this file give the following command at the command prompt.

```
more out1.pat
```

Describing the core of the chip

The behavioral file "dec2to4.vbe" is the description of the decoder as is seen at the pins of the chip. We have not thought about the pads that drive the pins. When the chip is described physically in Alliance, it consists if two separate parts that are brought together, the core and the pads. In Alliance, the core and the pads are brought together in a **C** description file. This file when treated with **Genlib**, produces the structural description of the chip with the pads. In practice, the core can be synthesized automatically from a behavioral description, whereas the pads should be placed physically, one by one in the **C** file. Placing the pads require the structural knowledge of the pads. One of the type of pads that is used in this example is the po_sp output pad.

Give the following command at the command line to see a description of this pad.

```
man po_sp
```

Behavioral Description of the Core

We can now describe the core in such a way, we get at the outputs of the chip.

Copy the file "dec2to4.vbe" to the file "dec0core.vbe" and edit it as shown below. Read the comments carefully.

```
-- Port declaration of the simple 2 to 4 decoder

ENTITY dec2to4 IS

PORT ( A, B, enable, vdd, vss : in BIT ; -- <--nc, Vdde and Vsse have been
removed
        Y : out bit_vector (0 to 3));     --   <--since the core does not need them


end dec2to4;

ARCHITECTURE dEc2to4_data_flow OF dec2to4 IS

signal A_bar, B_bar : bit ;
signal a1, a2, a3, a4 : bit;

 begin

     A_bar <= not A ;
     B_bar <= not B ;

     a1 <= A_bar and B_bar and enable ;
     a2 <= A_bar and B and enable ;
     a3 <= A and B_bar and enable ;
     a4 <= A and B and enable ;
```

```
    Y(0) <= a1;
    Y(1) <= a2;
    Y(2) <= a3;
    Y(3) <= a4;

end DeC2to4_data_flow;
```

Synthesizing the logic and the structure

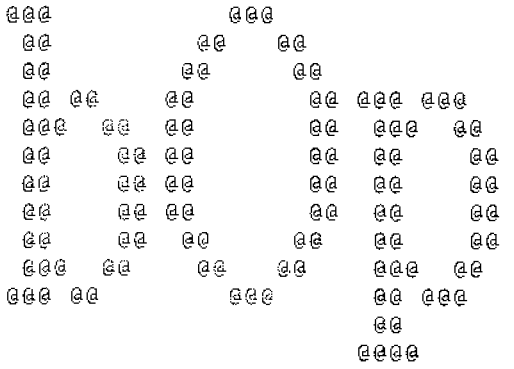
The file dec0core.vbe describes the behavior of the core in VHDL language, that is at the highest available level. From this file it is possible to start a synthesis procedure towards lower levels of description. The first step is to generate an equivalent file but in terms of Boolean expressions, performing at the same time some optimization regarding number of Boolean operator, number of intermediate signals, etc. To do this we use **Bop**, a Boolean optimizer, which takes the file dec0core.vbe to create the new dec0corel.vbe, that is still a behavioral description. There are mainly two kind of available optimizations: global and local, and can be choose with an appropriate option. Detailed information on **Bop** is available in the man pages (man bop).

```
bop -o dec0core dec0corel
```

-o	-	option for global optimization
dec0core	-	dec0core.vbe (input file)
dec0corel	-	dec0corel.vbe (output file)

The following is typically displayed:

```
[cicuttin@mlab-42]$ bop -o dec0core dec0corel
```



```
    Boolean OPtimization

Alliance CAD System 3.2b, bop 4.20 [1997/10/09]
Copyright (c) 1990-1999,      ASIM/LIP6/UPMC
E-mail support:   alliance-support@asim.lip6.fr

================================ Environment ================================
MBK_WORK_LIB      = .
MBK_CATA_LIB      = .:/alliance/archi/Linux_elf/cells/sclib:/alliance/archi/Linux_elf/cells/padlib
============================ Files, Options and Parameters ============================
VHDL file         = dec0core.vbe
```

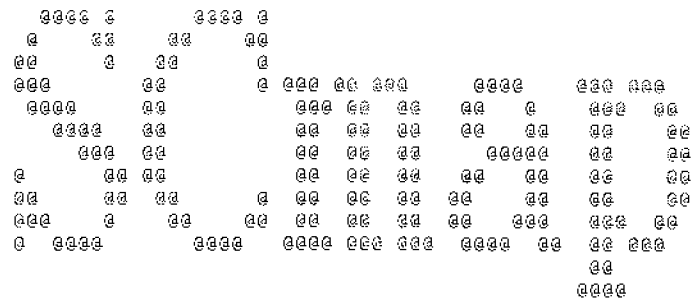
```
output file       = dec0corel.vbe
Parameter file    = default.lax
Mode              = Global optimization
Optimization mode  = 50% area - 50% delay optimization
Optimization level  = 2
================================================================================

Compiling 'dec0core'...

Running abl ordonnancer on 'dec2to4'
....

Running Abl2Bdd on 'dec2to4'

---> Final number of nodes = 13(9)

Running Global Optimizer on 'dec2to4'

============================== INITIAL COST ==================================
Total number of literals   = 12
Number of reduced literals  = 18
Number of latches          = 0
Maximum logical depth      = 2
Maximum delay              = 1.000
================================================================================
.
...
============================== FINAL COST ====================================
Total number of literals   = 12
Number of reduced literals  = 12
Number of latches          = 0
Maximum logical depth      = 2
Maximum delay              = 1.000
================================================================================
BEH : Saving 'dec0corel' in a vhdl file (vbe)


[cicuttin@mlab-42]$
```

The second step is the synthesis of the structural view of the circuit. The structural description consists in a set of elementary interconnected blocks. At this level must be describe which blocks are used and how they are connected each other (gate network). The behavior of each block is supposed to be known. In our case since the circuit is very simple, we map with the Alliance standard cell library. This is performed by **Scmap** which can accomplish further optimization. Detailed information on **Scmap** is available in the man pages (man scmap). To pass from the optimized behavioral dec0corel.vbe to the structural dec0core.vst, give the following command at the command prompt.

```
scmap dec0corel dec0core
```

The following is typically displayed:

```
[cicuttin@mlab-42]$ scmap dec0corel dec0core
```



Mapping Standard Cells

Alliance CAD System 3.2b, scmap 4.20 [1997/10/09]
Copyright (c) 1990-1999, ASIM/LIP6/UPMC
E-mail support: alliance-support@asim.lip6.fr

```
================================== Environment ==================================
MBK_WORK_LIB      = .
MBK_CATA_LIB      = .:/alliance/archi/Linux_elf/cells/sclib:/alliance/archi/Linux_elf/cells/padlib
MBK_TARGET_LIB    = /alliance/archi/Linux_elf/cells/sclib
MBK_IN_LO         = vst
MBK_OUT_LO        = vst
============================ Files, Options and Parameters =======================
VHDL file         = dec0corel.vbe
output file       = dec0core.vst
Parameter file    = default.lax
Mode              = Mapping standard cell
Optimization mode = 50% area - 50% delay optimization
Optimization level = 2
=================================================================================

Compiling 'dec0corel' ...
Running Standard Cell Mapping
================================ INITIAL COST ===================================
Total number of literals  = 12
Number of reduced literals = 12
Number of latches         = 0
Maximum logical depth     = 2
Maximum delay             = 1.000
```

```
================================================================================
Compiling library '/alliance/archi/Linux_elf/cells/sclib'
Generating Expert System ...
Cell 'cmx2_y' Unused
Cell 'cry_y' Unused
Cell 'sum_y' Unused
Cell 'tie_y' Unused
162 rules generated
================================ FINAL COST ================================
Number of cells used  = 3
Number of gates used  = 7
Number of inverters   = 3
Number of grids       = 6552
Depth max. (gates)        = 2
      (eq. neg. gates) = 2
================================================================================

MBK Driving './dec0core.vst'...

[cicuttin@mlab-42]$
```

The structural description file generated by **Scmap** can be examined by giving the Unix "more" command:

```
more dec0core.vst
```

Using Asimut to simulate the structural description

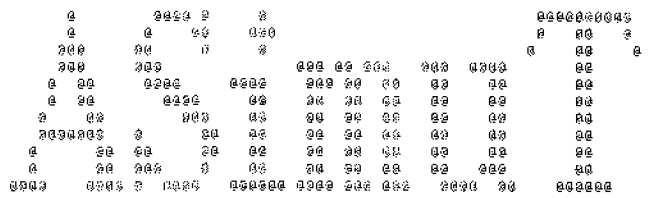
You can do the simulation with the structural description with the same pattern files that are used for the behavioral description.
To do the simulation on the structural description, give the following command at the command prompt.

```
asimut dec0core dec2to4 r3
```

no option	-	takes the structural description by default
dec0core	-	dec2to4core1.vst
dec2to4	-	dec2to4.pat
r3	-	result of simulation in r3.pat

The following screen is typically displayed

```
[cicuttin@mlab-42]$ asimut dec2to4corel dec2to4 r3
```



A SIMUlation Tool

```
Alliance CAD System 3.2b,      asimut v2.01
Copyright (c) 1991-1999,       ASIM/LIP6/UPMC
E-mail support: alliance-support@asim.lip6.fr

Paris, France, Europe, Earth, Solar system, Milky Way, ...
Initializing ...
Searching 'dec0core' ...
Compiling 'dec0core' (Structural) ...

Flattening the root figure ...

Searching 'a3_y' ...
BEH : Compiling 'a3_y.vbe' (Behaviour) ...
Making GEX ...

Searching 'no3_y' ...
BEH : Compiling 'no3_y.vbe' (Behaviour) ...
Making GEX ...

Searching 'n1_y' ...
BEH : Compiling 'n1_y.vbe' (Behaviour) ...
Making GEX ...

Searching pattern file : 'dec2to4' ...
Restoring ...

Linking ...
###----- processing pattern 0 -----###
###----- processing pattern 1 -----###
###----- processing pattern 2 -----###
###----- processing pattern 3 -----###
###----- processing pattern 4 -----###
###----- processing pattern 5 -----###
###----- processing pattern 6 -----###
###----- processing pattern 7 -----###
###----- processing pattern 8 -----###
###----- processing pattern 9 -----###

[cicuttin@mlab-42]$
```

Optimizing for Fanout and Timing

The structural description created above has been created without worrying about the standard cells fanout limits and critical path signals. **Glop** can analyze the structural description and create

a new description by adding buffers to the appropriate nets so as to solve fanout problems and to optimize on signal delays. Detailed information on **Glop** is available in the man pages (man glop). Give the command:

```
glop -g dec0core dec0opt -i -t
```

-g	invokes a timing optimization.
dec0core	dec0core.vst input file
dec0opt	dec0opt.vst output file
-i	gives fanout information about the gate netlist.
-t	gives timing information about the gate netlist.

The following is typically displayed.

```
[cicuttin@mlab-42]$ glop -g dec0core dec0opt -i -t

                  @@@@ @    @@@@         @@@
                 @@     @@       @@      @@    @@
                @@        @      @@     @@      @@
               @@                @@    @@        @@ @@@ @@@
               @@                @@    @@        @@  @@@   @@
               @@       @@@@@    @@    @@        @@  @@     @@
               @@       @ @@     @@    @@        @@  @@     @@
               @@     @   @@     @@    @@        @@  @@     @@
                @@        @@     @@     @@      @@    @@     @@
                 @@     @@       @@      @@    @@     @@@   @@
                   @@@@        @@@@@@       @@@        @@ @@@
                                                       @@
                                                      @@@@

                        Gate Level OPtimizer

        Alliance CAD System 3.2b, glop 4.20 [1997/10/09]
        Copyright (c) 1990-1999,        ASIM/LIP6/UPMC
        E-mail support:    alliance-support@asim.lip6.fr

   ========================== Environnement ==========================
   MBK_WORK_LIB      = .
   MBK_CATA_LIB      = .:/alliance/archi/Linux_elf/cells/sclib:/alliance/archi/Linux_elf/cells/padlib
   ================== Files, Options and Parameters ==================
   Netlist file      = dec0core.vst
   Output file       = dec0opt.vst
   Parameters file   = default.lax
   Mode              = global optimization with timing analysis
   ===========================================================================

Loading dec0core...
Flattening dec0core...
Loading models ... in /alliance/archi/Linux_elf/cells/sclib
  3 models - 7 cells - 6552 pitchs
        Critical path  UP  = 2486 ps
                from external connector enable
                to external connector y 0
[enable]->auxsc3->y_0->[y 0]
        => 2 gates


        Critical path DOWN = 2407 ps
                from external connector enable
```

```
                to external connector y 1
[enable]->auxsc3->y_1->[y 1]
        => 2 gates

Power Gate Optimization
  ==> Model to use for y_3 [a3_y] : a3p_y
 Timing Analysis : Delay UP 2498 (y_0) - Delay DW 2432 (y_1)

         1 repowered gates


        Critical path  UP  = 2498 ps
                from external connector enable
                to external connector y 0
[enable]->auxsc3->y_0->[y 0]
        => 2 gates

        Critical path DOWN = 2432 ps
                from external connector enable
                to external connector y 1
[enable]->auxsc3->y_1->[y 1]
        => 2 gates

Buffer Optimization

         0 inserted buffers


        Critical path  UP  = 2498 ps
                from external connector enable
                to external connector y 0
[enable]->auxsc3->y_0->[y 0]
        => 2 gates

        Critical path DOWN = 2432 ps
                from external connector enable
                to external connector y 1
[enable]->auxsc3->y_1->[y 1]
        => 2 gates

NO_FACTOR = 1201
  3 models - 7 cells - 6552 pitchs

Saving ./dec0opt...
```

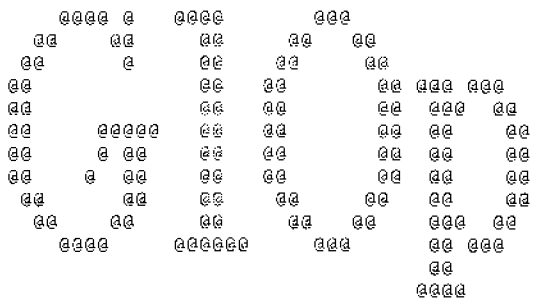
This command takes "dec0core.vst" structural description and generates a "dec0opt.vst" file after buffers have been added to the critical paths. We can run **Glop** again, this time with the option –f to optimize the critical path and the cells interface. Give the command:

```
glop  -f  dec0opt dec0opt
```

-f	invokes the fanout optimization option.
dec0opt	dec0opt.vst structural file to be modified.

The following is typically displayed.

```
[cicuttin@mlab-42]$ glop -f dec0opt dec0opt
```



Gate Level OPtimizer

```
Alliance CAD System 3.2b, glop 4.20 [1997/10/09]
Copyright (c) 1990-1999,        ASIM/LIP6/UPMC
E-mail support:    alliance-support@asim.lip6.fr

========================== Environnement ==========================
MBK_WORK_LIB      = .
MBK_CATA_LIB      = .:/alliance/archi/Linux_elf/cells/sclib:/alliance/archi/Linux_elf/cells/padlib
================== Files, Options and Parameters ==================
Netlist file      = dec0core.vst
Output file       = dec0opt.vst
Parameters file   = default.lax
Mode              = local optimization with max fanout
===========================================================================

Loading dec0core...
Flattening dec0core...
Loading models ... in /alliance/archi/Linux_elf/cells/sclib
  3 models - 7 cells - 6552 pitchs


  ==> BUFFER added after b : netopi16 [p1_y / area = 756]
  ==> BUFFER added after a : netopi17 [p1_y / area = 756]
  4 models - 9 cells - 8064 pitchs (+23.08 %)

Saving ./dec0opt...
Generation of statistics file : ./dec0opt.stat
[cicuttin@mlab-42]$
```

Using the Standard Cell Router Scr

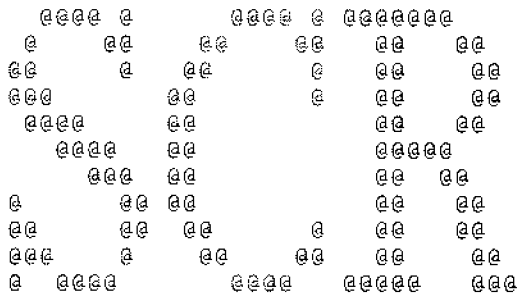
The Standard Cell Router **Scr** is used to place and route the cells of the core. By doing this we synthesize a physical description of the core from the structural view More information on **Scr** is available in the man pages (man scr).. This tool takes the structural file dec0opt.vst and generates the physical file dec0opt.ap. The extension "ap" stands for the Alliance internal physical format, this format is described in man pages (man ap). Type the following command.

```
scr -p -r -l 3 -i 5000  dec0opt
```

-p	-	invokes the automatic placement process.
-r	-	invokes the automatic routing process.
-l	-	allows the designer to set the number of rows.
-i	-	iteration number (to improve the placement quality).
dec0opt	-	input (vst) filename, and output (ap) filename.

The following is typically displayed.

```
[cicuttin@mlab-42]$ scr -p -r -l 3 -i 5000  dec0opt
```



```
  Standard Cell router

Alliance CAD System 3.2b,          scr 5.2
Copyright (c) 1991-1999,       ASIM/LIP6/UPMC
E-mail support: alliance-support@asim.lip6.fr

Loading logical view : dec0opt
Placing logical view : dec0opt
Loading SCP data base ...
Generating initial placement ...
9 cells 14 nets in 3 rows
Placement in process of treatment : 100%
5% saved in 13.1 s
Saving placement 100%
Checking consistency between logical and physical views
Loading SCR data base ...
Deleting MBK data base ...
Global routing ...
Channel routing ...
|_____Routing Channel : scr_p2
|_____Routing Channel : scr_p4
|_____Routing Channel : scr_p6
|_____Routing Channel : scr_p8
Making vertical power and ground wires
Saving layout : dec0opt
[cicuttin@mlab-42]$
```

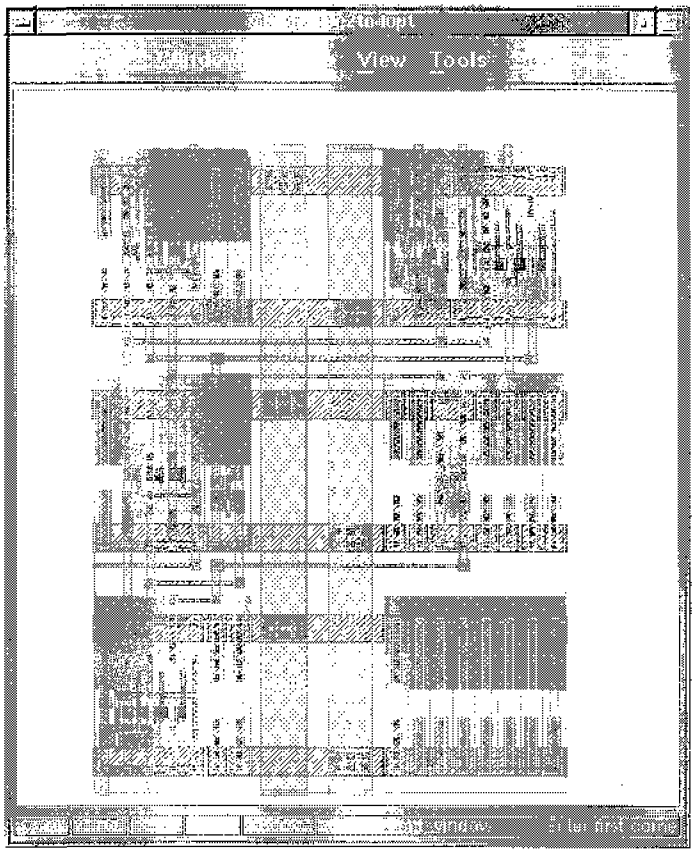
Using the symbolic layout editor

To see the core layout, "dec0opt.ap" we use **Graal** a symbolic layout editor.

Give the command

```
graal
```

A new window is opened. Choose the **File** menu from the menu bar and choose the **open** option from the menu that pops up. Another sub window inside the main window will be opened. In this window the files with the extension .ap will be listed. Choose the dec0opt.ap file and press the **ok** button. The layout will appear on the screen, but only at the standard cell level. Now you can choose the **Tool** option from the menu bar and from the pop-up menu choose the **peek** option. Now with the mouse mark the window where you want to "peek" at the layout. If the whole layout is "peek"ed typically you will see the layout as shown below:



Now the core is ready to be connected to the pads, which interface the heart of the chip to the outside environment.

