First steps in using **Alliance** the **addaccu** tutorial

Abstract

*This tutorial introduces the design flow to be used in the **Alliance** CAD framework for the design and verification of a standard cells circuit, including the pads. Each step of the desgin flow is supported by one or more specific tools, whose use is briefly explained.*

This text is meant to be simple and comprehensive, and is to be used to get into *the system. Should something be unclear or wrong, please indicate this by sending an e-mail to* `alliance-support@lip6.fr`.

1 Introduction

In this tutorial, you will learn the practical use of some basic **Alliance** tools by building a very simple circuit from scratch. It is recommended that you read the `overview.ps` file before proceeding, as it describes the main steps of the design conceptually.

Before we proceed to the tutorial, you must make sure that the **Alliance** tools are readilly available when invoking them at the prompt. The prompt in represented in the following text by the symbol :

```
~fred/addaccu %-)
```

In this system, `fred` is the user, `addaccu` is the current directory, and `%-)` is supposed to give us courage!

Try issuing the following command to check that **Alliance** is correctly installed:

```
~fred/addaccu %-) yagle
```

If everything is working, you get the following result:

```
YAGLE 2.00
Usage   : yagle [options] <file1> [file2]
Options : -i     reads the '.inf' file
          -v     vectorises the vhdl description
          -p=n   n is the depth for functional analysis
          -nc    no detection of complex gates
          -nl    no latch detection
          -fcl   transistor netlist detection
          -elp   use the technology file '.elp'
          -d     generates a .cns file
          -b     transistor orientation
          -z     functional analysis through HZ nodes
          -os    only one vdd and vss in the vhdl description
          -nh    generates a hierarchical cone netlist
          file2  is the vhdl file to be generated (default is file1)
          -t     display execution trace
```

If it does not work, please abort the tutorial and fix the pathnames.

We will assume that the user is running a `c-like` shell, like `csh` or `tcsh`. If you run a `sh-like` shell, please refer to your documentation.

Typically do:

```
~fred/addaccu %-) setenv ALLIANCE_TOP  /usr/local/cad/alliance
~fred/addaccu %-) setenv PATH $ALLIANCE_TOP/bin:$PATH
```

Where we assume that **Alliance** has been installed under `/usr/local/cad/alliance` directory. In the sequel, we will assume that the `$ALLIANCE_TOP` variable is properly set up.

All the tools used in this tutorial are documented at least with a manual page. Each manual can be accessed using the `man` *tool* command. You may have to do a:

```
~fred/addaccu %-) setenv MANPATH $ALLIANCE_TOP/man:$MANPATH
```

in order to have the manual accessible.

The tutorial is organized around the following sections:

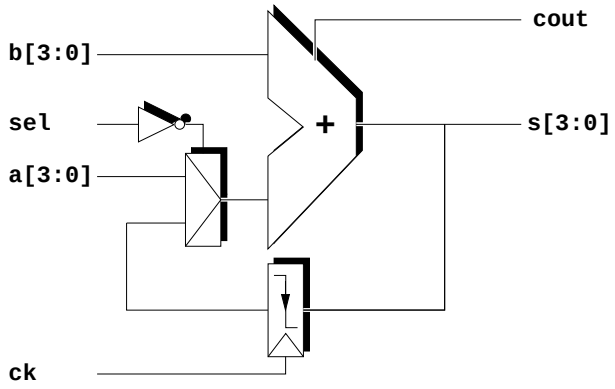

Figure 1: Architecture of the `addaccu` circuit.

2 Chip overview

The picture in the Figure **??** shows the architecture of the small chip that will be our example all along this tutorial. As you can see, the circuit is pretty small: it mostly consists in a four bit adder, a four bit register, and a 2 to 1 four bit multiplexer.

Inputs are located on the left side of the picture, and outputs are located on the right side. We did not represent the power supplies on this schematic, but you'll need them in order for the chip to work!

The circuit performs an addition between either the `b[3:0]` and `a[3:0]` inputs when `sel` is set to 0, or between `b[3:0]` and the contents of the four bit register when `sel` is set to 1. The content of the register is overwriten by the value of the outputs `s[3:0]` on each falling edge of the clock, `ck`.

3 Design flow

You are now ready to actually design the chip and use the **Alliance** tools. The design flow for this little example is composed of 5 main steps:

1. behavioral capture and simulation;
2. netlist capture and validation;
3. physical layout generation;
4. design validation;
5. symbolic to real conversion.

As you will see, points 2 and 3 must be performed for each level of hierarchy. In this example we distinguish two levels of hierarchy: the core level and the chip level. At the core level, the leaf cells of the design belong to the **Alliance** standard cells library. At the chip level, the previous core and pads belonging to the **Alliance** pad library are used.

The Figure **??** below describes the circuit's hierarchy.

