Alliance: A Complete CAD System for **VLSI** Design

Équipe Achitecture des Systèmes et Micro-Électronique
Laboratoire d'Informatique de Paris 6
Université Pierre et Marie Curie
4, Place Jussieu 75252 Paris Cedex 05,
France
`http://www-asim.lip6.fr/alliance/`
`ftp://ftp-asim.lip6.fr/pub/alliance/`
`mailto:alliance-support@asim.lip6.fr`

Abstract

*The **Alliance** package is a complete set of CAD tools for the specification, design and validation of digital **VLSI** circuits. Beside the tools, **Alliance** includes also a set of cell libraries, from standard cells for automatic place and route tools, to custom block generators to be used in high performance circuits. This package is used in more than 250 universities worldwide.*

*Each **Alliance** tool can operate as a standalone program as well as a part of the complete design framework. After introducing briefly the design methodology, we outline the functionnality of the tools. Experiemental results conclude the presentation.*

***Alliance** runs on any **Unix** system and has been recently ported to **Windows** NT. It is freely available on `ftp`, and includes binaries, leaf cells libraries, on-line documentation, and tutorials.*

1 Introduction

The **Alliance** package is the result of a ten years effort spent at the **LIP6** Laboratory (formerly **MASI**) of the Pierre et Marie Curie University (UPMC), in Paris. During these years, our major goal was to provide our undergraduate and graduate students with a complete CAD framework, designed to assist them in digital **VLSI CMOS** course. The *Architecture* team at **LIP6** focuses its activity on two key issues: computer architectures using high complexity ASICs, and innovative CAD tools for **VLSI** design. Strong interaction exists between the people working on computer architectures and the one working on CAD tools. The main CAD action aims at fulfilling both the needs of experienced designers by providing practical answers to state-of-the-art problems (logic synthesis, procedural generation, layout verification, test and interoperability), and novice designers, by providing a simple and consistent set of tools. Our **VLSI** design flow is therefore based on both advanced CAD tools that are not available within commercial CAD systems, such as functional abstraction or static timing analysis, and standard design/validation tools.

Alliance VLSI CAD System is free software. Binaries, source code and cells libraries are freely available under the GNU General Public Licence (**GPL**). You are welcome to use the software package even for commercial designs whithout any fee. You are just required to mention : "Designed with Alliance © LIP6/Université Pierre et Marie Curie". For any questions please mail to : `alliance-support@asim.lip6.fr`.

1.1 Process independence

To be useful, a CAD system must provide a way to the silicon, therefore **Alliance** provides a large set of cell libraries also available at the layout level. The target technologies of **Alliance** is **CMOS**. The layout libraries rely on a symbolic layout approach that provides process independence in order to allow the designers to easily port their designs from one silicon supplier to another. The main point in this approach is that the pitch matching constraints in both x and y direction are kept through technological retargetting. The translation, fully automated, relies on a technological file suited to a given process.

These files can be generated directly from the process design rules. Also technological files for several processes are available through the CMP and EuroPractice services, provided you signed a NDA for the process.

1.2 Software portability

The **Alliance** package has been designed so to run on an heterogeneous network of workstations. The only requirements are a **C** compiler and a **Unix** system. For the graphical applications, the XWindow library is used. Several hardware platforms, from Intel 386 based microcomputers to SparcStations and DEC Stations, are supported.

1.3 Modularity

According to the interoperability constraints, each **Alliance** tool can operate as a standalone program as well as a part of the complete **Alliance** design framework. Each **Alliance** tool therefore supports several standard **VLSI** description formats : **SPICE**, **EDIF**, **VHDL**, **CIF**, **GDS2**. In that respect, the tools ouputs are fully usable under the **Compass** and **Cadence Opus** environnement, provided these tools have the necessary configuration files. The **Alliance** tools support a zero-default top-down design methodology with not only construction tools — layout editor, automatic place & route — but also validation tools, from design rule checker to functional abstraction and formal proof.

1.4 Compactness

Unlike commercially available CAD systems, the **Alliance** CAD Framework suits the limited ressources of low-cost workstations. For small educational projects — 5000 gates —, a **Unix** system with 8 to 20 Mbytes of memory, appropriate disk storage (30 Mbytes per user), and graphic capabilities (X-Window) is sufficient.