Here we require to edit two files one with extension ".c" and the other with ".rin"

Create a file with the text editor called "dec0chip.c" and enter the following:

```
#include <genlib.h>

main()
{

  DEF_LOFIG("dec0chip");

  LOCON("A",    IN,    "A"   ); /* input B */
  LOCON("B",    IN,    "B"   ); /* input A */
  LOCON("NC",   IN,    "NC"  ); /* not connected */
  LOCON("enable", IN, "enable" ); /* input enable */
  LOCON("vdd",  IN,  "vdd"  ); /* core power supply */
  LOCON("vss",  IN,  "vss"  ); /* core ground       */
  LOCON("vdde",  IN, "vdde" ); /* pads power supply */
  LOCON("vssc",  IN, "vssc" ); /* pads ground       */
  LOCON("Y[0:3]", OUT, "Y[0:3]"); /* output          */
/*          power supplies:
  pxxxe_sp are external power supplies, ie used only by the buffers inside the pads.
```

```
    pxxxi_sp are internal power supplies, for core logic only.      */
    LOINS ("pvsse_sp", "p16", "cki", "vdde", "vdd", "vsse", "vss", 0);
    LOINS ("pvdde_sp", "p20", "cki", "vdde", "vdd", "vsse", "vss", 0);
    LOINS ("pvssi_sp", "p18", "cki", "vddc", "vdd", "vsse", "vss", 0);
    LOINS ("pvddi_sp", "p19", "cki", "vdde", "vdd", "vsse", "vss", 0);

    LOINS("pck_sp","NCpad","NC","cki","vdde","vdd","vsse","vss",0);

    LOINS("pi_sp", "p12", "enable", "en", "cki", "vdde", "vdd", "vsse", "vss", 0);

    LOINS("pi_sp", "p13", "a", "aa", "cki", "vdde", "vdd", "vsse", "vss", 0);
    LOINS("pi_sp", "p14", "b", "bb", "cki", "vdde", "vdd", "vsse", "vss", 0);

    LOINS("po_sp", "p0", "yy[0]", "y[0]", "cki", "vdde", "vdd", "vsse", "vss", 0);
    LOINS("po_sp", "p1", "yy[1]", "y[1]", "cki", "vdde", "vdd", "vsse", "vss", 0);
    LOINS("po_sp", "p2", "yy[2]", "y[2]", "cki", "vdde", "vdd", "vsse", "vss", 0);
    LOINS("po_sp", "p3", "yy[3]", "y[3]", "cki", "vddc", "vdd", "vsse", "vss", 0);

    LOINS("dec0opt", "core", "aa", "bb", "en", "vdd", "vss", "yy[0:3]", 0);

    SAVE_LOFIG();
    exit(0);
}
```

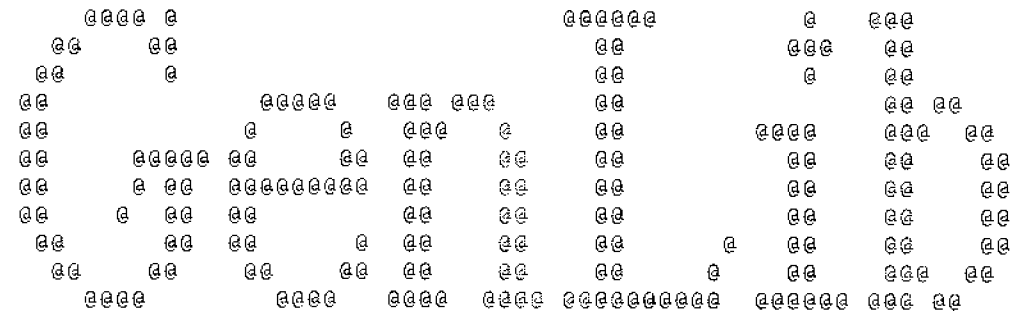
This file describes the external chip pins, and the connectivity between the pads and the core. It is written in the **Genlib** procedural language, which is basically a set of **C** macro functions. The connection between the pads and the core is described in this language using the *Netlist capture* macro functions. More details on **Genlib** and related macro functions can be obtained from the man pages (man genlib).

Now give the following command at the command line:

```
genlib -v  dec0chip
```

The following screen is typically generated.

```
[cicuttin@mlab-42]$ genlib -v  dec0chip
```



Procedural Generation Language

Alliance CAD System 3.2b, genlib 3.3
Copyright (c) 1991-1999, ASIM/LIP6/UPMC
E-mail support: alliance-support@asim.lip6.fr

```
Generating the Makefile
Compiling, ...
Current execution environment
MBK_CATA_LIB  : .:/alliance/archi/Linux_elf/cells/sclib:/alliance/archi/Linux_elf/cells/padlib
MBK_WORK_LIB  : .
MBK_IN_LO     : vst
MBK_OUT_LO    : vst
MBK_IN_PH     : ap
MBK_OUT_PH    : ap
MBK_CATAL_NAME : CATAL
Executing ...
Removing tmp files ...

[cicuttin@mlab-42]$
```

This generates the structural description file dec0chip.vst, which has the core and the pads put together.

Simulating the completed chip

The chip with the pads can be simulated with the original pattern file. The pads need a separate ground and power supply, then two pins of the chip provides these specific signals. The file dec2to4.pat can be used to test the structural view of the chip but as it is does not provide the stimulus to the signals vdde and vsse needed by the pads. Thus warning messages could appear telling something like "power supply is missing on po_sp".

```
asimut dec0chip dec2to4 r5
```

no option	-	takes the structural description (.vst) by default
dec0chip	-	dec0chip.vst
dec2to4	-	dec2to4.pat
r5	-	r5.pat simulation result file

Now check the output file r5.pat by using the more command.

Placing and routing the pads

Now the chip's pads and the core has to be connected together physically in a layout. The physical placement and routing of the core to the pads is done by using **Ring**. The relative position of the pads on the four sides of the chip is described in a ".rin" file. For more information on **Ring** and its capabilities, see the man pages (man ring).

Edit a file called "dec0chip.rin" and enter the following as is:

```
width (vdd 20 vss 20)

west (p0 p19 p20)

north (p1 p2 p3)

east (p14 p18 p16)

south (p12 p13 NCpad)
```

Now give the command:

```
ring dec0chip dec0chip
```

The following is typically displayed.

```
[cicuttin@mlab-42]$ ring dec0chip dec0chip

                @@@@@@@@          @                    @@@@ @
                  @@    @@      @@@                  @@     @@
                  @@     @@      @                  @@       @
                  @@     @@           @@@ @@@     @@
                  @@    @@    @@@@     @@@    @    @@
                  @@@@@          @@     @@     @@   @@      @@@@@
                  @@   @@        @@     @@     @@   @@      @ @@
                  @@    @@       @@     @@     @@   @@     @   @@
                  @@    @@       @@     @@     @@   @@        @@
                  @@     @@      @@     @@     @@    @@     @@
                @@@@@    @@@ @@@@@@@ @@@@   @@@@      @@@@

                              PAD ring router

                        Alliance CAD System 3.2b,            ring 2.9
                        Copyright (c) 1991-1999,       ASIM/LIP6/UPMC
                        E-mail support: alliance-support@asim.lip6.fr

        o reading netlists, layout views of core and pads.
        o reading file of parameters, including the placements of pads.
        o making equipotential list.
        o making the first placement of pads.
        o filling data internal structures.
        o reading the connectors positions of the core.
        o computing the best placement of the pads.
        o reading the connectors positions of the pads.
        o routing deportation of connectors.
        o routing supply tracks.
        o routing equipotentials.
        o compressing channels.
        o saving in MBK data structure.
lucky, no error.
[cicuttin@mlab-42]$
```

As before with **Graal** you can see the complete chip file "dec0chip.ap".

Static Timing analysis

The "dec0chip.ap" contains the layout information. However we do not know if the physical description produced reflects the desired behavior. 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 analyzer 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 dec0chip dec0chip
```

-v	-	verbose
-t	-	build the netlist to the transistor level.

first dec0chip - take the "dec0chip.ap" layout file as input.
second dec0chip - generate the "dec0chip.al" netlist file.

The following is typically displayed.

```
[cicuttin@mlab-42]$ lynx -v -t dec0chip dec0chip

@@@@@@
  @@
  @@
  @@          @@@@@ @@@ @@@ @@@     @@@@  @@@
  @@           @@     @   @@@   @     @@     @
  @@            @@    @   @@    @@    @@   @
  @@            @@  @     @@    @@      @@@
  @@              @@ @     @@    @@      @@@
  @@       @     @@@       @@    @@     @  @@
  @@     @         @@      @@    @@    @     @@
@@@@@@@@@@@  @@   @       @@@@  @@@@ @@@  @@@@
             @@  @
              @@@
                        Netlist extractor

   Alliance CAD System 3.2b,        lynx 1.16
   Copyright (c) 1997-1999,      ASIM/LIP6/UPMC
   E-mail support: alliance-support@asim.lip6.fr

---> Extracts symbolic figure dec0chip

        ---> Flatten figure

        ---> Translate Mbk -> Rds
        ---> Build windows
        <--- 2401

        ---> Rectangles    : 63720
        ---> Figure size   : ( -7055, -6655 )
                             ( 8135, 8595 )
        ---> Cut transistors
        <--- 0
        ---> Build equis
        <--- 52
        ---> Delete windows
        ---> Build signals
        <--- 52
        ---> Build instances
        <--- 0
        ---> Build transistors
        <--- 368
        ---> Save netlist

    <--- done !
[cicuttin@mlab-42]$
```

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.clp dec0chip
```

-tec - selects the technology file prol10.elp

[cicuttin@mlab-42]$ tas -tec=/alliance/archi/Linux_elf/etc/prol10.elp dec0chip

```
                    @
  @                 @
 @@                @@@
 @@               @@@        @@@@@@
@@@@@@@@         @  @@      @@     @
 @@              @  @@      @@@
 @@             @     @@     @@@@
 @@             @@@@@@@       @@@@
 @@           @       @@   @     @@@
 @@    @     @        @@   @@     @@
  @@@@  @@@@      @@@@ @ @@@@@
```

CMOS-VLSI Timing Analyzer

Alliance CAD System 3.2b, tas 5.21
Copyright (c) 1990-1999. ASIM/LIP6/UPMC
E-mail support: alliance-support@asim.lip6.fr

TECHNOLOGY FILE IS : '/alliance/archi/Linux_elf/etc/prol10.elp'
TECHNOLOGY : prol10 VERSION : 2.00
REFERENCE : HSPICE, LEVEL = 2.00

LOADING FILE dec0chip.al :
00min 00s
tas user : 00'00.0"
system : 00'00.0"

DISASSEMBLING :
[YAG MES] Transistor netlist checking 00m00s u:00m00.0 s:00m00.0
[YAG MES] Extracting CMOS duals 00m00s u:00m00.0 s:00m00.0
[YAG MES] Extracting bleeders 00m00s u:00m00.0 s:00m00.0
[YAG MES] Making gates 00m00s u:00m00.0 s:00m00.0
[YAG MES] Latches detection 00m00s u:00m00.0 s:00m00.0
[YAG MES] External connector verification 00m00s u:00m00.0 s:00m00.0
[YAG MES] Checking the yagle figure 00m00s u:00m00.0 s:00m00.0
00min 00s
tas user : 00'00.0"
system : 00'00.0"

COMPUTING GATE DELAYS :
00min 00s
tas user : 00'00.0"
system : 00'00.0"

SEARCHING OF CRITICAL PATHES :
00min 00s
tas user : 00'00.0"
system : 00'00.0"

GENERAL PERFMODULE dec0chip.ttx :
00min 00s
tas user : 00'00.0"
system : 00'00.0"

TIMING ANALYSIS REPORT :
complexity = 86
the circuit worst case delay is 7303pS

```
--> from -\_ b
--> to -\_ y[3]
nb couple = 24
nb chain = 0
 00min 00s
tas user   : 00'00.0"
   system : 00'00.0"
---------------------------------
TOTAL RUN TIME :
 00min 00s
tas user   : 00'00.0"
   system : 00'00.0"
0.00% CPU---end!!!
---------------------------------
[cicuttin@mlab-42]$
```

Inspect the result on the screen, in the sessions "SEARCHING OF CRITICAL PATHES" and "TIMING ANALYSIS REPORT" to know the worst case delay.

Layout Extraction and Netlist Comparison

The "dec0chip.ap" contains the layout information. However we do not know if the physical description produced reflects the behavioral 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.

Give the command at the command line:

```
lynx -v -f dec0chip dec0chip
```

-v	-	verbose
-f	-	asks **Lynx** to generate the netlist from the Standard-cells level.
first dec0chip	-	Take the "dec0chip.ap" layout file as input.
second dec0chip	-	Generate the "dec0chip.al" netlist file.

The following is typically displayed.

```
[cicuttin@mlab-42]$ lynx -v -f dec0chip dec0chip

                @@@@@@
                  @@
                  @@
                  @@          @@@@@ @@@ @@@ @@@     @@@@   @@@
                  @@            @@     @    @@@    @     @@     @
                  @@              @@    @    @@     @@     @@   @
                  @@              @@   @     @@     @@      @@@
                  @@                @@ @     @@     @@      @@@
                  @@         @      @@@      @@     @@     @   @@
                  @@       @          @@      @@     @@    @     @@
                @@@@@@@@@@@  @@   @      @@@@   @@@@ @@@   @@@@
                             @@   @
                              @@@
```

Netlist extractor

```
Alliance CAD System 3.2b,        lynx 1.16
Copyright (c) 1997-1999,     ASIM/LIP6/UPMC
E-mail support: alliance-support@asim.lip6.fr


---> Extracts symbolic figure dec0chip

        ---> Flatten figure

        ---> Translate Mbk -> Rds
        ---> Build windows
        <--- 2401

        ---> Rectangles   : 3146
        ---> Figure size  : ( -7055, -6655 )
                            ( 8135, 8595 )

        ---> Cut transistors
        <--- 0
        ---> Build equis
        <--- 25
        ---> Delete windows
        ---> Build signals
        <--- 25
        ---> Build instances
        <--- 25
        ---> Build transistors
        <--- 0
        ---> Save netlist

<--- done !

[cicuttin@mlab-42]$
```

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.
More information on **Lvx** can be obtained from the man pages (man lvx).
Give the command at the command line

```
lvx vst al dec0chip dec0chip -f -o
```

-f	-	build the netlist to the standard cell level.
-o	-	to have ordered connectors in the output netlist
vst	-	take the first file in .vst format.
al	-	take the second file in .al format.
first dec0chip	-	"dec0chip.vst" input file.
second dec0chip	-	"dec0chip.al" output file.

The following is typically displayed.

```
[cicuttin@mlab-42]$ lvx vst al dec0chip dec0chip -f -o

@@@@@@       @@@@     @@@ @@@@    @@@@
  @@          @@       @   @@       @
  @@          @@       @    @@    @
  @@           @@     @       @@ @
  @@           @@     @         @@
  @@            @@   @          @@
  @@            @@   @          @@@
  @@             @@@           @   @@
  @@       @      @@@         @     @@
  @@     @          @       @       @@
@@@@@@@@@@@        @      @@@     @@@@
```

```
                    Gate Netlist Comparator

          Alliance CAD System 3.2b,        lvx 2.23
          Copyright (c) 1992-1999,    ASIM/LIP6/UPMC
          E-mail support: alliance-support@asim.lip6.fr

***** Loading and flattening dec0chip (vst)...

***** Loading and flattening dec0chip (al)...


***** Compare Terminals .....
***** O.K.       (0 sec)

***** Compare Instances .....
***** O.K.       (0 sec)

***** Compare Connections .....
***** O.K.       (0 sec)

===== Terminals ..... 12
===== Instances ..... 21
===== Connectors ..... 131

***** Netlists are Identical. *****  (0 sec)

***** Ordering .....
***** O.K. (0 sec)

***** Saving dec0chip (al)...

[cicuttin@mlab-42]$
```

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** 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 "dec0chip.al" can be simulated using **Asimut** and the test vector file "dec2to4.pat".
The input file format for **Asimut** must be "al".

Give the following command at the command line.

```
asimut  dec0chip dec2to4 r6
```

dec0chip	-	take the "dec0chip.al" as input file
dec2to4	-	take the "dec2to4.pat" test vector file
r6	-	deliver the results in file "r6.pat".

Any error 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 transistor netlist the behavior of the circuit. Essentially a **VHDL** file is created from a standard cell connectivity list! This **VHDL** file can be simulated in turn to verify the function of the chip.

Give the command at the command line:

```
yagle -v dec0chip
```

-v	-	signal vectorized
dec0chip	-	takes the "dec0chip.al" as input.

The extracted VHDL description is in the file "dec0chip.vbc".

The following is typically displayed.

```
[cicuttin@mlab-42]$ yagle -v dec0chip
```

```
                                        @@@@
                                          @@
                                          @@
@@@@@ @@@   @@@@      @@@@@@   @@      @@@@@
 @@    @   @@   @    @@  @@    @@     @    @
  @@   @   @@   @@   @   @@    @@    @@    @@
  @@  @      @@@@@   @   @     @@    @@@@@@@@@
   @@ @    @@   @@    @@@      @@    @@
   @@@    @@    @@   @@        @@    @@      @
    @@    @@   @@@   @@@@@@    @@    @@    @@
@@  @      @@@@  @@ @@   @@@ @@@@@@    @@@@
@@  @               @     @
 @@@                 @@@@@
```

Yet Another Gate Level Extractor

Alliance CAD System 3.2b, yagle 2.02
Copyright (c) 1994-1999, ASIM/LIP6/UPMC
E-mail support: alliance-support@asim.lip6.fr

```
[YAG MES] Loading the figure dec0chip          00m00s u:00m00.0 s:00m00.0
[YAG MES] Flattening the figure               00m00s u:00m00.0 s:00m00.0
[YAG MES] Transistor netlist checking          00m00s u:00m00.0 s:00m00.0
[YAG MES] Extracting CMOS duals                00m00s u:00m00.0 s:00m00.0
[YAG MES] Extracting bleeders                 00m00s u:00m00.0 s:00m00.0
[YAG MES] Making gates                       00m00s u:00m00.0 s:00m00.0
[YAG MES] Latches detection                   00m00s u:00m00.0 s:00m00.0
[YAG MES] Making cells                       00m00s u:00m00.0 s:00m00.0
[YAG MES] External connector verification      00m00s u:00m00.0 s:00m00.0
[YAG MES] Checking the yagle figure           00m00s u:00m00.0 s:00m00.0
[YAG MES] Building the behavioural figure      00m00s u:00m00.0 s:00m00.0
TOTAL DISASSEMBLY TIME                         00m00s u:00m00.0 s:00m00.0
-------------------------------------------------------
[YAG MES] Erasing the transistor netlist
[YAG MES] Generating the VHDL Data Flow
[YAG MES] Execution COMPLETED
-------------------------------------------------------
[YAG WAR 04] 32 transistors are always off
[YAG WAR 07] 32 transistors are not used
[YAG WAR 13] 1 signals do not drive anything
See file 'dec0chip.rep' for more information

[cicuttin@mlab-42]$
```

Give the command:

```
asimut -b dec0chip dec2to4 r4
```

to simulate the extracted behavioral file.

Alliance has a program called **Proof** that compares the extracted behavioral file with the original behavioral file to formally prove the functional congruence of the described and the extracted circuit.

Give the command:

```
proof -d dec0chip dec2to4
```

-d - displays logical functions as they are processed
dec0chip - extracted "dec0chip.vbe" file.
dec2to4 - original "dec2to4.vbe" file.

The following is typically displayed.

```
[cicuttin@mlab-42]$ proof -d dec0chip dec2to4

@@@@@@@                                                  @@@
  @@    @@                                               @  @@
  @@     @@                                             @@  @@
  @@     @@ @@@ @@@      @@@        @@@       @@
  @@    @@   @@@  @@  @@    @@    @@    @@  @@@@@@@@@
  @@@@@       @@   @@ @@      @@ @@       @@    @@
  @@          @@       @@      @@ @@       @@    @@
  @@          @@       @@      @@ @@       @@    @@
  @@          @@       @@      @@ @@       @@    @@
  @@          @@        @@    @@   @@    @@     @@
@@@@@@      @@@@          @@@        @@@     @@@@@@

                        Formal Proof

Alliance CAD System 3.2b, proof 4.20 [1997/10/09]
Copyright (c) 1990-1999,        ASIM/LIP6/UPMC
E-mail support:    alliance-support@asim.lip6.fr

========================================= Environment ==========================================
MBK_WORK_LIB            = .
MBK_CATA_LIB            = .:/alliance/archi/Linux_elf/cells/sclib:/alliance/archi/Linux_elf/cells/padlib
====================== Files, Options and Parameters ======================
First VHDL file         = dec2to4.vbe
Second VHDL file  = dec0chip.vbe
The auxiliary signals are erased
Errors are displayed
================================================================================================

Compiling 'dec2to4' ...
Compiling 'dec0chip' ...

Running abl ordonnancer on 'dec2to4'
....

Running Abl2Bdd on 'dec2to4'
---> final number of nodes = 16(9)

Running Abl2Bdd on 'dec0chip'

------------------------------------------------------------------------
        Formal proof with Ordered Binary Decision Diagrams between

        './dec2to4' and './dec0chip'
------------------------------------------------------------------------
============================== PRIMARY OUTPUT ==============================
============================== AUXILIARY SIGNAL ==============================
============================== REGISTER SIGNAL ==============================
============================== EXTERNAL BUS ==============================
============================== INTERNAL BUS ==============================
```

```
                Formal Proof : OK

pppppppppppppppppppppppprrrrrrrroooooooooooooooooooooooooffffffffffffff
----------------------------------------------------------------------

|cicuttin@mlab-42|S
```

This formal functional verification is much powerful than any verification done with **Asimut**. If both behavioral descriptions, original and extracted, are formally equivalent, then they will give the same response to any stimulus. On the other hand **Asimut** allows checking the response to the stimulus defined in the input pattern file only.

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/archie/Linux_elf/etc/proll0_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.