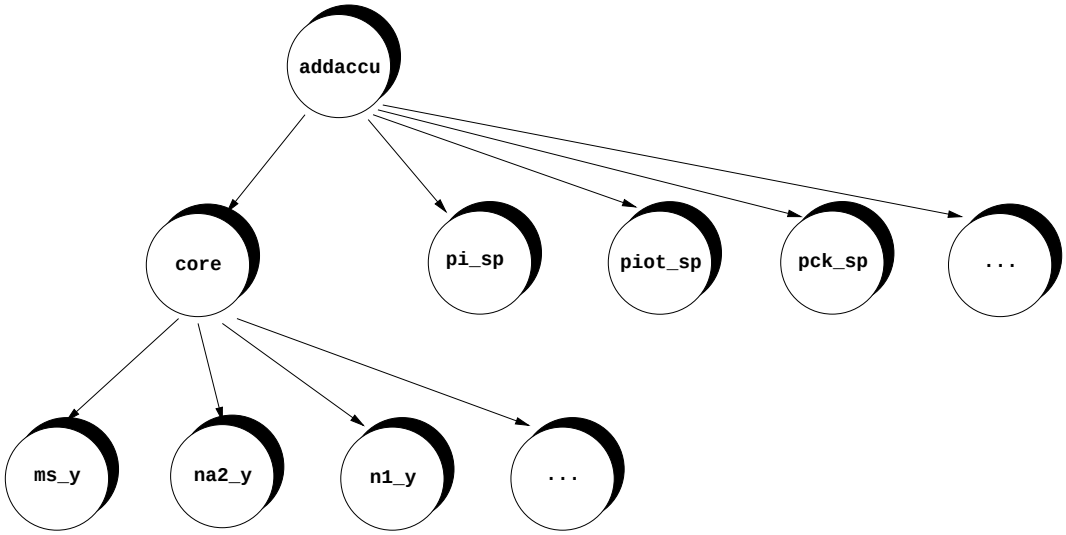

Figure 2: Hierarchical partitionning.

4 Execution environment set-up

Before you start examining the first phase of the design methodology, you first have to set up an execution environment for the **Alliance** tools.

If the **Alliance** installation is set up properly, most of the tools can be executed without problem. If the installation is only partial, or if you want to set a special feature of a tool, you will have to set some environment variables up.

The environment variables upon the which each tool depends are documented in each tool manual page. However, some variables are really useful, and are in part documented here.

4.1 Cell libraries path

The first thing to know for this circuit is the actual location of the standard-cell library. This library is located into `$ALLIANCE_TOP/cells/sclib`. But indicating only one library is not enough, and as you can see in doing a `ls $ALLIANCE_TOP/cells`, several libraries are available.

The other library of interest here is the pad library, located in `$ALLIANCE_TOP/cells/ring`.

To set up the cell library path, the following command is required:

```
~fred/addaccu %-) setenv MBK_CATA_LIB $ALLIANCE_TOP/cells/scr:$ALLIANCE_TOP/cells/ring
```

This `setenv` instructs **Alliance** tools to search its cells in `/alliance/cells/scr` and then in `/alliance/cells/ring`, that respectively contains the standard cells and the pads.

4.2 User working directory

You must specify the directory where the files generated by an **Alliance** tool are to be located.

Please enter the following command:

```
~fred/addaccu %-) setenv MBK_WORK_LIB .
```

In this case, the working directory is the current directory. This library is scanned before the ones set in `MBK_CATA_LIB` when a tool loads a cell, and the only one used for writing a cell. Note that `.` is the default value of this variable, so it is not required to set it up usually.

4.3 File formats

One of the interesting features of **Alliance** is that different file formats can be used for both netlist and layout view. However, in the design methodology we wish to promote, some formats are recommended. The `vst`, structural **VHDL**, is dedicated to netlist specification. The `al` format is dedicated to extracted layout representation. The `ap` format is the usual layout format.

So, prior to generate a specification netlist, you shall type:

```
~fred/addaccu %-) setenv MBK_OUT_LO vst
```

But if you wish to extract a netlist from the layout then you'll do:

```
~fred/addaccu %-) setenv MBK_OUT_LO al
```

5 Behavioral capture and simulation

When designing a chip, the first thing to do is to write its behavior, based on its functional specifications. In our example `addaccu`, we have to modelize an adder, a register and a multiplexer. We also have to specify in the resulting behavior file that the adder may take as an input the result of a previous calculation or a new input stimuli.

Modern behavioral descriptions are written using the **VHDL** language, the most promoted and supported hardware description language.

Let us now edit the behavioral description of addaccu by issuing the following command:

```
~fred/addaccu %-) vi addaccu.vbe
```

The `addaccu.vbe` file contains the behavioral description of the circuit. `Vbe` stands for **VHDL** behavioral description. Although this tutorial does no intend to explain the arcanes of **VHDL** programming, it's worth noting some interesting points:

1. notice that the entity name, `addaccu`, identifies the circuit;
2. examine the circuit interface. You can recognize the terminals of the previous picture, plus 4 special terminals, known as supply terminals. The supply terminals are needed for the last stage of the design, as the original behavioral description must be matched with an "extracted" behavior. If this does not make immediate sense to you, do not panic, everything will be clear in a moment;
3. take a look to the functional architecture of `addaccu`. When examining it, pay attention to the `reg_bit register` names of the accumulator `reg`, as they will be used later in the validation stage;

4. read carefully the functional description. With a little work, you can recognize the description of the multiplexer, the equations of the adder, and the accumulator. It is very important to notice that the functional description of the circuit assumes that edge-triggered latches are used.

