A Novel Analog Module Generator Environment

M. Wolf *, U. Kleine * and B. J. Hosticka **

* Otto-von-Guericke-Universität Magdeburg
IPE, PO Box 4120, D-39016 Magdeburg, Germany
email: mwolf@ipe.et.uni-magdeburg.de

** Fraunhofer Institute of Microelectronic Circuits and Systems,
D-47057 Duisburg, Germany

Abstract

This paper describes a novel analog module generator environment for the automatic layout development of analog circuits. The C++ tool features a novel procedural layout description language that drastically eases the creation of analog modules. Due to the object oriented programming the designer can specify the modules in a hierarchical way using elementary geometrical primitives and conditional statements. The primitive objects are placed relatively and are abutted with the help of a special compactor. An optimization routine with backtracking capability facilitates the creation of high quality analog layouts. A layout example of a broad-band BiCMOS amplifier will demonstrate the usability of the tool.

1. Introduction

Analog circuits have become a bottleneck in the development of integrated mixed mode analog-digital systems. Due to the necessary constraints of analog layouts (large variations of the device sizes, sensitivity to parasitic capacitances, crosstalk, device matching and symmetry requirements) to meet the various specifications, existing automated digital design tools are not well suited for the generation of analog layouts. As a consequence most analog layouts today are still hand-drafted by specialists. In recent years several automated design tools for analog circuits have been developed [1-8]. In general, in these programs the layout is generated in three steps: knowledge based partitioning of the schematic into predefined more or less complex modules, placement of the modules either by the slicing tree method [1-3] or with the simulated annealing approach [4], and finally routing of these blocks. Some programs perform an additional compaction step.

Besides the mentioned three or four steps for automatic layout generation, a further preceding step has to be performed in which a dedicated module library is created. Only a few different module types (e. g. different current mirrors, differential pairs, stacked transistors, diode connected transistors) are required in analog circuits. However, these modules differ greatly in their functionality, their external connections, and their device dimensions. Therefore technology-independent parameterizable module generators are procedurally tedious to construct and to maintain, the complexity of the modules is often limited to single device modules [4]. This keeps the size of the module library small, but shifts the essential module creation problems to the placement and routing steps by adding numerous constraints. On the other hand the availability of complex generators, like a centroidal cross-coupled differential pair with its internal wiring and with substrate or well contacts, simplifies the placement and routing problem drastically and yields more optimal layouts. To overcome these problems, a novel module generator environment is developed with a procedural layout description language, which permits the technology independent creation of parameterizable analog layouts. The design rules are stored in a technology description file. The environment features a special compactor and an optimization routine to generate dense layouts with a quality comparable to hand-drafted ones. The optimization routine can also handle electrical constraints. Several routing routines support the internal wiring of the modules. In the next sections the environment is discussed in more detail.

2. Module Generator Environment

No general guideline exists for the synthesis of optimal analog layouts. Even experienced analog designers have to perform the layout iteratively and cannot consider every possible topology variation because of limited time. Due to electrical constraints like matching requirements, it is necessary to cluster groups of devices. Thus a modular layout style results naturally. To support an automatic layout tool, a library with parameterizable modules is necessary. The programming of parameterizable modules to create optimal error-free layouts is a very complicated and time consuming task. The parameterizability of modules increases the complexity drastically. To ease the module creation a novel module generator environment has been

developed which supports the iterative design style of analog experts.

2.1 Procedural Layout Description Language

In contrast to former approaches of procedural languages for module generation [1, 3, 6, 7, 15-16], this paper presents a natural description language. The new procedural language enables the designer to describe parameterizable modules for analog integrated circuits hierarchically and design-rule independent. This language features loops, conditional statements and a set of simple functions to create and to wire primitive geometries without considering exact coordinates. Previous work has been done to write module generators in a graphical way [12-14]. For a graphical description of a module no special language is required but loops and conditions are graphically difficult to handle. Therefore a simple procedural language that yields natural and short code has been developed to write parameterizable modules.