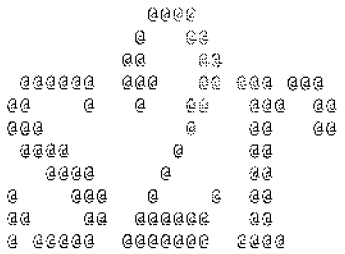
Give the command:

```
s2r -cv dec0chip dec0chip
```

-c	-	deletes connectors at the highest hierarchy. (Use man to see full description)
-v	-	verbose mode on
first dec0chip	-	"dec0chip.ap" file as input
second dec0chip	-	"dec0chip.cif" file as output.

The following is typically displayed.

```
[cicuttin@mlab-42]$ s2r -cv dec0chip dec0chip
```



Symbolic to Real layout converter

```
Alliance CAD System 3.2b,          s2r 3.6
Copyright (c) 1991-1998,       ASIM/LIP6/UPMC
E-mail support: alliance-support@asim.lip6.fr

o loading technology file : /alliance/archi/Linux_elf/etc/prol10_7.rds
o loading all level of symbolic layout : dec0chip
o removing symbolic data structure
o layout post-treating without connector, with scotchs.
--> post-treating model pali_sp
    rectangle merging :
    . RDS_NWELL .....
    . RDS_NIMP .....
    . RDS_PIMP .....
    . RDS_ACTIV .....
    . RDS_POLY .....
    . RDS_ALU1 .....
    . RDS_ALU2 .....
--> post-treating model pi_sp
    ring flattenning  :
    . RDS_NWELL .....
    . RDS_NIMP .....
.....
.....
.....

o replacing black boxes
--> replace cell padreal
o saving dec0chip.cif
o memory allocation informations
--> required rectangles = 3112  really allocated = 7
--> required scotchs = 0  really created = 0
--> Number of allocated bytes: 183724

[cicuttin@mlab-42]$
```

This completes the design of the decoder chip.

SEVENTH COURSE
ON
BASIC VLSI DESIGN TECHNIQUES

MICROPROCESSOR LABORATORY
ICTP-UNESCO

29 October- 23 November, 2001

Trieste, Italy

Exercise 3

Design of an Octal Tri-state Transceiver chip

Problem Description

This design example is a transceiver chip similar to the 74HC245. In this design example you will learn to:

- Describe and simulate the behaviour of the transceiver in VHDL.
- Simulate bi-directional signals and about IO pads.
- Make 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 you have created, to complete formal verification.

This design example consists of two phases. The first phase is to describe the behaviour of the chip as is seen at the pins of the chip. The second phase is to describe the functions of the core of the chip, and then connect it to the pads.

In the first phase you will:

- Describe the transceiver's behaviour using **VHDL** (xceiver.vbe).
- Write test pattern files.
- Simulate the behavioural description using the pattern file by using **Asimut**.

In the second phase you will:

- Describe the behaviour of the core in **VHDL** as is seen inside the chip by the pads (xceivercore.vbe).
- Synthesise the logic and structural descriptions using **Bop** and **Scmap** (xceivercorel.vbe and xceivercorel.vst).
- Use the standard cell router called **Scr** to place and route the core (xceivercorel.ap).
- Add the necessary pads for the chip and compile using **Genlib** (xceiverchip.vst).
- Use **Asimut** to simulate the 'xceiverchip.vst' file using a pattern file created by Genpat in the first phase.
- Place the pads and generate the layout of the chip with pads using **Ring** (xceiverchip.ap).
- Use **Lynx** to extract the netlist from the layout file 'xceiverchip.ap' (xceiverchip.al).
- Use **Tas** to perform the static timing analysis.
- Use **Lvx** to compare the extracted circuit 'xceiverchip.al' and the original 'xceiverchip.vst' file created by **Genlib**.

- Use **Yagle** to extract the behaviour, 'xceiverchip.vbe' from the 'xceiverchip.al' netlist file.
- Use **Proof** to compare the extracted behaviour file, 'xceiverchip.vbe' and the behavioural file created in the first phase, 'xceiver.vbe'.

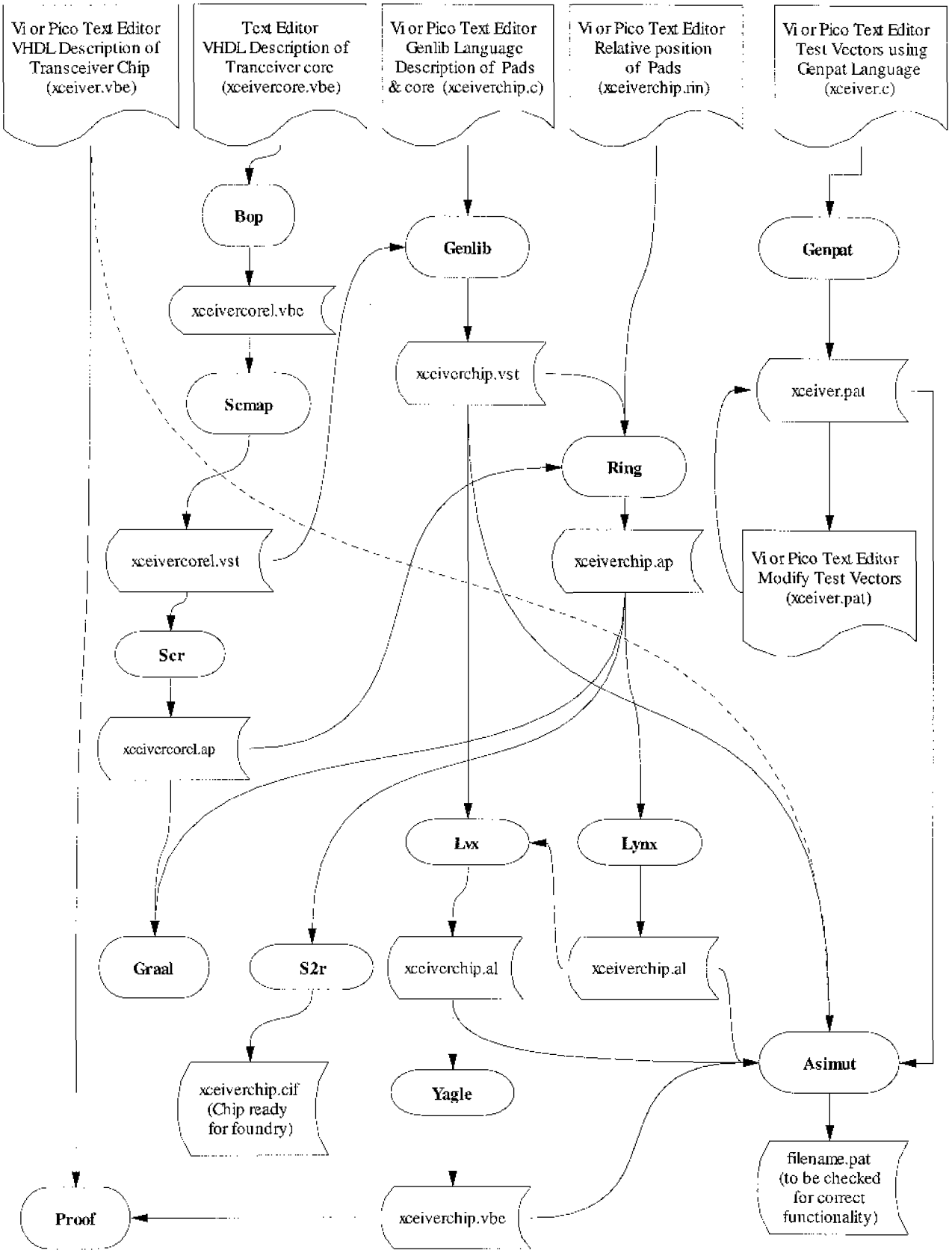


Fig 1. Design flow for the Transceiver Chip

Transceiver Chip General Description

The transceiver chip that is proposed in this example is similar to the 74HC245 transceiver chip. The pin diagram of the transceiver chip and the truth table of the controls shown below explain the operation of the chip as looked from outside.

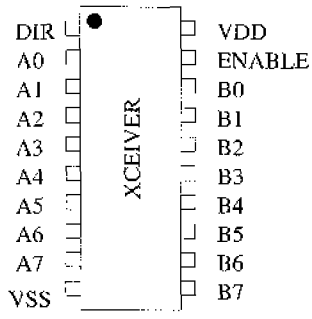


Fig. 2. Pinout of 74HC245 transceiver chip.

ENABLE	DIR	Operation
L	L	Data Transmitted from Bus B to Bus A
L	H	Data Transmitted from Bus A to Bus B
H	X	Busses Isolated (High-Impedance State)

Table 1. Truth Table for the controls of the transceiver chip

Solution

Legend

Give the command that appear 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 xceiver
```

Change into this directory:

```
cd xceiver
```

Before starting the design you will have to set the environmental variables as shown below so that you will not run into problems later.

```
setenv MBK_CATA_LIB .:/alliance/archi/Linux_elf/cells/sclib:
                      /alliance/archi/Linux_elf/cells/padlib

setenv MBK_IN_LO vst

setenv MBK_OUT_LO vst

setenv MBK_IN_PH ap

setenv MBK_OUT_PH ap

setenv MBK_WORK_LIB .
```

Create with the "pico" (or "vi") editor a file called "xceiver.vbe". Enter the following and save the file.

```
-- Octal Tristate Non-inverting Bus transceiver ---
-- 6th Course on VLSI Design - Trieste

ENTITY xceiver IS
PORT (Vdd, Vss, Vdde, Vsse: IN BIT;
       A, B: inout MUX_VECTOR (7 downto 0) BUS;
       dir, enable : IN BIT);
END xceiver;

ARCHITECTURE xceiver_b OF xceiver IS

begin

b1: BLOCK (dir ='0' and enable = '0')
BEGIN
       A <= guarded B;
END BLOCK b1;
b2: BLOCK (dir = '1' and enable = '0')
BEGIN
       B <= guarded A;
END BLOCK b2;

end xceiver_b;
```

Typographical or syntax errors can be found when the file is passed through **Asimut** in the compilation mode. Before using Asimut you will have to set the environmental variables as shown previously.

Give the following command at the command line

```
asimut -b -c xceiver
```

Creating the test pattern for simulation

If the above step functions without giving syntax errors, then the behavioural description is ready for simulation.
A file with the test patterns in the **pat** format is required for the simulation. The **pat** format file has a declaration part and a description part of the signals. The declaration part consists of a list of inputs, outputs, internal signals and registers of the design. Inputs are forced to a particular value while the outputs are observed during the simulation.

Edit this file and make changes to the file like the one shown below. Save the modified pattern file.

```
-- description generated by Pat driver v104
--
-- date : Sep 14 21:00:18 1999


--             sequence : xceiver

-- input / output list :
in       vdd B;
in       vss B;
in       vdde B;
in       vsse B;;;;;;
inout    a (7 downto 0) X;;
inout    b (7 downto 0) X;;;
in       dir B;;;
in       enable B;;;

begin

-- Pattern description :

--        vvvv      a   b   d  e
--        dsds              i  n
--        dsds              r  a
--          ee                 b
--                             1
--                             e


-- Beware : unprocessed patterns

        : 1010     ?00  00  0  0  ;
        : 1010     ?55  55  0  0  ;
        : 1010     ?00  00  0  0  ;
        : 1010      00 ?00  1  0  ;
        : 1010      55 ?55  1  0  ;
        : 1010      00 ?00  1  0  ;
        : 1010     ?00  00  0  0  ;
        : 1010     ?AA  AA  0  0  ;
        : 1010     ?00  00  0  0  ;
        : 1010      00 ?00  1  0  ;
        : 1010      AA ?AA  1  0  ;
        : 1010      00 ?00  1  0  ;
        : 1010     ?55  55  0  0  ;
        : 1010     ?55  55  0  0  ;
        : 1010     ?55  55  0  0  ;
        : 1010      55 ?55  1  0  ;
        : 1010      55 ?55  1  0  ;
        : 1010      55 ?55  1  0  ;
        : 1010     ?AA  AA  0  0  ;
        : 1010     ?AA  AA  0  0  ;
        : 1010     ?AA  AA  0  0  ;
        : 1010      AA ?AA  1  0  ;
        : 1010      AA ?AA  1  0  ;
        : 1010      AA ?AA  1  0  ;
        : 1010     ?55  55  0  0  ;
        : 1010     ?55  55  0  0  ;
        : 1010     ?55  55  0  0  ;
        : 1010      AA ?AA  1  0  ;
        : 1010      AA ?AA  1  0  ;
        : 1010      AA ?AA  1  0  ;
        : 1010     ?AA  AA  0  0  ;
        : 1010     ?AA  AA  0  0  ;
        : 1010     ?AA  AA  0  0  ;
        : 1010      55 ?55  1  0  ;
        : 1010      55 ?55  1  0  ;
        : 1010      55 ?55  1  0  ;
        : 1010     ?** ?**  0  1  ;
        : 1010      55  55  0  1  ;
        : 1010     ?** ?**  0  1  ;
        : 1010     ?** ?**  1  1  ;
        : 1010      AA  AA  1  1  ;
        : 1010     ?** ?**  1  1  ;

end;
```

****

********** ***Optional*** **********

We suggest you to skip this part (thtat isn't essential to the exercise) the first time. You can come back to this part later or in a second time. Go ahead to the next paragraph: ***Simulating the Behavioural Description*** *(page 10).*

Genpat *is a set of* ***C*** *functions that allows a procedural description of input patterns file for the logic simulator* ***Asimut****. The* ***genpat*** *command accepts a* ***C*** *file as input and produces a pattern description file as output. Information on the functions that are allowed in* ***genpat*** *is given in the man pages (man genpat).*
A file with test patterns is required for the simulation. You will have to write a C file that when treated with ***Genpat*** *will generate the pattern file for you.*

Create a file called "xceiver.c" and enter the following:

```
/*************************************************************************\
      Trieste, microprocessor laboratory

      File              :     xceiver
      Date        :     10 21 1999
      Version           :     3

\*************************************************************************/

/*-----------------------------------------------------------\
                      Includes
\-----------------------------------------------------------*/
#include <mut321.h>
#include <stdio.h>
#include <genpat.h>

/*-----------------------------------------------------------\
                      defines
\-----------------------------------------------------------*/

#define maxcycle  5

/*-----------------------------------------------------------\
                      inttostr
\-----------------------------------------------------------*/
char *inttostr(integer, len)
  int integer;
  int len;
{
 char *str;

 str = (char *) mbkalloc (len * sizeof (char) + 1);
 sprintf (str, "%.32d", integer);
 return(&str[32-len]);
}

/*-----------------------------------------------------------\
\-----------------------------------------------------------*/
void dir()
```

```
{
 int i;

 for (i=0;i < (maxcycle*6);i=i+6)
 {
  AFFECT(inttostr(i,32),"dir","0b0");
  AFFECT(inttostr(i+1,32),"dir","0b0");
  AFFECT(inttostr(i+2,32),"dir","0b0");
  AFFECT(inttostr(i+3,32),"dir","0b1");
  AFFECT(inttostr(i+4,32),"dir","0b1");
  AFFECT(inttostr(i+5,32),"dir","0b1");
 }
}


/*-------------------------------------------------------------\
                        power
\-------------------------------------------------------------*/
void power()
{
  AFFECT(inttostr(0,32),"vdd","0b1");
  AFFECT(inttostr(0,32),"vss","0b0");
  AFFECT(inttostr(0,32),"vdde","0b1");
  AFFECT(inttostr(0,32),"vsse","0b0");
}

/*-------------------------------------------------------------\
                        enable
\-------------------------------------------------------------*/
void enable()
{
 AFFECT(inttostr(0,32),"B","0B0");
 AFFECT(inttostr(0,32),"enable","0B0");
}


/*-------------------------------------------------------------\
                        main
\-------------------------------------------------------------*/
int main()
{
/*  int i,j;*/

 /* Declaring name of pattern file */
  DEF_GENPAT("xceiver");

  DECLAR("vdd",":0","B",IN,"");
  DECLAR("vss",":5","B",IN,"");
  DECLAR("vdde",":0","B",IN,"");
  DECLAR("vsse",":5","B",IN,"");
  DECLAR("A",":1","x",INOUT,"7 downto 0");
  DECLAR("B",":2","x",INOUT,"7 downto 0");
  DECLAR("dir",":2","B",IN,"");
  DECLAR("enable",":2","B",IN,"");


 /* initilisation of the enable, vdd, vss, and it */
 dir();
 power();
```

```
 enable();

 /* the end */
 SAV_GENPAT();

}
```

Give the following command at the command prompt:

```
genpat xceiver
```

This command typically generates the following display.

```
[cicuttin@mlab-42]$ genpat xceiver

      @@@@ @                          @@@@@@@@
    @@     @@                           @@    @@                 @
   @@       @                           @@     @@               @@
  @@                @@@@@    @@@ @@@     @@     @@    @@@@       @@
  @@              @      @    @@@    @     @@    @@    @@    @    @@@@@@@@@
  @@       @@@@@ @@      @@   @@     @@    @@@@@       @@    @@     @@
  @@       @ @@  @@@@@@@@@@   @@     @@    @@           @@@@@     @@
  @@     @   @@  @@           @@     @@    @@         @@    @@     @@
   @@       @@  @@        @   @@     @@    @@        @@     @@     @@
    @@     @@    @@     @@   @@     @@    @@        @@    @@@     @@    @
      @@@@          @@@@    @@@@   @@@@  @@@@@@      @@@@   @@     @@@@

                Procedural GENeration of test PATterns

             Alliance CAD System 3.2b,           genpat 3.1
             Copyright (c) 1991-1999,        ASIM/LIP6/UPMC
             E-mail support: alliance-support@asim.lip6.fr

[cicuttin@mlab-42]$
```

A pattern file "xceiver.pat" is created by ***Genpat****. You can easily learn how the C file works by changing some of the parameters in the C code and inspecting the correspondent changes in the generated pattern file "xceiver.pat".*

```
************************************************************************************
************************************************************************************
************************************************************************************
```

Simulating the Behavioural Description

Now the behavioural file "xceiver.vbe" can be simulated with this pattern file.

Give the following command at the command prompt to start simulating.

```
asimut -b xceiver xceiver r1
```

-b	-	chooses the behavioural simulation option
first xceiver	-	takes the xceiver.vbe as input
second xceiver	-	takes the xceiver.pat vector file for simulation
r1	-	result of simulation is put in r1.pat

The simulation should proceed without any errors. If errors appear, check the xceiver.vbe or the xceiver.pat file.

Describing the core of the chip

The above description that we have made in the "xceiver.vbe" file simulates the transceiver as is seen from the pins of the chip. We did not care about the pads that drive the pins. However when a chip is described physically in Alliance, it consists of two separate parts that are brought together, the core or heart of the chip and the pads. In Alliance, the core and the pads are brought together in a C description file. This file when treated with **Genlib**, produces the structural description of the chip with the pads. In practice the core can be synthesised automatically form a behavioural description, whereas the pads should be placed physically, one by one in the C file. Placing the pads require the structural knowledge of the pads. One of the type of pads that is used in this example is the piot_sp IO pad, a cell of "padlib", a library of pads provided with Alliance.

Give the following command at the command line to see a description of this pad.

