Using C to Write Portable CMOS VLSI Module Generators

Alain Greiner and Frédéric Pétrot
Laboratoire MASI, Université Pierre et Marie Curie
4, place Jussieu, 75252 Paris cedex 05, France

Abstract

We describe the use of the C *programming language as* procedural layout *language oriented toward development of portable parameterized generators for CMOS* VLSI. *Mostly dedicated to libraries design,* **genlib** *addresses both the process and software portability issues. Advantages of using a superset of* C *functions for symbolic layout and netlist description over proprietary vendor languages as* **L** *or* **Skill** *are discussed. Seven optimized generators using a tiler and leaf cell approach and three netlist parameterized generators have been developed at UPMC MASI using the procedural language* **genlib** *and the graphical debugger* **genview**.

1 Introduction

In *ASIC* design, the time-to-market is the critical issue. In order to reduce the design time, designers use synthesis tools that accept a behavioral input description. By now, the output from those synthesis tools is a gate netlist that uses a specific standard cell library. But standard cell implementations give poor results when high density or high speed is required. In order to reach high performances, synthesis tools must use more complex libraries containing optimized components like *RAM*, *ROM*, datapath operators, *etc.* Due to the large range of possible parameters, such complex blocks must be provided by parameterized generators[1]. We present a language that allows to design process independent portable parameterized generators. This work has been done at the University Pierre et Marie Curie, the MASI laboratory acting as an associated partner of BULL in the framework of the ESPRIT2 IDPS project. In the IDPS project, five European silicon suppliers — ST, SIEMENS, PHILIPS, PLESSEY AND ES2 — joined to develop a common portable ASIC library for a generic 0.8 μm CMOS technology.

Developing a module generator is a multi-disciplinary task that requires knowledge in the following areas:

- definition and evaluation of algorithms for integrated operators;
- logic design and electrical optimization;
- topological partitioning of the layout;
- software development and software verification;

As such generators are complex to design, portability is a must.

This paper focus on a layout assembling language and its related layout methodology. Section 2 introduces the major existing approaches to module generation. Section 3 defines the principles and goals for the definition of a new module generation language. Process independence is addressed using the symbolic layout methodology described Section 4. Section 5 justifies the choice of the general purpose C language as procedural generation language. An overview of the C library **genlib** is given Section 6, and the generator debugger **genview** is introduced Section 7. Practical results using these tools for generator design are given Section 8.

2 Existing approaches to module generation

The choice of a module generation language is much more than the choice of a syntax, it also implies the choice of a layout design methodology. Most layout generators rely on a "tiler and leaf cells" approach. The problem is to determine a set of basic cells that will be used as tiles in a macro-cell. The leaf cells are abutted, in a tiling fashion, to carry out the actual layout. This offers high densities and good electrical performances. The price to pay is the complexity of the abutment scheme, and the resulting number of leaf cells needed to satisfy all the generators parameters. For example, only 10 different logical gates are needed to design the netlist of a recurrence solver adder generator[2], but because of the topological constraints of the approach, 55 layout leaf cells had to be drawn.

The design of a generator is complex and time consuming. In order to preserve this investment, process independence of module generators is addressed by almost all existing approaches. The whole problem is to warranty that pitch matching constraints in both x and y directions are kept through technological retargetting. Two major methodologies are currently used.

Some languages, like L[3] or Slic[4], make use of parameterized leaf cells. The leaf cells are described procedurally as functions under a text editor, using the same language for the leaf cells and the tiler. Each layout primitive is placed using variable coordinates which will be given a process dependent value at generation time. This approach, called *dynamic virtual grid compaction*[5], provides a stretchable representation of the layout that is supposed to be usable with several technology files. It has two defaults. First, it's not intuitive, as from a designer point of view a cell is mainly a graphical representation. Leaf cell are naturally described under a graphical editor, and the visual aspect of the layout is very helpful in practice to integrate the complexity of the pitch matching constraints between neighbors. Second, the actual process portability is yet to prove. Integrating all the abutment constraints into a parameterized program is a very difficult task, that looks much like a compaction algorithm. The parameterized expressions using design rules to ensure correct abutment between two cells for a given technology won't necessarilly give the same result with other technologies. This can be done only by using the very same expressions for all objects of the facing edges of abutted cells, but this seems neither easy to enforce, nor very good for design quality because of constraint propagations. The verification of the generated objects must be done for many technologies to warranty actual process independence.

Other languages make use of a compactor, like **Skill**[6] or **Modgen**[7]. The cells are drawn under a stick diagram editor, and then tiled together by program. With compaction, cell size and pins' positions are not predictable. In order to warranty pitch matching, the tiler must explicitly contain "constraints"[8] definitions. Theses constraints have to be defined for each pin of each instance for proper abutment. The tiling problem becomes a very complicated problem, since all the pins of facing instances must be indicated with constraints so to match after the compaction process. For **Skill**, five compaction algorithms and three constraint types can be chosen from for each pin. This shows that such programs are not anymore "tilers". Moreover, the use of hierarchical compaction requires a half design rule guard distance between abutted cells, as internal elements cannot be taken into account, leading to potential area loss already at the leaf cell level.

These methodologies put the burden of layout portability on the designers, either making leaf cell design a complex software problem and giving a hardly portable result, or having the tiler describe a constraints graph between instances connectors that the compacter may not be able to solve for various technologies[9].

3 Principles of the proposed approach

Defining a new language for procedural generation is useful only if it provides a more portable, more structured and simpler approach to generator design. Therefore, we define the following principles:

Process independence: the generated objects must be portable on a wide set of technologies. Ensuring the respect of pitch matching constraints in both directions is obtained by a symbolic layout on fixed grid approach. It is described in Section 4.

Separation of leaf cell and tiler: leaf cells are graphical objects, and designers are used to draw them under a layout editor. A custom cell is shown Figure 1. The natural way to design them is graphically. Placing rectangular instances side by side is a far more regular task than leaf cell design, and it can be done efficiently by the tiling program.

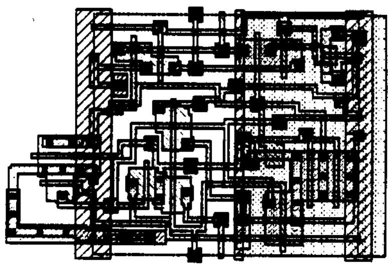


Figure 1: A typical cell of an optimized generator. (The abutment box is drawn as a bold rectangle.)

Separation of netlist and layout description: some generation languages need a netlist to be able to compact a layout correctly. But algorithms for placement and interconnections are not alike, and merging both may introduce many constraints on the designer.

Use of a general purpose portable language: what's the point with defining a new language, with specific syntax and flow control statements? Using a general purpose language enhanced with dedicated functions seems a reasonable choice. A generator is too much of an investment to be developed in the restricted framework of a commercial CAD system. Interoperability with these system is however a need for practical use of the generated blocks.

Wide range of file formats: in order to be usable within different design flows and different CAD environnements, several file formats must be available for the differents views produced by a generator.

It is useful to be able to describe two kinds of generators:

- some functional blocks have an intrinsic topological regularity: *ROM*, *RAM*, integer multiplier or adder. For those blocks, optimized layout can be generated using a "tiler and leaf cells" approach. Each generator uses a private cell library with the right abutment properties.

- other functional blocks are more easily described as parameterized soft macros because they do not have good topological properties: floating point arithmetic operators are a good example. What is actually generated is a standard cell netlist.

Both approaches rely on a procedural language providing either sophisticated placement functions for procedural layout, or connectivity functions for procedural netlist.

4 Portable layout

Process independence is addressed by a symbolic approach on fixed grid[10]. Our symbolic methodology is not based on compaction, but keeps the advantages of design speed. It uses graphical primitives laid out on a thin fixed grid and a restricted set of symbolic layers. The symbolic to target process is an improved linear shrink. Unlike the "usual" λ[11], what is snapped to the grid are not polygones edges but the center or axis of the basic symbolic primitives. As a result, the output of a generator is still portable across technologies, and can be instanciated as it is in a chip. The translation to the target process may take place at the chip level if the whole design uses the same symbolic approach.

Careful examination of over twenty different processes ranging from 2 μm to .6 μm have leaded to the definition of a generic set of symbolic design rules. The important idea is that while minimum widths and spacings are quite different through this sample of technologies, the pitches — axis to axis distances — vary more homogeneously.

The mask artwork is produced from the symbolic layout with a fully automated tool that uses a technological file parameterized for the target process. The translation is done hierarchically, and its primary steps are:

- compute the value of the λ for the process. The value of the λ is determined by an expert from the micron design rules.
- shrink the size of the symbolic grid to the λ value chosen for the process. Since the grid changes homothetically, the relative position of the center of the symbols is kept.
- adjust the width, W, of the runs, using the formula:

  $$W_{real} = (W_{symbolic} - W_{symbolic\ min})\lambda + W_{real\ min}$$

  This shows that only non minimal runs depend on the value of the λ, and that each layer is treated separately. Transistors are runs of a specific layer and are macro generated with a similar formula.
- adjust the transistor channel widths. Different technologies produce different transistor current ratio I_P/I_N. Note that changing the channel lengths would decrease electrical performances.
- expand the symbolic primitives defined with a single point, such as contacts.

This is done with a linear time algorithm. Notches and minimum distance between implant areas are corrected with a $O(\sqrt[3]{n})$ post treatment, where n is the number of rectangles of the layer for the given hierarchical level. The result is a **cif** or **gdsII** file suitable for the foundry.

Regarding process independence, this method is close to the one used in [12]. Its main advantage though is a more structured approach using symbolic objects for leaf cell drawing, that warranties higher productivity. Predictable area loss ranges from 10% to 20% compared to a process dedicated custom approach. Approaches using compaction lead to similar results, but require a much more complicated tiler.

5 Justification of the C language choice

Software portability is addressed using the C language. We believe that a tool being dedicated to libraries development should be independent of any CAD vendor and the software environment required to compile and run the generators should be as simple as possible.

The C language is a good candidate. C in itself allows computations, flow control and function calls. The tiling language is actually a library of predefined C functions. The generation language **genlib** can be easily extended by integrating a new function into a dedicated library. Unlike proprietary languages as L[3], **Skill**[6] or **Modgen**[7], C is universally taught and known. It's not "yet an other language", is independent of any vendor, and is today the standard portable language. Type checking of function parameters was the main reason in the past for the introduction of languages dedicated to procedural generation, but this is not anymore applicable with modern compilers with strong static type checking capabilities. The only interest of *ad hoc* languages, since they are interpreted, is their debug capability. We present Section 7 a tool that provide this functionality.

Other general purpose languages could be used, like **lisp**, **pascal** or **fortran**. But these, unlike C, are not available on any UNIX machine.

6 Overview of genlib functionalities

Although **genlib** allows procedural description of leaf cells, the main purpose of the language is to write tiling code in order to generate complex parameterized block. In our tiling approach, using fixed grid symbolic layout, the two main difficulties are cell placement and reliable physical interface definition. Powerful symbolic placement functions avoid any coordinates computation in the generator: all coordinates are implicitly retrieved from the already placed cells. With this approach, interconnections are embedded in the leaf cell layout. Nevertheless, in somes case it can be necessary to make explicit routing between distant cells or between two sub-blocks. The **genlib** library provides a channel routing function. Figure 2 shows a simple block done using relative placement functions and procedural wires and contacts. Wires can be drawn between two cells, with either an L or S shape under designer control. The wires' extremities are given by either connectors or points inside cells called references. The references may also be used for procedural contacts placement, as in a *RAM* address decoder, so to minimize the number of leaf cells. When two blocks, such as a register file and its decoders, have been design without paying much attention to the exact connector locations on the instances faces, ie not allowing direct abutment, the channel router function can be called to materialize the connections.

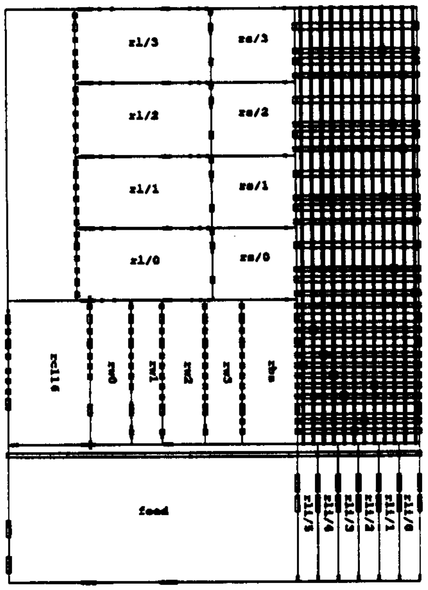


Figure 2: a small placement example.

As for the layout view, **genlib** provides the basic functions for interface definition and cell instantiation for the procedural description of a parameterized gate netlist. All the semantics founded in **VHDL** structural descriptions can be described using **genlib** functions. But **VHDL** functions are high level behavioral descriptions, and therefore not easily applicable to produce netlists of gates. **Genlib** provides a procedural approach to parameterized netlist generation with the full power of C and two advanced functions. The first one is a flattening utility that suppresses a hierarchical level in a hierarchical netlist. The other one "unflattens" a netlist: it creates a new hierarchical level into a hierarchical netlist without having to rewrite any interface. It is very useful when floor plan constraints requires to split a big netlist of gates into several pieces. The Figure 3 shows the behavior of the function.

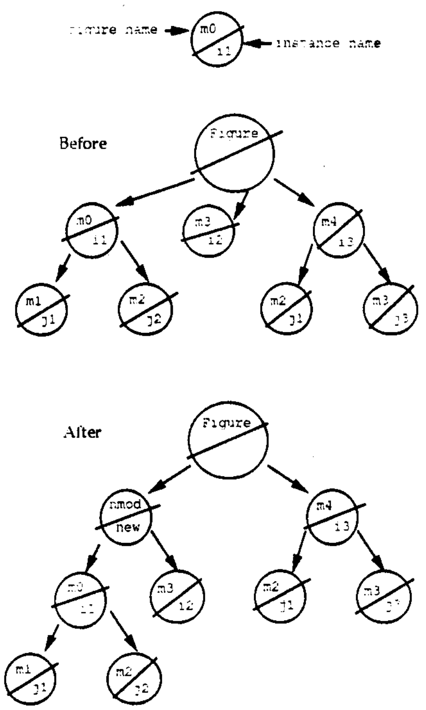


Figure 3: "Unflaten" behavior, the hierarchical level nmod is created with the instance name new.

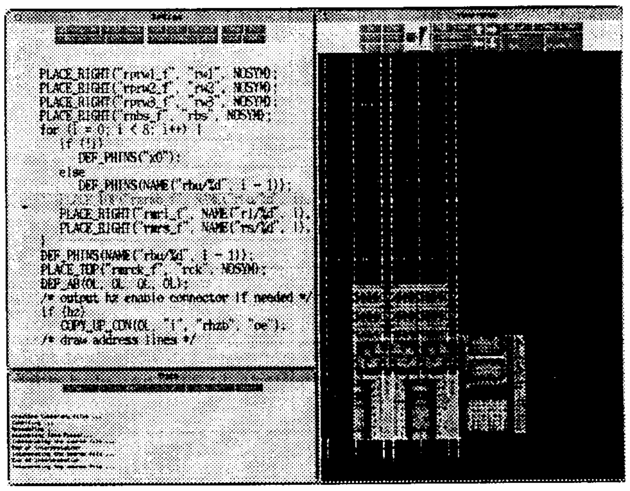
7 Graphical C debugger genview

As any complex piece of software, a generator must be debugged[13]. Given the large number of possibly generated blocks, this requires a powerful dedicated debugger. **Genview**[14] is a graphical C interpreter that allows step by step visualization of the tiling process, with many possible debug functionalities, like break points, variable trace, *etc*.

The heart of the C interpreter is based upon the *Gnu* gcc[15] compiler. Writing such a interpreter for a dedicated language would have required much more work with probably less efficiency. A typical view of the tools is given Figure 4.

8 Practical results

The **genlib** C library contains about 70 functions, with UNIX on line documentation. The **genlib** functions have been successfully used for both education and industrial projects.



```
PLACE_RIGHT("rprw1_f", "rw1", NOSYM);
PLACE_RIGHT("rprw2_f", "rw2", NOSYM);
PLACE_RIGHT("rprw3_f", "rw3", NOSYM);
PLACE_RIGHT("rnbs_f", "rbs", NOSYM);
for (i = 0; i < 8; i++) {
   if (!i)
      DEF_PHINS("x0");
   else
      DEF_PHINS(NAME("rbu%d", i - 1));
   [illegible]
   PLACE_RIGHT("rwrl_f", NAME("rl%d", i),
   PLACE_RIGHT("rwrs_f", NAME("rs%d", i),
}
DEF_PHINS(NAME("rbu%d", i - 1));
PLACE_TOP("rwrck_f", "rck", NOSYM);
DEF_AB(0L, 0L, 0L, 0L);
/* output hz enable connector if needed */
if (hz)
   COPY_UP_CON(0L, "i", "rhzb", "oe");
/* draw address lines */
```

Figure 4: The three main windows of genview.

Seven layout generators have been developed in two years using **genlib** and the graphical interactive symbolic debugging environment **genview**. The estimated design time for each of them ranges from one half to one man year, depending upon designer experience, complexity of the layout and software, *etc*. Four of them have been designed to fit in a datapath structure with over the cell routing capabilities[16]. The remaining three are *RAM*, *ROM* and *FIFO* generators. The tables below give run times — measured on a Sparc 2 —, densities, and delays — simulation of the extracted critical path for worst case parameters at 70°C — for a 1 μm technology. Entry delay in the tables is the read access time for the sequential blocks.

- A fast adder generator[2], with propagation time in $\log N$ and size in $N \log N$, where N is the number of bit. This adder has been used a test vehicle and fabricated on a large set of technologies: 1.2 μm, 1.0 μm, 0.8 μm, 0.7 μm and 0.5 μm.
- A static register file generator. It may be operated as a set of level-sensitive latches or edge triggered flip-flops.
- A barrel shifter generator.
- An integer modified booth algorithm array multiplier.

block	run time	trans-istors	Ktrs/ mm^2	delay (ns)
32 bit, 2 inputs adder	3 s	1513	4.7	10.5
32x32 multiplier	10 s	23875	5.4	52.0
32 bit shifter	5 s	2828	2.5	9.5
32x32 register file	14 s	11656	5.3	8.0

Use of these operators in datapath structures leads to density up to 5Kt/mm^2 using a dedicated over the cell router[17].

- A high speed *ROM* generator, with three state outputs.
- A static *RAM* generator, with selectable aspect ratio.
- An asynchronous *FIFO* generator.

block	run time	trans-istors	Ktrs/ mm^2	delay (ns)
64 Kb *ROM*	47 s	75048	14.3	8.5
1Kx32b *RAM*	11 s	201686	11.0	11.5
32x32 *FIFO*	7 s	12176	6.1	8.0

This table gives quantitative values on the generators.

generator	lines of code	number of leaf cells	options
adder	1000	53	10
multiplier	1400	78	5
shifter	600	56	2
register file	2600	81	10
ROM	1100	100	7
RAM	1000	69	5
FIFO	300	66	2

Most of those generators have been used in several academic and industrial circuits.

• The parameterized netlist generators have been essentially dedicated to floating point arithmetic, for an adder, a multiplier and floating point format converters[18].

An other possible use of the **genlib** language is one shot procedural layout for custom blocs:
The cache of a BULL DPS7 chip has been assembled with **genlib**. This block contains about 800.000 transistors, and the code to generate it, written by a non-programmer, is 2000 lines.
The StaCS[19] circuit, high complexity demonstrator of the **IDPS** project containing about 875.000 transistors, has been implemented using several generators and dedicated one-shot custom blocks written in **genlib**. The target process is 0.8 μm CMOS.

9 Conclusion

A procedural generation language is required for fast development of libraries. Compared with existing languages, **genlib** main advantage is to ensure a portable and simple approach to the tiling problem. The layout portability is granted thanks to the fixed grid symbolic methodology. Portability from the software point of view is provided by the standard C compiler. Fast design cycle is obtained with the graphical debugger **genview**. Silicon compilation evolution depends on the availability of powerful parameterized module generators. **Genlib** actually provides an efficient development environment for those portable generator libraries.

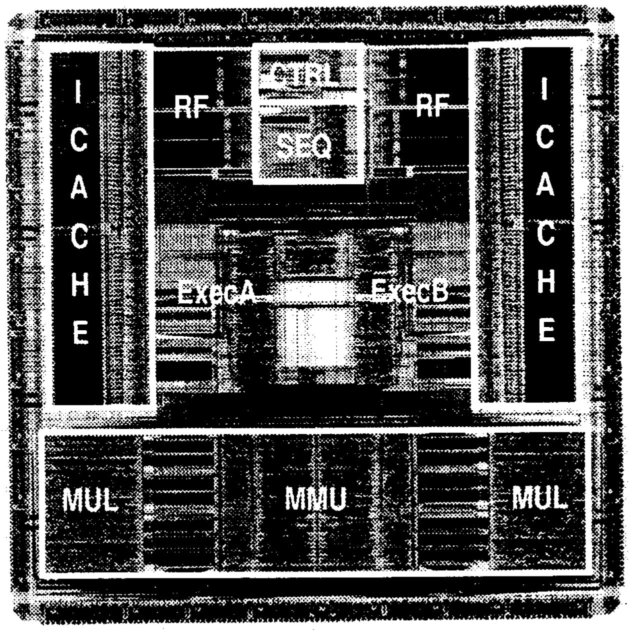


Figure 5: StaCS floor-plan.

References

[1] Daniel D. Gajski, Nikil Dutt, Allen Wu, and Steve Lin. *High Level Synthesis, Introduction to Chip and System Design*, chapter 1, pages 18–19. Kuwer Academic Publisher, 1992.

[2] Luis Lucas, Alain Greiner, and Michel Thill. High speed adder generator. In *5th International Conference on Microelectronic*, pages 136–139, December 1993.

[3] Mentor Graphics Corporation, 2045 Hamilton Avenue, San Jose, CA 95125. *GDT Designer, L Database and Language Users Guide*, September 1990.

[4] Cascade Design Automation Corporation, 3650 131st Avenue SE, Bellevue, WA 98006. *CDS Users's Manual and CDS Reference Manual*, April 1993.

[5] M.R. Burich. Design of module generators and silicon compiler. In D. D. Gajski, editor, *Silicon Compilation*, chapter 2, pages 49–94. Addison Wesley, 1988.

[6] Cadence Design Systems, Inc, 555 River Oaks Parkway, San Jose, California 95134. *SKILL Reference Manual*, October 1991.

[7] Henry Janssen. *Modgen and graphMG User Manual*. Philips Research Laboratories, Eindhoven, The Netherlands, 1991. Modgen has been chosen by the IDPS ESPRIT project as generator exchange language.

[8] Y.Z. Liao and C.K. Wong. An algorithm to compact vlsi symbolic layouts with mixed constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 2.2:62–69, April 1983.

[9] Cyrus S. Bamji and Ravi Varadarajan. Mtsc: A method for identifying overconstraints during hierarchical compaction. In *30th Design Automation Conference*, pages 389–394, 1993.

[10] Alain Greiner and Jean-Pierre Leroy. A symbolic layout view in edif for process independant design. In *Fourth European Edif Forum proceedings*, Daresbury, Cheshire, UK, 1990.

[11] Carver Mead and Lin Conway. *Introduction to VLSI systems*. Addison Welsey, 1980.

[12] E.K. Cheng and S. Mazor. The genesil silicon compiler. In Daniel D. Gajski, editor, *Silicon Compilation*, chapter 9, pages 361–404. Addison Wesley, 1988.

[13] G. Wood and H-F.S. Law. Skill - an interactive procedural design environment. In *IEEE Custom Integrated Circuits Conference*, pages 544–547, Portland, OR, May 1986. IEEE, Inc.

[14] Arnaud Compan, Francois Pêcheux, Alain Greiner, and Frédéric Pétrot. A portable source level debugger for macro cell generators. In *The European Conference on Design Automation*, pages 408–412, February 1991.

[15] Richard Stallman et al. Gnu gcc c compiler version 1.42. 675 Mass avenue, Cambridge, MA 02139, USA, 1989.

[16] Lotfi Ben Ammar and Alain Greiner. A high density datapath compiler mixing random logic with optimized blocks. In *The European Conference on Design Automation*, pages 294–298, February 1993.

[17] Philippe Chaisemartin. Comparaison and evaluation of different datapath generators. Thomson ST Internal technical report, June 1992.

[18] Jean-Arnaud Francois. *F-RISC:Machine risc modulaire intégrant une unité flottante*. PhD thesis, Laboratoire cao-vlsi, 4, place Jussieu, 75252 Paris cedex 05, France, 1992.

[19] Alain Greiner, Luis Lucas, Franck Wasjbürt, and Laurent Winckel. Design of a high complexity superscalar microprocessor with the portable idps asic library. In *The European Conference on Design Automation*, March 1994.