A design flow for performance planning : new paradigms for iteration free synthesis

A Design Flow for Performance Planning: New Paradigms for Iteration Free Synthesis

Ralph H.J.M. Otten

Eindhoven University of Technology, Faculty of Electrical Engineering, P.O. Box 513, 5600 MB Eindhoven, The Netherlands*

Abstract. In conventional design, higher levels of synthesis produce a netlist, from which layout synthesis builds a mask specification for manufacturing. Timing analysis is built into a feedback loop to detect timing violations which are then used to update specifications to synthesis. Such iteration is undesirable, and for very high performance designs, infeasible. The problem is likely to become much worse with future generations of technology. To achieve a non-iterative design flow, early synthesis stages should use *wire planning* to distribute delays over the functional elements and interconnect, and layout synthesis should use its degrees of freedom to realize those delays.

1 Introduction

Layout synthesis has always relied on wire length and area minimization under the constraints of a technology file (design rule set) to generate masks for chips that showed acceptable functionality, yield and performance. Interconnect served merely as the realization of the net list and its influence on performance was negligible.

This enabled a technique that was iteration free in the sense that there was a flow that started with *functional synthesis*, transforming the initial specification into a net list of modules and interconnections, that was handed to the back-end part in which a mask specification was to be constructed. Figure 1 shows schematically such a straight-line design flow. Mostly a *library* is available which either contains complete layouts of modules or procedures that can generate these layouts. The *technology file* consists of design rules, a compact, sufficient representation of what is possible in the target technology. The *footprint* captures the properties of the carrier, for example the image of the array on which the modules have to be mapped (such as with gate array and sea-of-gates realizations), the positions of bonding pads and possibly supply rails, maybe even preplaced modules (memory arrays or sensitive circuitry).

One particular approach from the eighties was summarized in [16], using principles from programming [15], and naming it *stepwise layout refinement* after a fundamental paper about program development earlier in that decade

* Part of this work was done at Delft University of Technology, Delft, The Netherlands, and part at the University of California at Berkeley, CA, USA.

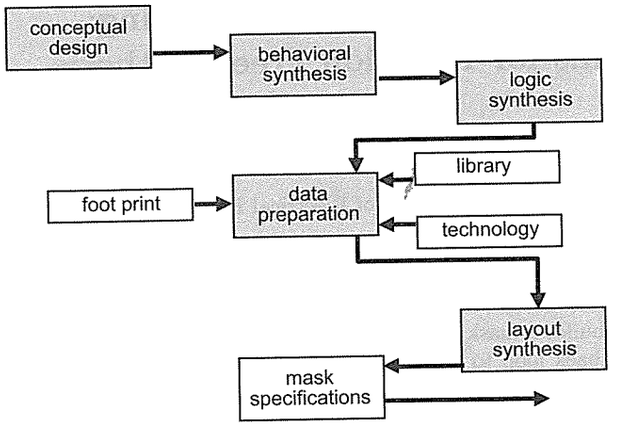


Fig. 1. The straight line design flow

[26]. The techniques were based on "postponing implementation decisions to avoid premature commitments that cause unnecessary constraints in the later stages of the design". It assumed that functional decompositions that were inherited from higher (behavioral) levels, contained useful information for layout synthesis, and also postulated that these decompositions were to remain recoverable as recognizable blocks for the designer. Layout synthesis was mainly the refinement and ordering of that "functional" decomposition.

Benefits were expected from such an approach because of a presumably high correlation between functional interdependence and connectivity. The latter, stored in *net lists*, was the main driver in layout synthesis, as many "flat" approaches minimized total wire length, while stepwise refinement tried to contain wires as much as possible within the lower levels of the refined hierarchy (that is within slices). Also this principle had a counterpart in "structured programming" [27].

As technology moves deeper into sub-micron feature sizes, and more components are integrated on a single chip, interconnect effects become more problematic, and those principles have to be reconsidered, and maybe more aspects of stepwise refinement. Especially, the blind acceptance of the functional hierarchy with gate and net lists in order to come to a layout by consulting technology files and libraries, often hampers achieving the required performance for today's designs, mainly because delay, both of gates and interconnect, are the more or less arbitrary outcome of total wire length minimization and subsequent sizings.