```
man piot_sp
```

As will be seen from the behavioural description, this pad has towards the outside a tri-state, while towards the core, a data input, a data output and a control line that controls the direction of the data.

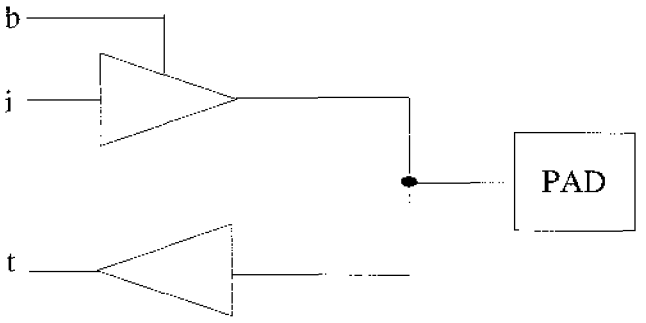


Fig. 3. Schematic of the IO pad piot_sp

b	PAD	t
1	i	PAD
0	High Z when looked from i	PAD

Table 2. Truth Table for controls of the IO pad piot_sp

Thus for an IO pad, the core will have
- a data output that is connected to the data input of the pad,
- a data input that will be connected to the data output of the pad and,
- a control line output that will be connected to the control line input of the pad.

Behavioural Description of the Core

With the above knowledge of the IO pads, we are now ready to describe the functions of the core.

Edit a new file called "xceivercore.vbe" and give the description as shown below:

```
-- Octal Tristate Non-inverting Bus transceiver ---
-- 6th Course on VLSI design TRIESTE ---

ENTITY xceivercore IS
PORT (Vdd, Vss: IN BIT;
      AIN, BIN: in BIT_VECTOR (7 downto 0);
      AOUT, BOUT: out BIT_VECTOR (7 downto 0);
      ACONT, BCONT: OUT BIT;
      dir, enable : IN BIT);
END xceivercore;

ARCHITECTURE xceiver_b OF xceivercore IS
signal enab: BIT_VECTOR (7 downto 0);
begin

ASSERT (vdd = '1' and vss = '0')
REPORT "Wrong power supplies"
```

```
SEVERITY WARNING;

enab(0) <= enable;
enab(1) <= enable;
enab(2) <= enable;
enab(3) <= enable;
enab(4) <= enable;
enab(5) <= enable;
enab(6) <= enable;
enab(7) <= enable;
AOUT <= BIN ;
BOUT <= AIN ;
ACONT <= (not dir) and (not enable);
BCONT <= dir and (not enable);

end xceiver_b;
```

Synthesising the Logic and the Structure of the Core

We use **Bop** to synthesise the logic and **Scmap** to synthesise the structural description of the transceiver core.

Give the following command at the command line:

```
bop -o xceivercore xceivercorel
```

-o	-	option for global optimization
xceivercore	-	xceivercore.vbe (input file)
xceivercorel	-	xceivercorel.vbe (output file)

The logic description of the core is created in the file "xceivercorel.vbe". From this file we proceed to synthesise the structural view of the core. To do this give the following command at the command line:

```
scmap xceivercorel xceivercorel
```

The structural description of the core is created in the file "xceivercorel.vst".

Placement and Routing of the core

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

```
scr -p -r -l 5 -i 1000 xceivercorel
```

-p	-	placement option
-r	-	routing option
-l	-	number of rows
-i	-	iteration number

A "xceivercorel.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 (structural view).

Edit and save into the file "xceiverchip.c" the following:

```
/* Transceiver chip */
/* Date: 07-17-99 */

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

 LOCON("VDD", 'I', "VDD");
 LOCON("VSS", 'I', "VSS");
 LOCON("VDDE", 'I', "VDDE");
 LOCON("VSSE", 'I', "VSSE");
 LOCON("A[0:7]", 'X', "A[0:7]");
 LOCON("B[0:7]", 'X', "B[0:7]");
 LOCON("DIR", 'I', "DIR");
 LOCON("ENABLE", 'I', "ENABLE");
 LOCON("NC",   IN,   "NC"   ); /*  */

 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", "vssc", "vss", 0);
 LOINS("pvssi_sp", "ivss","cki","vdde", "vdd", "vsse", "vss", 0);

 LOINS("pck_sp","NCpad","NC","cki","vdde","vdd","vsse","vss",0);

 LOINS("piot_sp", "A0", "AOUT[0]", "ACONT", "AIN[0]","A[0]","cki", "vdde", "vdd",
"vssc", "vss", 0);
 LOINS("piot_sp", "A1", "AOUT[1]", "ACONT", "AIN[1]","A[1]","cki", "vdde", "vdd",
"vssc", "vss", 0);
 LOINS("piot_sp", "A2", "AOUT[2]", "ACONT", "AIN[2]","A[2]","cki", "vdde", "vdd",
"vsse", "vss", 0);
```

```
 LOINS("piot_sp", "A3", "AOUT[3]", "ACONT", "AIN[3]","A[3]","cki", "vdde", "vdd",
"vsse", "vss", 0);
 LOINS("piot_sp", "A4", "AOUT[4]", "ACONT", "AIN[4]","A[4]","cki", "vdde", "vdd",
"vsse", "vss", 0);
 LOINS("piot_sp", "A5", "AOUT[5]", "ACONT", "AIN[5]","A[5]","cki", "vdde", "vdd",
"vsse", "vss", 0);
 LOINS("piot_sp", "A6", "AOUT[6]", "ACONT", "AIN[6]","A[6]","cki", "vdde", "vdd",
"vsse", "vss", 0);
 LOINS("piot_sp", "A7", "AOUT[7]", "ACONT", "AIN[7]","A[7]","cki", "vdde", "vdd",
"vsse", "vss", 0);


 LOINS("piot_sp", "B0", "BOUT[0]", "BCONT", "BIN[0]","B[0]","cki", "vdde", "vdd",
"vsse", "vss", 0);
 LOINS("piot_sp", "B1", "BOUT[1]", "BCONT", "BIN[1]","B[1]","cki", "vdde", "vdd",
"vsse", "vss", 0);
 LOINS("piot_sp", "B2", "BOUT[2]", "BCONT", "BIN[2]","B[2]","cki", "vdde", "vdd",
"vsse", "vss", 0);
 LOINS("piot_sp", "B3", "BOUT[3]", "BCONT", "BIN[3]","B[3]","cki", "vdde", "vdd",
"vsse", "vss", 0);
 LOINS("piot_sp", "B4", "BOUT[4]", "BCONT", "BIN[4]","B[4]","cki", "vdde", "vdd",
"vsse", "vss", 0);
 LOINS("piot_sp", "B5", "BOUT[5]", "BCONT", "BIN[5]","B[5]","cki", "vdde", "vdd",
"vsse", "vss", 0);
 LOINS("piot_sp", "B6", "BOUT[6]", "BCONT", "BIN[6]","B[6]","cki", "vdde", "vdd",
"vsse", "vss", 0);
 LOINS("piot_sp", "B7", "BOUT[7]", "BCONT", "BIN[7]","B[7]","cki", "vdde", "vdd",
"vsse", "vss", 0);


 LOINS("pi_sp", "dir", "dir", "pdir", "cki","vdde", "vdd", "vsse", "vss", 0);
 LOINS("pi_sp", "enable", "enable", "penable", "cki","vdde", "vdd", "vsse", "vss", 0);



LOINS("xceivercorel","xceiver","vdd","vss","ain[7:0]","bin[7:0]","aout[7:0]","bout[7:0]
","acont", "bcont", "pdir", "penable", 0);

 SAVE_LOFIG();

 exit(0);
}
```

Give the command at the command line:

```
genlib -v xceiverchip
```

This creates a "xceiverchip.vst" structural description file with pads. Use "more" to browse through the structural description.

Simulating the Structural Description

You can now simulate this structural description with the test vector file that we developed for "xceiver.vbe".

Give the command at the command line:

```
asimut xceiverchip xceiver r2
```

xceiverchip	-	The structural description "xceiverchip.vst" with pads
xceiver	-	The "xceiver.pat" test vector file.
r2	-	Result to be place in "r2.pat" file.

There should be no errors, which means that the structural description is functionally equivalent to the behavioural 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 "xceiverchip.rin":

```
# File used by ring tool  to define the relative position of pads
north (a0 dir vdd cnable b0 b1)
west ( a4 a3 ivss a2 a1)
south ( a5 a6 NCpad vss a7 b7 b6)
east (b5 b4 ivdd b3 b2)
```

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 xceiverchip xceiverchip
```

The physical file "xceiverchip.ap" is created that can be examined by using **Graal**.

Examine the layout using **Graal**.

Static Timing Analysis

The "xceiverchip.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 xceiverchip xceiverchip
```

-v	-	verbose
-t	-	build the netlist to the transistor level.
first xceiverchip	-	take the "xceiverchip.ap" layout file as input.
second xceiverchip	-	generate the "xceiverchip.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 xceiverchip
```

-tec - selects the technology file prol10.elp.

Layout Extraction and Netlist Comparison

The "xceiverchip.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**.

Give the command at the command line:

```
lynx -v -f xceiverchip xceiverchip
```

-v	-	verbose
-f	-	asks **Lynx** to generate the netlist from the Standard-cells level.
first xceiverchip	-	Takes the "xceiverchip.ap" layout file as input.
second xceiverchip	-	Generate the "xceiverchip.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 xceiverchip xceiverchip -f -o
```

vst	-	takes the first file in .vst format.
al	-	takes the second file in .al format.
first xceiverchip	-	"xceiverchip.vst" file.
second xceiverchip	-	"xceiverchip.al" file.
-f	-	build the netlist to the standard cell level.
-o	-	to have ordered connectors in the output netlist

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** 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 "xceiverchip.al" can be simulated using **Asimut** and the test vector file "xceiver.pat".

Give the following command at the command line:

```
setenv MBK_IN_LO al
```

This sets the input file format for **Asimut** for the ".al" format.

Give the following command at the command line.

```
asimut  xceiverchip xceiver r3
```

xceiverchip	-	take the "xceiverchip.al" as input file
xceiver	-	take the "xceiver.pat" test vector file
r3	-	deliver the results in file "r3.pat".

Any error 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 transistor netlist the behaviour of the circuit. Essentially a **VHDL** file is created from a standard cell connectivity list! This **VHDL** file can be simulated in turn to verify the function of the chip.

Give the command at the command line:

```
yagle -v xceiverchip
```

-v	-	vectorize
xceiverchip	-	Takes the "xceiverchip.al" as input.

The extracted VHDL description is put in the file "xceiverchip.vbe". Give the command:

```
asimut -b xceiverchip xceiver r4
```

to simulate the extracted behavioural file.

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.

Give the command:

```
proof -d xceiverchip xceiver
```

-d	-	displays logical functions as they are processed
xceiverchip	-	extracted "xceiverchip.vbe" file.
xceiver	-	original "xceiver.vbe" file.

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**.

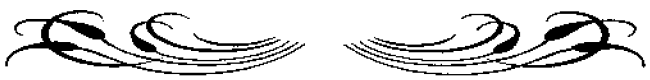
```
setenv RDS_TECHNO_NAME /alliance/archi/Linux_elf/etc/pro110_7.rds
setenv RDS_OUT cif
setenv RDS_IN cif
```

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 xceiverchip xceiverchip
```

-c	-	deletes connectors at the highest hierarchy. (Use man to see full description)
-v	-	verbose mode on
first xceiverchip	-	"xceiverchip.ap" file as input
second xceiverchip	-	"xceiverchip.cif" file as output.

This completes the design of the transceiver chip.



SEVENTH COURSE
ON
BASIC VLSI DESIGN TECHNIQUES

MICROPROCESSOR LABORATORY
ICTP-UNESCO

29 October- 23 November, 2001

Trieste, Italy

Exercise 4

Design of a 4 bit Presettable Synchronous Binary Counter using VHDL dataflow

Problem Description

In this example you will design a 4-bit presettable synchronous binary counter using VHDL dataflow. In this design example you will learn to:

- Specify the behaviour of the counter using VHDL and simulate it.
- Generate the structural description of the counter and simulate it.
- Place the necessary pads and re-simulate the structural description of the counter.
- Make 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.

This design example consists of two phases. The first phase is to describe the behaviour of the chip as is seen at the pins of the chip. The second phase is to describe the functions of the core of the chip, and then connect it to the pads.

In the first phase you will:

- Describe the counter's behaviour using VHDL (counter.vbe).
- Write test patterns files.
- Simulate the behavioural description using the pattern file by using **Asimut**.

In the second phase you will:

- Describe the behaviour of the core in VHDL as is seen inside the chip by the pads (countcore.vbe).
- Synthesise the logic and structural descriptions using **Bop** and **Scmap** (countcorel.vst).
- Use **Glop** to optimise for critical path and fanout (countopt.vst).
- Use the standard cell router called **Scr** to place and route the core (countopt.ap).
- Add the necessary pads for the chip and compile using **Genlib** (countchip.vst).
- Use **Asimut** to simulate the 'countchip.vst' file using the pattern file 'counter.pat'.
- Place the pads and generate the layout of the chip with pads using **Ring** (countchip.ap).
- Use **Tas** to perform the static timing analysis.
- Use **Lynx** to extract the netlist from the layout file 'countchip.ap' (countchip.al).
- Use **Lvx** to compare the extracted circuit 'countchip.al' and the original 'countchip.vst' file created by **Genlib**.
- Use **Yagle** to extract the behaviour, 'countchip.vbe' from the 'countchip.al' netlist file.
- Use **Proof** to compare the extracted behaviour file, 'countchip.vbe' and the behavioural file created in the first phase, 'counter.vbe'.

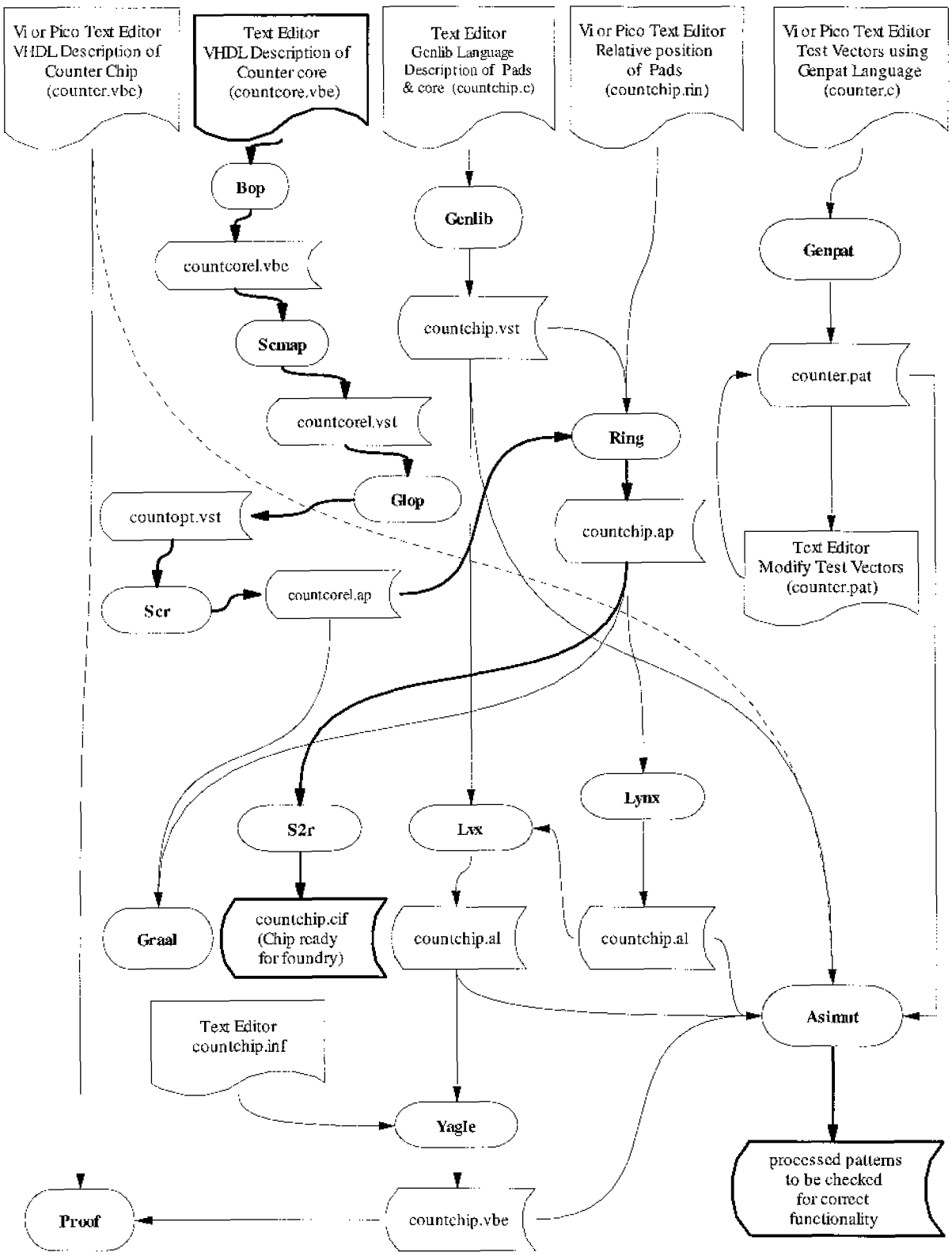


Fig 1. Design Flow for the counter chip.

A 4-Bit Presettable Synchronous Binary Counter

The present exercise is a 4-bit presettable synchronous binary counter. The counter has an "enable" which when at logic '1' allows the counter to count. The counter is presettable to the value given on the input bus when "preset=1". The counter counts forward starting from this value. There is a synchronous reset, which puts the counter to zero when it is '0'.

A possible pin diagram of the counter is shown in Fig. 2.

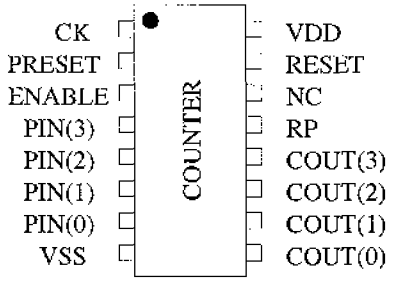


Fig. 3 Counter chip (a possible pinout diagram).

CK	RESET	PRESET	ENABLE	COUT(3:0)
Rising Edge	0	X	X	0000
Rising Edge	1	1	X	PIN(3:0)
Rising Edge	1	0	1	COUT(3:0) +1
Rising Edge	1	0	0	COUT(3:0)
No Rising Edge	X	X	X	COUT(3:0)

Table 1. Truth Table for the 4-bit presettable counter

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 counter
```

Change into this directory:

```
cd counter
```

Before starting the design you will have to set the environmental variables as shown below so that you will not run into problems later.

```
setenv MBK_CATA_LIB .:/alliance/archi/Linux_elf/cells/sclib:
                      /alliance/archi/Linux_elf/cells/padlib
setenv MBK_IN_LO vst
setenv MBK_OUT_LO vst
setenv MBK_IN_PH ap
setenv MBK_OUT_PH ap
setenv MBK_WORK_LIB .
```

Create with the text editor a file called "counter.vbe". Enter the following and save the file.

```
-- Behavioural description using VHDL
-- 6th Workshop on VLSI Design - Trieste

ENTITY counter IS
  PORT(
     Vdd, Vss, Vddc, Vssc: in BIT;
     Pin: in BIT_VECTOR (3 downto 0);
     Cout: out BIT_VECTOR (3 downto 0);
     ck: in BIT;
     reset: in BIT;
     enable: in BIT;
     preset: in BIT;
     rp: out BIT
    );
END counter;

ARCHITECTURE dataflow OF counter IS

SIGNAL count: REG_VECTOR (3 downto 0) REGISTER;

BEGIN

lcount : BLOCK(ck='1' and not ck'STABLE)
  BEGIN
       count <= GUARDED B"0000" when (reset ='0') else
       Pin     WHEN  (preset = '1') else
       B"0001"  WHEN  ((enable='1') and (count = B"0000"))  else
       B"0010"  WHEN  ((enable='1') and (count = B"0001"))  else
       B"0011"  WHEN  ((enable='1') and (count = B"0010"))  else
       B"0100"  WHEN  ((enable='1') and (count = B"0011"))  else
       B"0101"  WHEN  ((enable='1') and (count = B"0100"))  else
       B"0110"  WHEN  ((enable='1') and (count = B"0101"))  else
       B"0111"  WHEN  ((enable='1') and (count = B"0110"))  else
       B"1000"  WHEN  ((enable='1') and (count = B"0111"))  else
       B"1001"  WHEN  ((enable='1') and (count = B"1000"))  else
       B"1010"  WHEN  ((enable='1') and (count = B"1001"))  else
       B"1011"  WHEN  ((enable='1') and (count = B"1010"))  else
       B"1100"  WHEN  ((enable='1') and (count = B"1011"))  else
       B"1101"  WHEN  ((enable='1') and (count = B"1100"))  else
       B"1110"  WHEN  ((enable='1') and (count = B"1101"))  else
       B"1111"  WHEN  ((enable='1') and (count = B"1110"))  else
       B"0000"  WHEN  ((enable='1') and (count = B"1111"))  else
       count;
END BLOCK lcount;

Cout <= count;

rp <= count(0) and count(1) and count(2) and count(3);

end dataflow;
```

Test Pattern File and Simulation of the Behavioural Description

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 if it is necessary by editing it and simulate using **Asimut** with the **-b** option and check that the counter performs satisfactorily.

Describing the core of the chip

The behavioural file "counter.vbe" is the description of the counter as is seen at the pins of the chip. We have not thought about the pads that drive the pins. When the chip is described physically in Alliance, it consists if two separate parts that are brought together, the core and the pads. In Alliance, the core and the pads are brought together in a C description file. This file when treated with **Genlib**, produces the structural description of the chip with the pads. In practice the core can be synthesised automatically form a behavioural description, whereas the pads should be placed physically, one by one in the C file. Placing the pads require the structural knowledge of the pads. One of the type of pads that is used in this example is the pck_sp clock pad, a cell of "padlib", a library of pads provided with Alliance.

Give the following command at the command line to see a description of this pad.

```
man pck_sp
```

Behavioural Description of the Core

Copy the file "counter.vbe" to the file "countcore.vbe", edit it and delete the Vdde and Vsse input signals since they are used only for the Pads.

Logic and Structural Synthesis of the Core

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

Give the command:

```
bop -o countcore countcore1
```

This takes as input the "countcore.vbe" description and creates an optimised behavioural description file "countcore1.vbe". Now to synthesise the structural description give the command:

```
scmap countcorel countcorel
```

This takes as input the optimised behavioural description "countcorel.vbe" and creates a structural description file "countcorel.vst" using the elementary components from the standard cell library.

Optimising for Fanout and Timing

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 countcorel countopt -i -t
```

-g	-	invokes timing optimization.
countcorel	-	countcorel.vst input file
countopt	-	countopt.vst output file
-i	-	gives fanout information about the gate netlist.
-t	-	gives timing information about the gate netlist.

This command takes "countcorel.vst" structural description and generates a "countopt.vst" file after buffers have been added to the critical paths.

