Proc. 1st int. conf. on ASIC, Beijing, China,
W. YangYuan (Ed.), Oct. 18–21, 1994, pp. 75 – 78

CMOS cell generation for Logic Synthesis

dr.ir. J.T.J. van Eijndhoven
Eindhoven University of Technology
The Netherlands

J.T.J.v.Eijndhoven@ele.tue.nl

In commercial ASIC systems, fixed libraries of standard CMOS cells are much more popular than programs which generate such cell layouts on the fly upon demand. However, using cell generators designs can be mapped to fewer cells due to the large choice, and with more timing freedom. This requires a cell layout style with predictable timing performance, and logic synthesis with a consistant timing model. The basic algorithm to derive the cell topology shows two orders of magnitude improvement over originally published results.

1. Introduction

Generation of mask layout for static CMOS cells from a boolean specification has been done for many years now, especially in research environments. However in commercial systems standard cell libraries providing a limited choice of cells are most popular. With logic synthesis widely available in commercial systems now, a cell generator can show remarkable advantages. This requires particular attention for cell timing behavior and routing transparency, in combination with a consistent timing model in the logic synthesis package. As result high performance circuits can be generated, with a reduced cell count and fast and reliable timing.

2. Static CMOS cell style

Static CMOS cells have a general circuit topology as depicted in figure 1. Such a function is normally specified as the inverse of a nested expression built from input identifiers, *and* symbols, *or* symbols, and braces. The internal structure of the N– and P– networks is easily represented in a graph, with a node representing a net, and an edge representing a transistor source–drain path. The edges are labelled with the input identifier. The N– and P– graphs normally have dual topologies, such that each input identifier occurs exactly once in both the N– and P– graph. In our approach these graphs are also series–parallel, directly mirroring the recursive specification of the boolean function. The classical cell layout style is used as depicted in figure 2, where source–drain regions are shared between neighboring transistors, and the inputs vertically aligned for a compact cell layout [3]. A particular boolean function is realized by interconnecting diffusion areas with horizontal wires for parallel transistors, and making connections to output and power or ground.

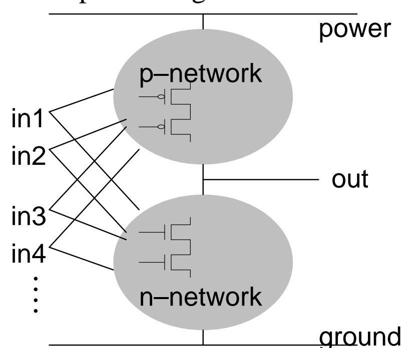


figure 1: general circuit topology

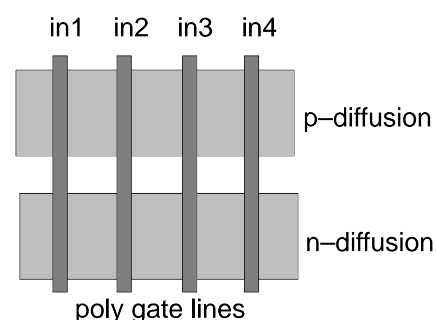


figure 2: classic layout style

3. Cell topology

The main problem in creating the cell topology is deciding the transistor (input identifier) order. A so called dual Eulerpath has to be found in the dual series–parallel graphs of the network structure. If the Eulerpath doesn't exist, the graph edges should be covered with a set of disjoint paths. In this case the

resulting layout becomes wider due to separation between the different paths (transistor sequences), horizontally placed next to each other. The problem is thus finding a dual graph coverage with a minimum number of dual paths. For given series–parallel graphs this can be solved optimally in linear CPU time [1]. However in general the resulting number of paths can be influenced by reordering of the graph, which does not modify the boolean function. Thus the problem becomes finding a coverage in a minimum number of dual paths over all possible graph orderings. Unfortunately this problem is NP–hard. However Masiasz published an exact algorithm to solve this relatively efficiently [1]: his algorithm is only worst–case exponential in the maximum degree found in the expression tree. The algorithm proceeds bottom–up in the expression tree, cataloging for each node all possible externally different sub–graph covers. The efficiency of the method is due to the fact that the maximum number of such covers in a single node can never exceed 42, independent of the expression complexity.