Once you have carefully examined this file, close it and get back to the shell prompt, for the **VHDL** compilation stage.

5.1 Behavior compilation

Once the behavioral description of the circuit is written, it is time for **VHDL** compilation.

You are now ready to compile `addaccu.vbe`. As we do not have written simulation patterns yet, the only thing we can do is to test the syntax of the file we have just edited.

At the shell prompt level, please type:

```
~fred/addaccu %-) asimut -b -c addaccu
```

This command instructs `asimut` to compile the behavioral file `addaccu.vbe`. Option `-b` means behavioral description, and `-c` means compilation only, i.e no simulation.

```
~fred/addaccu %-) asimut -b -c addaccu

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

                            A SIMUlation Tool

                 Alliance CAD System 3.2,         asimut v2.01
                 Copyright (c) 1991-1997,       ASIM/LIP6/UPMC
                 E-mail support: alliance-support@asim.lip6.fr

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

Figure 3: `Asimut` compiling the behavioral specifications.

5.2 Behavioral simulation and validation

A behavioral description without simulation patterns is useless. In order to see if the behavior you have just written is functionally correct, you must write simulation patterns and use the `asimut`

simulator function.

Simulation patterns are contained in a plain text file, `addaccu.pat`. For more informations about the `pat` format you can read the appropriate on-line manual — `man 5 pat` — or read the printed documentation.

To get acquainted with `addaccu.pat`, please issue the following command:

```
~fred/addaccu %-) vi addaccu.pat
```

There are several interesting things here. First, you can recognize the circuit interface again. Second, you can see that output terminals are now located at the end of the interface. Ordering terminals is very important for `pat` files. The order you use to specify terminals greatly influences the way you will have to write simulation patterns. It is important to notice that simulation patterns contain both input and output values. The behavior must be checked using these values. Comments are prefixed by a #, and are of much interest here.

Once you have understood the structure of this file, you are able to simulate the previous behavioral description.

In order to simulate, please type:

```
~fred/addaccu %-) asimut -b addaccu addaccu specifications
```

to obtain:

```
~fred/addaccu %-) asimut -b addaccu addaccu specifications

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

                           A SIMUlation Tool


                Alliance CAD System 3.2,         asimut v2.01
                Copyright (c) 1991-1997,       ASIM/LIP6/UPMC
                E-mail support: alliance-support@asim.lip6.fr

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

Searching pattern file : addaccu ...
Restoring ...

Linking ...
###----- processing pattern 0 -----###
###----- processing pattern 1 -----###
###----- processing pattern 2 -----###
###----- processing pattern 3 -----###
###----- processing pattern 4 -----###
###----- processing pattern 5 -----###
###----- processing pattern 6 -----###
```

Figure 4: `Asimut` checking the behavior with a few functionnal test vectors.

Feel free to add new simulation vectors, as it's a very good practice. You can also write obviously wrong patterns to see how `asimut` behaves when it encounters errors.

6 Netlist capture and validation

You are now about to capture the logical view of `addaccu`, commonly known as netlist. The design methodology prevents the designer from building the entire netlist in one shot, including pads. VLSI design is based on hierarchy and incremental approaches.

6.1 Netlist capture

The circuit netlist you have to write will be captured in two steps:

1. capture of the core netlist;

2. capture of the chip netlist, linking the core with pads.

The core netlist contains standard-cells describing the logical functions needed to design the adder, the multiplexer and the accumulator, and the internal wires.

We do not use a schematic editor for netlist capture, but rather a textual approach.

```
~fred/addaccu %-) vi core.c
```

The core netlist is nothing more than a **C** file containing specific function calls for the creation of VLSI objects. Notice that the include file `genlib.h` must systematically be included at the top of **genlib**'s files.

The description of a netlist in **C** is mainly based on the `DEF_LOFIG, LOCON` and `LOINS` functions. `DEF_LOFIG` defines the name of the resulting netlist view, `LOCON` instanciates a new terminal in the currently opened figure, and `LOINS` instanciates pre-existing cells or blocks. For more informations, please read the on-line manual associated with each function.

To compile and run the **C** file, you must use the `genlib` program.

You must specify the input format of cells that are instanciated in the **genlib**'s code.

Therefore enter the following command:

```
~fred/addaccu %-) setenv MBK_IN_LO vst
~fred/addaccu %-) setenv MBK_OUT_LO vst
```

These `setenv` define the input and output formats for the specification netlist.

`vst` stands for structural **VHDL** description. The design methodology assumes that user defined netlists must use extension `vst` as we've already said. The `al` format is mostly used for extracted netlist with specific informations like capacitances that cannot be coded in **VHDL**.

At last, you can execute the genlib program:

```
~fred/addaccu %-) genlib -v core
```

to get:

```
~fred/addaccu %-) setenv MBK_IN_LO vst
~fred/addaccu %-) setenv MBK_OUT_LO vst
~fred/addaccu %-) setenv MBK_CATA_LIB /alliance/cells/scr
~fred/addaccu %-) genlib -v core

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

                        Procedural Generation Language

                  Alliance CAD System 3.2,           genlib 3.3
                  Copyright (c) 1991-1998,       ASIM/LIP6/UPMC
                  E-mail support: alliance-support@asim.lip6.fr

Generating the Makefile
Compiling, ...
Current execution environment
MBK_CATA_LIB   : /users/soft5/newlabo/Solaris/cells/sclib
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 ...
```

Figure 5: Netlist core generation using `genlib`.

You can now look in the current directory for a file named `core.vst`. This file contains the resulting description of the core.

The `addaccu.c` file contains the actual chip netlist. You can edit it to see how pads are instanciated.

Running the following command:

```
~fred/addaccu %-) genlib -v addaccu
```

produces:

```
~fred/addaccu %-) setenv MBK_IN_LO vst
~fred/addaccu %-) setenv MBK_OUT_LO vst
~fred/addaccu %-) setenv MBK_CATA_LIB /alliance/cells/scr:/alliance/cells/ring
~fred/addaccu %-) genlib addaccu

                Procedural Generation Language

          Alliance CAD System 3.2,           genlib 3.3
          Copyright (c) 1991-1998,       ASIM/LIP6/UPMC
          E-mail support: alliance-support@asim.lip6.fr
```

Figure 6: Core to pad netlist generation using `genlib`.

This creates `addaccu.vst`, the resulting netlist, in the current directory.

6.2 Simulation of the specification netlist

Once the complete netlist has been captured, we can simulate the whole chip with the previous simulation patterns.

You can run asimut by entering the following command:

```
~fred/addaccu %-) asimut addaccu addaccu schema
```

It produces the output of Figure **??**.

The first `addaccu` stands for `addaccu.vst`. The second `addaccu` stands `addaccu.pat`, the pattern file with input and output values. `schema` stands for `schema.pat`, the generated list of patterns.

Because option `-b` is not set and because `MBK_IN_LO` contains `vst`, `asimut` simulates the specification netlist we have captured with `genlib`.

Errors during this execution mean that something went wrong between your current position in the design flow and the behavioral simulation stage.

```
~fred/addaccu %-) setenv MBK_IN_LO vst
~fred/addaccu %-) setenv MBK_CATA_LIB /alliance/cells/scr:/alliance/cells/ringxx
~fred/addaccu %-) asimut addaccu addaccu schema
Warning 2 : consistency checks will be disabled

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

                            A SIMUlation Tool

                  Alliance CAD System 3.2,         asimut v2.01
                  Copyright (c) 1991-1997,       ASIM/LIP6/UPMC
                  E-mail support: alliance-support@asim.lip6.fr

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

Flattening the root figure ...

Searching a2_y ...
BEH : Compiling a2_y.vbe (Behaviour) ...
Making GEX ...
.
.
.
Searching pi_sp ...
BEH : Compiling pi_sp.vbe (Behaviour) ...
Making GEX ...

Searching pck_sp ...
BEH : Compiling pck_sp.vbe (Behaviour) ...
Making GEX ...

Searching pattern file : addaccu ...
Restoring ...

Linking ...
###----- processing pattern 0 -----###
###----- processing pattern 1 -----###
###----- processing pattern 2 -----###
###----- processing pattern 3 -----###
###----- processing pattern 4 -----###
###----- processing pattern 5 -----###
###----- processing pattern 6 -----###
```

Figure 7: `Asimut` checking the netlist description.

7 Core layout generation

Incremental design relies on hierarchy. Before you design the core to pads netlist, the circuit core must be successfully built. In this example, the layout generation relies on:

1. standard-cells placement (automatic or manual);
2. automatic routing of standard-cells.

`scr` is a place and route tool that can produce automatically the layout view from a netlist description.

As stated in the previous section, before you run `scr`, you must specify environment variables. Needless to say that these variables will deal with physical aspects of the circuit.

First, you must specify the input layout format you use for the standard-cells. The **Alliance** distribution only contains `ap` symbolic layout files so you must issue the following command:

```
~fred/addaccu %-) setenv MBK_IN_PH ap
```

This command instructs `scr` to use the `ap` cell layout format as input. Remember that the variable `MBK_CATA_LIB` is still set, so **Alliance** tools know where to find cells.

Second, you must accordingly specify the output layout format. Again, the format is `ap`, so:

```
~fred/addaccu %-) setenv MBK_OUT_PH ap
```

This command instructs `scr` to use `ap` representation of cells as an output.

Everything is now ready for the place and route phase. The command is:

```
~fred/addaccu %-) scr -p -r core core
```

Option `-p` means automatic placement, and option `-r` means automatic routing.

Once `scr` has been executed, the core layout is completed.