1.5 Easiness

All tools and the proposed design flow are simple to teach and to learn. In any situation, easiness and simplicity have been prefered to sophisticated approaches.

To each tool correspond a unique behavior and utility. Each step of the design methodology corresponds to the use of one or a few tools, for which the usage is well identified.

From a pratical point of view, both on-line documentation (**Unix** `man`) and paper are available with each tool of the **Alliance** package.

2 Alliance design flow

We refer to the term "design flow" as a sequenced set of operations performed when realizing a **VLSI** circuit. In the design flow, we rely on a strict definition of all the objects and design functions found in the process of designing a **VLSI** chip. The design flow is based on the Mead-Conway model and is characterized by its top-down aspect. Below we introduce the major steps of the basic design methodology. It emphasizes the top-down aspect of the design flow, and points out that our methodology is breaked up into 5 distinct parts, the latter being not available yet within **Alliance**:

- capture and simulation of the behavioral view,
- capture and validation of the structural view,
- physical implementation of the design,
- layout verification,
- test and coverage evaluation.

The design flow also includes miscellaneous tools like layout editor for the design of the cell libraries, and a PostScript plotter for documentation.

2.1 Capture and simulation of the behavioral view

Like we just saw, the capture of the behavioral view is the very first step of our design flow. Within **Alliance**, any **VLSI** design begins with a timing independent description of the circuit with a subset of **VHDL** behavior primitives. This subset of **VHDL**, called `vbe`, is fairly restricted: it is the data-flow subset of this language. It is not very easy to modelize an architecture using this subset, but it has the great advantage of allowing simulation, logic synthesis and bit level formal proof on the same files.

Patterns, **VHDL** simulation stimuli, are described in a specific formalism that can be captured using a dedicated language `genpat`. Once a **VHDL** behavioral description written and a set of test vectors have been determined, a functional simulation is ran. The behavioral **VHDL** simulator is called `asimut`. It validates the input behavior, according to the input/output vectors.

2.2 Capture and validation of the structural view

The structural view can be captured once the data flow description is validated. The actual capture of the netlist relies either on specific description languages, `genlib` for standard cells or `fpgen` for data-path, or on direct synthesis from the data flow using the `bop` tool for optimization and the `scmap` tool to map on a cell library. `Genlib` and `fpgen` are netlist-oriented libraries of C functions. In the design methodology, it is essential for the students to get acquainted with the **C** language basics. The advantage of such an approach is that designers do not have to learn several language with specific syntax and semantics.

Usually, the main behavior is partitionned in several sub-behaviors. Some are described recursively using the `genlib` language, other using `fpgen`, and the other ones can be directly synthesized from a **VHDL** description of the corresponding sub-behaviors. The `scmap` tool takes an **RTL** description and generates a netlist of standard cell gates. An other subset of **VHDL** allows to capture finite state machines. This subset, called `fsm`, can be translated into a **RTL** description using the tool `syf`, and then the resulting description optimized usign `bop` and finally syntesized as a netlist using once more `scmap`.

Since asimut can operate on both **RTL** and structural views, the structural description is checked against the behavioral description by using the same set of patterns that has been used for behavioral validation.

2.3 Physical design

Once the circuit netlist has been captured and validated, each leaf of the hierarchy has to be physically implemented. A netlist issued from scmap is usually placed and routed using the standard cell router scr. If the netlist has been captured using genlib and if it has a high degree of regularity, it can be placed manually for optimisation using other genlib functions. The netlist resulting from the use of fpgen are placed and routed using the datapath router dpr.

These part can be assembled together using a gridless channel router called bbr, and this generates what we call a *core*. The circuit core is now ready to be connected to external pads. The core-to-pads router, ring, aims at doing this operation automatically, provided the user has given an appropriate netlist and some indications on pad placement.

The last stage of the physical implementation is the translation of the symbolic layout to a foundry compliant layout using the s2r tool. After that, the tape containing the circuit can be processed by the silicon supplier.

2.4 Verification

In our **VLSI** class, we intend to show that **VLSI** verification is at least as important as **VLSI** physical design. For that reason, we have introduced in the design flow powerful tools to perform behavior, netlist and layout verifications.

