An Integrated Layout System for Sea-of-Gates Module Generation

P. Duchene[1], M. Declercq[1] and S. M. Kang[2]

[1] : Electronics Labs, Federal Institute of Technology, Lausanne (EPFL)
EL-Ecublens, CH-1015 Lausanne, Switzerland
[2] : Coordinated Science Laboratory, University of Illinois at Urbana-Champaign
1101 West Springfield Avenue, Urbana, IL 61801

Abstract

This paper presents a sea-of-gates layout system able to design medium-size logic circuits in a true channelless fashion. The methodology relies on flexible leaf cell generation, systematic cell terminal abutment, a global routing scheme using integer linear programming methods, and a step-wise compaction-rerouting refinement. Modules up to several hundred transistors have been laid out compactly with more than 80 % transistor utilization with two layers of metal. With top-down hierarchy, those modules can be used as macrocells.

1. Introduction

Second generation gate arrays, or Sea-of-Gates (SOG), have emerged as a design methodology of choice in the ASIC market. They are distinguished from older gate arrays by architectural features, such as gate isolation and channelless layout [1]. The methodology used for customization, however, has not changed much : it is still based on standard cell (or macro) libraries and on channel routing. Except for some special macrocells (e.g. RAMs), designers actually put *channelled* designs on *channelless* structures, and do not use the full flexibility offered by modern SOG architectures.

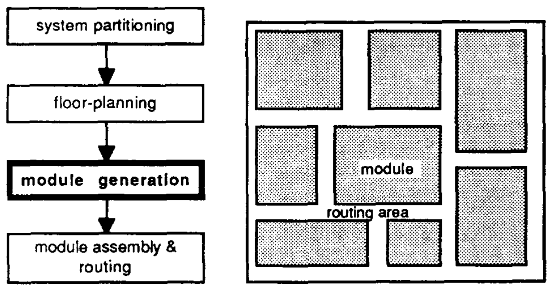
Few attempts have been made to investigate design methodologies better suited to SOG arrays. The Gate Forest project [1] emphasizes methodologies based on hierarchy and advanced floorplanning techniques. Compilers are used for special macrocells. However, large parts of the design still rely on standard cell libraries. The PROUD [2], SOGR [3] and MOLE [4] programs propose efficient solutions to the placement, global routing and detailed routing problems on SOG arrays, respectively. Those tools are oriented toward the design of very large circuits. All of them, however, keep standard macros as the basic building blocks.

The SOG layout system presented here, in combination with a novel SOG architecture [5], generates medium-size subcircuits, or *modules*, without channels. Its key features are : cell generation at the transistor level (no standard library), cell stretching, routing by abutment, and through-the-cell routing.

This work is part of the EEC ESPRIT II - I.D.P.S project (task II.A.2.6.4) and was supported by SGS-Thomson Microelectronics, Grenoble, France, and by the C.E.R.S., Bern, Switzerland.

2. Global strategy

The layout program presented here aims to be used in a macrocell design flow (Fig. 1). A complete circuit is first decomposed into smaller modules whose size, shape and location are determined by an appropriate floor-planning tool. Those modules are then individually generated. Finally inter-module connections are routed. Although this methodology has demonstrated its efficiency in numerous full-custom designs, it is still not commonly accepted in semi-custom design.



(a) flow-chart (b) floor-plan

Fig. 1 Macrocell design style

This paper focuses on the module generation phase. Partitioning, floor-planning and routing tools for full custom have received considerable attention these last few years. At those levels, however, the distinction between full- and semi-custom is not significant, and the same methods and algorithms can be applied.

The program has three main phases, each being composed of elementary steps (Fig. 2) :

- The *initial placement-routing* places cells with their terminals in such a way that most connections are made by vertical and horizontal abutment.

- A *global routing* phase achieves routing of all nets.

- Finally a *step-wise improvement* procedure compacts the layout and re-routes some nets, in order to decrease the total area.