```
~fred/addaccu %-) setenv MBK_IN_LO vst
~fred/addaccu %-) setenv MBK_IN_PH ap
~fred/addaccu %-) setenv MBK_OUT_PH ap
~fred/addaccu %-) setenv MBK_CATA_LIB /alliance/cells/scr
~fred/addaccu %-) scr -p -r core

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

                              Standard Cell router

                    Alliance CAD System 3.2,              scr 5.2
                    Copyright (c) 1991-1997,       ASIM/LIP6/UPMC
                    E-mail support: alliance-support@asim.lip6.fr

Loading logical view : core
Placing logical view : core
Loading SCP data base ...
Generating initial placement ...
25 cells 37 nets in 2 rows
Placement in process of treatment :   100%
49% saved in  0.3 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
Making vertical power and ground wires
Saving layout : core
```

Figure 8: Placing and Routing the core netlist with `scr`.

Note that `scr` knows how to place and route only cells having a standard cell topology.

8 Core layout verification

Routers may have bugs, and produce shorts or open circuits. So serious checks must be made on the produced layout. To check the layout, we use two **Alliance** tools, `lynx` and `lvx`.

`lynx` is the **Alliance** netlist extractor. From a physical layout it extracts a netlist representation of the circuit, in terms of blocks, gates or transistors. In our case, we want to extract the core netlist at the gate level.

`lvx` is the **Alliance** netlist comparator. Its main function is to verify that an extracted netlist corresponds to the specification netlist.

Hence, the design methodology is quite simple. We must extract the core `core.ap` with `lynx` and check the resulting netlist with the original `addaccu.vst` file using `lvx`.

8.1 Extraction of the core netlist

Now, back to practice. You have to extract the layout using `lynx`. Remember two things:

1. **Alliance** tools deeply rely on appropriate environment variables. For instance, you have to specify the format the extracted netlist will be generated with.
2. in the usual design flow, extracted netlist files are written in `al` format.

So you must:

```
~fred/addaccu %-) setenv MBK_OUT_LO al
```

to generate a extracted netlist file called `core.al`.

The actual execution of lynx is quite easy. Just type:

```
~fred/addaccu %-) lynx -v core
```

and you get your extracted netlist in terms of interconnected standard cells.

```
~fred/addaccu %-) setenv MBK_IN_PH ap
~fred/addaccu %-) setenv MBK_OUT_LO al
~fred/addaccu %-) setenv RDS_TECHNO_NAME /alliance/etc/cmos_5.rds
~fred/addaccu %-) setenv MBK_CATA_LIB /alliance/cells/scr:/alliance/cells/ring
~fred/addaccu %-) lynx -v core core

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

                                 Netlist extractor

                              Alliance CAD System 3.2,            lynx 1.16
                              Copyright (c) 1993-1997,       ASIM/LIP6/UPMC
                              E-mail support: alliance-support@asim.lip6.fr



        ---> Extracts symbolic figure core

                ---> Translate Mbk -> Rds
                ---> Build windows
                <--- 108

                ---> Rectangles    : 894
                ---> Figure size   : (    -50,   -100 )
                                     (  55250,  16600 )

                ---> Cut transistors
                <--- 0
                ---> Build equis
                <--- 37
                ---> Delete windows
                ---> Build signals
                <--- 37
                ---> Build instances
                <--- 29
                ---> Build transistors
                <--- 0
                ---> Save netlist

        <--- done !
```

Figure 9: Extracting a hierarchical netlist in terms of standard cells with `lynx`.

8.2 Netlist cross-checking

Now it's time for netlist verifications. Since `lvx` takes two netlist in input, the formats have to be specified on the command line.

```
~fred/addaccu %-) lvx vst al core core
```

`lvx` works somewhat differently from other tools. Its two former arguments are the formats of the first netlist, here `vst` (for the original netlist) and `al` (for the extracted netlist). The two following arguments are the names of the netlist. We have done our best to make these names identical. Once again, remember that specification files are postfixed with `vst` and extracted files with `al`.

```
~fred/addaccu %-) setenv MBK_CATA_LIB /alliance/cells/scr:/alliance/cells/ring
~fred/addaccu %-) lvx vst al core core -f

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

                             Gate Netlist Comparator

                  Alliance CAD System 3.2,             lvx 2.23
                  Copyright (c) 1992-1997,       ASIM/LIP6/UPMC
                  E-mail support: alliance-support@asim.lip6.fr

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

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

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

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

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

===== Terminals .......... 16
===== Instances .......... 25
===== Connectors ......... 150

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

Figure 10: Doing the specification netlist versus extracted netlist compaison with `lvx`.

As a result, you now have a correct `al` file for the circuit's core. Its name is `core.al`. This file may now be simulated with `asimut`, but this is only for cross checking.