The correctness of the design rules is checked using the design rule checker druc.

An extracted netlist can be obtained from the resulting layout. Lynx, the layout extractor operates on both hierarchical and flattened layout and can output both flattened netlists (transistor netlist) and hierarchical netlists. The transistor netlist is the input of the yagle functional abstractor. Yagle provides a **VHDL** data-flow behavioral description, identical to the one that feeds asimut, from the transistor netlist of a circuit. The resulting behavior can be compared to the initial specifications using either asimut with the functionnal vectors used for the validation of the behavioral specification, or formally proved equivalent, thanks to the formal proof analyzer proof.

When extracted hierarchically, the resulting netlist can be compared with the original netlist by using the lvx tool. Lvx, that stands for Logical Versus Extracted, is a netlist comparator that matches every design object found in both netlists.

The critical path of the circuit, and an estimate of its delay, can be obtained using the static timming analyzer tas.

2.5 Test and coverage evaluation

For now, the fault coverage provided by the functional patterns is evaluated using a commercial fault simulator, as **Alliance** doesn't provide one yet.

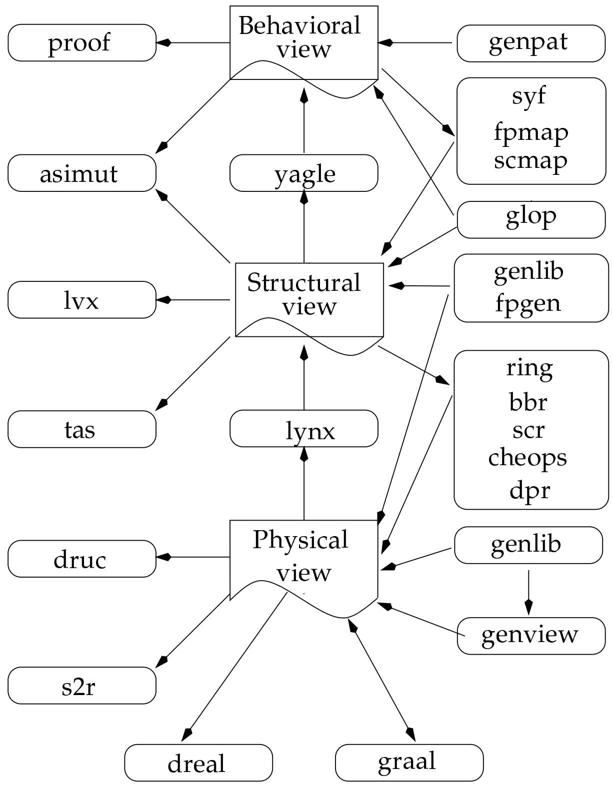

Figure 1: How the tools are linked on the data structures.

3 Tools and layout libraries of the Alliance package

Every **Alliance** tool has been designed to simply interface with each other, in order to support the proposed design flow. Nevertheless, each tool can also be used independently, thanks to the multiple standard formats used for input and output files.

One of the most important characteristics of the **Alliance** system is that it provides a common internal data structure to represent the three basic views of a chip:

- the behavioral view,
- the structural view,
- the physical view.

Figure 1 details how all the **Alliance** tools are linked together around the basic behavioral, structural and physical data structures.

The process independence goal is achieved with a thin fixed-grid symbolic layout approach. All the library of the system use this approach successfully. Layouts have been targetted to ES2 2μm, 1.5μm, 1.2μm, 1.0μm and 0.7μm technologies, the AMS 1.2μm technology and SGS-Thomson 0.5μm technology. Chips have been fabricated successfully through the **CMP** services on these technologies.

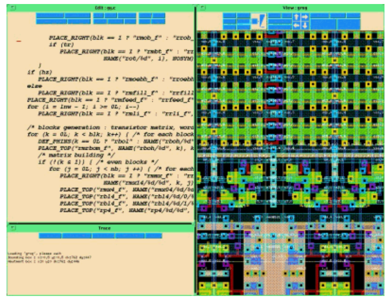

Figure 2: A typical run of `genview`.