Give the command:

```
glop -f countopt countopt
```

-f	-	invokes fanout optimization.
countopt	-	countopt.vst modified structural file

This command should add buffers to the appropriate nets to resolve fanout problems and write over the "countopt.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 -l 4 -i 100 countopt
```

-p	-	placement option
-r	-	routing option
-l 4	-	asks to place and route the core in 4 rows
-i 100	-	use 100 iterations to improve placement quality

A "countopt.ap" layout file is created which can be viewed with **Graal**. Inspect it using **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.
Edit and save into the file "countchip.c" the following:

```
#include <genlib.h>

main()
{
int i;

  DEF_LOFIG("countchip");

  LOCON("ck",    IN,    "ck"   );
  LOCON("reset",   IN,   "reset" );
  LOCON("preset",   IN,   "preset" );
  LOCON("enable", IN, "enable"  );
  LOCON("vdd",   IN,  "vdd"   ); /* core power supply */
  LOCON("vss",   IN,  "vss"   ); /* core ground      */
  LOCON("vdde",   IN,  "vdde"  ); /* pads power supply */
  LOCON("vsse",   IN,  "vsse"  ); /* pads ground      */
  LOCON("PIN[3:0]",  IN, "PIN[3:0]"); /*   preset input      */
  LOCON("COUT[3:0]",  OUT, "COUT[3:0]"); /*     output        */
  LOCON("rp",    OUT,    "rp"   );

/*
  power supplies:
  pxxxe_sp are external power supplies, i.e. used only by the buffers
     inside the pads.
  pxxxi_sp are internal power supplies, for core logic only.
*/

  LOINS ("pvsse_sp", "p30", "cki", "vdde", "vdd", "vsse", "vss", 0);
  LOINS ("pvsse_sp", "p31", "cki", "vdde", "vdd", "vsse", "vss", 0);
  LOINS ("pvdde_sp", "p32", "cki", "vdde", "vdd", "vsse", "vss", 0);
  LOINS ("pvdde_sp", "p33", "cki", "vdde", "vdd", "vsse", "vss", 0);
  LOINS ("pvssi_sp", "p34", "cki", "vdde", "vdd", "vsse", "vss", 0);
  LOINS ("pvddi_sp", "p35", "cki", "vdde", "vdd", "vsse", "vss", 0);

  LOINS("pck_sp", "p0", "ck", "cki", "vdde", "vdd", "vsse", "vss", 0);

  LOINS("pvssick_sp", "p1","clock", "cki","vdde", "vdd", "vsse", "vss", 0);

  LOINS("pi_sp", "p2","reset", "res","cki", "vdde", "vdd", "vsse", "vss", 0);
  LOINS("pi_sp", "p3","preset", "pres","cki", "vdde", "vdd", "vsse", "vss", 0);
  LOINS("pi_sp", "p4","enable", "en","cki", "vddc", "vdd", "vsse", "vss", 0);

  LOINS("po_sp", "p5","rprp", "rp","cki", "vdde", "vdd", "vsse", "vss", 0);
```

```
  LOINS("po_sp", "p10","usc[0]", "cout[0]","cki", "vdde", "vdd", "vssc", "vss", 0);
  LOINS("po_sp", "p11","usc[1]", "cout[1]","cki", "vdde", "vdd", "vsse", "vss", 0);
  LOINS("po_sp", "p12","usc[2]", "cout[2]","cki", "vdde", "vdd", "vsse", "vss", 0);
  LOINS("po_sp", "p13","usc[3]", "cout[3]","cki", "vdde", "vdd", "vsse", "vss", 0);


  LOINS("pi_sp", "p20","pin[0]", "ingr[0]","cki", "vddc", "vdd", "vsse", "vss", 0);
  LOINS("pi_sp", "p21","pin[1]", "ingr[1]","cki", "vdde", "vdd", "vsse", "vss", 0);
  LOINS("pi_sp", "p22","pin[2]", "ingr[2]","cki", "vdde", "vdd", "vsse", "vss", 0);
  LOINS("pi_sp", "p23","pin[3]", "ingr[3]","cki", "vdde", "vdd", "vsse", "vss", 0);

  LOINS("countopt", "core","vdd", "vss", "ingr[3:0]", "usc[3:0]", "clock", "res",
"cn","prcs", "rprp", 0);

  SAVE_LOFIG();
  exit(0); /* necessary for the proper run of the Makefile */
}
```

Give the command at the command line:

```
genlib -v countchip
```

This creates the "countchip.vst" structural description file with pads.

Simulating the Structural Description

You can now simulate this structural description with the test vector file that you used for "counter.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 "countchip.rin":

```
width (vdd 20 vss 20)

west (p0 p32 p35 p33 p23)

south (p3 p2 p1 p4 p5)

north (p10 p30 p34 p31 p21)

east (p11 p12 p13 p20 p22)
```

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

Give the command at the command line:

```
ring countchip countchip
```

The "countchip.ap" layout file is created that can be examined by using **Graal**.
Examine the layout using **Graal**.

Static Timing Analysis

The "countchip.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 the 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 countchip countchip
```

-v	-	verbose
-t	-	build the netlist to the transistor level.
first countchip	-	take the "countchip.ap" layout file as input.
second countchip	-	generate the "countchip.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 countchip
```

-tec - selects the technology file prol10.elp.

Layout Extraction and Netlist Comparison

The "countchip.ap" contains the layout information. However we do not know if the physical description produced reflect the initial 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 countchip countchip
```

-v	-	verbose
-f	-	asks **Lynx** to generate the netlist at the Standard-cells level.
first countchip	-	Take the "countchip.ap" layout file as input.
second countchip	-	Generate the "countchip.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 countchip countchip -f -o
```

vst	-	take the first file in .vst format.
al	-	take the second file in .al format.
first countchip	-	"countchip.vst" input file.
second countchip	-	"countchip.al" output file.
-f	-	build the netlist to the standard cell level.
-o	-	to have ordered connectors in the output netlist

The comparison should not produce any errors. If errors are produced by the program, then there is something wrong 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** 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 "countchip.al" can be simulated using **Asimut** and the test vector file that has been created to test the behavioural file "counter.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 several structural descriptions (man yagle), the behaviour of the circuit. Essentially a VHDL file is created from a standard cell connectivity description or from a SPICE transistor netlist! This VHDL file can be simulated in turn to verify the function of the chip

Give the command at the command line:

```
yagle -v countchip
```

-v	-	vectorize
countchip	-	Takes the "countchip.al" as input.

The extracted VHDL description is put in the file "countchip.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 "countchip.inf". If we do a "more" of the "countchip.vbe" file we see that the registers have a different name from the one that we have given in "counter.vbe".

Edit and save a file "countchip.inf" with the following:

```
rename
core.count_1.dff_s : count_1 ;
core.count_0.dff_s : count_0 ;
core.count_2.dff_s : count_2 ;
core.count_3.dff_s : count_3 ;
end
```

Give the command:

```
yagle -i -v countchip
```

-i	-	asks **Yagle** to read the "countchip.inf" file and rename the registers in the "countchip.vbe" file as given in the list.
-v	-	vectorize

Now a formal verification compares the *extracted* and the *original* behavioural descriptions. Give the command:

```
proof -p -d counter countchip
```

-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.

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 -v countchip countchip
```

-c	-	deletes connectors at the highest hierarchy. (Use man to see full description)
-v	-	verbose mode on
first countchip	-	"countchip.ap" file as input
second countchip	-	"countchip.cif" file as output.

This completes the design of the counter chip.

SEVENTH COURSE
ON
BASIC VLSI DESIGN TECHNIQUES

MICROPROCESSOR LABORATORY
ICTP-UNESCO

29 October- 23 November, 2001

Trieste, Italy

Exercise 5

Design of a 4 bit Adder Accumulator using VHDL Dataflow

Problem Description

In this example you will design a 4-bit binary adder accumulator using VHDL dataflow. In this design example you will learn to:

- Specify the behaviour of the adder using VHDL and simulate it.
- Generate the structural description of the adder and simulate it.
- Place the necessary pads and re-simulate the structural description of the adder.
- 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.

This design example consists of two phases. The first phase is to describe the behaviour of the chip as is seen at the pins of the chip. The second phase is to describe the functions of the core of the chip, and then connect it to the pads.

In the first phase you will:

- Describe the adder's behaviour using VHDL (adder.vbe).
- Write test pattern files.
- Simulate the behavioural description using the pattern file by using **Asimut**.

In the second phase you will:

- Describe the behaviour of the core in VHDL as is seen inside the chip by the pads (addercore.vbe).
- Synthesise the logic and structural descriptions using **Bop** and **Scmap** (addercorel.vbe & addercorel.vst).
- Use **Glop** to optimise for critical path and fanout (addopt.vst).
- Use the standard cell router called **Scr** to place and route the core (addopt.ap).
- Add the necessary pads for the chip and compile using **Genlib** (addchip.vst).
- Use **Asimut** to simulate the 'addchip.vst' file using the pattern file 'adder.pat'.
- Place the pads and generate the layout of the chip with pads using **Ring** (addchip.ap).
- Use **Tas** to perform the static timing analysis.
- Use **Lynx** to extract the netlist from the layout file 'addchip.ap' (addchip.al).
- Use **Lvx** to compare the extracted circuit 'addchip.al' and the original 'addchip.vst' file created by **Genlib**.
- Use **Yagle** to extract the behaviour, 'addchip.vbe' from the 'addchip.al' netlist file.
- Use **Proof** to compare the extracted behaviour file, 'addchip.vbe' and the behavioural file created in the first phase, 'adder.vbe'.

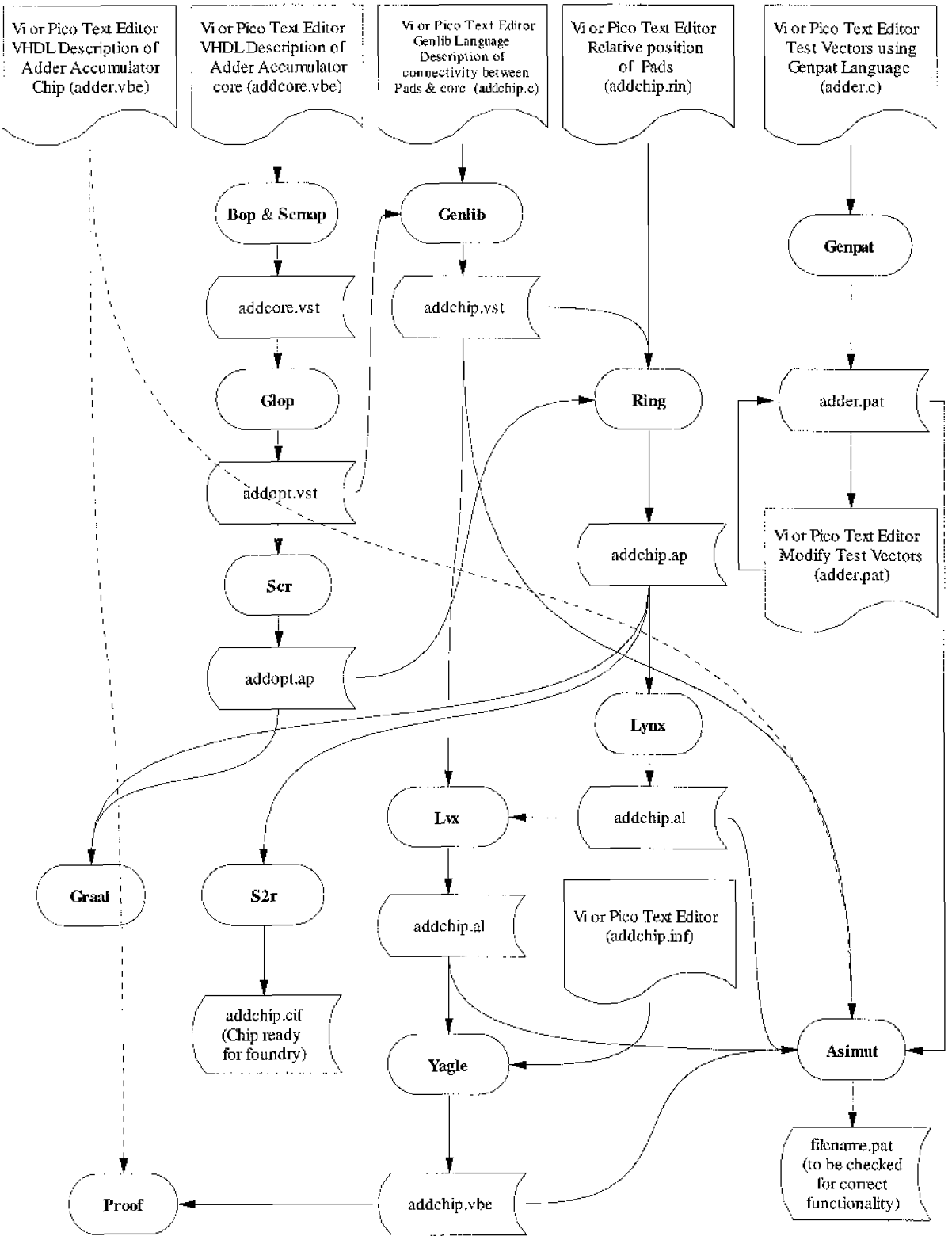


Fig 1. Design flow of the adder accumulator chip

A 4-Bit Binary adder accumulator

The present exercise is a 4-bit binary adder accumulator. The adder has a "select" which when at logic '0' allows the adder to sum the two inputs 4-bit buses: A and B, and when at logic '1' the input 4-bit bus A is added to the result stored in a 4-bit register which we call the accumulator. The accumulator is updated at the rising edge of the clock. The result of the sum is presented at the 4-bit output bus Y.

A possible pin diagram of the counter is shown in Fig. 2.

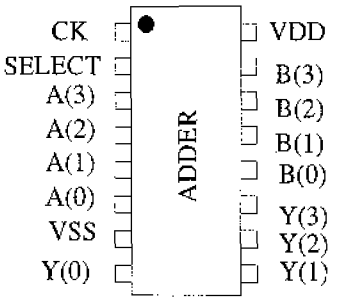


Fig 2. Adder chip (a possible pinout diagram).

CK	SELECT	Y(3:0)
Rising Edge	0	A+B
Rising Edge	1	A+Y
No Rising Edge	X	Y(3:0)

Table 1. Truth Table for the 4-bit binary adder

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 adder
```

Change into this directory:

```
cd adder
```

Before starting the design you will have to set the environmental variables as shown below so that you will not run into problems later.

```
setenv MBK_CATA_LIB .:/alliance/archi/Linux_elf/cells/sclib:
                     /alliance/archi/Linux_elf/cells/padlib
setenv MBK_IN_LO vst
setenv MBK_OUT_LO vst
setenv MBK_IN_PH ap
setenv MBK_OUT_PH ap
setenv MBK_WORK_LIB .
```

Create with the text editor a file called "adder.vbe". Enter the following and save the file.

```
ENTITY adder IS

PORT(
 vdd, vss, vdde, vsse : in BIT ;
 ck              : in BIT ;
 sel             : in BIT ;
 a               : in  BIT_VECTOR (3 DOWNTO 0) ;
 b               : in  BIT_VECTOR (3 DOWNTO 0) ;
 y               : out BIT_VECTOR (3 DOWNTO 0)
 );

END adder;

ARCHITECTURE data_flow OF adder IS

SIGNAL regstr       : REG_VECTOR (3 DOWNTO 0) REGISTER;
SIGNAL mux          : BIT_VECTOR (3 DOWNTO 0) ;
SIGNAL sum          : BIT_VECTOR (3 DOWNTO 0) ;
SIGNAL carry        : BIT_VECTOR (2 DOWNTO 0) ;

BEGIN


 WITH sel SELECT

 mux     <= b WHEN '0', regstr WHEN '1' ;

 sum(0)   <= a(0) xor mux(0) ;
 carry(0) <= a(0) and mux(0) ;

 sum(1)   <= a(1) xor mux(1) xor carry(0) ;
 carry(1) <= (a(1) and mux(1))    or
          (mux(1) and carry(0)) or
          (a(1) and carry(0)) ;

 sum(2)   <= a(2) xor mux(2) xor carry(1) ;
 carry(2) <= (a(2) and mux(2))    or
          (mux(2) and carry(1)) or
          (a(2) and carry(1)) ;

 sum(3)   <= a(3) xor mux(3) xor carry(2) ;


  store : BLOCK ((ck = '1') and not ck'STABLE)

       BEGIN
       regstr <= GUARDED sum ;

       END BLOCK ;


  y <= regstr;

END;
```

Test Pattern File and Simulation of the Behavioural Description

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 if it is necessary by editing it and simulate using **Asimut** with the **-b** option and check that the counter performs satisfactorily.

Describing the core of the chip

The behavioural file "adder.vbe" is the description of the adder as is seen at the pins of the chip. We have not thought about the pads that drive the pins. When the chip is described physically in Alliance, it consists of two separate parts that are brought together, the core and the pads. In Alliance, the core and the pads are brought together in a C description file. This file when treated with **Genlib**, produces the structural description of the chip with the pads. In practice the core can be synthesised automatically from a behavioural description, whereas the pads should be placed physically, one by one in the C file. Placing the pads require the structural knowledge of the pads. One of the types of pads that is used in this example is the pi_sp input pad, a cell of PAD-Lib, a library of pads provided with Alliance.

Give the following command at the command line to see a description of this pad.

```
man pi_sp
```

Behavioural Description of the Core

Copy the file "adder.vbe" to the file "addercore.vbe", edit it and delete the Vdde and Vsse input signals since they are not necessary for the core.

Logic and Structural Synthesis of the Core

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

Give the command:

```
bop -o addercore addercore1
```

This takes as input the "addercore.vbe" description and creates an optimised behavioural description file "addercore1.vbe".

To synthesise the structural description give the command:

```
scmap addercorel addercorel
```

This takes as input the optimise behavioural description "addercorel.vbe" and creates a structural description file "addercorel.vst" using the components from the standard cell library.

Optimising for Fanout and Timing

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 addercorel addopt -i -t
```

-g	-	invokes timing optimization.
-i	-	gives fanout information about the gate netlist.
-t	-	gives timing information about the gate netlist.

This command takes "addercorel.vst" structural description and generates a "addopt.vst" file after buffers have been added to the critical paths.

Give the command:

```
glop -f addopt addopt
```

This command should add buffers to the appropriate nets to resolve fanout problems and write over the "addopt.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 -l 4 -i 1000 addopt
```

-p	-	placement option
-r	-	routing option
-l 4	-	asks to place and route the core in 4 rows
-i 1000	-	use 1000 iterations to improve placement quality

A "addopt.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 "addchip.c" the following:

```
#include <genlib.h>
main()
{
 DEF_LOFIG("addchip");
 LOCON("a[3:0]",'I',"a[3:0]");
 LOCON("b[3:0]",'I',"b[3:0]");
 LOCON("y[3:0]",'O',"y[3:0]");
 LOCON("sel",'I',"sel");
 LOCON("ck",'I',"ck");
 LOCON("vdde",'I',"vdde");
 LOCON("vsse",'I',"vsse");
 LOCON("vdd",'I',"vdd");
 LOCON("vss",'I',"vss");

  LOINS ("pvssc_sp", "Vss", "cki", "vdde", "vdd", "vsse", "vss", 0);
  LOINS ("pvdde_sp", "Vdd", "cki", "vdde", "vdd", "vsse", "vss", 0);
  LOINS ("pvssi_sp", "Vssi", "cki", "vdde", "vdd", "vsse", "vss", 0);
  LOINS ("pvddi_sp", "Vddi", "cki", "vdde", "vdd", "vsse", "vss", 0);

  LOINS("pi_sp","sl","sel","sl","cki","vdde","vdd","vsse","vss",0);

  LOINS("pck_sp", "clk", "ck", "cki", "vdde", "vdd", "vsse", "vss", 0);

  LOINS("pvsseck_sp", "clkcore", "clkcore", "cki",
                                        "vdde", "vdd", "vsse", "vss", 0);

  LOINS("pi_sp","a0","a[0]","ina[0]","cki","vdde","vdd","vsse","vss",0);
  LOINS("pi_sp","a1","a[1]","ina[1]","cki","vdde","vdd","vsse","vss",0);
  LOINS("pi_sp","a2","a[2]","ina[2]","cki","vdde","vdd","vsse","vss",0);
  LOINS("pi_sp","a3","a[3]","ina[3]","cki","vdde","vdd","vsse","vss",0);

  LOINS("pi_sp","b0","b[0]","inb[0]","cki","vdde","vdd","vsse","vss",0);
  LOINS("pi_sp","b1","b[1]","inb[1]","cki","vdde","vdd","vsse","vss",0);
  LOINS("pi_sp","b2","b[2]","inb[2]","cki","vdde","vdd","vsse","vss",0);
  LOINS("pi_sp","b3","b[3]","inb[3]","cki","vdde","vdd","vsse","vss",0);

  LOINS("po_sp","y0","out[0]","y[0]","cki","vdde","vdd","vsse","vss",0);
  LOINS("po_sp","y1","out[1]","y[1]","cki","vdde","vdd","vsse","vss",0);
  LOINS("po_sp","y2","out[2]","y[2]","cki","vdde","vdd","vsse","vss",0);
  LOINS("po_sp","y3","out[3]","y[3]","cki","vdde","vdd","vssc","vss",0);

LOINS("addopt","adder1","vdd","vss","clkcore","sl","ina[3:0]","inb[3:0]","out[3:0]",0);

SAVE_LOFIG();
  }
```

Give the command at the command line:

```
genlib -v addchip
```

This creates a "addchip.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 "adder.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 "addchip.rin":

```
north ( clk sl b0 b1 b2 b3 )
west ( a0 a1 vssi a2 a3 )
south ( y0 y1 clkcore y2 y3 )
east ( vdd vddi vss )
```

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

Give the command at the command line:

```
ring addchip addchip
```

A "addchip.ap" file is created that can be examined by using **Graal**.

Static Timing Analysis

The "addchip.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 analyzer 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 addchip addchip
```

-v	-	verbose
-t	-	build the netlist to the transistor level.
first addchip	-	take the "addchip.ap" layout file as input.
second addchip	-	generate the "addchip.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 addchip

-tec - selects the technology file prol10.elp.