Execution environment is set by issuing:

```
~fred/addaccu %-) setenv MBK_IN_LO al
```

because we want to use the extracted `core.al` file as input to `asimut`.

Then you can run asimut by entering the following command:

```
~fred/addaccu %-) asimut core addaccu result
```

9 Core to pads routing

Having built and checked the core layout, you can run the core-to-pads router. Its function is to locate pads appropriately around the core and to create wiring segments between pads and chip terminals.

Before we set up the execution environment, take a look at the `addaccu.rin` file. This file instructs `ring`, the core to pads router, to place pads according to designer's wishes. It also instructs `ring` to use segments with reasonable width for power routing.

The syntax of this file is quite simple. The names mentioned here are pads' instance names.

Run:

```
~fred/addaccu %-) ring addaccu addaccu
```

```
~fred/addaccu %-) setenv MBK_IN_LO vst
~fred/addaccu %-) setenv MBK_IN_PH ap
~fred/addaccu %-) setenv MBK_OUT_PH ap
~fred/addaccu %-) setenv MBK_CATA_LIB /alliance/cells/scr:/alliance/cells/ring
~fred/addaccu %-) ring addaccu addaccu

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

                                PAD ring router

                Alliance CAD System 3.2,             ring 2.9
                Copyright (c) 1991-1997,       ASIM/LIP6/UPMC
                E-mail support: alliance-support@asim.lip6.fr

        o reading netlists, layout views of core and pads.
Warning 2 : consistency checks will be disabled
        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.
```

Figure 11: `Ring` routes the core to pad netlist.

Like for the core, the core to pad routing must be verified. Using the same approach leads to:

```
~fred/addaccu %-) setenv MBK_OUT_LO al
~fred/addaccu %-) setenv MBK_IN_PH ap
~fred/addaccu %-) lynx -v addaccu
~fred/addaccu %-) lvx vst al addaccu
```

It is possible to check both core a core to pad routing at once also:

```
~fred/addaccu %-) setenv MBK_OUT_LO al
~fred/addaccu %-) setenv MBK_IN_PH ap
~fred/addaccu %-) lynx -v -f addaccu
~fred/addaccu %-) lvx vst al addaccu -f
```

Here the `-f` option indicate the tools to express the netlist in terms of leaf cells — either pads

or standard cells —. Obtaining a transistor netlist from the layout is possible using the `-t` option of `lynx`, but this is more time and memory consuming, and fairly useless as we rely on correct cells.

10 Chip layout visualization

At that point, you are able to see the actual layout by executing the symbolic layout editor `graal`. You can run the editor by issuing the following command:

```
~fred/addaccu %-) graal -l addaccu
```

Now, the screen should be filled with the `graal` window, and should contain the whole layout. The graphical interface of `graal` is meant to be self explanatory.

11 Functional abstraction

`yagle`, a functional asbtractor that extracts a behavior from a transistor netlist can relieve the designer from many headaches. From a transistor netlist,`yagle` finally outputs a **VHDL** file that can be simulated.

The command is:

```
~fred/addaccu %-) setenv MBK_IN_LO al
~fred/addaccu %-) yagle -i -v addaccu
```

```
~fred/addaccu %-) setenv MBK_IN_LO al
~fred/addaccu %-) yagle addaccue -i -v

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

                        Yet Another Gate Level Extractor

                  Alliance CAD System 3.2,           yagle 2.01
                  Copyright (c) 1994-1997,       ASIM/LIP6/UPMC
                  E-mail support: alliance-support@asim.lip6.fr

[YAG MES] Reading file 'addaccue.inf'
[YAG MES] Loading the figure addaccue              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] 80 transistors are always off
[YAG WAR 07] 80 transistors are not used
[YAG WAR 09] 8 latches detected
See file 'addaccue.rep' for more information
```

Figure 12: `Desb` abstract the extracted netlist into boolean equations.

Then, you can simulate the resulting behavior file by issuing:

```
~fred/addaccu %-) asimut -b addaccu addaccu result
```

Simulation of extracted behavior with asimut is not the only mean to see if the chip is correct. The extracted behavior can also be used for formal proof.

In **Alliance**, formal proof is the ultimate way to validate your circuit.

Before you can use `proof`, the formal prover, you must get acquainted with some specific concepts of the formal proof theory.

If you look at the extracted behavior, `chip.vbe`, you can notice that `yagle` has found four

memorizing elements, specified as `reg_bit register`. In the formal proof, these four elements must match the four edge-triggered latches of the former behavioral specification, because `proof` can only compare combinatorial logic between external connectors and/or sequential elements.

If you examine the current directory, you will find a file called `addaccu.inf`.

This file contains specific informations for `yagle`. It instructs `yagle` to rename the internal node name of every latch *core.l** by its corresponding name in the first behavior file. In the `cells/scr` directory you can see that the name of the internal node of the used latch, `ms_y.vbe`, is *dff_s*.