3.1 Tools

- `asimut` is a **VHDL** logic simulator. The supported **VHDL** subset allows both structural and behavioral data-flow description (without timing information). Complex systems and microprocessors, including **INTEL** 8086 and **MIPS** R3000 have been successfully simulated with `asimut`. `Asimut` is based on an event-driven algorithm and powerful representation of boolean functions using binary decision diagrams.

- `genpat` is a language interpreter dedicated to efficient descriptions of simulation stimuli. It generates an **ASCII** file that can act as an input of `asimut`. A `genpat` file format to **MSA** translator allows the generation of appropriate simulation patterns for the Tektronix LV500 tester. This allows to perform functional tests when the circuits comes back from the foundry.

- `glop` is a gate level netlist optimizer. If the output of the logic synthesis takes into account the internal delays of the cells during the mapping phase, it doesn't take into account the fan-out problems. `Netoptim` work is to ensure that the drive capabilities of all cells are correct, and to try to minimize the delays on the critical pathes in inserting buffers where appropriate.

- `genlib` is a procedural language for netlist capture and placement description (there is no schematic editor in the **Alliance** system). `Genlib` provides a consistent set of **C** primitives, giving the designers the ability to describe **VLSI** circuit netlists in terms of terminals, signals and instances, or circuit topologies in terms of placement of abutment boxes. `Genlib` is mainly used to build parameterized netlist and layout generators.

- `genview` is a debugging tool for the development of the layout view of parameterized generators. It is a graphical environment that integrates a `genlib` interpreter, a step by step debugger, and a window in the which the circuit under construction is visualized. All the parameterized generators of **Alliance** have been developed using this tool. Part of the *ROM* generator `grog` under construction is shown figure 2.

  `Genview` uses the GNU `gcc` compiler parameterized for a virtual architecture as basis to its `genlib` interpreter.

- `fpgen` is a language that has moreorless the same functionalities as `genlib`, but it is dedicated to datapath description. Its primary difference with `genlib` is that it allows to manipulate vectors of cells, like 32 two inputs `nand` gates or a 32 bits adder. It contains many primitives that greatly simplify the description of operative parts, in an optimized manner.

- `bop` is a logic optimizer and logic synthesis tool. The input file is a behavioral description of the circuit using the same **VHDL** subset as the logic simulator. The boolean equations described in **VHDL** are optimized so to minimize the number of boolean operators. The output is a new, optimized, data flow description.

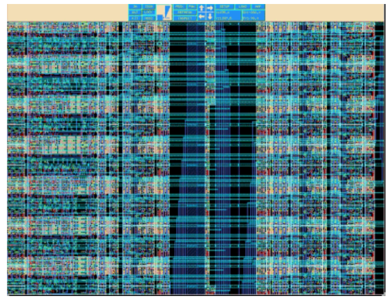

Figure 3: Part of a datapath.

- scmap is a logic synthesis tool. The output is a netlist of gates. scmap can map a data-flow description on any standard-cell library, as long as a **VHDL** data-flow description is provided with each cell.
- c4map is a logic synthesis tool. It has the same functionnality than scmap, but runs without a predefined standard-cell library, thanks to an internal cell compiler.
- syf is a finite state machine synthesizer. More precisely, syf assigns values to the symbolic states used for the automaton description, and aims at minimizing the resulting logic for both state transistion and output generation. The input is a fsm description, using a dedicated subset of **VHDL** that includes process description. The output is a behavioral description of the circuit using the same **VHDL** subset as the logic simulator. The output of syf is to be synthesized into a netlsit of gates using scmap.
- scr is a place and route tool for standard-cells. The placement system is based on simulated annealing. The channel router is an adaptation of the greedy router of Rivest-Fidducia. Feed-throughs and power routing wires are automatically inserted where needed. The input is a netlist of gates. The output is either an hierarchical (channels are instanciated) or flattened (channels are inserted) chip core layout without external pads. A specialized router is used for core to pad routing.
- Dpr is a place and route tool for bit slice oriented datapath. It privilegies the direct connexions between cells, and allows to used optimized blocks, like a fast multiplier or a register file, within the datapath. Most parameterized generators available in **Alliance** follow the bit-slice structure of this datapath compiler. This tool allows to mix some glue logic directly within a datapath. This functionnality doesn't exist in commercial tools.
- bbr is a gridless channel router that allows to route together two blocks having different topologies. For example the control part of a microprocessor realized in standard cell, and its operative part done as a datapath. Bbr is pretty tricky, and should be used with care.
- Ring is a specific router dedicated to the final routing of chip core and input/output pads. Ring takes into account the various problems of pad placement optimization, power and ground distribution. A set of symbolic pads is included in the package.
- S2r is the ultimate tool used in our design flow to perform process mapping. S2r stands for "symbolic to real", and translates the hierarchical symbolic layout description into physical layout required by a given silicon supplier. The translation process involves complex operations such as denotching, oversizing, gap-filling and layer adaptation. Output formats are either **CIF** or **GDSII**. S2r requires a parameter file for each technology aimed at. This file is shared with druc, lynx, graal, dreal and genview. From an implementation point of view, these tools use a bin-based data-structure that has very good performances.