The implemented language interpreter evaluates and fulfills the design rules automatically. If a rule cannot be fulfilled an error message occurs. A complex example of a rule check is the evaluation of the latch-up rule for substrate contacts. This rule determines if temporary rectangles which are placed around the substrate contacts enclose all locos areas of MOS-transistors. The size of these temporary rectangles is specified in the design rules. If not all active areas are enclosed additional substrate contacts have to be inserted. Fig. 1 shows the systematic check of this rule.

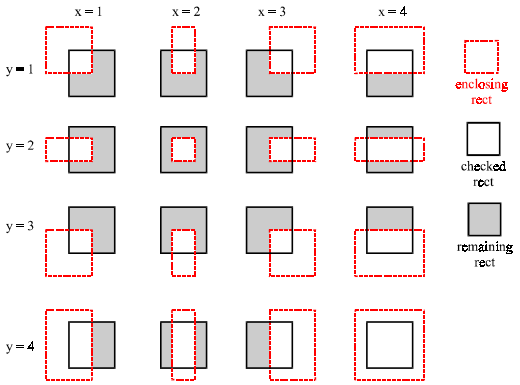

Fig. 1: Examination of the latch-up-rule

The temporary rectangles which should enclose the other rectangles are drawn with dashed lines. If these rectangles do not enclose completely the other rectangles only the overlapping part is cut while the remaining part of the rectangle is still stored in the database. If after examining all enclosing rectangles no parts of the solid rectangles are remaining, the latch-up rule is fulfilled. In Fig. 1 all possible 16 cases of overlapping are depicted. The four columns depict the four possible horizontal overlap cases while the rows consider the four possible vertical overlap cases. The not overlapped parts of the rectangle are converted to single rectangles that have to be enclosed by other temporary rectangles to fulfill the rule.

Due to design-rule constraints, the designer has to specify different topology alternatives for parameterizable modules. For this purpose backtracking is supported which eases the writing of different variants of a module because no complex if-then-structures with deep hierarchies have to be programmed. This shortens the code of the module because identical parts of two different variants do not appear multiple in different program branches. To keep the layout data structure efficient, polygons are converted into simple rectangular structures. During programming the environment supports two windows, a text window for the source code and a corresponding graphical view of the module. The source code is automatically translated into C++.

2.2 Functions for Generation of Primitive Geometries

Basic geometric objects are generated by calling primitive shape functions. The relevant design-rules are regarded automatically. Implemented functions are: inserting a rectangle inside other rectangles, placing a rectangle around a structure, creating an array of rectangles inside other rectangles, placing a ring around a structure, creating two overlapping rectangles and producing an angle adaptor for wiring purposes. Each geometry contains special properties that define if its edges are fixed or variable for moving inwards or outwards. If an edge is variable it can be moved automatically in a following step to improve the layout. Numerous internal C++-functions have been developed to support these geometric functions. The C++-functions are hidden for the module environment user.

```
...
gateCon = ContactRow(layer = "poly", W = 10000)
...
ENT ContactRow(layer, <W>, <L>)
  INBOX(layer, W, L)
  INBOX("metal1")
  ARRAY("contact")
END ENT
```

Fig. 2: Source code and call of the contact row

To illustrate the comfortable use of the module generator environment, Fig. 2 shows the source code for a contact row module. The description of the module consists of a calling sequence (first line) and an object declaration.

The entity "ContactRow" expects the parameters "layer" and the optional parameters "W" and "L" which define the width and length of the contact row, respectively. If an optional parameter is omitted, a default value is used. In the example the length of the contact row is omitted and therefore the minimum possible length for this value is selected according to the design-rules. After the declaration of the entity parameters three geometric function calls follow. The first call creates a rectangle on the specified layer (in this example: poly) with the dimensions W and L. The second function places a "metal1"-rectangle completely inside the first rectangle. The necessary overlap

between all involved layers is considered automatically. If the new rectangle cannot be placed inside the other rectangles, all outer rectangles are expanded. The third function creates a corresponding array of "contact"-rectangles inside the whole structure. The maximum number of rectangles which fits horizontally and vertically into the structure is calculated according to the necessary overlap and the contacts are placed equidistantly to minimize the contact resistance. If no rectangle can be placed, the outer geometries are expanded so that at least one rectangle can be generated. With these three primitive function-calls a complete parameterizable contact row is described without specifying or calculating an exact coordinate and without evaluating a design rule. These operations are performed automatically by the environment.