The equivalent name in `addaccu.vbe` is *reg*.

When you run `yagle` with option `-i`, the program searches for the file `chip.inf`, builds the gate netlist and replaces the node names by their equivalent names in the behavior file, in order to make `proof` work.

Then, you can run `proof`, the formal prover:

```
~fred/addaccu %-) proof -d -p addaccu addaccue
```

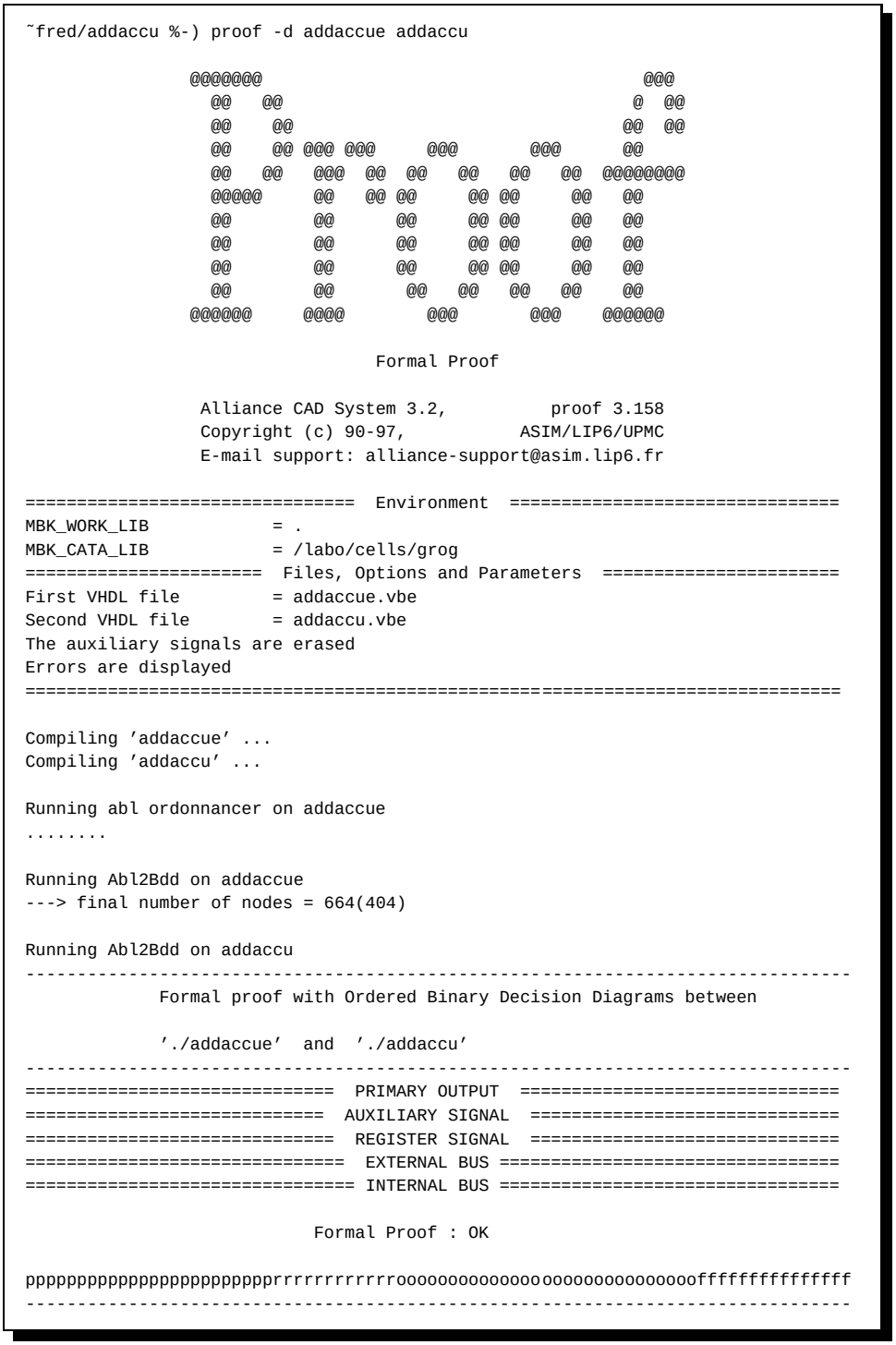

Figure 13: Doing a formal proof between the abstracted behavior and the specifications using `proof`.

Option `-d` displays the logical functions that do not match, and the `-p` option tells `proof` to

ignore the internal polarity of the flip-flops: we described our behavioral with a register

12 Further verifications

The last verification concerns design rules. Before the chip can actually be targetted on a process, a symbolic design rule check — DRC — must be performed.

In **Alliance**, the design rule checker is `druc`.