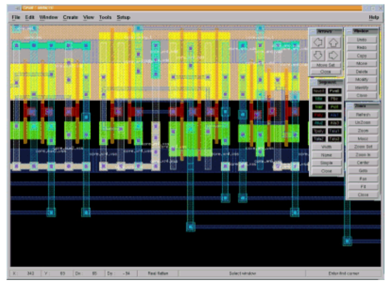

Figure 4: Editing some custom layout using `graal`.

- `druc` is a design rule checker. The input file is a - possibly hierarchical - symbolic layout. It checks that a layout is correct regarding the set of symbolic design rules. This correctness must be ensured in order for `s2r` to produce a layout compatible with the target silicon foundry.

- `Lynx` is a layout extractor. The input is a - possibly hierarchical - layout. The layout can be either symbolic or real. The output is an extracted netlist with parasitic capacitances. The resulting netlist can either be hierarchical or flattened (transistor netlist).

- `Lvx` is a logical versus extracted net-compare tool. The result of a run indicates if the two netlist match together, or if there are different. Note that `lvx` doesn't work at the transistor level.

- `yagle` is a functional asbtractor/disassembler for **CMOS** circuits. It provides a **VHDL** Data-Flow behavioral description from the transistor netlist of a circuit, by first extracting a pseudo-gate netlist, and second translating each pseudo-gate in boolean equations. The input file is a - possibly extracted - flattened transistor netlist. The output is a simulable behavioral **VHDL** model (data-flow without timing informations). `Yagle` can be distinguished from commercial CAD abstractors by the fact that it does not need a predefined cell library or transistor patterns. Furthermore, the use of a purely algorithmic approach compared to a pattern matching one implies a huge gain in performance. Yagle is not anymore part of Alliance, but is freely available at `http://www.avertec.com`.

- `tas` is a static timing analyzer. It takes as input a transistor netlist and produces a file containing all the combinatorial paths of the circuit, the critical path being outlined. Tas is not anymore part of Alliance, but is freely available at `http://www.avertec.com`.

- `proof` performs a formal comparison between two data flow **VHDL** descriptions that share the same register set. `Proof` supports the same subset of **VHDL** as `asimut`, `bop`, `scmap` and `yagle`.

- `graal` is an hierarchical symbolic layout editor. It requires a X-Window graphical environment and the Motif libraries. `Graal` is used for cell layout design or hierarchical block construction. It provides an on-line **DRC** and automatic display of equipotential nets. Editing a cell under `graal` is shown figure 4.

- `L2p` creates a Postscript file from a layout, symbolic or real.

3.2 Cell libraries

The **Alliance** package provide a wide range of libraries, either static, ie. fixed cells, or dynamic, as the block is produced by running a parameterized generator. These libraries are compatible with any two metals/one polysilicon technology.

Each object in the library has, for static ones, or produces, for dynamics ones, three views at least :

- the symbolic layout, that describes the cell topology.
- the netlist, in terms of transistor interconnections.
- the behavior, specified in **VHDL** data flow form.

Area loss due to the symbolic layout compared to micron design has been estimated ranging from 10% to 20%. In any case, loosing area is affordable, where loosing years is not.