Layout Extraction and Netlist Comparison

The "addchip.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 addchip addchip
```

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

vst	-	take the first file in .vst format.
al	-	take the second file in .al format.
first addchip	-	"addchip.vst" file.
second addchip	-	"addchip.al" file.
-f	-	build the netlist to the standard cell level.
-o	-	to have ordered connectors in the output netlist

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.

Lvx 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 "addchip.al" can be simulated using **Asimut** and the test vector file that has been created to test the behavioural file "adder.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 addchip
```

-v	-	vectorize
addchip	-	Takes the "addchip.al" as input.

The extracted VHDL description is put in the file "addchip.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 "addchip.inf". If we do a "more" of the "addchip.vbe" file we see that the registers have a different name from the one that we have given in "adder.vbe".

Edit and save a file "addchip.inf" with the following:

```
rename
adder1.regstr_0.dff_s : regstr_0 ;
adder1.regstr_1.dff_s : regstr_1 ;
adder1.regstr_2.dff_s : regstr_2 ;
adder1.regstr_3.dff_s : regstr_3 ;
end
```

Give the command:

```
yagle -i -v addchip
```

-i - asks **Yagle** to read the "addchip.inf" file and rename the registers in the "addchip.vbe" file as given in the list.

Give the command:

```
proof -p -d adder addchip
```

-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.

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.

Give the command:

```
setenv RDS_TECHNO_NAME /alliance/archi/Linux_elf/etc/pro110_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 addchip addchip
```

-c	-	deletes connectors at the highest hierarchy. (Use man to see full description)
-v	-	verbose mode on
first addchip	-	"addchip.ap" file as input
second addchip	-	"addchip.cif" file as output.

This completes the design of the counter chip.

SEVENTH COURSE
ON
BASIC VLSI DESIGN TECHNIQUES

MICROPROCESSOR LABORATORY
ICTP-UNESCO

29 October- 23 November, 2001

Trieste, Italy

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'.

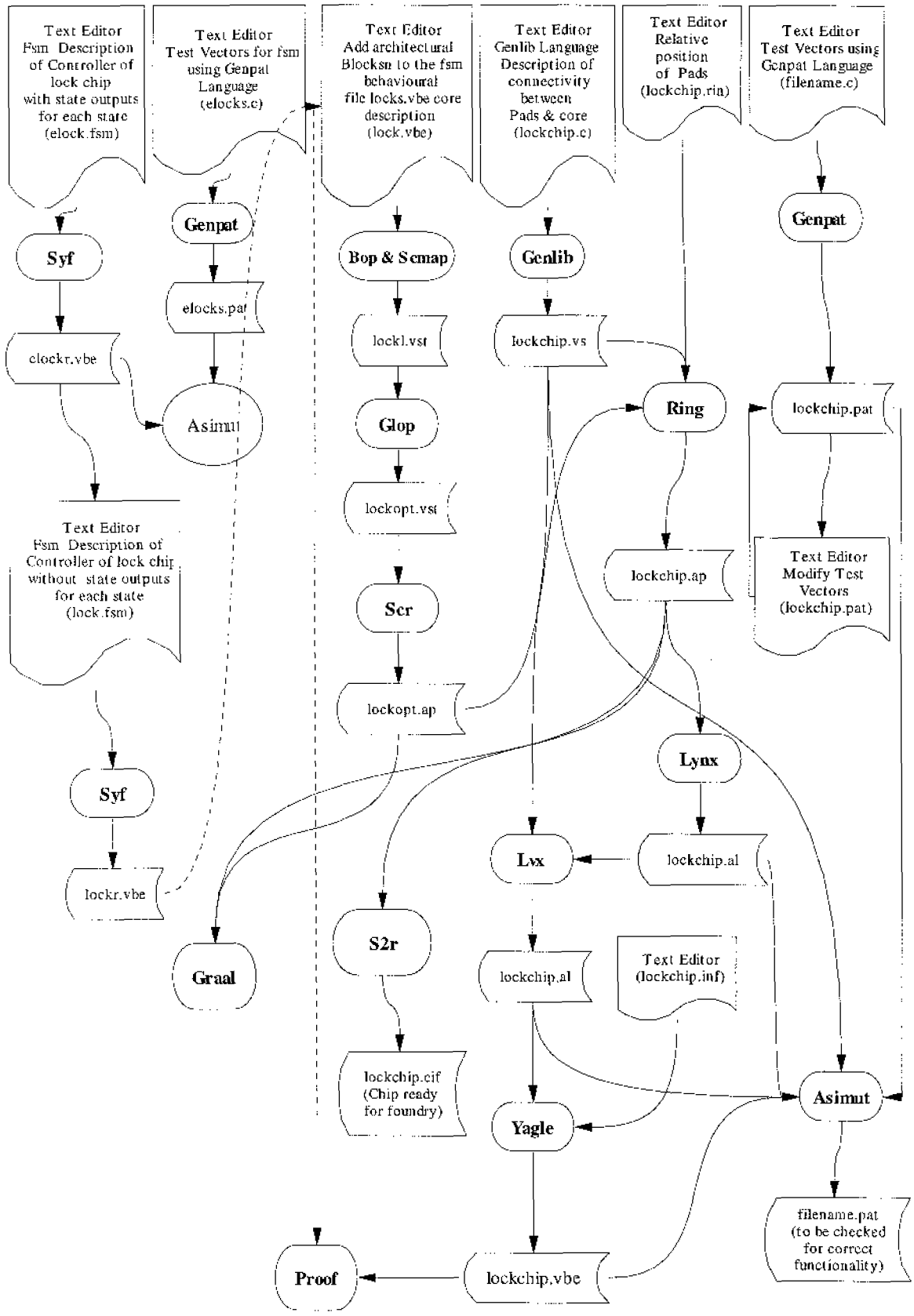


Fig 1. Design flow for the Hex Combination Lock

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.

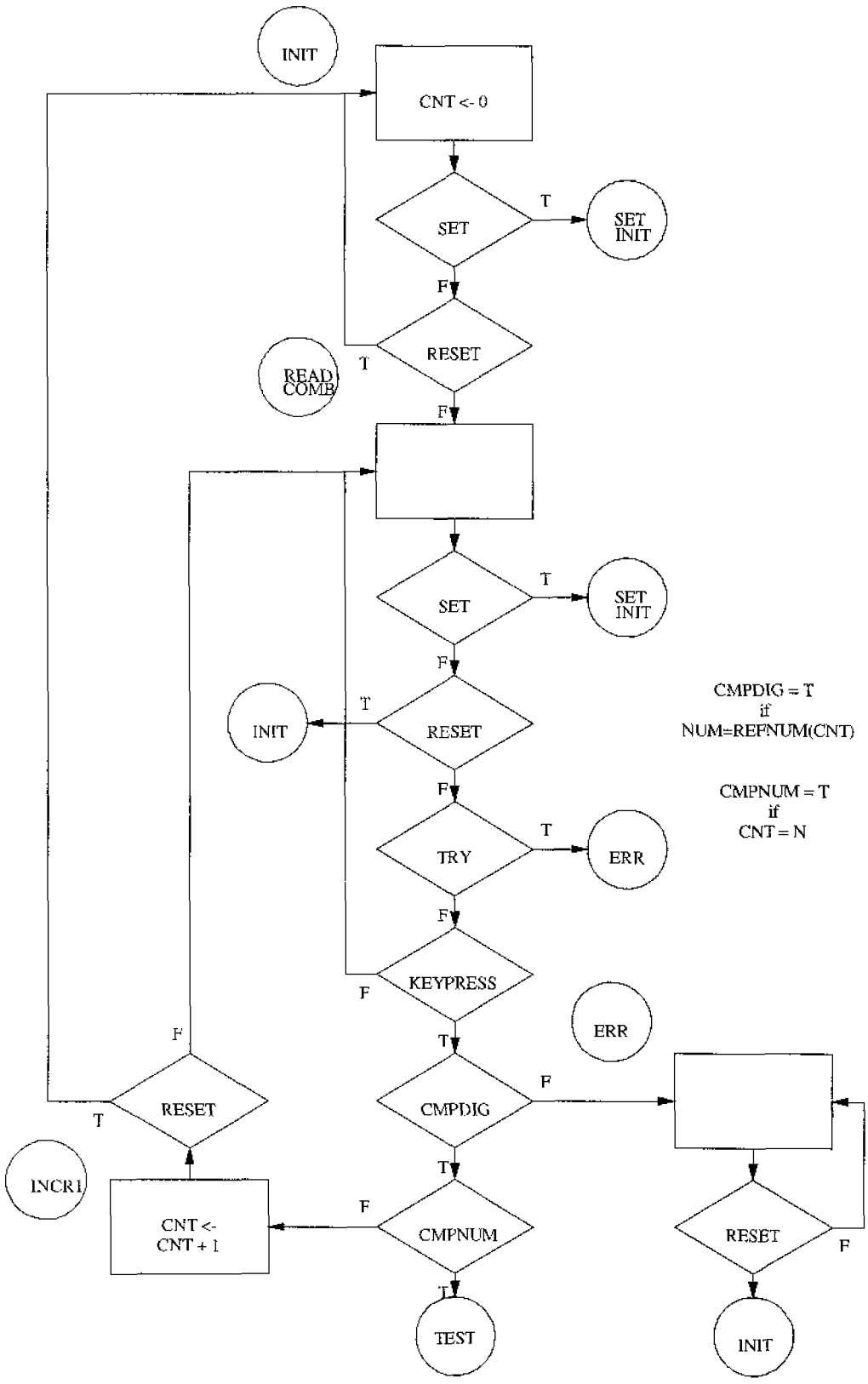


Fig.2 ASM chart of the Serial Combination Lock

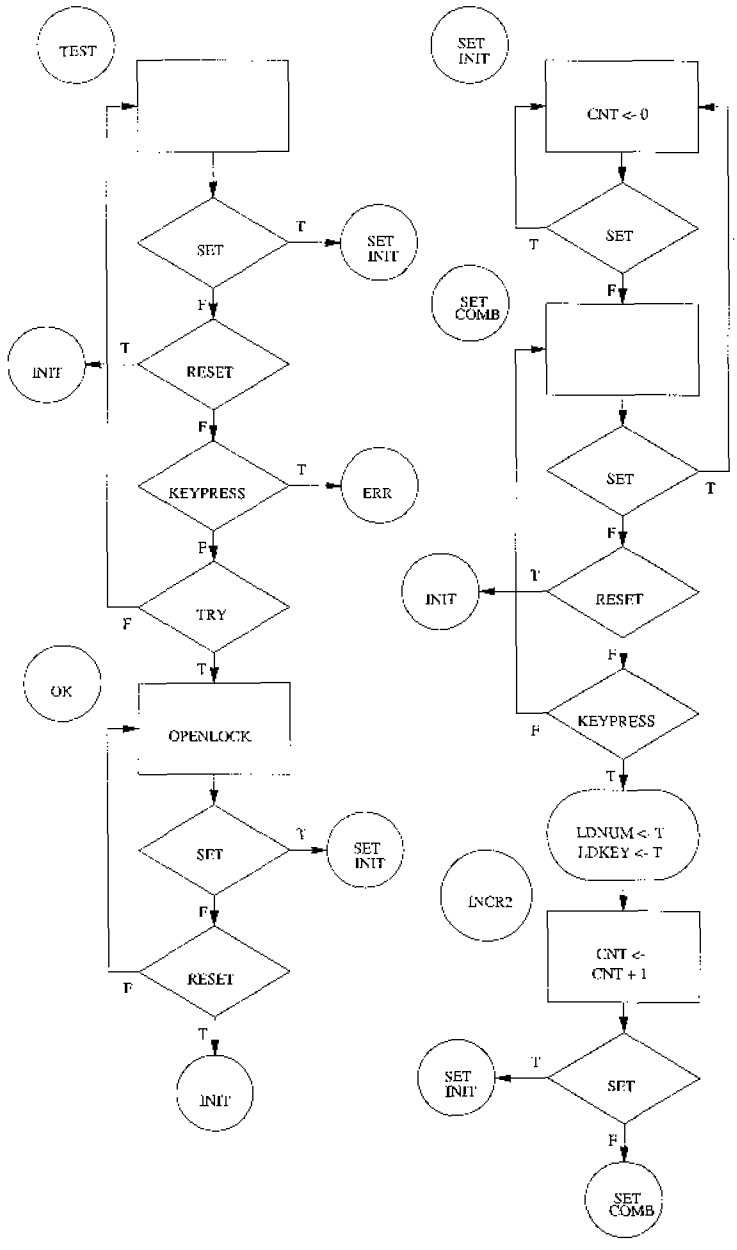


Fig. 2 (cont'd) ASM of the Serial Combination Lock

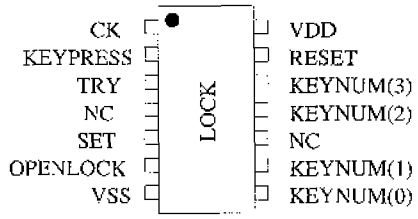


Fig. 3 Lock chip (a possible pinout diagram).

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;
        incent : out bit;
        resent : 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 "clock.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;
        incent : 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;
                        incent <= '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, incent, 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 incl_s : BIT;          -- incl_s
  SIGNAL incl_m : BIT;          -- incl_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 ((incent='1') and (counter = B"000"))
else
                       B"010" when ((incent='1') and (counter = B"001"))
else
                       B"011" when ((incent='1') and (counter = B"010"))
else
                       B"100" when ((incent='1') and (counter = B"011"))
else
                       B"101" when ((incent='1') and (counter = B"100"))
else
                       B"110" when ((incent='1') and (counter = B"101"))
else
                       B"111" when ((incent='1') and (counter = B"110"))
else
                       B"000" when ((incent='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;

incent <= ((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 (roset ) 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 lock1
```

This takes as input the "lock.vbe" description and creates an optimised Boolean behavioural description file "lock1.vbe".

To synthesise the structural description give the command:

```
scmap lock1 lock1
```

This takes as input the optimised behavioural description "lock1.vbe" and creates the structural description file "lock1.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
```

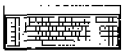
-p - placement option
-r - routing option

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","VSSE","
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","VSSE",
"VSS",0);

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

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

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


LOINS("po_sp","OPENLOCK","POPENLOCK","OPENLOCK","cki","VDDE","VDD","VSSE","
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.vhe". 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.

SEVENTH COURSE
ON
BASIC VLSI DESIGN TECHNIQUES

MICROPROCESSOR LABORATORY
ICTP-UNESCO

29 October- 23 November, 2001

Trieste, Italy

Exercise 7

Adder Accumulator using Datapath Entities

Problem Description

Re-design the 4-bit adder accumulator that you designed in Exercise 4 using datapath entities. In this design example you will learn to:

- Describe the circuit in the **Fpgen** language.
- Compile and generate the structural description using **Fpgen**.
- Place and route the chip and generate the layout file using **Dpr**.
- Place the pads using **Genlib** language and generate the structural description with pads.
- Simulate the structural using **Asimut**.
- Place and route the pads using **Ring**.
- Extract back the circuit from the layout using **Lynx**.
- Use **Tas** to perform the static timing analysis.
- Make a layout verification by comparing the extracted netlist with the structural description using **Lvx**.

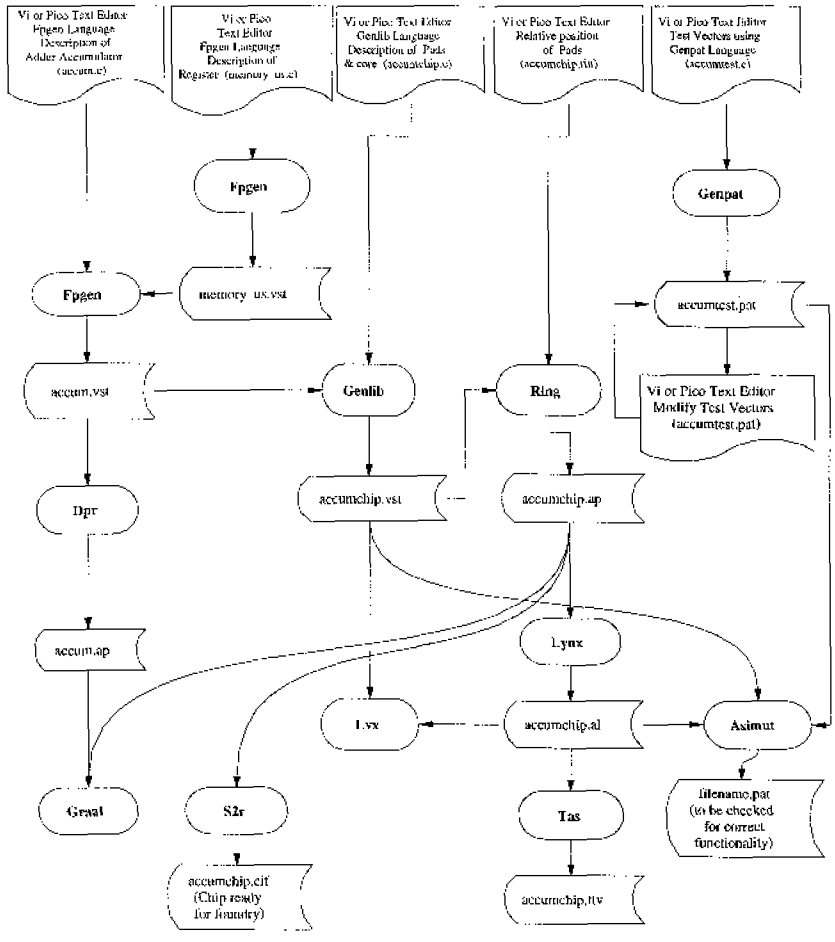


Fig 1. Design Flow for the Adder Accumulator

A 4-Bit Adder Accumulator Using Data Path Entities

This design example differs from other examples in the sense that you will not make the behavioural description of the circuit. Instead the circuit will be described as a netlist of components from the data-path elements library. Fig 1 below shows the block diagram of the circuit with the components from the data-path library with names of the intermediate nodes. This circuit will be translated into the **Fpgen** language. Fig 3 shows a possible pin out for the chip. Table 1 summarises the function of the chip.

4 Bit Slices

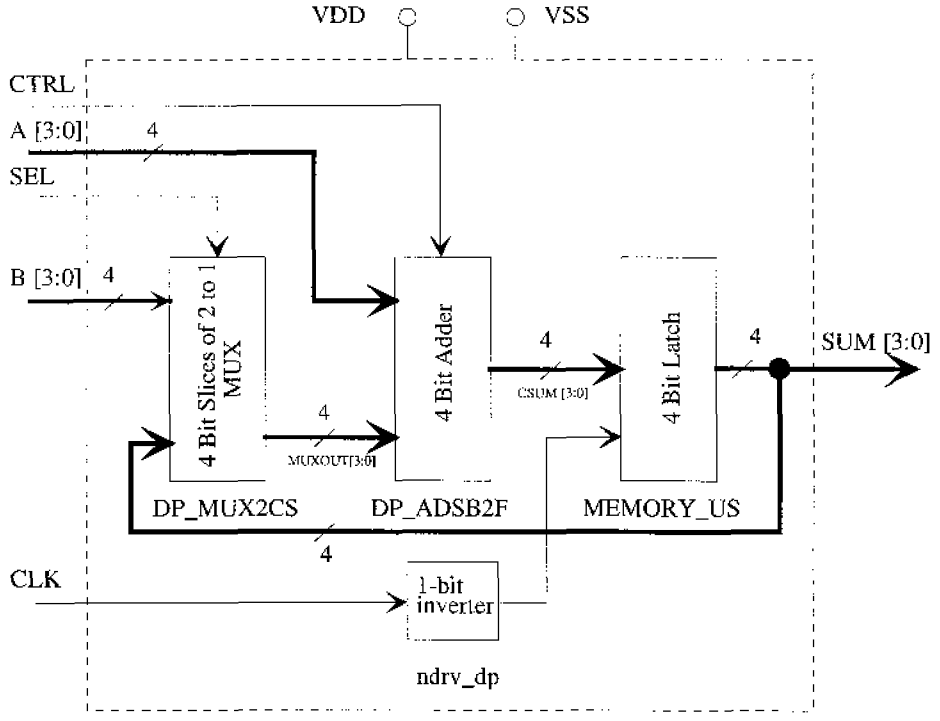


Fig 2. Block Diagram of the Adder Accumulator using Data Path Components.

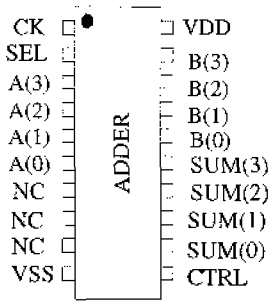


Fig. 3 The Adder Accumulator chip (a possible pinout diagram).

CLK	SEL	CTRL	SUM
Rising Edge	0	0	A + B
Rising Edge	1	0	A + SUM
No Rising Edge	X	X	SUM

Table 1. Truth Table for the 4-bit presetable counter

When the SEL is '0' two 4-Bit numbers A and B are added, latched and presented, at the 4-Bit SUM output at the rising edge of the CLK. When SEL is '1' the SUM output is fed back to the adder and is added with A. The value is latched and presented at the SUM output at the rising edge of the CLK.