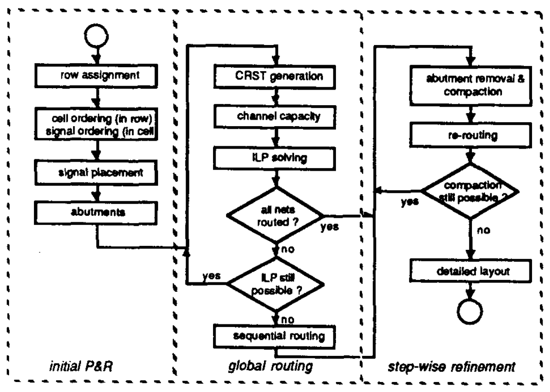


Fig. 2 : flow-chart of the program

3. Initial placement-routing

The initial placement phase is made by a modified version of the GRAPES program [6]. The basic building block is the leaf cell, which can be an AOI cell of arbitrary complexity or some other cell such as XOR or latch. Cell layout is not stored in a library but is described by two objects : a cell *template*, defining all allowed I/O permutations, and a cell *generator*, able to lay out internal connections once the terminal locations have been fixed. By this way, any permutation of the inputs or output are allowed, provided that sub-gates inputs are grouped. The cell can be stretched and crossed over by feed-through wires at arbitrary positions. GRAPES first places cells in rows, then looks in each row for the best cell ordering, and in each cell for the best terminal ordering, in order to optimize the number of terminal abutments.

After placement, typically 50-60% of the nets are connected by abutment. The original GRAPES program used a conventional channel routing scheme to connect the remaining nets. This, and the numerous constraints imposed by vertical cell abutment were at the origin of a sometimes poor transistor utilization rate. That is why improved routing and compaction schemes have been needed.

4. Global routing using integer linear programming techniques

Several authors have highlighted that the global routing problem can be formulated as an *Integer Linear Programming* (ILP) problem [7]. However, few attempts have been made to solve it by conventional ILP methods. The reason is that full size VLSI global routing problems are too large to be solved by those methods, and thus require some heuristics. In our system, however, the problem size is greatly reduced by the previous placement phase, and the related cell terminal abutments. Direct ILP techniques are then applicable.

Before the stating the problem, let us define several terms :

- for each net, we must find one *Constrained Rectilinear Steiner Tree* (CRST) that spans all the net *terminals.*

- the layout area is partitioned into vertical or horizontal *channels*. The channel definition here is somewhat different from its usual meaning. Horizontal channels fit in the predefined rows of the master, while the vertical channels correspond to transistor columns (Fig. 3). Vertical and horizontal channels thus overlap. But since vertical and horizontal connections use a different layer of metal, they can be considered separately.

- the horizontal channel *capacity* initially equals the number of routing tracks available on the master cell height, minus the tracks needed for the internal routing of functional cells, which are given by the cell template for each cell configuration. The vertical channel capacity is 1 if the underlying column is free, 0 otherwise.

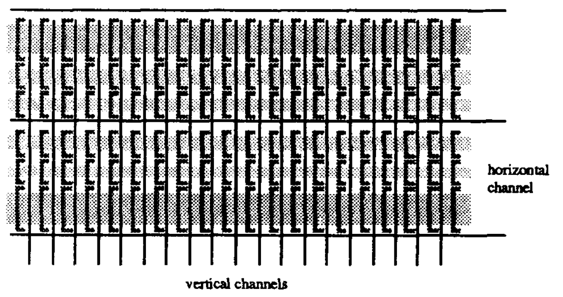


Fig. 3 : Channel definition on SOG

As usual, the routing problem can be decomposed into a global and a detailed routing phase. The global routing problem can be stated as follows :

Find a set of CRSTs each spanning all the terminals of each net to be routed, such that :

- there is exactly one CRST per net

- the total number of CRSTs is lower or equal to the channel capacity in each channel

- the total cost (e.g. routing length) is minimized

A more formal expression of this problem is :

Given :
- N nets $n_1, \dots, n_N$ to be routed
- m_i CRSTs candidate for routing for each net n_i
- $M = \sum_{i=1}^{N} m_i$, the total number of CRSTs
- M variables y_{ij} (i=1,...,N ; j=1,m_i) that designate the jth CRST of the net n_i
- M constants b_{ij} (i=1,...,N ; j=1,m_i) representing the *cost* (e.g. total wire length) of each CRST
- C channels respectively with capacity $c_1, \dots, c_C$
- $M \times C$ matrix $[a_{ij}]$ $i = 1,\dots,M$ $j = 1,\dots,C$
where $a_{ij} = 1$ if CRST number i uses channel number j, 0 otherwise
- cost function $\mathrm{COST}(y_{ij}) = \sum_{i=1}^{N} \sum_{j=1}^{m_i} b_{ij}\, y_{ij}$