When revising the methodology its salient feature, a strictly top-down flow in layout synthesis, should not be given up however. The answer of the early nineties, still dominating the back end tools of today, is not complying with that precept (Figure 2). The effect of wiring on delay was determined by timing analysis tools that detect timing violations and produce either input

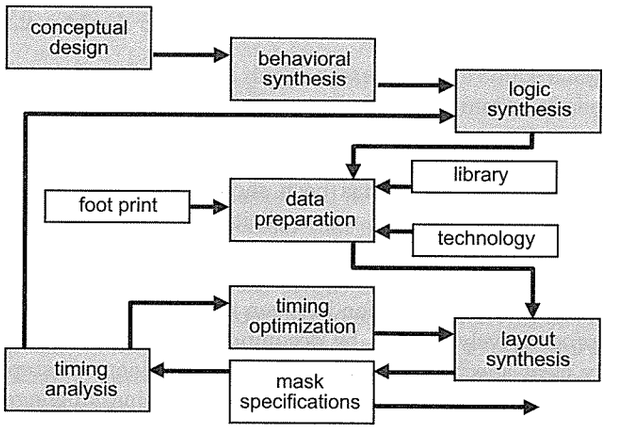


Fig. 2. Iterative flows: timing optimization, or even resynthesis whenever a timing violation is detected

for timing optimization procedures (such as transistor sizing, buffer insertion and fanout trees) or an updated specification file for higher level synthesis, expecting an improved gate and net list for layout synthesis. Essentially the back-end of the design process has become a slow iterative scheme with no guarantee of convergence. Even if the process converges, it is uncertain how the final solution compares with the optimum.

In this chapter we want to lay the foundation of approaches that effectively avoid global iteration loops. Obviously, the early design stages have to be integrated with layout synthesis, or at least able to incorporate sufficient layout considerations without unnecessary constraints for the back end. This will require a completely new approach, especially for complex designs with very tight performance constraints. The required performance must be guaranteed by construction (and not be left as the arbitrary outcome of indirect optimizations). This affects not only the way layout synthesis should be organized, but also higher levels of synthesis, and logic synthesis in particular.

We first study components of conventional flows to identify the biases that require revision. This leads to a new look at the concept of a *global wire*. After observing that the delay of "long" wires can be made linear in their length (and thus total interconnect delay on a path independent of the position of restoring circuits), and discovering that delay between buffers in an optimally buffered interconnection is a constant of the target technology, we can extend the notion of wireplanning as the task to lay out the interconnection structure before deciding on the functional content of the nodes. Assuming that functional synthesis can provide the delay distribution over interconnect and gates, we declare the need for algorithms to produce networks not violating the timing constraints under the linear wire delay model, and gate-based synthesis with fixed delays.

2 Flow Components

2.1 Introduction

Stepwise refinement is a technique that has been shown to be effective in the development of computer programs. It was explicitly formulated in a famous paper by Niklaus Wirth [26]. In that paper the design of a structured program was viewed as a sequence of refinement steps. Starting with a clear problem statement, that specifies the relation between the input and the output data, the task is progressively refined, by decomposing it into subtasks, each having an equally clear specification. The sequence of refinement steps terminates when all tasks are specified in a chosen programming language. The constructs of that language should be a direct translation of tasks resulting from the final refinement steps. To be effective, they have to form a small but powerful enough repertoire. This method thus entails a hierarchical structure. (A *hierarchy* is, either a set of hierarchies, or an atom. In this case each hierarchy represents a task, and each task translatable in a construct is an atom.)

Stepwise refinement can also be viewed as postponing implementation decisions, to avoid committing the program prematurely to a specific implementation. Each decision should leave enough freedom to following stages to satisfy the constraints it created, and at the same time rearrange the available data such that further meaningful decisions are possible in the next step. So, concurrent with the gradual stiffening of the design, the information is progressively organized so that more and more detailed decisions can be derived.

The principles of stepwise refinement obviously apply to any complex design task based on a top-down strategy rather than to a process of combining independently developed subdesigns. Completely specified subdesigns, in general, are difficult to handle, because the flexibility and the information for adapting them to their environment is often not available when they are designed.