Solution

Legend

commands are to be given at the command line.

Edit and save into a file

Explanation of a topic

Environmental variables are to be set

Creating the Design

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

```
mkdir accum
```

Change into this directory:

```
cd accum
```

Create with the text editor a file called "accum.c". Enter the following and save the file.

```
#include <genlib.h>
#include <fpgen.h>

main()
{
DP_DEFLOFIG("ACCUM", 4, LSB_INDEX_ZERO);

/*  Interface declaration  */

printf( "Interface\n" );

 DP_LOCON("vdd",IN,"vdd");
 DP_LOCON("vss",IN,"vss");
 DP_LOCON("A[3:0]",IN,"A[3:0]");
 DP_LOCON("B[3:0]",IN,"B[3:0]");
 DP_LOCON("SUM[3:0]",OUT,"SUM[3:0]");
 DP_LOCON("SEL",IN,"SEL");
 DP_LOCON("CLK",IN,"CLK");
 DP_LOCON("CTRL",IN,"CTRL");

           DP_MUX2CS(        "MUXINST",
                              4,
                              0,
                              "SEL",
                              "SUM[3:0]",
                              "B[3:0]",
                              "MUXOUT[3:0]",
                               EOL  );

           DP_ADSB2F(         "ADDER",
                              "A[3:0]",
                              "MUXOUT[3:0]",
                              "CARRY",
                              "OVF",
                              "CSUM[3:0]",
                              "CTRL",
                                EOL  );

/* heterogeneous operator */
LOINS ("ndrv_dp","CLKINV","CLK","NCLK","vdd","vss",0);

DP_IMPORT("memory_us","MEMINS","CSUM[3:0]","SUM[3:0]","NCLK",EOL);

DP_SAVLOFIG();

        exit( 0 );
}
```

Fpgen is a set of **C** functions dedicated to data path synthesis. **Fpgen** creates a *hierarchical netlist* that can be given to the data path route tool **Dpr**.

To compile with **Fpgen**, two include files, "genlib.h" and "fpgen.h" are required which have to be declared through the **C** include statement at the top of the file. Then the circuit is described inside a procedure like any normal main procedure in **C**.

```
main()
{

        Here is your circuit description.

        exit(0);

}
```

Inside the main procedure , the circuit is described as **macro-functions**. The man pages of **fpgen** or **fplib** (man fpgen or man fplib) contains a list of **macro-functions** that are allowed inside the main procedure. The macro-functions consists of gate level logical functions like **inverter**, **and**, **or**, **xor**, etc. It also consists of generator functions like **adder** and **barrel shifter**. Register function like **Dflip-flop** is also provided. With these functions most data paths can be constructed.

Each of the **macro-functions** has its man pages and it is recommended that they be consulted before the circuit is constructed !.

Coming to our circuit, the adder accumulator has been described in the above file. In this file the **DP_IMPORT** function has been use to *instanciate* a part called "*memory_us*" that has been constructed out of heterogeneous functions. We have to generate this file too, if our circuit has to work. The man pages of **dplib** (man dplib) gives a list of heterogeneous operators that are allowed. The man pages of a particular heterogeneous operator gives in detail the order and type of the arguments for that operator (e.g. man **ms_dp**).

Create with the text editor a file called "memory_us.c". Enter the following and save the file. This file describes the 4-bit edge triggered register that has been built from a heterogeneous block "ms_dp". The instance name of the heterogeneous operator **ms_dp** end with a "_#" so that the data path router, **Dpr** knows that "#" is the slice number (the level) at which the block is to be placed.

```
#include <genlib.h>
#include <fpgen.h>

main ()
{
  /* creating a new data-path figure for accumulator-adder */
  DEF_LOFIG("memory_us");

  /* logical connectors */
  LOCON("i[3:0]", IN ,"i[3:0]");
  LOCON("o[3:0]", OUT ,"o[3:0]");
```

```
  LOCON("clk", IN ,"clk");
  LOCON("vdd", IN ,"vdd");
  LOCON("vss", IN ,"vss");

  /* data path netlist description */
  LOINS("ms_dp","mem_0","i[0]","clk","o[0]","vdd","vss", EOL);
  LOINS("ms_dp","mem_1","i[1]","clk","o[1]","vdd","vss", EOL);
  LOINS("ms_dp","mem_2","i[2]","clk","o[2]","vdd","vss", EOL);
  LOINS("ms_dp","mem_3","i[3]","clk","o[3]","vdd","vss", EOL);

  /* save the model on disk */
  SAVE_LOFIG();

 }
```

Set the environmental variables as shown below.

```
setenv MBK_CATA_LIB .:/alliance/archi/Linux_elf/cells/fplib:
                      /alliance/archi/Linux_elf/cells/dplib:
                      /alliance/archi/Linux_elf/cells/rsa:
                      /alliance/archi/Linux_elf/cells/sclib:
                      /alliance/archi/Linux_elf/cells/padlib

setenv MBK_IN_LO  vst
setenv MBK_OUT_LO  vst
setenv MBK_IN_PH  ap
setenv MBK_OUT_PH  ap
setenv MBK_WORK_LIB .
```

```
fpgen  -v memory_us
```

Give the following command at the command line.

```
fpgen -v accum
```

These commands generate the structural of the respective parts of the hierarchy.

Placement and Routing of the core

The core consisting of datapath elements, is routed using the data path router **Dpr**.
This tool can use some information from a *<filename>.dpr* file in order to customise the resulting layout. By mean of this file it is possible to define the abutment-box, the width of the power supplies tracks, the exact position of the connectors, etc. Type *man dpr* for a detailed information on.

Create this small file called "accum.dpr". Enter the following and save the file.

```
DP_DEFAB -20 +20
DP_POWER  0  10
```

Now give the following command at the command line:

```
dpr -p -r accum accum
```

-p	-	placement option
-r	-	routing option

The "accum.ap" file is created which can be viewed with **Graal**.

Describing the Pads and Core using the Procedural Design Language

When the chip is described physically in Alliance, it consists of two separate parts that are brought together, the core and the pads. In Alliance, the core and the pads are brought together in a **C** description file. This file when treated with **Genlib**, produces the structural description of the chip with the pads. The pads are placed physically, one by one in the **C** file. Placing the pads require the structural and functional knowledge of the pads. One of the types of pads that is used in this example is the pvsseck_sp, a cell of PAD-Lib, a library of pads provided with Alliance.

Give the following command at the command line to see a description of this pad.

```
man pvsseck_sp
```

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, edit and save into the file "accumchip.c" the following:

```
#include <genlib.h>
main()
{
 DEF_LOFIG("accumchip");
 LOCON("a[3:0]",'I',"a[3:0]");
 LOCON("b[3:0]",'I',"b[3:0]");
 LOCON("y[3:0]",'O',"y[3:0]");
 LOCON("sel",'I',"sel");
 LOCON("ck",'I',"ck");
 LOCON("ctrl",'I',"ctrl");
 LOCON("vdde",'I',"vdde");
 LOCON("vsse",'I',"vsse");
 LOCON("vdd",'I',"vdd");
 LOCON("vss",'I',"vss");
```

```
/* 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 ("pvssi_sp", "Vssi", "cki", "vdde", "vdd", "vsse", "vss", 0);
   LOINS ("pvddi_sp", "Vddi", "cki", "vdde", "vdd", "vsse", "vss", 0);

   LOINS("pi_sp","si","si","si","cki","vdde","vdd","vsse","vss",0);
   LOINS("pi_sp","ct","ctrl","ct","cki","vdde","vdd","vsse","vss",0);

   LOINS("pck_sp", "clk", "ck", "cki", "vdde", "vdd", "vsse", "vss", 0);

   LOINS("pvsseck_sp", "clkcore",
         "clkcore", "cki",
         "vdde", "vdd", "vsse", "vss", 0);

   LOINS("pi_sp","a0","a[0]","ina[0]","cki","vdde","vdd","vsse","vss",0);
   LOINS("pi_sp","a1","a[1]","ina[1]","cki","vdde","vdd","vsse","vss",0);
   LOINS("pi_sp","a2","a[2]","ina[2]","cki","vdde","vdd","vsse","vss",0);
   LOINS("pi_sp","a3","a[3]","ina[3]","cki","vdde","vdd","vsse","vss",0);

   LOINS("pi_sp","b0","b[0]","inb[0]","cki","vdde","vdd","vsse","vss",0);
   LOINS("pi_sp","b1","b[1]","inb[1]","cki","vdde","vdd","vsse","vss",0);
   LOINS("pi_sp","b2","b[2]","inb[2]","cki","vdde","vdd","vsse","vss",0);
   LOINS("pi_sp","b3","b[3]","inb[3]","cki","vdde","vdd","vsse","vss",0);

   LOINS("po_sp","y0","out[0]","y[0]","cki","vdde","vdd","vsse","vss",0);
   LOINS("po_sp","y1","out[1]","y[1]","cki","vdde","vdd","vsse","vss",0);
   LOINS("po_sp","y2","out[2]","y[2]","cki","vdde","vdd","vsse","vss",0);
   LOINS("po_sp","y3","out[3]","y[3]","cki","vdde","vdd","vsse","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("accum","core","vdd","vss","ina[3:0]","inb[3:0]","out[3:0]","si","clk
core","ct",0);

SAVE_LOFIG();

}
```

```
genlib -v accumchip
```

This creates a "accumchip.vst" structural description file with pads.

Test Pattern Generation and Simulation of the Structural Description

Write a pattern file for simulation and validation with **Asimut**.
Check that the adder accumulator performs satisfactorily.

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 "accumchip.rin":

```
east ( clk sl b0 b1 b2 b3 )
south ( a0 al vssi a2 a3 )
west ( y0 y1 clkcore y2 y3)
north ( vdd vddi ct vss  )
```

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

Give the command at the command line:

```
ring accumchip accumchip
```

The "accumchip.ap" file is created that can be examined by using **Graal**.

Examine the layout using **Graal**.

Static Timing analysis

The "accumchip.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 accumchip accumchip
```

-v	-	verbose
-t	-	build the netlist to the transistor level.
first accumchip	-	take the "accumchip.ap" layout file as input.
second accumchip		generate the "accumchip.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 accumchip

-tec - selects the technology file prol10.elp.

Layout Extraction and Netlist Comparison

The "accumchip.ap" contains the layout information. However we do not know if the physical description produced reflect the initial 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.

Set the environmental variable **MBK_OUT_LO** as shown below:

```
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 accumchip accumchip
```

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

-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 accumchip	-	"accumchip.vst" file.
second accumchip	-	"accumchip.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** 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 "accumchip.al" can be simulated using **Asimut** and the test vector file that has been created to test the structural file "accumchip.vst".

Give the following command at the command line:

```
setenv MBK_IN_LO al
```

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

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**.

Set the environmental variables, as shown below:

```
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 generic CMOS process whose technology file is the prol10.rds. The output format of the chip is in *cif* format. The symbolic pads are replaced with their real equivalent. The pads due to their technology dependence are maintained as a *cif* file in the library.

Give the command:

```
s2r -cv accumchip accumchip
```

-c	-	deletes connectors at the highest hierarchy. (Use man to see full description)
-v	-	verbose mode on
first accumchip	-	"accumchip.ap" file as input
second accumchip	-	"accumchip.cif" file as output.

This completes the design of the adder accumulator chip.

SEVENTH COURSE
ON
BASIC VLSI DESIGN TECHNIQUES

MICROPROCESSOR LABORATORY
ICTP-UNESCO

29 October- 23 November, 2001

Trieste, Italy

Project

Design of a Programmable Traffic Signal Controller

PROGRAMMABLE TRAFFIC SIGNAL CONTROLLER

INTRODUCTION

In any city, the streets constitute a complex urban network and there are many "traffic signal" nodes in this network, in such a way that they put some order to the traffic increasing the safety and the "efficiency". The concept of "efficiency" is not very well defined, but everybody has an intuitive idea about what "traffic efficiency" means. To define accurately what "Vehicular Traffic efficiency" is, it is important to establish what the objective parameters are that permit to us to evaluate the quality of the vehicular traffic. These are parameters that we should be able to measure. The problem doesn't finish here because every individual interested in "efficiency" and "optimality" of the network expects a different thing. For example if we take as parameter of quality of the traffic like the average velocity of the cars in the urban network, the drivers would like it to be high, but the pedestrians will like it to be low for security reasons. This example shows that the problem is not only technical but also political, in the sense that a city administration may decide the definition of optimum.

However, once a criterion is fixed to evaluate the quality of the vehicular traffic, it is important to have the means to bring the traffic towards an optimal condition. Among the means to reach that situation, we have the "traffic signal" (TS), the experiences of which indicate its extreme importance. Then the quality of the vehicular traffic is sensitive to how the "traffic signals" are configured. We will assume that configuration of the "traffic signal lights", as the set of parameters that characterises completely the state of the traffic signals of the network.

There are normally two modes in which the light traffic works. They are the intermittent yellow and the cyclic mode that alternates between yellow, red and green. In the last case, the colour is a periodic function of the time. We need four parameters to characterise this function: the duration of each colour (3 parameters), and the phase of the signal, e.g. the instant in which the yellow, for a determined street, starts.

We assume a simple traffic light (in the sense that it regulates only two crossing streets) and of equal duration for yellow on the two streets. With this assumption, we propose a "programmable light traffic controller". This device is capable of receiving information containing the working mode and the colour's duration, which is updated when a synchronisation signal arrives. This controller is capable of avoiding dangerous and traumatic situations of discontinuity in the traffic. The instant at which **syncro** (synchronisation signal) arrives, fixes the phase of the traffic light. With the signal **syncro**, also arrive the working mode (**mode**) and the colour's duration: **tyel**, **tred** and **tgre** (duration of yellow, red and green respectively).

We are looking for a device with the external ports as shown in Fig. 1. Table 1. shows how the internal registers can be set-up for operation. Table 2. essentially gives the output associated with each of the states of the state machine. The state machine flow diagrams are described in the following pages.

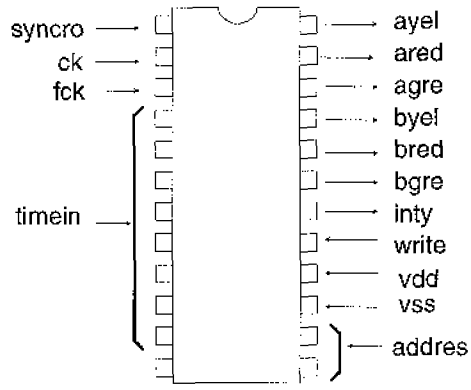


Fig. 1. Possible pinout of the Programmable Traffic Controller Chip

FCK	write	address	tyel(7-0)	tred(7-0)	tgre(7-0)	mode(1-0)
L to H	L	X	X	X	X	X
L to H	H	'00'	timein(7-0)	X	X	X
L to H	H	'01'	X	timein(7-0)	X	X
L to H	H	'10'	X	X	timein(7-0)	X
L to H	H	'11'	X	X	X	timein(1-0)

Table 1. Setting up of the internal registers.

CK	MODE	STATE	duration (In number of periods of 'ck')	ayel	ared	agre	byel	bred	bgre	inty
L to H	'01'	INTYEL	permanent	L	L	L	L	L	L	H
L to H	'00'	YELLOW	tyel	H	L	L	L	H	L	L
L to H	'00'	RED	tred	L	H	L	L	L	H	L
L to H	'00'	REDINT	tyel	L	H	L	L	L	H	L
L to H	'00'	REDYEL	tyel	L	H	L	H	L	L	L
L to H	'00'	GREEN	tgre	L	L	H	L	H	L	L
L to H	'10'	RED	permanent	L	H	L	L	L	H	L
L to H	'11'	GREEN	permanent	1	L	H	L	H	L	L

Table 2. The outputs that are associated with each state

We will divide the complete architecture in three parts: **(1)** a synchronous variable mod counter, **(2)** the registers of the data (times plus mode) controlled by an address

(**address**) and an external fast clock (**fck**), and (**3**) the logic part to control the state of the lamps.

The Algorithmic Finite State Machine

We start with some definitions:

- The device has clock (**ck**) as input, its period will be taken as the unit of time in which the duration will be expressed, and a fast clock (**fck**) used to introduce the data into the internal registers.

- The state of the TS is defined by the state of each colour (**ayel, ared, byel, bgre**, etc.) and its duration (**timef**).

- **mode**:(1 downto 0), indicates the working mode.

  mode <= '00' normal cyclic mode
  mode <= '01' intermittent yellow
  mode <= '10' permanent red (in a determined direction)
  mode <= '11' permanent green (in the same determined direction)

- **timef** : (7 downto 0) , indicates the normal duration of the state. In some cases that the FSM (finite state machine) goes into a new state, the counter is reset to zero putting the signal **cntreset = '1'** . When the counter reaches the value of **timef** the flag **cntflag** is raised (**cntflag = '1'**) and this fact will be used to decide about the change of the state. The values that **timef** can assume are: **tyel, tred** or **tgre.**

- For the street A

  ayel <= '1' means "on" , '0' means "off"
  ared <= '1' means "on" , '0' means "off"
  agre <= '1' means "on" , '0' means "off"

  The same for the street B with **byel, bred** and **bgre**
  and for both streets
  inty <= '1' means "on" , '0' means "off"

- For us the lights **ayel** , **ared** and **agre** are mutually exclusive (in Trieste the style is **ayel** and **agre** at the same time slightly before the end of **agre,** and remains till the end when the signal changes to red). The same for the other street. The signal **intyel** is incompatible with the other lights.

- The situation : **agre** <= '1' & **bgre** <= '1' is absolutely forbidden.

- The change from green to red must be done by means of an intermediate yellow

- Each time that the configuration is updated, the new parameters must be simultaneously provided to the chip along with a **syncro** signal with duration of one clock period (**ck** single pulse).

- There are six States of the Finite State Machine (SFSM) called: INTYEL, YELLOW, RED, REDYELL, GREEN and REDINT. In each SFSM the colours have a specified constant state ('on' or 'off'), then they change only if the state changes.

In the Fig. 2. below we represent the colours on each street, as a function of time, with the different cases in which the synchronisation signal can arrive.