3.2.1 Standard cell library

The `sclib` library contains boolean functions, buffers, mux, latches, flip-flops, . . . (around 70 cells). All the cells have the same height, share the power and ground lines on east and west side, and have pitched I/Os in metal2 on north and south side. They are supposed to be used with a usual standard cells place and route tool, such as **Alliance**'s `scr`, **Compass** or **Cadence**. These cells are to be used primary for glue logic, since optimized operators can be obtained using dedicated generators, as stated paragraph 3.2.2. The `logic` tool can map a behaviral VHDL onto this library.

The figure 3.2.1 below shows the difference between `sclib` and `dplib` regarding the shape and contents of a cell.

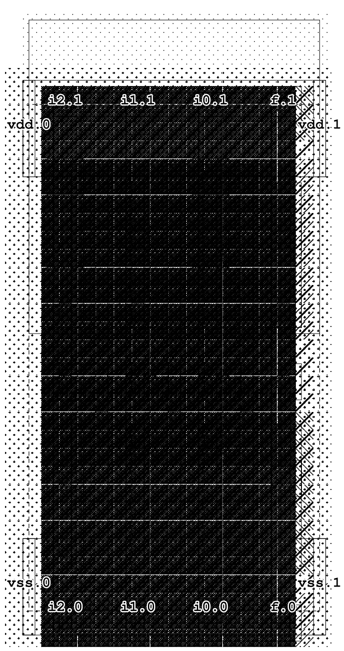

Figure 5: *Sclib* version of a three inputs `and` gate

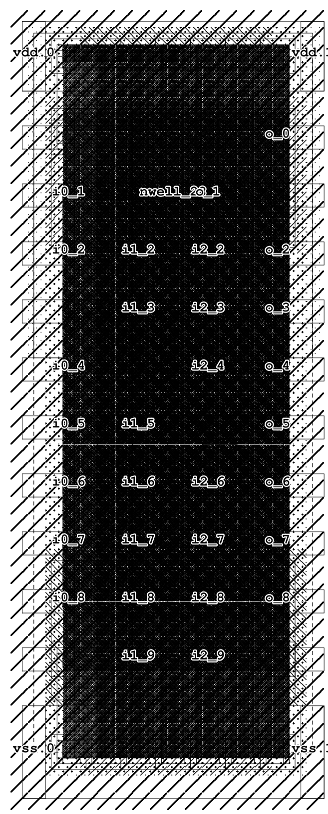

Figure 6: *Dplib* version of a three inputs `and` gate

3.2.2 Datapath libraries

There are two kinds of datapath libraries:

- `dplib` is a cell library dedicated to high density data-paths. It must be used in conjunction with the data-path tools `fpgen` and `dpr`. The cells in `dplib` have the same functionnalities as the ones in `sclib`, but have a topology that is usable only within a datapath. `Scmap` can also map a behavior onto the `dplib` library.
- `fplib` is a set of above 30 regular functions that are useful in the design of a datapath. These functions range from a n inputs `nand` gate to a n *times* m register file.

Here the cells share the power and ground lines in metal2. A powerful dedicated over the cell router can route custom blocks and logic glue in the same structure. Among the `fplib` functionnalities, four optimized blocks generators should be presented in more details, as they reflect the quality of this library. All the generators are build with a tiler using a dedicated leaf cell library. Their output is a symbolic layout, a **VHDL** behavior, a set of pattern for test purpose, a netlist, an icon, and a datasheet indicating size and timing estimation for a given technology. The structural parameters varies according to their functionalities.

- optimized generators for datapath operators:

  `rsa`, a fast adder generator, with propagation time in log nb and size in nb log nb, where nb is the number of bits. Its has 2 or 3 input buses, and if needed a carry input. It may be used as a substractor or adder/substractor.

Params	Meaning	Range
nb	number of bits	3 to 128
cin	carry in	true or false
csa	three inputs adder	true or false
ovr	overflow flag	true or false

  `rfg`, a static register file generator. It has one write address , and one or two read address. It may be operated as a set of level-sensitive latches or edge triggered flip-flops.

Params	Meaning	Range
nb	number of bits	2 to 64
nw	number of words	2 to 256
bus	number of read bus	1 or 2
op	mode of operation	latch or flip-flop
low power	reduce power consumption	true or false

  `bsg`, a barrel shifter generator. Possible operations are :

  - logical right shift
  - arithmetical right shift
  - logical left shift
  - arithmetical left shift
  - right rotation
  - left rotation

Params	Meaning	Range
nb	number of bits	3 to 64

`amg`, an integer modified booth algorithm array multiplier. the x and y inputs are independent, and pipeline stages can be inserted in the circuit.

Params	Meaning	Range
nx	number of bits of the x operand	8 to 64
ny	number of bits of the y operand	8 to 64
ps	number of pipeline stages to be inserted in the circuit	0 to $\min(\frac{nx}{2}, \frac{ny}{2})$-1

3.2.3 Custom libraries

Two full-custom parameterized generators are also available. They produce stand-alone blocks, that are to be routed only at the floorplan level with other blocks, using either `bbr` or better `x-cheops`.

- *ROM* and *RAM* generators:

  `grog`, a generic *ROM* generator. The interface is an address bus, a clock and an output enable signal, and a data out bus. The coding format to specify the *ROM* contents is a limited subset of VHDL.

Params	Meaning	Range
nb	number of bits	1 to 64
nw	number of words	64, 128, 256, n 512, $1 \leq n \leq 8$
hz	tri-state output	true or false

  `rage`, a *RAM* generator. The interface has a read/write address, a write signal indicating if a read or a write is to be performed, and a clock.

Params	Meaning	Range
nb	number of bits	2 to 128
nw	number of words	128 to 4096
aspect	aspect ratio	narrow, medium or large
ud	unidirectional, ie share the same bus for data in and out	true or false

All these generators have been designed using the **Alliance** CAD tools, for both design and verification phases.

3.2.4 Pad library

Alliance provides also a pad library. This library also uses a symbolic layout approach, and therefore a whole chip can be targeted on several technology without even the core to pad routing. A very robust approach has been enforced, as the pads are subject to electrostatic discharge, and also more sensible to latch-up than the other parts of the circuit due to the amount of current that flows through them.

Chips using these pads have been fabricated on ES2 1.0μm, AMS 1.2μm and SGS-Thomson 0.5μm technology and work as expected.

4 Supported exchange formats

The **Alliance** CAD system handles many file formats. They are summarized here. A file can be either read, using a *parser*, or written, using a *driver*.

- Behavioral view:
  - dataflow **VHDL** parser and driver.
- Structural view:
  - **VHDL** parser and driver.
  - **EDIF** parser and driver.
  - **Spice** parser and driver.
  - **Compass** parser and driver.
  - **Alliance** parser and driver.
  - **Hilo** driver
- Physical view:
  - **Alliance** parser and driver, for symbolic layout.
  - **Compass** parser and driver, for symbolic layout.
  - **Modgen** parser and driver, for symbolic layout.
  - **CIF** parser and driver, for real layout.
  - **GDSII** parser and driver, for real layout.

Being able to understand and write many file formats is a must. First, in a development environment, as it allows to check the validity of tools on other CAD systems. Second, because some tools are not available or desirable within **Alliance**, but may be useful however: it is possible to feed an other software with a design in that situation.

The experience showa that many of these formats are used daily. For example, the design that we fabricate through the CMP services are transmitted using the **GDSII** format. The final **DRC** on these files are performed using **Cadence** `pdverify`.

An other example: **Alliance** does not have a fault simulator yet. However this kind of tool is very useful to evaluate the fault coverage of a set of vectors and must be introduced in a **VLSI** class. This is hopefully easilly done using the **Hilo** output of **Alliance** that feed the `hifault` simulator.

5 Alliance internal organization

The complete **Alliance** CAD system contains about 600 000 lines of C code, and over 600 leaf cells. It compiles and runs on most **Unix** system, and requires the basic X-Window library X11 plus Motif. The distribution tape shows that there are three kinds of files:

- common data structures and manipulation primitives.
- parsers/drivers to read and write external file formats.
- actual tools.

Project	transistors	Functionality
Smal	17 000	one bit processor for SIMD architectures
Data-safe	35 000	dynamic data encryption chips
TNT	60 000	switch-router for T800 transputerss
FRISC	200 000	floating-point **RISC** microprocessor
StaCS	875 000	Very Long Instruction Word processor
Rapid2	650 000	SIMD systolic and associative processor
Rcube	350 000	Message router for parallel machines

Figure 7: Various chips designed with **Alliance**.

Alliance as been developed in order to simplify cooperative work between the CAD tool designers. The existence of a common data structure framework releaves the developer of many burdens: reading and writing many file format, conceptualizing the VLSI objects, writing classical high level and nevertheless complex functions, ... All the **Alliance** tools share these data structures and their related functions. So each tool communicates with the other ones smoothly, by construction.

6 Use of Alliance inside our laboratory

Alliance is used for both educational and research purposes. We relate our experience below.

Educational aspects

The **Alliance** System has been extensively used during the past eight academic years (89-97) as a practical support of two undergraduate courses: one on **CMOS VLSI** design, the other one on **advanced computer architecture**. These initiation courses lasts 13 weeks with a 2 hours lecture and 4 hours spent using the **Alliance** system per week, and involves 60 students and 3 teachers.

The '**VLSI** design' course is for students that have no previous knowledge on **VLSI** design and mainly come from two distinct channels: "computer science" and "electrical engineering" masters of sciences. During this course, students are required to design and implement an **AMD2901** compatible processor, starting from a commercial data-sheet. The chip, with a complexity of about 2000 transistors, is designed by groups of 2 or 3 students. The main interest in this course is to teach the design methodology. Most of the **Alliance** tools are used during this class.

The 'architecture' course focuses on the way processor architecture, from the system point of view and not from an implementation one. Typical **CISC** and **RISC** processors are studied, and part of them modelized using our **VHDL** subset. In that class, only the `asimut` simulator is used.

Alliance is also used in an intensive graduate course, for the design of the 32 bits microprocessor `dlx` **RISC** processor – 30000 transistors –. This course lasts two months, and aims only at the implementation : the high level behavioral model of the processor is given to the students. During that period of time, all the **Alliance** tools are used.

Research projects

These projects range from medium complexity ASICs developed in 6 months by a couple of designers **Data-safe**, **TNT**, **Smal**, **Rf264**,etc...) to high complexity circuits (**FRISC**, **Multick**, **StaCS**, **Rapid2**, **Rcube**) developed by a team of PhD students.

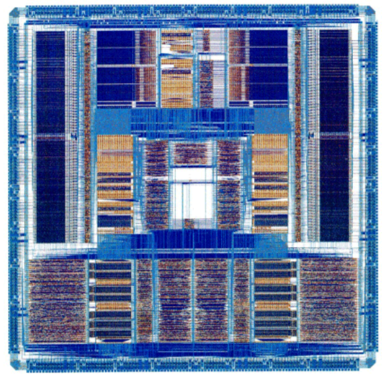

Figure 8: The 875 000 VLIW StaCS processor.

The three largest circuits described in table 7 use not only standard-cells but also parameterized generators for regular blocks like *RAM*s, data-paths, or floating-point operators. The **FRISC** and **TNT** projects successfully used the **Cadence** and **Compass** place and route tools, and therefore prove the interoperability of the **Alliance** system.

A picture of the **StaCS** processor is shown figure 8.

7 Conclusion

We are very satisfied to use a set of tools of our own for teaching **CMOS VLSI** design for two good reasons. First, we simply can't afford 50 high end workstations to run commercial CAD systems like **Compass**, **Mentor Graphics** or **Cadence**. Second, both the **Compass** and **Cadence** system have been used in research project at **LIP6**. They are powerful and sophisticated environments but are much too complex for novice undergraduate students. The great advantage of the **Alliance** CAD system is that we have done our best to stick to the basic yet powerful concepts of **VLSI** design. To each tool correspond a unique functionnality, that cannot be changed or worked around by parameter files. At last, we experienced that the technology migration and process independence are key issues. Hence, it is crucial to rely on a portable library at the symbolic layout level.

The **Alliance** package is now in use all over the world, and more than 250 sites have registered today. It is available through anonymous `ftp` at `ftp://ftp-asim.lip6.fr/pub/alliance/distribution/`, or through a `Web` browser at `http://www-asim.lip6.fr/pub/alliance/distribution/`.

There is an **Alliance** mailing list, where users can share their views and problems, and our team is always ready to answer questions. The address of this mailing list is `alliance-users@asim.lip6.fr`. The support of **Alliance** can be joined at `alliance-support@asim.lip6.fr`.