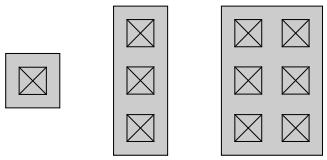

Fig. 3: Examples of the contact row

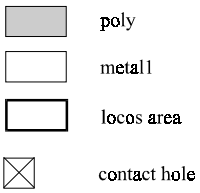

Fig. 4: Fill patterns for the layers

In Fig. 3 three resulting modules of the contact row are depicted. In the left example, both parameters W and L were omitted, in the middle example only the parameter L was omitted and in the right example W and L have been defined. The fill patterns for the layers are explained in Fig. 4.

2.3 Successive Compaction

Complex modules are constructed by compacting either geometric primitives or hierarchically built objects to an existing structure. In contrast to general compaction approaches [17, 18], the compaction is done successively by involving only one new object in each step. Thus, only outer edges of the main object have to be kept in the data structure and no general edge graph must be created. This speeds up the compaction time. Furthermore the designer is able to predict the result.

According to the design rules, the objects are placed with the minimum distance. A special property for every rectangle can avoid undesired overlaps (parasitic capacitances). Moreover, the implemented compactor supports special features to simplify the description and to optimize the layout. For instance edges on the same potential are not considered during compaction, because they can be merged. With this condition simple wiring can be performed by compacting a rectangle whose edges are on the same potential as the edges of the rectangles which shall be connected. The rectangle is compacted as close as possible and the rectangles on the same potential are merged. Fig. 5a show examples for this feature in the marked parts: A metal-rectangle was compacted to the top of the MOS-transistor and the outer diffusion contact rows were automatically connected to this rectangle.

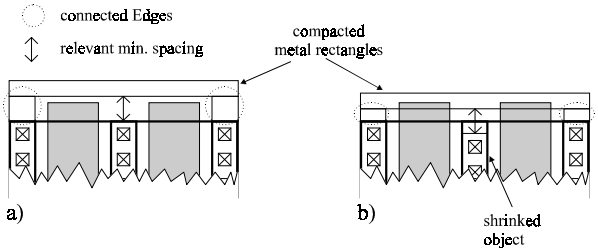

Fig. 5: Auto-connected edges (a and b) and optimization by shrinking objects (b)

Another feature of the compactor is the modification of objects during compacting. Each edge contains a property defining if it can be moved or not. This property is determined by the designer or defined automatically. If an edge is variable and defines the minimum distance between the two objects, the compactor tries to move it until it is no longer relevant. This step is repeated until fixed edges define the minimum distance. The objects affected by the movement are rebuilt automatically. An example for this feature is depicted in Fig. 5b. It is the same layout as in Fig. 5a but now the metal edges of the contact row are variable and the metal1-rectangle of the middle contact row was shrinked automatically. After that the contact row was rebuilt and the array of "contact"-rectangles was recalculated.

This concept of variable edges provides additional freedom in optimization because even already placed objects can be modified. Therefore optimizations are possible that are not obvious in previous steps. The benefit of this strategy is a substantial reduction of the layout area.

2.4 Optimization

The result of the above described compaction method depends on the compaction order, because one object could have a different minimum distance to a structure if another object was compacted previously. To find the optimal solution, the compactor can be used in an optimization mode. In this mode all different variations are generated by altering the order of the compacted objects. Each solution is evaluated by a rating function which considers the area and electrical conditions. If different topology variants exist for a module the rating function is also applied to select the best variant.

2.5 Simple MOS Differential Pair Example

Fig. 6 illustrates the generation steps for a simple MOS differential pair module which consists of two transistors, three diffusion-contact-rows and two poly-contacts. Fig. 7 shows the hierarchical source code of the differential pair. The description consists of a calling sequence in the first line and two object declarations. The declaration of the

contact row (Fig. 2) is omitted because it has already been explained in section 2.2.

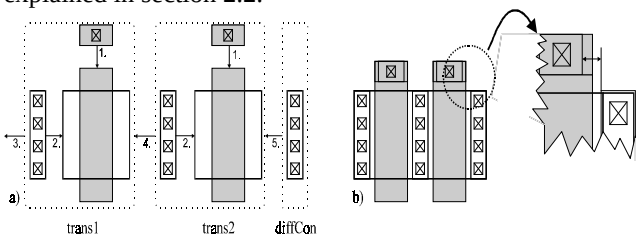

Fig. 6: MOS differential pair, a) before and b) after compaction

In the declaration of the object "Trans" the call of the primitive shape function "TWORECTS" creates a transistor geometry consisting of two overlapping rectangles in the data structure of the object "Trans". After that a poly contact and a diffusion contact are created by calling the object "ContactRow". The calls of the "compact"-function compact these two objects (step 1 and step 2 in Fig. 6) to the data structure of the object "Trans". The first parameter of this function is the compacted object, the second parameter is the direction and the third parameter determines layers which are not relevant during this compaction step. The geometries of these layers are connected automatically after the compaction if they are on the same potential.

```
...
diff = DiffPair(W = 10000, L = 5000)
...
ENT Trans(<W>, <L>)
  TWORECTS("poly", "pdiff", W, L)
  polyCon = ContactRow(layer = "poly", L = L)
  diffCon = ContactRow(layer = "pdiff", W = W)
  compact(polyCon, SOUTH, "poly")           // step 1
  compact(diffCon, SOUTH, "pdiff")          // step 2
END ENT

ENT DiffPair(<W>, <L>)
  trans1 = Trans(W = W, L = L)
  trans2 = trans1                    // copy of trans1
  diffCon = ContactRow(layer = "pdiff", W = W)
  compact(trans1, WEST, "pdiff")            // step 3
  compact(trans2, WEST, "pdiff")            // step 4
  compact(diffCon, WEST, "pdiff")           // step 5
END ENT
```

Fig. 7: Source code of the simple MOS differential pair

The declaration of the object "DiffPair" contains a call of this object "Trans" and a function call that copies the data structure of the first object to a new object. A diffusion contact is created by calling the object "ContactRow". After this creation of the objects the data structure of the object "DiffPair" is still empty, only the objects were generated. Therefore the first compaction command copies the first transistor into the data structure. The two next compaction commands compact the second transistor and the diffusion contact to the first transistor. The result of the compaction is shown in Fig. 6b.

The magnified part displays the above mentioned feature of the compactor which moves non-fixed edges in order to produce denser layouts. In this example the metal-edges of the poly-contacts were moved so that the diffusion-contacts could be placed closer to the transistors. Using this hierarchical description for the module, a very short and easy to read code results. Former methods for equivalent generation by describing each rectangle with its exact coordinates needed a multiple of this source code and were much more difficult to construct and to maintain [11].

3. Example of a BiCMOS Operational Amplifier Layout

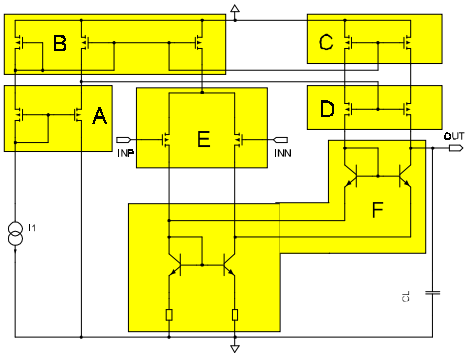

Fig. 8: High bandwidth BiCMOS amplifier [10]