We made an efficient implementation of this algorithm by a) not maintaining all possible detailed graph coverages in each node, but merely the different achievable cover types. This computation on just cover types proceeds upwards in the tree. At the tree root, the desired (minimum cardinality) cover is chosen, after which the corresponding dual paths are determined in detail. The efficiency is furthermore improved by b) deriving the different possible cover types only once for each different subexpression. For this purpose the expression is not really stored as a tree, but as a directed acyclic graph with each node representing a unique different (sub–)expression. As result we find a maximum CPU time of 0.09 seconds for separate program runs to exactly solve the NP–hard problem for any individual cell among the 425803 different cells in the 5x5 family, whereas Masiasz reported a maximum of 90 seconds among the 3503 cells in the 4x4 family [2].

In contrast to Masiasz, our CPU time also includes an algorithm for the horizontal placement of the paths of the chosen cover (their relative order and individual mirroring), to optimize the cell height: the number of internal routing tracks required to implement the boolean cell functionality. For comparing absolute CPU times one should note however that our experiments are done on a considerably faster HP 735/99 workstation.

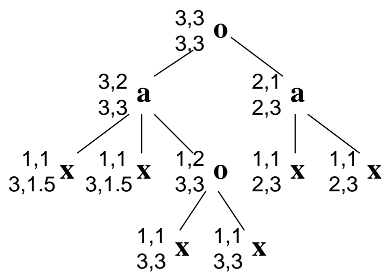
4. Transistor sizing

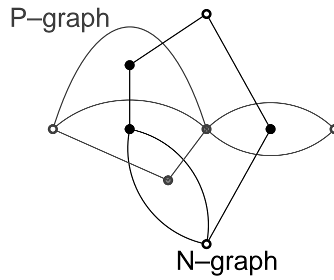
To obtain a predictable cell delay which is in first approximation independent of the boolean input pattern change, individual transistors in the cell are assigned different widths. The objective is to maintain a certain output conductance. Such width assignment can be done in different ways, as is shown in figure 3. For layout reasons we prefer the solution which minimizes the maximum used width, the rightmost

figure 3: two different width assignments for unit conductance

solution. It is determined by a simple linear time algorithm: First bottom–up in the expression tree the longest path in the N– sub–graph is denoted in each node as the sub–expression height. Secondly the N–size is assigned top–down, initialized with the height at the root, passed down to the arguments identical for *and* or multiplied with (height of argument)/(height of current node) for *or*. In the leaves of the tree, the N–size is assigned as N–transistor width. Correspondingly the P–size is assigned. Two other factors influence the final transistor sizes: First the p–transistors are always enlarged by a technology dependent factor, to compensate for their reduced conductivity. Secondly all transistors of the cell are scaled linearly with an individual cell ‘speed factor’, as discussed in the next section.

An example is shown in figure 4: The expressiontree is shown with **o** denoting an *or*, **a** denoting an *and*, and **x** denoting a primary *input*. The two comma separated numbers left above each such letter, indicates the size of the corresponding expression, which is calculated bottom–up in the tree, with **x** being 1,1 by definition. So a pair p,q at the top of the tree denotes p as the maximum number of N–transistors in series, and q such for the P–transistors. This pair of numbers is denoted as the ‘dimension’ of the cell. In general the logic optimization and technology mapping target towards cells with a prescribed maximum dimension: an AOI family. In a subsequent top–down phase the transistor sizes are derived as explained above, and denoted as a pair of numbers left–under each tree node. The corresponding layout shows the cell, simplified by removing well and well–contact features, and after input ordering. Note that each cell uses polysilicon in vertical strips, metal–1 in horizontal strips, and metal–2 not at all. This leaves maximal transparency for later over–the–cell routing and efficient area usage. Furthermore it allows changing the