On the other hand, the application of stepwise refinement in layout design raises a number of questions. Firstly, what information is available in the initial stage of layout design? A difficulty in answering this question is how to separate layout synthesis from the other design tasks, and yet make sure that these tasks are performed "with layout in mind" and provide enough information to preserve these decisions. Another question that immediately arises is what relevant information can be derived at the intermediate stages before fixing the geometrical details in the final stage? Finally, translation of the results of the last refinement steps has to be considered. Section 3 will be devoted to these questions. In this section the "environment" of layout synthesis (including "data preparation") is discussed.

2.2 Mask Specification

The ultimate task of a design system is to produce a *layout*, a set of data that uniquely and completely specifies the geometry of the circuit. Usually this data is an encoding of patterns, two-block partitions of the plane. The term *mask* will be used for each plane with a pattern, even if that plane is not exactly one of the real masks used in the fabrication. A layout is then translated into a sequence of processes that selectively change the characteristics of the silicon according to those patterns, thus realizing the functional specification available as input to the layout design procedures. Whereas the layout design system has considerable freedom in deriving geometry from functional specifications, the result of that translation procedure is fixed. Up to forty different patterns are sometimes used in the sequence of selective exposures of the wafer surface. However, many of these are implied by other patterns in the sequence. For present day technologies the geometrical specification of eight to fifteen planes suffices to specify the layout.

From device theory general restrictions on the shapes of the regions can seldom be derived. Lithography techniques, however, do sometimes have their limitations. Quite often only orthogonal artwork is acceptable. This leads to regions that are unions of iso-oriented rectangles. There are examples of circuits that indicate that other layout primitives are more efficient, such as hexagons in some systolic arrays. Rarely is a restriction to rectangles and combinations thereof detrimental, whereas the cell design algorithms and layout data bases profit from such a restraint. The rectangle is therefore accepted as the basic construct.

The rectangle constraint is also accepted for the compounds of layout primitives that form the atoms, and often even the hierarchies in the hierarchy entailed by the method of stepwise refinement. Consequently, each hierarchy will then be a rectangle dissection in the final layout, i.e. a rectangle subdivided into nonoverlapping rectangles. The restriction to rectangles might seem rather arbitrary. However, in a truly top-down design it is very difficult to be particular about these shapes, because the shapes of its constituent parts still have to be determined. A good estimate for the shape of the enclosing region is of great value in determining the shape and positions of the constituent parts, and hardly a constraint if these parts have a high degree of flexibility. The earlier estimates become in some sense even self-fulfilling, because the parts mostly can be fit nicely in the environment by using their flexibility. Besides, choosing rectangles as the only constructs in the repertoire simplifies the formulation of design decisions, and lowers the complexity of deriving these decisions, as will be seen later.

2.3 Technology File, Library and Footprint

To improve chances for successful integration of the circuit, and increase yield when the circuit goes into production, patterns are required to satisfy certain

rules, the so-called *design rules*, stored in a *technology file*. A first classification distinguishes roughly two classes of rules: *numeric rules* quantifying extensions of, and spacings between patterns in a plane and in combinations of planes, and *structural rules*, enforcing and prohibiting certain combinations.

There usually are a large number of numeric rules. Very few, however, are critical in a layout. For example, the spacing between two separate pieces of metal in the same layer is bounded below by different numbers depending on whether or not there are contacts to other layers in one or both of these metal pieces. In a wiring algorithm there rarely is a good reason for trying to use all these different minima. Instead, the maximum of all the rules that might apply is taken as the *pitch* for the metal in that layer. The reason for specifying the different rules for all special cases is mostly for the (manual) optimization of small pieces of layout that are used repetitively, such as memory cells. The numeric rules are almost exclusively specifications of lower bounds. This does not imply that the concerned extensions and spacings can be arbitrarily large. Making them arbitrarily large might impair the functioning of the devices in the circuit and increase delays, and decreases the yield. The rules are formulated as minimum rules only, because it is assumed that the layout design techniques will try to keep the total chip small.