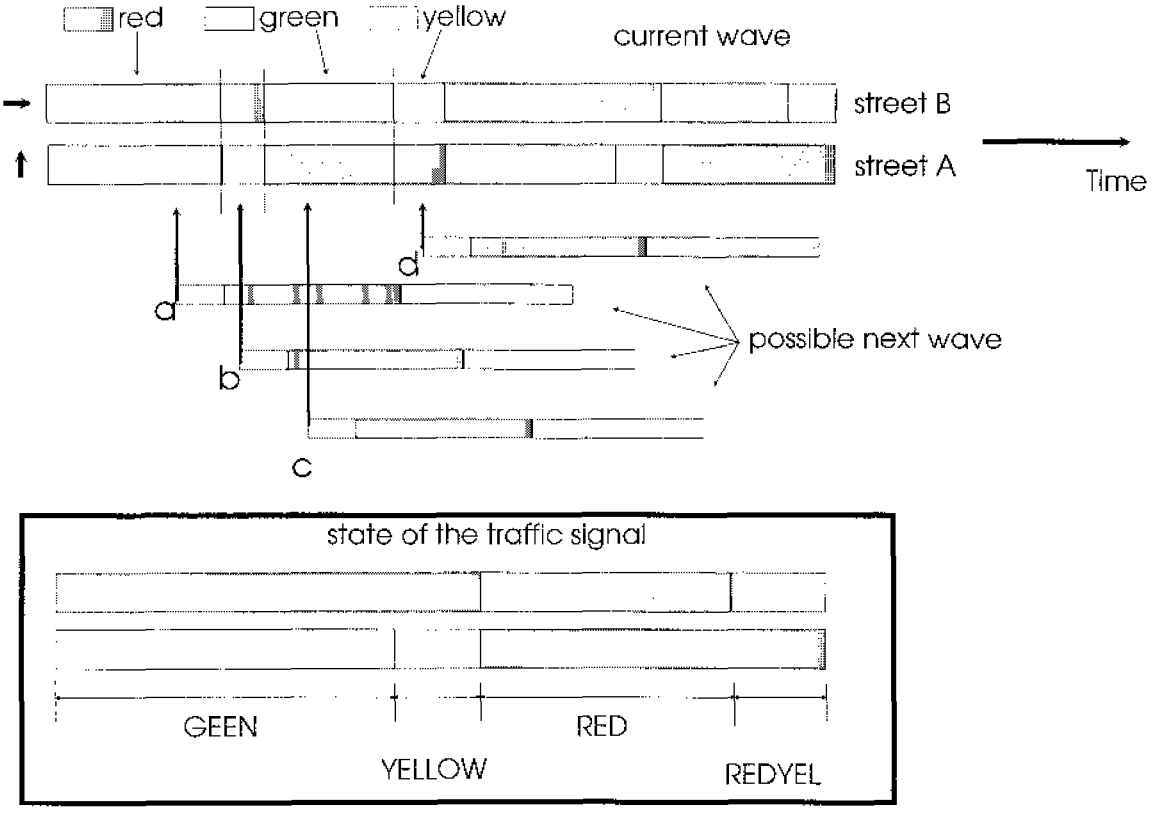
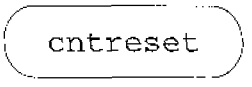


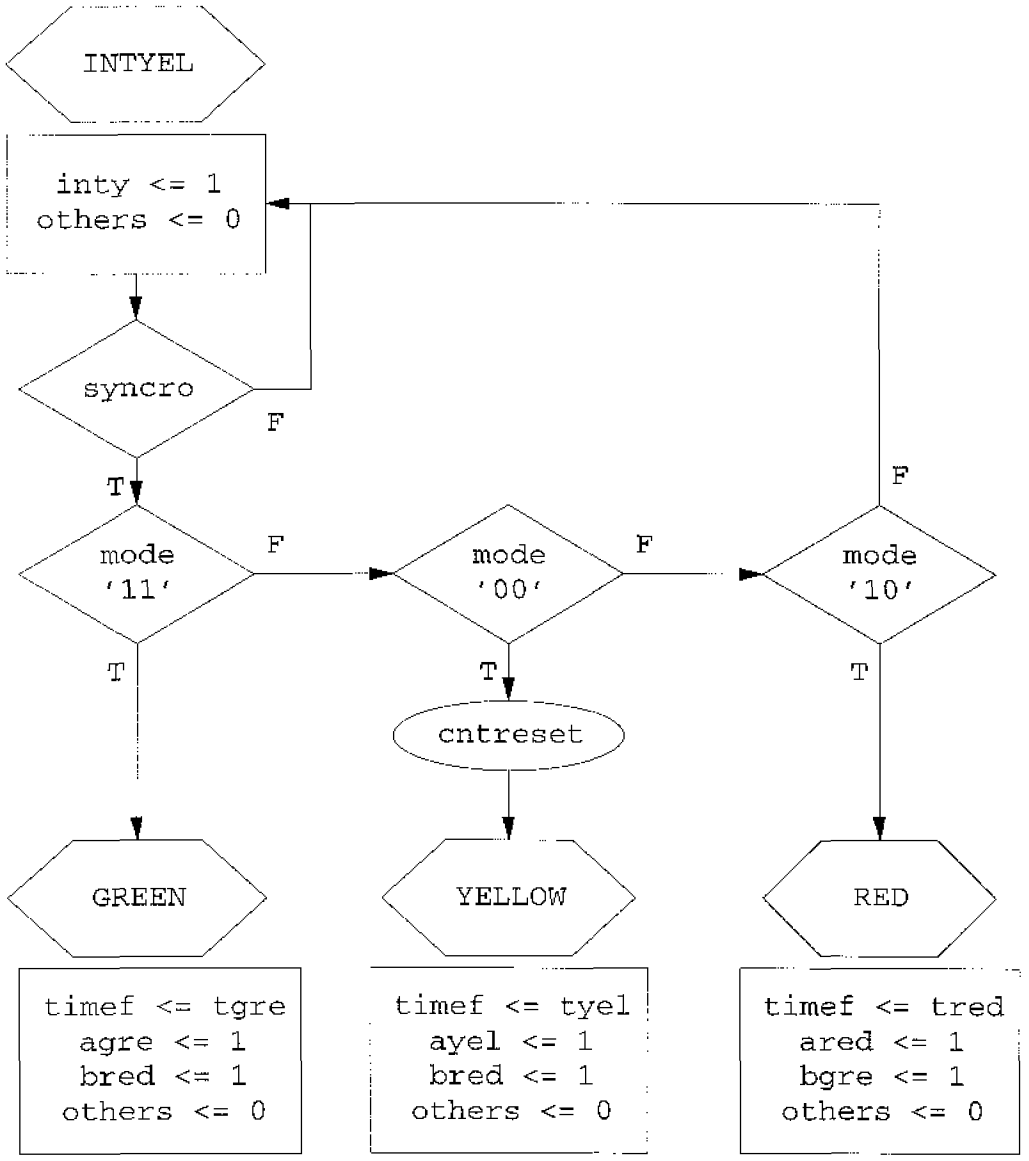
Fig. 2. Timing Relation between the Traffic signals

In the following flow diagrams we represent with circles the name of each state and within the lower rectangle the inherent constant values associated with each SFSM, with diamond the conditionals and with T the actual direction when the condition is truth, the same for F when the condition is false. The symbol:



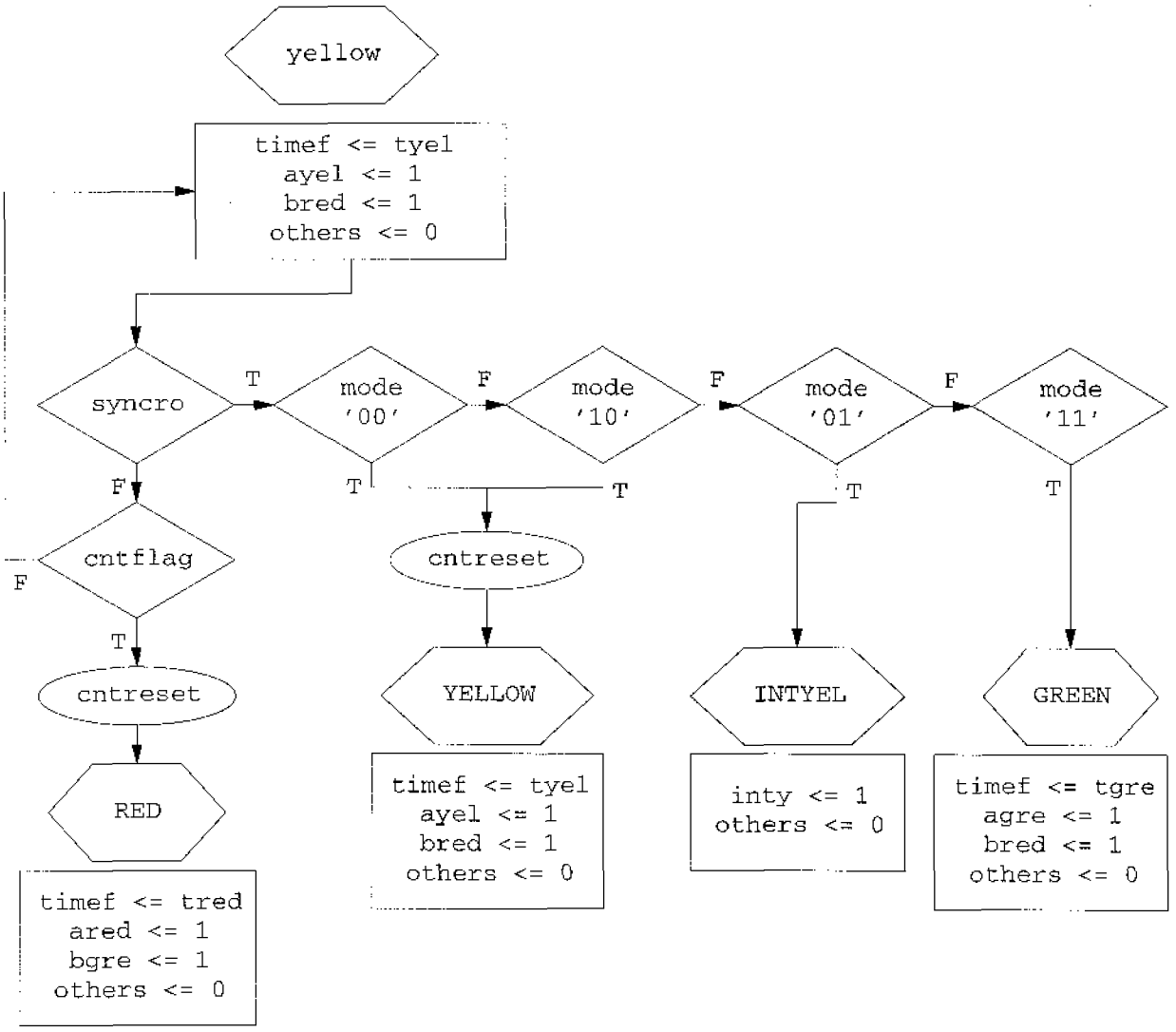
means that the internal signal **cntreset** takes instantaneously the value '1' when the condition is reached, resetting the synchronous counter when the state changes. The following flow diagrams show the transition conditions from each state:

State INTYEL:



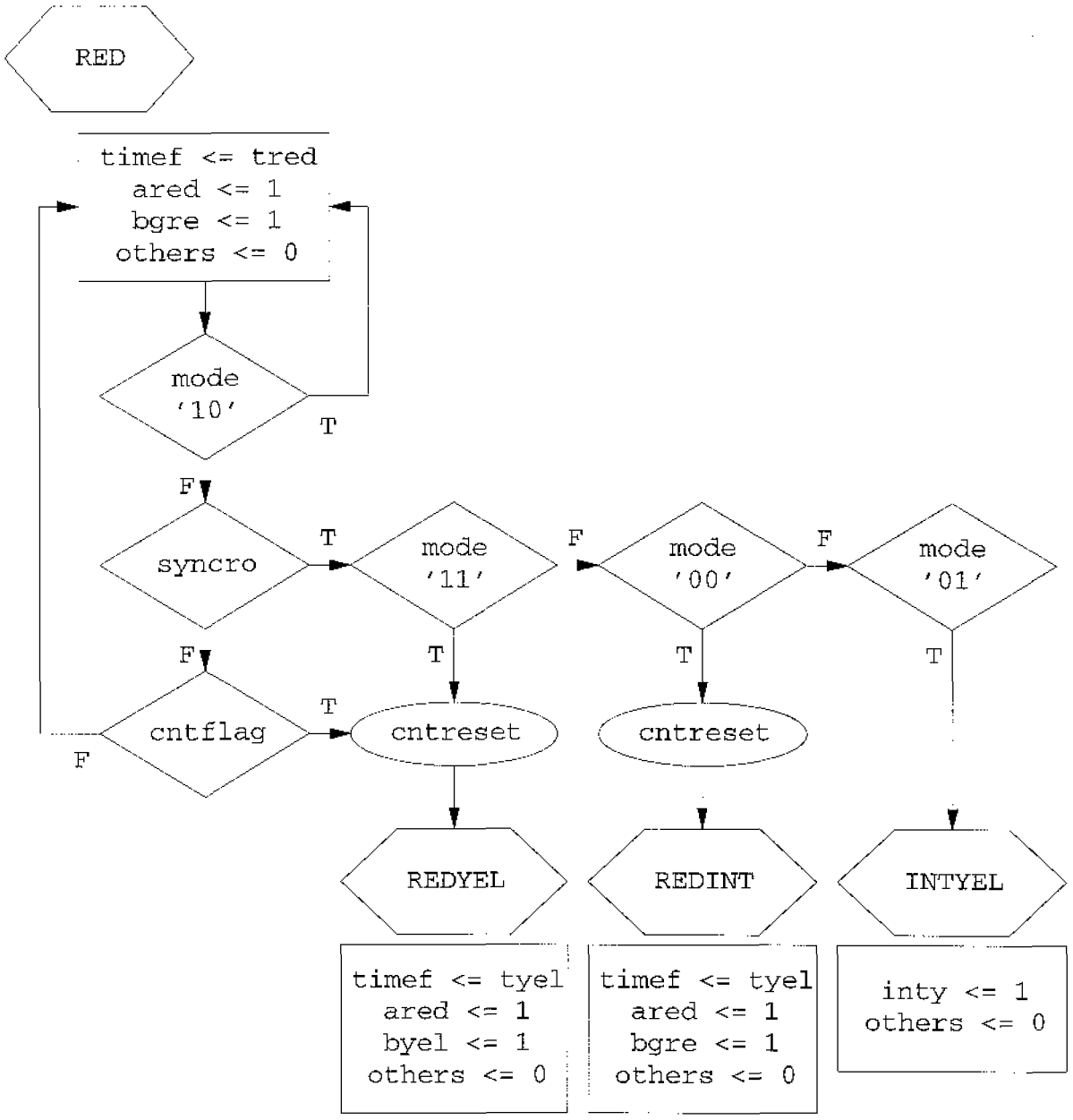
The state INTYEL corresponds to the intermittent yellow for both directions and will remain in such state until the signal **syncro** arrives together with the new working mode. If the new mode is permanent green or permanent red then the state of the counter is irrelevant. If the new mode is the normal cyclic mode then the new state will be yellow and immediately before this change the counter must be reset to zero in order to control the exact duration of the yellow state.

State YELLOW :



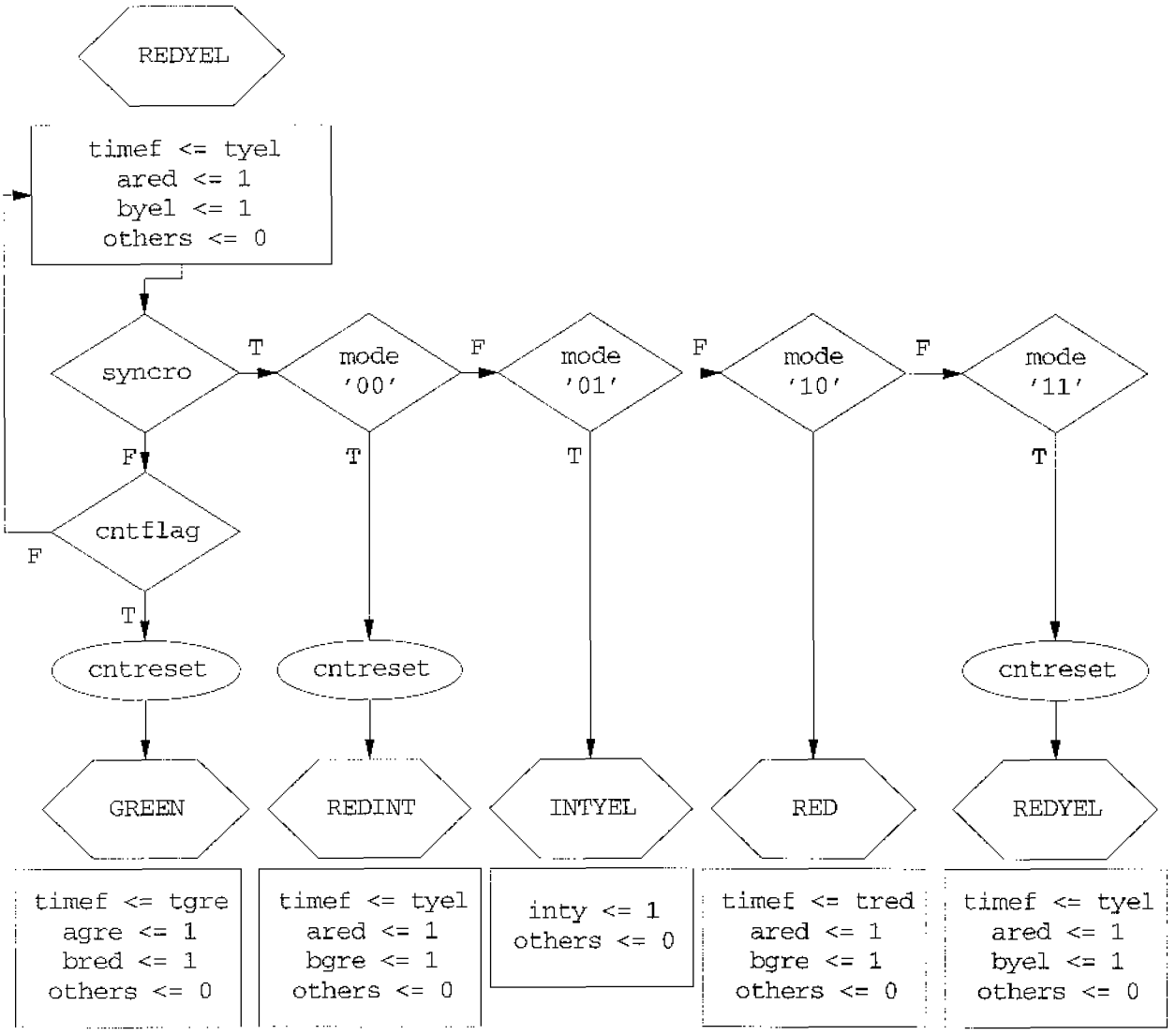
If the machine is in the yellow state and the **syncro** signal arrives, with the normal cyclic mode of operation, then the counter is reset to zero and the machine waits until the **cntflag** arrives, so as to change to the red state. If the new mode is permanent red then, as before, we put the counter to zero so as to be sure that at least the prescribed yellow duration be granted.

State RED:



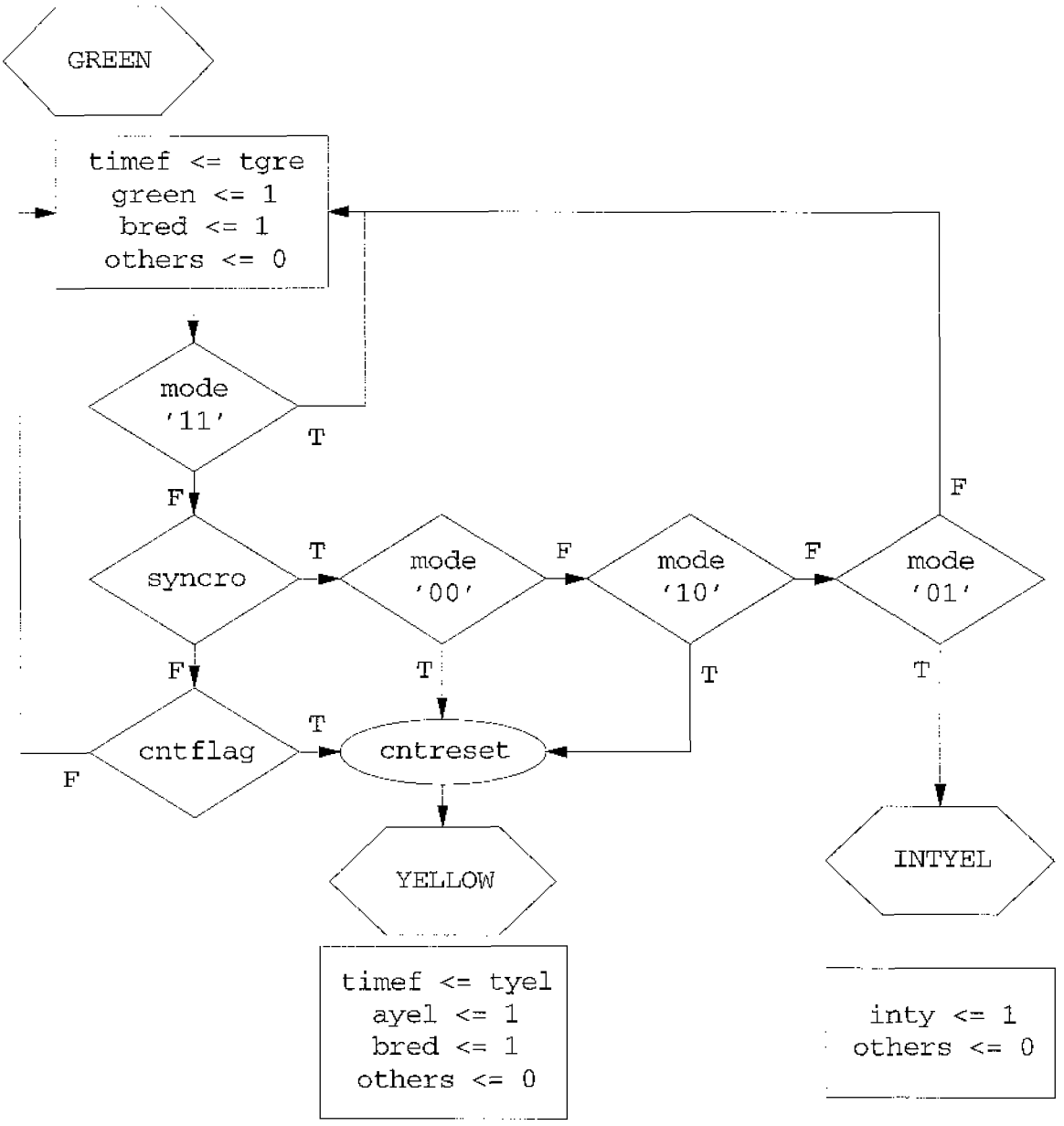
In this state, if the first conditional is true, the machine cycles permanently in this state. If the mode is different from "10" and the synchronisation signal (**syncro**) is not present, then the machine waits for the **cntflag** so as to change to the state REDYEL, while resetting the counter. If with the **syncro**, the normal cyclic mode arrives, then the next state will be REDINT, i.e. a red state but with a duration of the yellow, in order to fit the new "colour wave" without discontinuity, prolonging the actual state of the colours (see Fig. 2. a).

State REDYEL:



In this state if the **syncro** signal is not present, then the machine waits for the **cntflag,** changing to the green state after the prescribed period. If with the **syncro**, the mode is '00' (cyclic) then the next state will be REDINT (fitting the new wave) (See Fig. 2. b). If the new mode is permanent green then the state will be again REDYEL but with the counter starting from zero (to grant the duration prescribed for the state REDYEL) before the change to permanent green.

State GREEN:


GREEN
timef <= tgre
green <= 1
bred <= 1
others <= 0
mode '11'
T
F
syncro
T
mode '00'
F
mode '10'
F
mode '01'
F
T
T
T
F
T
F
cntflag
T
cntreset
YELLOW
INTYEL
timef <= tyel
ayel <= 1
bred <= 1
others <= 0
inty <= 1
others <= 0

State REDINT: