CAIRO: A HIERARCHICAL LAYOUT LANGUAGE FOR ANALOG CIRCUITS

M. A. Dessouky, A. Greiner, M.M. Louerat
LIP6, Université Pierre et Marie Curie, FRANCE

KEYWORDS : Analog Layout Automation, Design Reuse

Abstract: We present a dedicated language for layout generation of analog circuits. The language is based on the C programming language. It relies on the symbolic layout approach used in the ALLIANCE VLSI CAD system to achieve technology independence. The language includes a hierarchical area optimization algorithm based on slicing structures with top-down constraint propagation. Topological and electrical constraints specific to analog circuits are also taken into account. The code describing the layout is independent of the device dimensions and of the target technology.

INTRODUCTION

In recent years, the increasing density of integration (systems on chip) has lead to a big growth in mixed analog-digital IC's. For digital circuits, designers can use parameterized module generators for regular blocks, such as a RAM or a ROM. The design of analog module layout generators is more complicated since it has to meet the two following requirements:

The first issue is the genericity : for a given fabrication process, and a given analog function, a module generator must handle a range of functional specifications. Even a small variation in those specifications can introduce large and nonuniform variations in the device sizes. On the other hand, there are specific constraints for the layout of analog cells that can hardly be handled by a general-purpose place and route tool.

The second issue is the portability: For a given analog function together with a given functional specification, the physical layout strongly depends on the fabrication process. The design, the validation, and the documentation of an analog module generator is a major investment. So the process independence is highly desirable for design reuse and process migration.

The CAIRO language is intended to help the writing of portable, parameterized layout generators for analog cells. CAIRO, being a language, allows all the flexibility to the analog designer. The language provides parameterized, technology independent device (leaf-cell) generators, and helps the designer to solve the floor planning problem subjected to component size variations.

PREVIOUS WORK

We can classify the existing approaches for analog layout automation into two main groups: the knowledge based approaches and the optimization based ones.

In the first group the circuit topology is always fixed. A *sound* topological arrangement for the building blocks of the circuit is stored based on traditionally accumulated design experience. Knowledge storage can either be in the form of a procedural layout [1, 2], through the use of topology libraries (slicing trees) [3], by employing a *design by example* principle (layout templates) [4], or even through stored rules [5].

The second group of approaches employ an optimization algorithm to generate a suitable placement followed by an analog routing phase [6, 7]. They are fully automated and strive to take a large number of specific analog constraints into account. A recent modification is introduced by including quantitatively the performance degradation in the cost function of the numerical algorithm through the use of a set of simulated sensitivities [8, 9, 10].

All of the above approaches have been verified with respect to analog building blocks (op amps, comparators, ...). As the knowledge based approaches offer a fast generation time, and a reuse of the expert knowledge (which seems to be indispensable to the analog domain), they suffer from their high design cost and thus are best suited for circuits of frequent use. On the other hand, the optimization based approaches offer a user-independent layout generation which normally satisfies the required circuit performance, but they suffer from the large generation time, and they are thus best suited for circuits with small number of devices.

THE LANGUAGE

CAIRO is based on the C programming language, and relies on the symbolic layout approach used in the ALLIANCE VLSI CAD system [11] to achieve technology independence. The input is a SPICE net-list. The language offers powerful functions, including a dynamic optimization of the aspect ratio, while respecting the topological and electrical constraints specific to analog cells. The code describing the layout is therefore independent of the device dimensions and of the target technology.

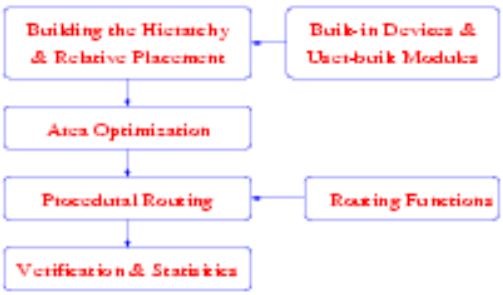

Figure 1: *Sections of CAIRO*

Figure 1 shows the different sections of the code describing a layout.
The language comprises two key features:
The first one is *a parameterized built-in device library*. The parameters of each device include its electrical characteristics (for example, the channel length and width of a MOS transistor). Another parameter is the device aspect ratio which determines the shape of the implemented layout (for example, the stacked MOS transistor [12]). The shape of the generated device is therefore determined by an *external* constraint. A symbolic layout approach is used to ensure portability.
The second feature is *the hierarchical, top-down area optimization algorithm* implemented by the *CAIRO_RESHAPE()* primitive.

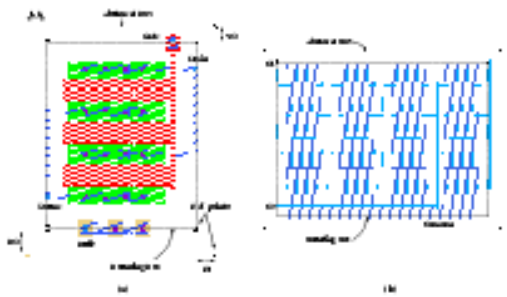

Figure 2: *Device characteristics: (a) A folded transistor, (b) A two capacitor array*

CAIRO allows the designer to use two types of blocks:
Built-in devices : Selected from the device generator library (Figure 2).
User-defined modules : The language allows the designer to build his own modules (sub-circuits), and to handle those created modules as if they were built-in devices. As for the built-in devices, user-defined modules can accept an aspect ratio parameter as an *external* constraint.

HIERARCHICAL PLACEMENT

While constructing a module, CAIRO supports a hierarchical approach based on *slicing trees* in order to facilitate the layout placement task. The predefined hierarchy is shown in Figure 3-a, while the corresponding slicing tree is shown in Figure 3-b. CAIRO follows the hierarchy provided by the designer to place all the elements.

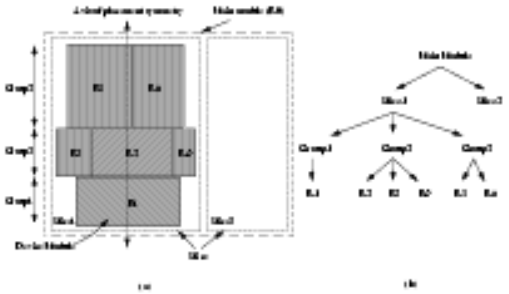

Figure 3: *(a) CAIRO predefined hierarchy. (b) the corresponding slicing tree*

The basic elements are:
The Device. This is the leaf cell of the module, it is selected from the language built-in parameterized components (transistors, transistor-pairs, resistances, capacitances, ...). It can also be a previously defined module.
The Group. This is composed of a horizontal arrangement (physical row) of devices and/or modules, placed besides each other in a specific order.
The Slice. This is composed of a vertical arrangement (physical column) of groups, placed on top of each other in a specific order. After layout generation, each slice preserves a vertical axis of symmetry passing by its center.
The Module. This is composed of a horizontal arrangement (physical row) of slices, placed besides each other in a specific order. A module is considered as a building block (a sub-circuit) that can be used to construct other modules, till the complete layout (*main module*) is described.

AREA OPTIMIZATION

The input to a layout generator written in CAIRO is a SPICE netlist that defines the electrical characteristics of the instantiated devices. A given netlist typically contains a large variety of device sizes and for each device size, a number of alternative implementations (aspect ratios) are possible. For exam-

ple, the aspect ratio of a large MOS transistor depends on the number of *folds* used (Figure 2). The main goal of the hierarchical top-down area optimization (HTDAO) algorithm is to determine the best aspect ratio for each device starting from a global external constraint.

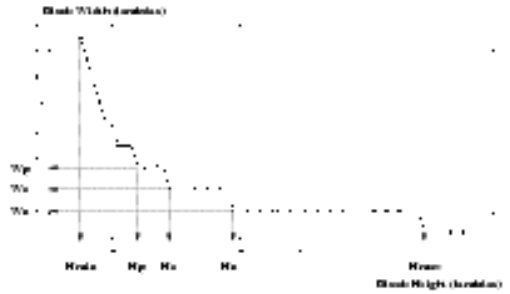

Figure 4: *Folded MOS transistor Shape function*

The HTDAO algorithm uses two *recursive* functions during the optimization process: the *AREA()* function and the *SHAPE()* function.

The AREA(module) function. Takes a pointer to the module as an input and returns an estimation of the area occupied by the module (for given device electrical parameters). This module estimated area is computed by summing the estimated areas of the child devices and modules. It is therefore sufficient to define, for each device generator in the built-in library, an area estimation function that depends on its electrical characteristics. The minimum and maximum values allowed for the module height, *Hmax* and *Hmin*, are also calculated. This function is used to initialize the area optimization process.

The SHAPE(module,height) function. Takes a pointer to the module and a given maximum allowable *height* as an input and returns the corresponding module *width*. A *SHAPE()* function is defined for each built-in device. This function is a discrete monotonic decreasing function since the *height.width* product is roughly constant, that depends on the electrical characteristics of the corresponding device. An example of this function is illustrated in Figure 4 for the case of a folded MOS transistor. Long intervals during which the device width is constant with respect to its height can be seen from Figure 4. At each transition point (Hc,Wc) we store the previous (Hp,Wp) and the next (Hn,Wn) transition points. These points are computed in the case of the folded MOS transistor by decrementing/incrementing the current number of folds by one respectively. This avoids unnecessary calculations between transitions. The minimum and maximum values allowed for the independent side, *Hmax* and *Hmin*, are also calculated to avoid infeasible device implementations. The core of the HTDAO algorithm under constraints consists of deriving the *SHAPE()* function of a module starting from its child devices and modules through *SHAPE()* function propagation [3].

The hierarchy has been chosen to facilitate the process of area optimization. Starting from the module level, each slice is independently optimized under the same height constraint. During this optimization process, slices call their internal groups which in turn call their internal devices. A propagation of the height constraint thus takes place similar to that in [3].

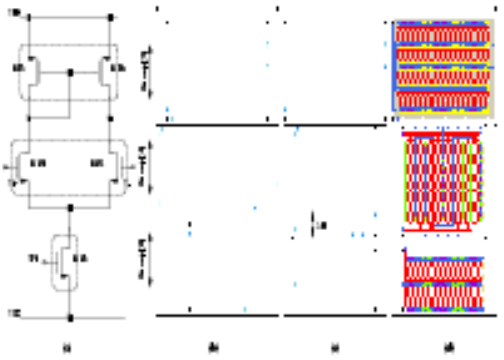

Figure 5: *Area optimization steps*

For a given slice, let H be the external height constraint, HS and WS be the slice height and width respectively, $\underline{h}$ be the set of group heights and $\underline{w}$ be the set of the corresponding widths, then the problem of slice area optimization can be formulated as follows:

$$\begin{aligned} \textit{minimize} \quad & WS = max(w_i) && (1)\\ & w_i = f_i(h_i) && (2)\\ \textit{subjected to} \quad & HS = \sum_i h_i \leq H && (3)\\ & h_{i\min} \leq h_i \leq h_{i\max} && (4) \end{aligned}$$

Since the slice height H is given as the external constraint, area minimization reduces to the minimization of the slice width WS as given by Equation 1. The function set $\underline{f}$ given in Equation 2 represents the corresponding shape functions for the groups. Equations 3 and 4 represent the optimization constraints.

The algorithm. The proposed algorithm is illustrated by the example in Figure 5. Figure 5-a shows the schematic of a simple OTA, and its corresponding hierarchy. The selected hierarchy is composed of three groups. Optimization starts with a given desired slice height and proceeds in two phases:
An initial set of group heights $\underline{h}$ is calculated such that the given slice height H is divided between

the groups in proportion to their estimated surface areas using the *AREA()* function (Figure 5-b). It is clear then that the slice width WS is determined by the width of the widest group (i.e. group[1]) as shown in Equation 1, while the slice height is given by Equation 3.
In order to decrease WS, the *height* of the widest group must be increased by a certain amount ΔH according to its shape function. This ΔH must then be subtracted from the other groups' heights in order to keep the total slice height $HS \leq H$. This is shown in Figure 5-c, where the height of group[1] has been increased by ΔH and the height of group[0] is reduced to compensate for this ΔH. The width of group[2] then becomes the new slice width WS. This process is then repeated till the smallest width is reached.
The algorithm is summarized by the following function:

```
OPTIMIZE_SLICE(H)
Phase 1:
   FIND the initial set of group heights h_i;
Phase 2:
   DO {
        FIND the group j such that w_j = WS;
        FIND ΔH needed to decrease w_j;
        FOR each group i not equal to j and
             WHILE (ΔH > 0)
             DO {
             Decrease h_i by Δh_i such that w_i < w_j;
             ΔH = ΔH - Δh_i;
                  };
        IF (ΔH <= 0) i.e. it is compensated by
             the other groups
             Conserve the new set of heights;
        ELSE
             The smallest WS has been found;
   } WHILE (Smallest WS has NOT been found);
}
```

Figure 5-c shows the slice after optimization.

The *CAIRO_RESHAPE()* function is a part of the CAIRO language. It takes the main module as an input and performs the area optimization as described above. All slices in a given module are forced to have the same height, then the above process is repeated independently for all slices in the module.
Note that in the case of the presence of a module in a slice (a module A instantiated inside another module B) the A module width is computed each time its dimensions are requested for. However, as the initial solution obtained in phase *1* based on the simple recursive *AREA()* function is generally a good guess, the number of iterations in phase *2* using the *SHAPE()* function is small. Moreover, due to the storage of the next and previous discrete dimensions of each device, many dimension re-calculations are avoided and the time needed for the optimization remains small.

ROUTING

Routing is done explicitly by the designer, i.e. the designer has to describe, using the language primitives, how each terminal is physically connected to other terminals. The language multi-segment routing functions use a relative approach. Those functions do not depend on the absolute coordinates of the terminals, all coordinate values are automatically retrieved from terminal names and reference points in the instantiated devices (Figure 2), these reference points are used during routing to specify wire break-points. This allows routing flexibility with respect to different shapes of the same layout. This task is the most time consuming one while describing a given layout. Several approaches to capture the routing are now under investigation: the use of a language *graphical debugger* [?], the graphical capture through the use of a symbolic graphical editor, and automatic channel routing dedicated for analog circuits.
Design rule checking is required to ensure the correctness of the resulting layout. This is done by a special verification function using a set of portable, symbolic layout rules [11].

PORTABILITY

Process independence is achieved using a symbolic approach on a fixed grid [?]. This approach uses graphical primitives laid out on a thin fixed grid and a restricted set of symbolic layers. Unlike the usual λ approach [?], what is snapped to the grid are not polygon edges but the center or axis of the basic symbolic primitives. As a result, the output of compiled code is still portable across technologies. It can also be instantiated, as it is, on a mixed analog-digital chip using the same approach.
Careful examination of over twenty different processes ranging from 2 to 0.6 μm has lead to the definition of a generic set of symbolic design rules. The basic idea is that while minimum widths and spacings are quite different through these sample technologies, the pitches (axis to axis distances) vary more homogeneously. Verification of the rules is also done on the symbolic view using a special tool. These design rules are transparent to the designer and are substituted by symbols in the language syntax.
The translation to the target process may take place after the generation of the layout or at the chip level if the whole design uses the same symbolic approach, using an automatic symbolic to real conversion tool that uses a technological file parameterized for the target process [11].

EXAMPLE

Figure 7 shows the CAIRO program describing the layout of the folded-cascode OTA circuit shown in Figure 6.
Module definition begins by the *CAIRO_OPEN_MODULE()* function which declares the corresponding module. For simplicity this example contains only *one* module.

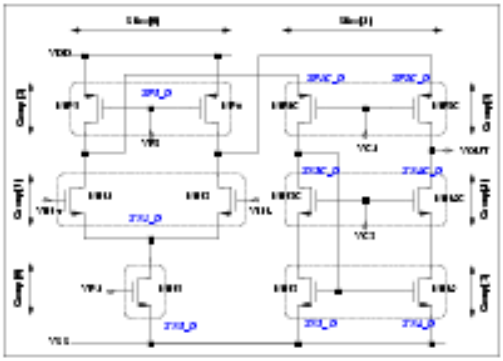

Figure 6: *FC-OTA module*

Association of built-in devices to layout components is performed while declaring all devices in the module.
As shown in Figure 6, the hierarchy chosen is composed of two slices, each contains three groups.

The hierarchy is constructed by the functions *CAIRO_ADD_DEVICE()* which adds a declared device to a group, *CAIRO_ADD_GROUP()* which adds a group to a slice, and *CAIRO_ADD_SLICE()* which adds a slice to a module.

Area optimization is performed using the *CAIRO_RESHAPE()* function which indicates the shape parameter used (either the main module height or aspect ratio).

Explicit routing then follows between the *CAIRO_BEGIN_ROUTE()* and the *CAIRO_END_ROUTE()* functions.

Figure 8 shows the corresponding generated layout. Two different processes (top: AMS 0.8 μ and bottom: SGS-Thomson 0.5 μ) are used with two different aspect ratios.

Another example is given in [13] and Figure 9 which shows the complete layout for a continuous-time $\Sigma\Delta$ modulator performed with CAIRO. The circuit is realized using continuous-time current-mode integrators and switched-current sources DAC. For this example running CAIRO with a given desired height lasts about 1 minute on a Sparc 10 workstation to generate the layout. The chip has been fabricated by AMS with a 0.6μm CMOS process.

```
#include <cairo.h>
This C function is the OTA generator
main(argc,argv)
int argc;
char ** argv;
/* Open a SPICE file */
CAIRO_OPEN_SPICE_FILE(argv[1]);
/******** Definition of Module OTA **********/
CAIRO_OPEN_MODULE("OTA");
/* Device Declaration */
CAIRO_DIFFPAIR_SPI("MN1",NTRANS,"MN1","MN2",S_G);
CAIRO_TRANSISTOR_SPI("MN5",NTRANS,"MN5",S_S);
---
/* Building Groups */
CAIRO_ADD_DEVICE("MN5","group_0","TN5",SYM_X)
CAIRO_ADD_DEVICE("MN1","group_1","TN1",NOT_P);
---
/* Building Slices */
CAIRO_ADD_GROUP("group_0","slice_0",0);
CAIRO_ADD_GROUP("group_1","slice_0",0);
---
/* Building Module */
CAIRO_ADD_SLICE("slice_0",0);
CAIRO_ADD_SLICE("slice_1",SD_NWELL_PTIE);
/* Reshaping all devices */
CAIRO_RESHAPE("OTA");
/* Routing */
CAIRO_BEGIN_ROUTE("OTA","OTA");
CAIRO_WIRE2(ALU2,ALU2,SW_ALU2,SW_ALU2,"TN5",
"drain","TN1","source",HOR);
---
/* Defining the module interfaces */
CAIRO_PLACE_CON1("TN1","gate1","V",ALU1,+
SW_ALU1,WEST);
----
CAIRO_END_ROUTE("OTA");
CAIRO_CLOSE_MODULE("OTA");
```

Figure 7: *CAIRO code for the OTA circuit shown in Fig 6.*

CONCLUSIONS

We have presented a language for designing analog layout generators. The time invested in writing the code is largely compensated through the reusability of the generator, thanks to the independence of the code with respect to the device dimensions and the target fabrication process. This is achieved using a hierarchical, top-down optimization algorithm and a symbolic layout approach respectively.

THE AUTHORS

Mohamed Dessouky, Pr. Alain Greiner and Dr. Marie-Minerve Louërat are with the ASIM Department of LIP6 Laboratory, Université Pierre et Marie Curie, Couloir 55-65, 2ème étage, 4 place Jussieu, 75252 Paris Cedex 05, France.

e-mail: Mohamed.Dessouky@@lip6.fr

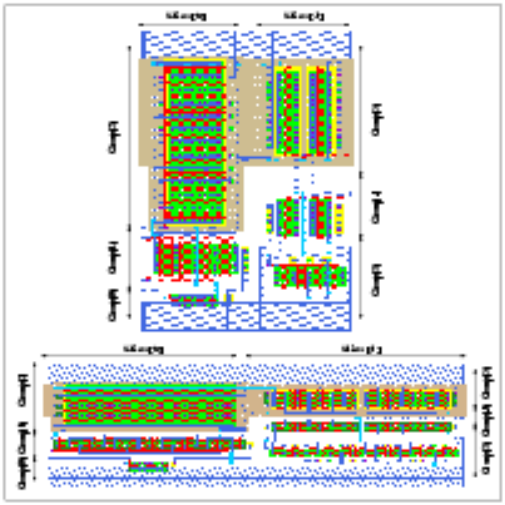

Figure 8: *Two layouts corresponding to the code in Figure 7, for 2 different technologies and two different heights*