In practice, quite often a *footprint* is prescribed, giving the geometry of the estate on which the complete circuit is to be placed. It is a rectangle with iso-oriented rectangles contained in it. These rectangles represent preplaced objects that are part of the circuit description, but cannot be freely placed and/or oriented. The remaining freedom varies. In so-called master image approaches all active components are placed and the circuit components (transistors) have to be "assigned" to these slots, and interconnections have to be realized in the metallization layers on top of them. But preplaced objects may also be complete layouts of circuit parts, complete with their wiring, leaving only part of the wiring space available for other interconnections. Such objects may occur in fully custom fabrication styles (that is all masks have to be produced for the circuit, whereas in master images the lower masks are a priori fixed, thus prescribing the component slots) as well as in other styles such as master image. Important to note is that, in contrast with the free area suggestion above (with only implicit insentives to keep the total size small), giving a footprint may make the design problem unsolvable. Of course, insufficient area for the components certainly excludes the existence of the circuit on that footprint, but other aspects for obtaining full functionality may be precluded by the properties of that constraint.

The other resource from which the *data preparation* program in Figure 1 and 2 draws to enable layout synthesis to produce the mask specification for the net lists that higher level syntheses have delivered is a library of *cells* or of procedures to generate these cells. Cells can be of various types. The type determines the cell's flexibility and how to obtain the layout of a cell. The most rigid cell type is the *inset cell*. Its configuration and pin positions are

fixed and stored, or completely implied by the algorithm generating the cell. The layout design system can only assign a location and an orientation to such a cell, within the restrictions of the footprint[1]. Cells can have a higher degree of flexibility. The algorithms that determine their layouts, are such that estimates about the environment can be taken into account. This certainly is true for general purpose cells such as *macros*. These cells have a decomposition of their own into circuits of a particular family. Another, less flexible, example of a general purpose cell is a programmable logic array. Its potential for adapting to its environment is sometimes further diminished by optimizations such as row and column folding that impose stricter constraints on the sequences in which nets enter the cell region. There are also cells, such as cells generated by algorithms that, depending on a few parameters, construct special purpose subcircuits such as arithmetic logic units, rotators and adders. These cells have limited flexibility, such as permitting stretching in one direction. Stretchability is often important in avoiding pitch adjustments in data buses.

Good cell generators working with the design rules produce valid layouts in a wide range of values for these lower bounds. Of course, the algorithms do not produce optimal layouts for all combinations of values in these rules, but they should produce acceptable solutions for all practical value sets. The latter requirement is much more difficult to maintain under changes in structural rules, because these changes often require completely different decisions. The rules usually increase the dependance between different masks. This is particularly problematic if the metal layers are involved. Rules that forbid or enforce certain overlaps between patterns in the metal layer masks and other masks affect the wiring routines which often are generic algorithms solving some cleverly isolated interconnection problem. Introducing structural constraints often invalidate the assumptions made during the isolation.

2.4 Conceptual Design and Synthesis

Conventional design flows are based on the dichotomy between the *front end* and the *back end* of a design system. The back end is dominated by layout synthesis, the topic of section 3. Its task is to accept a net list, possibly hierarchically organised, but complete in the sense that all components and all their interconnections are fully specified, and to produce a *layout* (see section 2.2). Part of this data, both in the input and in the output, is in pointers to library elements.

The front end has to produce these net lists for layout synthesis, together with the pointers into the library. Isolating tasks of an integrated approach is dangerous, because of their mutual dependence. Taking this dependance into account by iterations over several design tasks is highly undesirable, because of the time complexities involved and convergence properties. Clearly,

[1] Preplaced objects are typically inset cells.

since the final result has to be a complete specification of the masks, the later steps are mainly based on layout considerations. And, since the complete functional specification has to be available from behavioral synthesis on, the early decisions or refinement steps have to be predominantly based on function and testing arguments. In between, many steps, such as logic decomposition and data path definition, have a significant influence on the final layout, and many functionally almost equivalent decisions may have completely different consequences for the layout and its design process. The boundary between layout design and the other synthesis tasks was always therefore quite fuzzy. Nowadays the performance of integrated circuits depends heavily on the geometrical aspects of the final chip. A total dichotomy as described is no longer feasible. How to handle this will be main theme of the later sections of this chapter, and one conclusion already pursued by some start-up companies working toward new back-end tools, might be to remove all hierarchy constraints in layout synthesis. Early stages, however, will still be decomposing the overall function in order to make the task manageable. The earliest stages are called *conceptual design.*