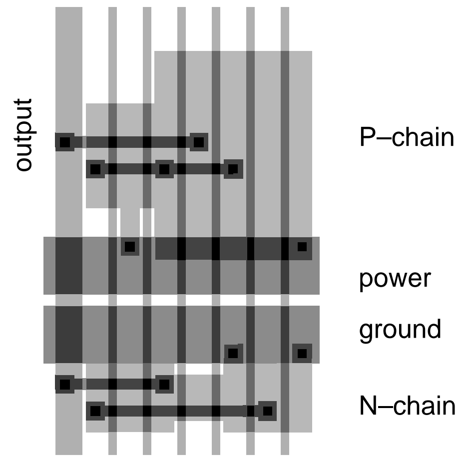


figure 4: transistorsizing and ordering for a cell example

cells speedfactor, after an initial placement and routing phase: The cell are horizontally abutted with the power lines, and can be adjusted for larger speedfactors by growing the transistors under the routing area. Besides the huge choice in cell boolean functionality, the freedom to size each individual cell allows a speed/power tradeoff of about a factor 3.

5. Timing issues

In commercial systems, fixed and well characterized cell libraries are preferred over automatic cell generators. The main reason for that is the confidence in their delay and power figures. However with well designed cells from generators and correspondingly adapted delay models in the logic synthesis package the same kind of reliability can be achieved, with the added advantages of more compact design realizations and simplified library maintenance [4]. The delay of a logic expression (cell) is modelled by the formula:

$$Delay = R_{int}\ x\ (C_{int} + (C_{wire} + C_{gates})/speedfactor))$$

Here *speedfactor* is a sizing parameter optionally assigned to individual cells, and is by default 1. C_{int}, models internal cell capacitances such as from diffusion to bulk, and grows with the cell complexity. C_{wire} is the wiring capacitance of the output net, which depends upon the placement and routing. Before placement and routing is done, this is estimated from the cell fanout, the total estimated circuit size, and averaged data gathered from previous designs. C_{gates} is the capacitive load of the fanout cells, which itself depends upon the fanout cells complexity and their speedfactors. Finally R_{int} is the cell driving resistance, which was made in principle independent of the cell type. Clearly all these R and C parameters also depend upon the target technology.

The delay model is kept consistently along the different phases of logic design, from multi–level optimization (creating a structured design from a two–level boolean specification, optimizing transistor count under a delay constraint), technology mapping (inverter optimization, fanout tree insertion, mapping boolean expressions to cells with prescribed maximum complexity), and cell sizing (globally minimizing total active area by assigning a speedfactor to all cells while satisfying a delay constraint). The cell sizing assigns a speedfactor from a prescribed interval (often 1–3, sometimes 0.5–5) to each individual cell, by means of Linear Programming, creating a globally optimal solution [5]. The LP method works fine for circuits up to several thousand cells, beyond which the numerical stability (not the CPU time) becomes problematic.

Already in the stage of logic optimization (before technology mapping) the delay estimates are accurate within 10%, compared with final figures from extracted layout, see figure 5. For final refinement and

5 different realizations of one circuit
delay estimates (ns) in the following stages:

- multilevel logic optimization
- technology mapping
- after layout extraction

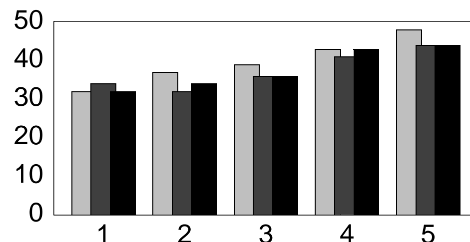


figure 5: delay estimates are in 10% accuracy

optimization of the delay, the system allows to use extracted net capacitances from the layout to be used

for a final (re–)sizing of the cells, in stead of the roughly estimated net capacitances from before placement and routing were done. The cells are then regenerated, and placed at their original positions. This is possible due to the layout style, where cells with different heights are placed in rows, a multi layer maze router is used allowing over the cell routing, and the cells are designed to be highly transparent for routing. As result the sizing can give a speedup of the total circuit of a factor 1.5 to 3, without really increasing the total layout area. Sizing for larger speedups tends to require excessive power.

As result compact blocks of cells are obtained, with a size ranging upto on the order of 10K cells, at first sight looking like standard cell layout. These blocks are subsequently placed and routed in a floorplanning/channelrout system. A small caption of a generated layout is shown in figure 6. Mapping towards more complex cells, allowing cells upto 4x4 or even 5x5 complexity, tends to increase the delay but can reduce both power and area.

6. Conclusion

A static CMOS cell generator has been made, with a fast implementation of a powerful algorithm. The cells are designed with circuit timing issues in mind, and made highly transparent to allow over the cell routing. In combination with a logic optimization system with a consistent delay model, high performance circuits are reliably designed.

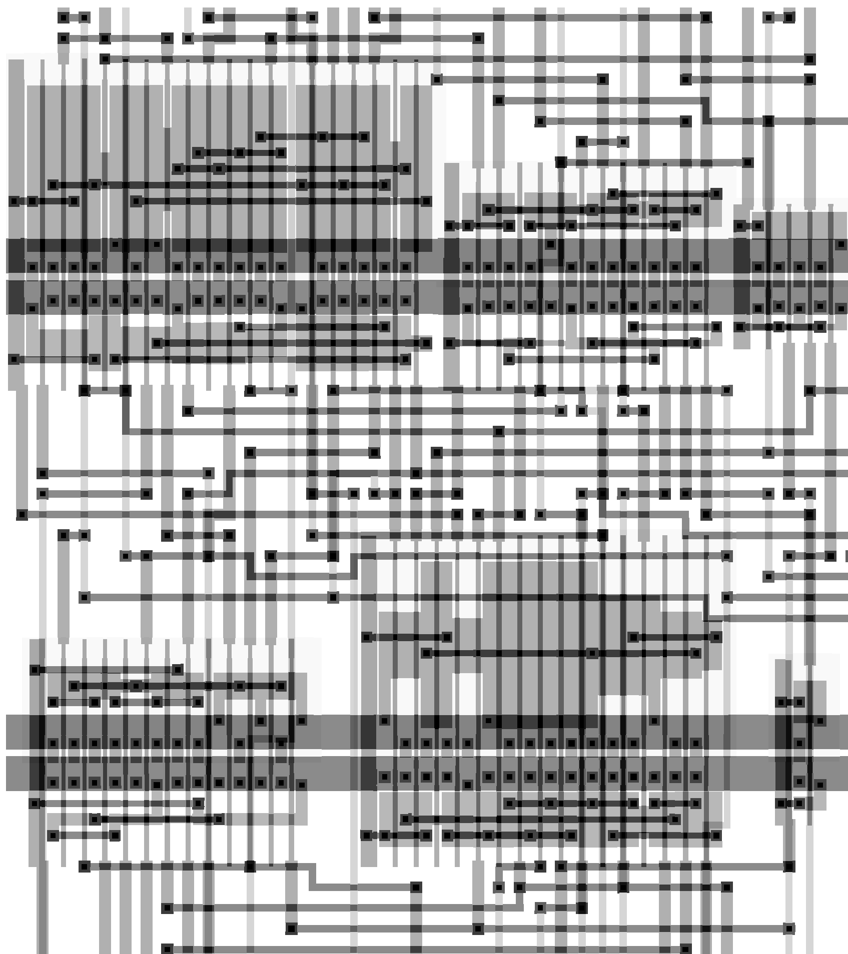


figure 6: layout caption with cells and routing

7. References

[1] R.L. MASIASZ, J.P. HAYES, Layout Optimization of static CMOS Functional Cells, *IEEE trans. on Comp. Aided Design*, Vol. 9, No. 7, pp. 708–719, July 1990.

[2] R.L. MASIASZ, J.P. HAYES, *Layout Minimization of CMOS Cells*, Kluwer Ac. Publ., 1992

[3] T. UEHARA, W.M. VAN CLEEMPUT, Optimal Layout of CMOS Functional Arrays, *IEEE Trans. on Computers*, Vol. C–30, No. 5, pp. 305–311, May 1981.

[4] M.R.C.M. BERKELAAR, J.F.M. THEEUWEN, Logic Synthesis with Emphasis on Area–Power–Delay Trade–Off, *J. of Semicustom ICs*, pp. 37–42, Sept. 1991

[5] M.R.C.M. BERKELAAR, J.A.G. JESS, Gate sizing in MOS Digital Circuits with Linear Programming, *Proc. European Design Aut. Conf. (EDAC)*, pp. 217–221, Sept. 1990