Fig. 8 shows the schematic of a broad-band BiCMOS amplifier [10] and the corresponding layout is depicted in Fig. 9. The knowledge based partitioning of the modules takes additional analog properties like matching and symmetry requirements, minimization of parasitic capacitances of nodes in the signal paths, minimal wire widths of the different modules and poly-wire resistance into account. The partitioning is indicated in the schematic. In order to reach the high performance requirements of the broad-band amplifier, each module has to satisfy special analog properties. Block **A** contains the cascode transistors of the bias circuit. This module is composed of two inter-digital MOS transistors because no special matching or symmetry requirements has been specified for these transistors. No special matching requirements exist for module **D**. Only moderate matching requirements has been specified for the current mirror of block **B**. Therefore a symmetrical layout module is chosen with the diode transistor in the middle. For the current sources of block **C** high symmetry and matching requirements exist. Thus a cross-coupled arrangement of inter-digital transistors is selected. To improve the matching properties, the differential pair in block **E** consists of centroidal cross-coupled inter-digital transistors with eight dummy transistors in the middle and four dummy transistors on the right and left side, respectively. An expanded view of this module is depicted in Fig. 10. As can be seen from the figure the wiring is fully symmetrical and every net has identical crossings. The source code for this complex module has a length of about 180 lines. The computation time for building this module is five seconds. The bipolar transistors of block **F** are composed symmetrically. The internal wiring and the substrate

or well contacts are included into the modules. The layout area (592 x 481 μm^2 in a 1µ Siemens-BiCMOS-technology) and the quality (parasitic capacitances of the internal nodes) of the amplifier are comparable to an optimal hand-drafted version or even better. The placement of the modules and the global routing were done manually.

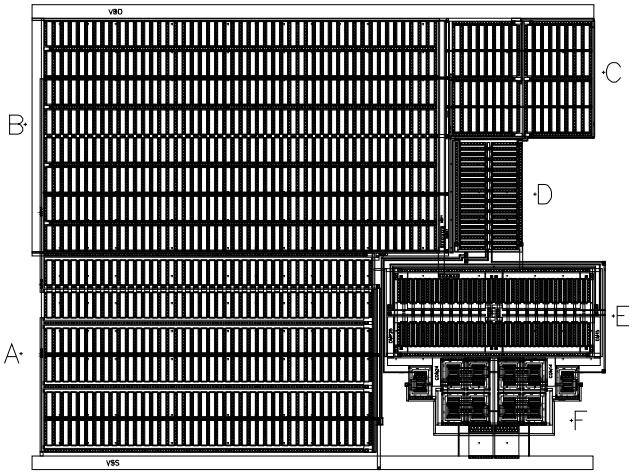

Fig. 9: Automatically generated layout of the BiCMOS amplifier

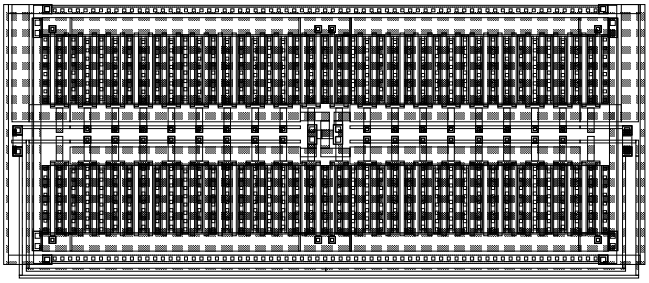

Fig. 10: Expanded view of the differential pair of module E

4. Conclusion

A novel module generator environment has been presented which simplifies the creation and shortens the source code of complex analog parameterizable modules drastically. With the natural layout description language the modules are written in a technology independent way because the environment considers the design rules automatically. The hierarchical description method makes the source code more readable and permits the reuse of the code. The simple successive compaction algorithm is very fast and produces predictable results. The concept of the variable edges offers a possibility for automatic and post-placed optimization. This approach is similar to the successful layout generation style of analog experts. The backtracking capability of the environment enables the user to create different topology variants easily for iterative optimization purposes. Due to its easy use, analog designers can construct and maintain their modules themselves. The performance of the module generator environment has been demonstrated with the automatic layout creation of a high bandwidth BiCMOS amplifier example.

Acknowledgments: The authors acknowledge V. Meyer zu Bexten and G. Nebel for their helpful discussions. This work has been supported by the JESSI project AC12 and Siemens.

References

[1] J. Rijmenants, et al., "ILAC: An Automated Layout Tool for Analog CMOS Circuits," IEEE J. Solid-State Circuits, Vol. 24, No. 2, pp. 417-425, April 1989.

[2] H. Y. Koh, et al., "OPASYN: A Compiler for CMOS Operational Amplifiers," IEEE Trans. Computer-Aided Design, Vol. 9, No. 2, pp. 113-125, Feb. 1990.

[3] H. Onodera, et al., "Operational-Amplifier Compilation with Performance Optimization," IEEE J. Solid-State Circuits, Vol. 25, No. 2, pp. 466-473, April 1990.

[4] J. M. Cohn, et al., "KOAN/ANAGRAM II: New Tools for Device-Level Analog Placement and Routing," IEEE J. Solid-State Circuits, Vol. 26, No. 3, pp. 330-342, March 1991.

[5] S. W. Mehranfar, "A Technology-Independent Approach to Custom Analog Cell Generation," IEEE J. Solid-State Circuits, Vol. 26, No. 3, pp. 386-393, March 1991.

[6] J. D. Conway and G. G. Schrooten, "An Automatic Layout Generator for Analog Circuits," Proc. European Design Automation Conf., pp. 513-519, 1992.

[7] J. P. Harley, M. I. Elmasry, and B. L. Leung, "STAIC: An Interactive Framework for Synthesizing CMOS and BiCMOS Analog Circuits," IEEE Trans. Computer-Aided Design, Vol. 11, No. 11, pp. 1402-1417, Nov. 1992.

[8] V. Meyer zu Bexten, et al., "ALSYN: Flexible Rule-Based Layout Synthesis for Analog IC's," IEEE J. Solid-State Circuits, Vol. 28, No. 3, pp. 261-268, March 1993.

[9] W. Schardein, et al., "Analog Module Generator for Effective Design Assistance," Proc. ESSCIRC'94, Ulm, pp. 160-163, Sept. 1994.

[10] G. Nebel, U. Kleine, H. J. Pfleiderer, "Large Bandwidth BiCMOS Operational Amplifiers for SC-Video-Applications," Proc. ISCAS, pp. 5.85-5.88, 1994.

[11] D. S. Tjan, "Entwurf eines Layout Generators für analoge Schaltungen," Master Thesis, University of Kaiserslautern, Oct. 1992.

[12] J. Conway, G. Beenker, "A New Template Based Approach to Module Generation," IEEE International Conf. on CAD, ICCAD 1990, pp. 528-531, 1990.

[13] Cadence Design Systems, Inc, 555 River Oaks Parkway, San Jose, California 95134, "SKILL Reference Manual," Oct. 1991.

[14] D. Lacroix, S. Menkis, "An Interactive Graphical Approach to Module Generation Development," IEEE Custom Integrated Circuits Conference, pp. 30.1.1-30.1.5, 1990.

[15] J. Kuhn, "Analog Module Generators for Silicon Compilation," VLSI System Design, pp. 75-80, May 1987.

[16] A. Greiner, F. Pétrot, "Using C to Write Portable CMOS VLSI Module Generators," EURO-DAC 1994, pp. 676-681, 1994.

[17] E. Felt et al., "An Efficient Methodology for Symbolic Compaction of Analog IC´s with Multiple Symmetry Constraints," European Design Automation Conf., EDAC 1992, pp. 148-153, 1992.

[18] E. Felt et al., "Performance-Driven Compaction for Analog Integrated Circuits," IEEE Custom Integrated Ciruits Conf., CICC 1993, pp. 17.3.1-17.3.5, 1993.