Typically, conceptual design for complex integrated circuits is done in small teams of experts. These experts work for a large part on the basis of experience. Experience has learned them how size (or power) can be traded against speed. On that basis they divide for example time budgets over identified parts of the design. These identified parts are a first level of hierarchy. Beside dividing time budgets, also relative positions and estimated sizes are tried, often on a white board or scratch paper. Several iterations may follow, and further elaboration on parts, creating deeper levels of hierarchy, may follow before specifications, suitable for behavioral synthesis, are written, mostly in description languages, such as *vhdl* or *verilog.* The intermediate stages, the sketches on the white board and on paper, are in fact *wire plans*, and when more sharply defined can be the structure on which quick analyses can be carried out.

We will come back to these possibilities in section 6. For the present discussion we note that conceptual design will inevitably lead to a functional hierarchy, probably reflecting the functional interdependence of the hierarchies (modules). Functional interdependence and connectivity are often highly correlated, and the latter can be an important basis for decisions in layout design. The considerations that lead to a functional hierarchy mostly ignore other important aspects of layout synthesis. For example, in the design of digital systems the isolation and implementation of execution units is often established quite early. The remainder, control and interrupt, is left logically completely specified, but mainly unstructured. A layout with a decomposed, or even partly duplicated control unit, might be more efficient than a layout in which this part has been kept together. Several sections of the control can be placed closer to specific execution units they are heavily connected with. If the connectivity with the rest of the control is relatively low, this might save

wiring area. It is also possible that the decomposition goes further than what is useful for layout decisions. The layout design part may, therefore, choose to ignore parts of the decomposition, initially or throughout. Nevertheless it is assumed that, possibly after some clustering around seeds and some pruning, the design data are completely hierarchically structured. That structure is considered part of the initial data for layout synthesis. The hierarchies and atoms are called modules in this context. The formal definition of a module implies such a hierarchy.

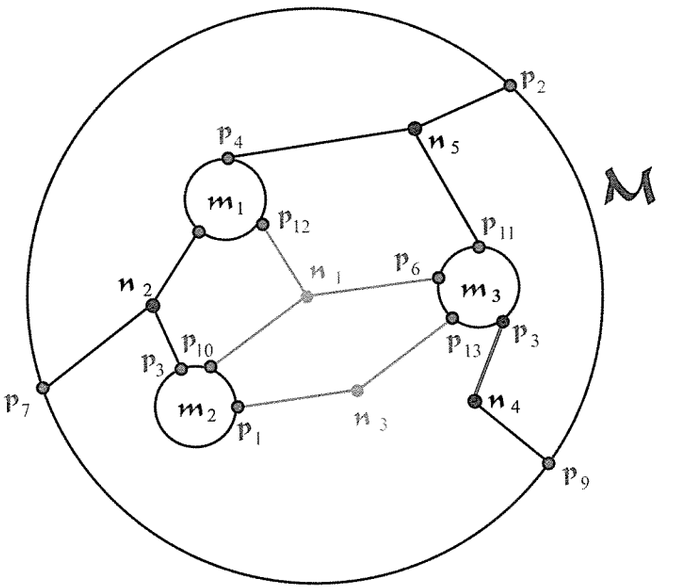


Fig. 3. A pictorial representation of an incidence structure

A *module* $\mathcal{M}$ is defined to be a collection of modules $\{m_1, m_2, \cdots, m_m\}$ where m =| $\mathcal{M}$ |, and an incidence structure $\mathbf{IS} = (\mathcal{M} \cup \{\mathcal{M}\}, \mathcal{P}, \mathcal{N})$. The modules m_i are the *submodules* of $\mathcal{M}$, and *comodules* of each other. $\mathcal{M}$ is their unique *supermodule.* There is exactly one module without a supermodule. This module represents the entire system to be integrated. *Cells* are modules with an empty set of submodules. The others are called *compounds.* The hierarchy can be represented as a rooted tree. The modules are represented by the nodes. The root represents the whole system or chip. The leaves represent the cells. The internal nodes represent the compounds. Each node representing a submodule is the end of an arc that started in the node representing its supermodule.