Determine y_{ij} $i=1,\dots,N \quad j=1,m_i$
so as to minimize $\mathrm{COST}(y_{ij})$ $i=1,\dots,N \quad j=1,m_i$
under the following constraints :

$$y_{ij} = 1 \text{ or } 0 \qquad i=1,\dots,N \quad j=1,m_i \qquad (1)$$

$$\sum_{j=1}^{m_i} y_{ij} = 1 \qquad i = 1, \dots , N \qquad (2)$$

$$\sum_{i=1}^{M} a_{ij}\, y_{ij} \le c_j \qquad j = 1, \dots , C \qquad (3)$$

The constraint (2) reflects that exactly one CRST must be retained for each net. The constraint (3) reflects the channel capacity. Formally, this constraint is too restrictive : indeed, two non-overlapping CRST branches could share the same track in a channel, so the number of CRSTs using a channel could be larger than the channel capacity. However, this constraint has been retained for simplicity. Because it tends to leave free tracks in most channels, it promotes subsequent re-routing.

The Steiner tree generation problem is known to be NP-complete. Our problem, however, is not to find the *optimal* Steiner tree for each net, but to generate a large set of various *acceptable* trees, hopefully including the optimal ones. A tree generation algorithm based on a modified branch-and-bound method [8] has been selected. By changing the starting vertex in the search procedure, several distinct trees can be generated.

The problem as previously stated is a *Zero-One Integer Linear Programming* problem, which is solved by conventional methods [9] : first the *linear relaxation* of the problem is solved, by relaxing the constraints stating that all variables must be integers. If no solution is found, the problem is unfeasible. If the solution is all-integer, the problem is done. Otherwise, a pivot-and-complement heuristic [10], and a branch-and-bound method are successively applied to find a near-optimal all-integer solution.

In some cases, when the problem is ill-defined, usually due to too restrictive constraints, it is possible that this method cannot provide feasible solutions. In such a case, the branch-and-bound algorithm is run to obtain a *partial solution* where several variables have 0-1 values, while the others are not integer. The CRSTs associated with the 1-valued variables are routed. The remaining nets form a problem similar to the previous one, but of smaller size. We then re-apply the same methods (from CRST generation to ILP problem solving), except that more CRSTs are generated. This is repeated until all nets are routed, or until no more net can be routed by this method. In this last case, remaining nets are routed sequentially using the method described in §6. If this also fails, we insert a routing channel. By this way, we insure a 100% routing completion. This last case, however, has not yet been encountered.

5. Step-wise improvement

At this point, all nets are routed and the layout is done. However, the layout can contain large empty *holes*, or areas without cells or routing. This is a consequence of the abutment methodology used in the initial placement : indeed, requirements for vertical abutments can be considered as constraints against a further compaction. If one or more vertical abutments are removed, the layout can usually be compacted (Fig. 4). On the other hand, nets previously connected by those abutments must be routed. While the layout is not too congested, this operation is possible. But as the array fills up, routing may not be feasible any more without stretching some cells. One can thus intuitively see that there is an optimal *abutment-against-routing* ratio. More abutments imply more holes in the layout, while more compaction implies more routing tracks.

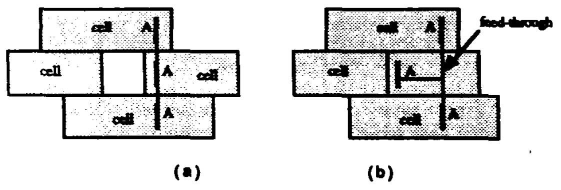


Fig. 4 : Abutment and compaction :

(a) the abutment A blocks any further compaction

(b) after removing the abutment, the layout is compacted

The method selected to approach this optimum is based on a greedy heuristic. The outline is given below :

```
repeat
   select an abutment to remove
   remove that abutment
   compact the layout
   route the net associated with that abutment
until
   there is no abutment to remove
   or routing is unfeasible for every net still
      abutted
   or the opening size required by routing is
      larger than the size gained by compaction for
      every net still abutted
```

To select the abutment to remove, the longest row is first selected ; then, in this row, we look for the largest hole. The selected abutment is the abutment responsible for this hole. It is placed just on the right of the hole. This method insures that the longest row will always be compacted first. Since the area of a module is fixed by its longest row, we make sure that the area is decreased at each step.

A Lee algorithm is used for routing the net. If it fails to find a connecting tree (too many obstacles), we heuristically *open* the layout, by inserting a new empty vertical track, and we use that track to join the disjoint subtrees obtained by the Lee algorithm. If this fails, we mark the net as unroutable and loop back to the abutment selection step, undoing the previous compaction.

6. Implementation and results

The layout system presented here has been implemented in a prototype program called SOGGY. SOGGY is written in Common Lisp with Flavors, an object-oriented environment. It runs on Apollo™ and Sun™ workstations.

The input consists of a set of Boolean equations (each one describing a functional cell), plus various geometrical (size, aspect ratio) and electrical (transistor sizes or maximum delay) constraints. The output is in EDIF or CIF format.

Several examples, ranging from 50 to 800 transistors, have been generated using this program. The observed transistor utilization ranges from 60 to 85 %. These numbers must be considered bearing in mind that :

- only two levels of metal are used, and
- the basic master cell is small (only 11 horizontal routing tracks) compared to most commercial arrays

Fig. 5 shows an design example of a fast 8-bit comparator circuit counting 302 transistors. Fig. 6 gives a separate view of what is (a) the internal routing of functional cells, (b) the routing-by-abutment, and (c) the global routing. Fig. 7 summarizes area and routing data for this circuit and others, in a 2 μm, double metal CMOS technology.

7. Conclusion

A sea-of-gates layout system has been presented. Contrary to conventional gate array or sea-of-gates systems, it doesn't rely on libraries of macros and on channel routing, but on advanced features such as cell generation, routing by abutment, and through-the-cell routing. An integer linear programming method has been followed to solve the global routing problem. This approach is very efficient to solve medium size routing problems. The presented system can handle efficiently circuits up to several hundred transistors without routing channels, resulting in a high transistor density and reduced wiring length.

References

[1] M. A. Beunder, B. Hoefflinger and J. P. Kernhof
"New Directions in Semi-custom Arrays"
IEEE J. Solid-State Circuits, vol SC-23, No 3, June 1988, pp 728-735

[2] R.-S. Tsay, E. S. Kuh and C.-P. Hsu
"PROUD : A Fast Sea-of-Gates Placement Algorithm"
Proc. 25th DAC, July 1988, pp 318-323

[3] T.-M. Parng and R.-S. Tsay
"A New Approach To Sea-of-Gates Global Routing"
Proc. ICCAD, Nov. 1989, Santa Clara, CA, pp 52-55

[4] A. Srinivasan and E. S. Kuh
"MOLE - A Sea-of-gates Detailed Router"
Proc. EDAC, March 1990, Glasgow, Scotland, pp 446-450

[5] P. Duchene and M. Declercq
"A Highly Flexible Sea-of-Gates Structure for Digital and Analog Applications"
IEEE J. Solid-State Circuits, vol SC-24, no 3, June 1989, pp 576-584

[6] H. Heeb and W. Fichtner
"GRAPES : A Module Generator Based on Graph Planarity"
Proc. ICCAD, Nov. 87, Santa Clara, CA, pp 428-431

[7] E. S. Kuh, M. Marek-Sadowska
"Global Routing"
in *Layout Design and Verification*, North-Holland, 1986, edited by T. Ohtsuki, pp.169-198

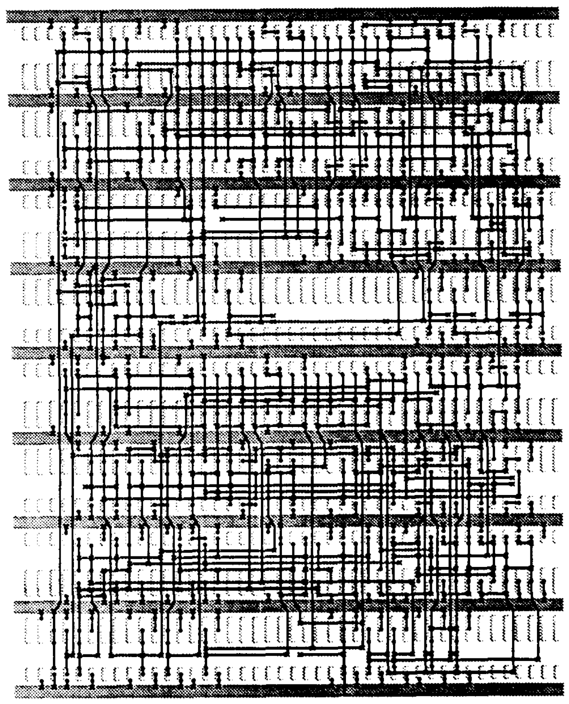
[8] Y. Y. Yang and O. Wing
"Suboptimal Algorithm for a Wire Routing Problem"
IEEE Transactions on Circuit Theory, vol CT-19, no 5, Sept. 1972, pp 508-510

[9] R. E. Marstein
"The Design of the XMP Linear Programming Library"
ACM Transactions on Mathematical Software, vol. 7, no. 4, Dec. 1981, pp 481-497

[10] E. Balas and C. H. Martin
"Pivot and Complement - A Heuristic for 0-1 Programming"
Management Science, vol. 26, no. 1, Jan. 1980, pp 86-96

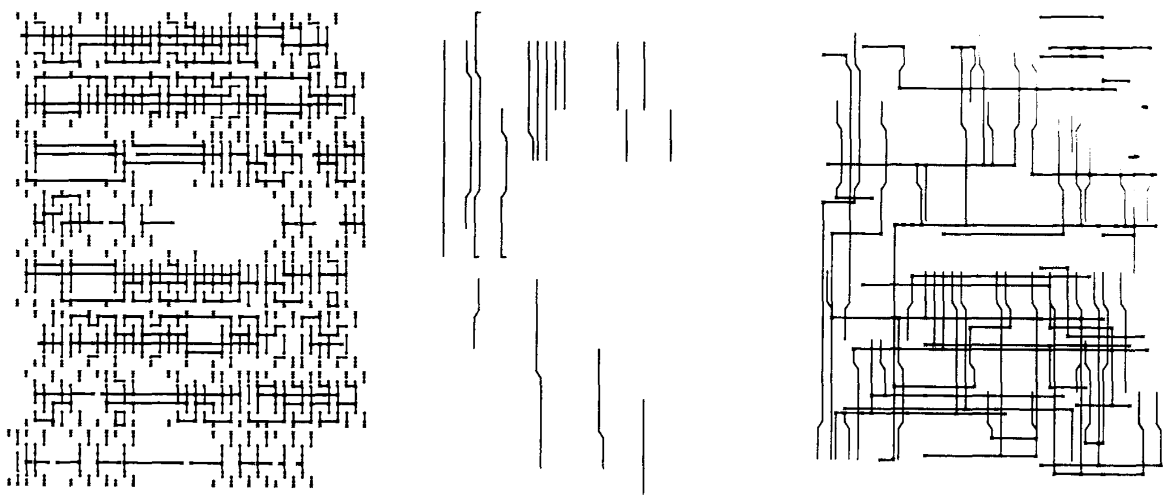
```
(
    (max-delay 10)
    (order P0 Q0 P1 Q1 P2 Q2 P3 Q3)
    (edge north P0 P1 P2 P3 Q0 Q1 Q2 Q3)
    (not PgQBn (or
       (and MSB4 P3 Q3n)
       (and MSB4 P2 Q2n PQ3)
       (and MSB4 P1 Q1n PQ2 PQ3)
       (and MSB4 P0 Q0n PQ1 PQ2 PQ3)
       (and MSB4 PQ0 PQ1 PQ2 PQ3 PgQin)))
    (not PQ0 (or (and P0 Q0n) (and P0n Q0)))
    (not Q0n Q0)
)
```

(a) part of the input file



(b) layout produced by SOGGY

Fig. 5 : Fast comparator design example



(a) Internal cell connections (b) abutment routing (c)global routing

Fig. 6 : Separate view of the routing parts

	functional transistors	area (μm^2)	transistor utilization*	total wire length (μm)	CPU time** Grapes (sec.)	CPU time** Soggy (sec.)
8 bits comparator	302	566 K	79 %	18 318	1 446	64
4 bits ALU	256	490 K	81 %	13 172	1 510	67
8 bits ADC control part	624	1 441 K	76 %	49 728	9 639	522

* includes functional and isolation transistors ** on a Apollo™ 3500 workstation

Fig. 7 : Results