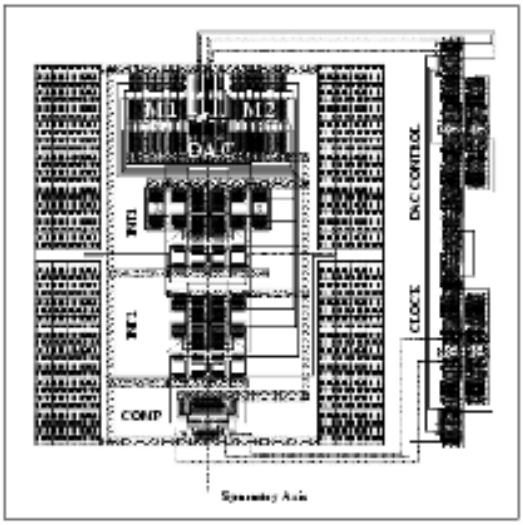

Figure 9: ΣΔ *modulator [13]*

References

[1] Hidetoshi Onodera, Hiroyuki Kanbara, and Keikichi Tamaru. Operational-amplifier compilation with performance optimization. *IEEE Journal of Solid-State Circuits*, 25(2):466–473, April 1990.

[2] Bryn R. Owen, R. Duncan, S Jantzi, C. Ouslis, S. Rezania, and K. Martin. BALLISTIC: An analog layout language. In *Proc. IEEE Custom Integrated Circuits Conference*, 1995.

[3] Han Young Koh, Carlo H. Sequin, and Paul R. Gray. OPASYN: A compiler for CMOS operational amplifiers. *IEEE Trans. Computer-Aided Design*, 9(2):113–125, February 1990.

[4] J. D. Conway and G. G. Schrooten. An automatic layout generator for analog circuits. In *Proc. European Design Automation Conference*, pages 513–519, 1992.

[5] Volker Meyer zu Bexten, Claudio Moraga, Roland Klinke, Werner Brockherde, and Klaus-Gunther. ALSYN: Flexible rule-based layout synthesis for analog IC's. *IEEE Journal of Solid-State Circuits*, 28(3):261–267, March 1993.

[6] Jef Rijmenants, James Litsios, Thomas R. Schwarz, and Marc G. R. Degrauwe. ILAC: An automated layout tool for analog CMOS circuits. *IEEE Journal of Solid-State Circuits*, 22(2):417–425, April 1989.

[7] J. M. Cohn, R. A. Rutenbar, and L. R. Carley. KOAN/ANAGRAM II: New tools for device-level analog placement and routing. *IEEE Journal of Solid-State Circuits*, 26(3):330–342, March 1991.

[8] Enrico Malavasi, Edoardo Charbon, Eric Felt, and Alberto Sangiovanni-Vincentelli. Automation of ICL layout with analog constraints. *IEEE Trans. Computer-Aided Design*, 15(8):923–942, August 1996.

[9] Koen Lampaert, Georges Gielen, and Willy M. Sansen. A performance-driven placement tool for analog integrated circuits. *IEEE Journal of Solid-State Circuits*, 30(7):773–780, July 1995.

[10] Juan A. Prieto, Adoraction Rueda, José M. Quintana, and José L. Huertas. A performance-driven placement algorithm with simultaneous place and route optimization for analog IC's. In *Proc. European Design and Test Conference*, pages 389–394, March 1997.

[11] Alain Greiner, Frédéric Pétrot, and Franck Wajsburt. Fixed grid symbolic layout translation into mask rectangles. In *2nd Advanced Training Course: Mixed Design of VLSI Circuits - Education of Computer Aided Design of Modern VLSI Circuits*, pages 281–286, Poland, May 1995.

[12] J. D. Bruce, H. W. Li, M. J. Dallabetta, and R. J. Baker. Analog layout using ALAS! *IEEE Journal of Solid-State Circuits*, 31(2):271–274, February 1996.

[13] H. Aboushady, E. L. Mendes, M. Dessouky, and P. Loumeau. A current-mode continuous-time $\sigma\delta$-modulator with delayed return-to-zero feedback. In *Proc. International Symposium on Circuits and Systems*, June 1999.