You just have to type:

```
~fred/addaccu %-) setenv MBK_IN_PH ap
~fred/addaccu %-) setenv RDS_TECHNO_NAME /alliance/etc/cmos_5.rds
~fred/addaccu %-) druc addaccu
```

The file pointed to by the `RDS_TECHNO_NAME` environement variable contains many technological information including the desription of the design rules.

```
~fred/addaccu %-) setenv MBK_IN_PH ap
~fred/addaccu %-) setenv RDS_OUT cif
~fred/addaccu %-) setenv RDS_TECHNO_NAME /alliance/etc/cmos_5.rds
~fred/addaccu %-) setenv MBK_CATA_LIB /alliance/cells/scr:/alliance/cells/ring
~fred/addaccu %-) druc addaccu

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

                               Design Rule Checker

                  Alliance CAD System 3.2,            druc 3.00
                  Copyright (c) 1993-1997,       ASIM/LIP6/UPMC
                  E-mail support: alliance-support@asim.lip6.fr

Flatten DRC on: addaccu
Delete MBK figure : addaccu
Load Flatten Rules : /usr/local/cad/alliance/etc/cmos_7.rds

Unify : addaccu

Create Ring : addaccu_rng
Merge Errorfiles:

Merge Error Instances:
instructionCourante : 50
End DRC on: addaccu
Saving the Error file figure
Done
  0

File: addaccu.drc is empty: no errors detected.
```

Figure 14: Symbolic design rule checking using `druc`.

This layout level verification has been optimized for speed. So it is fast, even on large circuits, but requires quite a few megs then.

13 Symbolic to real technology translation

The purpose of **Alliance** symbolic layout approach is to allow the designs to be targetted on several technologies. The whole chip is conceived with symbolic cells.

In order to send the chip to a specified foundry, the designer must perform the symbolic to real conversion. This stage is the last stage of the design methodology.

The only things you have to specify is the target technology and some environment variables.

The only target technology available in the distribution is `prol10`, a fake 1.0 process. You can

choose it by setting the RDS_TECHNO_NAME environment variable.

```
~fred/addaccu %-)setenv RDS_TECHNO_NAME /alliance/etc/prol10.rds
```

You also have to specify the format of the output file that will represent the foundry layout. **Alliance** provides two distinct formats: `gds` and `cif`.

```
~fred/addaccu %-) setenv RDS_OUT cif
```

This command instructs the converter to output the chip in the Caltech Intermediat Form. This form is an `ascii` format, that can include connectors, whereas `gds` is binary and doesn't include connectors.

A tricky thing is that the *pad* part of the symbolic IO pads must be replaced by its "real" equivalent. This is indeed the only technology dependant part of a circuit designed with our symbolic approach.

So you also have to set the RDS_IN environment variable, to indicate the format of the substitution pad.

```
~fred/addaccu %-) setenv RDS_IN cif
```

At last, you run the symbolic to real converter, `s2r`.

```
~fred/addaccu %-) s2r -cv addaccu
```

Your chip is now ready for the foundry that provides the prol10 technology.

```
~fred/addaccu %-) setenv MBK_IN_PH ap
~fred/addaccu %-) setenv MBK_CATA_LIB /alliance/cells/scr:/alliance/cells/ring
~fred/addaccu %-) setenv RDS_TECHNO_NAME /alliance/etc/prol10.rds
~fred/addaccu %-) setenv RDS_OUT cif
~fred/addaccu %-) setenv RDS_IN cif
~fred/addaccu %-) s2r -v addaccu

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

                              Symbolic to Real layout converter

                         Alliance CAD System 3.2,              s2r 3.6
                         Copyright (c) 1991-1997,       ASIM/LIP6/UPMC
                         E-mail support: alliance-support@asim.lip6.fr


        o loading technology file : /usr/local/cad/alliance/etc/prol10_7.rds
        o loading all level of symbolic layout : addaccu
        o removing symbolic data structure
        o layout post-treating with connectors, with scotchs.
        --> post-treating model palo_sp
            rectangle merging :
            . RDS_NWELL ................................
            . RDS_NIMP .................................
            . RDS_PIMP .................................
            . RDS_ACTIV ................................
            . RDS_POLY .................................
            . RDS_ALU1 .................................
            . RDS_ALU2 .................................
            .
            .
            .
        o replacing black boxes
        --> replace cell padreal
        o saving addaccu.cif
        o memory allocation informations
        --> required rectangles = 7318  really allocated = 7
        --> required scotchs = 3  really created = 3
        --> Number of allocated bytes: 341231
```

Figure 15: Translating the symbolic layout into process layout with `s2r`.

14 Conclusion

If you actually arrived here doing all the previous steps, congratulation!

You can see an automatic replay of this tutorial by executing the `Makefile` in the current

directory by issuing the command:

```
~fred/addaccu %-) make
```

Note that the `Makefile` uses Bourne-shell commands.

If you plan to design a chip with **Alliance**, it is a good idea to use `Makefile`s to ensure the consistency between the design and verifications of the differents views and hierarchies, but this is another story.

If anything went wrong or if you have any question, you can contact us by sending a mail to the support team of **Alliance**, located at `alliance-support@lip6.fr`.