With regard to the incidence structure **IS** the module and its submodules are considered to be subsets of the set of pins $\mathcal{P} = \{p_1, p_2, \cdots, p_p\}$ Also the *signal nets*, forming $\mathcal{N} = \{n_1, n_2, \cdots, n_n\}$ are considered to be subsets of $\mathcal{P}$. *Pins* are for the moment merely a mechanism for relating modules and their supermodule with signal nets. The incidence structure can be represented by a bipartite graph $(\{\mathcal{M}\} \cup \mathcal{M} \cup \mathcal{N}, \mathcal{P})$ the *potential graph*. Figure 3 illustrates the terminology.

2.5 Timing Analysis

The single most important consideration in designing complex systems is conceptual integrity. An important aspect of this integrity is how to store the data of a design between the various stages. As pointed out in the previous subsection, the design has a hierarchical structure while being treated by higher level synthesis and being prepared for layout synthesis. The modules in that hierarchy may have specific meaning for certain parts of the system. For example, they may have a functional model associated with them. Such a model makes simulation of that module in its environment possible. Extensive circuit simulation will have been performed on the system before the layout is considered. Yet, certain important performance aspects heavily depend on parasitic elements and final device parameters, and these are not known until the layout is determined. Simulation is therefore also important during and after establishing the layout. This requires that the simulation part must be able to find the modules for which a model is known, and assign values to parameters that represent the influence of device realizations and parasitics. It is therefore expedient that the results of the layout design process are stored in a way compatible with the data representation delivered by previous design procedures. In the preceding subsection it was established that that data is hierarchically structured. A hierarchy is mostly represented by an unordered tree. It would be convenient if the layout design procedure could preserve that structure, possibly refined and ordered. Refinement here means that leaves can be replaced by hierarchies of which the root takes the place of the replaced leaf, and subtrees consisting of a module with all its submodules can be replaced by any tree with the same root and the same leaves, but a number of additional internal nodes. Section 3 will describe such refinement steps.

Recently, simulations after layout synthesis have become necessary to check whether the system as a whole satisfies the performance requirements. A timing analyser is presently integrated in most design flows (Figure 2). From the result of layout synthesis a network, together with its parasitics and wire properties has to be extracted. Consequently, a network far more complex than the original net list produced by higher level synthesis has to be analysed. Besides, accurate timing models are very complex, and with such models timing analysis will be become a very time consuming procedure.

Considerable simplifications are therefore introduced, hoping that all timing violations can still be reliably detected.

Worse than the inaccuracy of the result is the fact that it is not clear whether something can be done to remove the violation. Identifying critical paths, and speeding them up by transistor sizing, fanout buffering, and path isolation, may help, but failure to do so, does not mean that a timing correct solution does not exist. Anything produced is constrained by the result of layout synthesis, that in conventional design systems optimizes metrics like size, wire length, and the like, but not speed directly. Relying on higher synthesis stages to propose repairs or even complete redesign is not always converging either (even when an acceptable solution exists), slows the design process even further down, and is also likely to get trapped locally.

If performance is the requirement, then all optimization should be under that constraint, and no longer rely on minimization of wire length and area to get acceptable performance. Rather higher level synthesis should fix the allowed delays on interconnect and in the gates, and layout synthesis should have as its main task the realization of these delays. This requires radical changes on both sides. These issues will be addressed at the end of this chapter.

3 Layout Synthesis

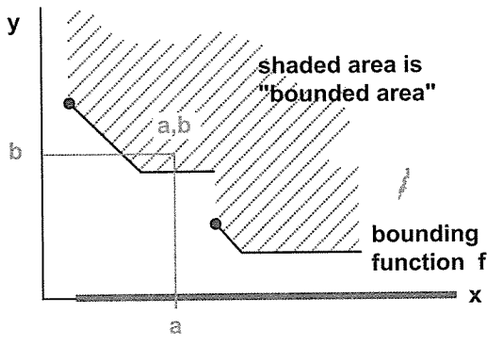
3.1 Shape Constraints

For every cell in the hierarchy there is an algorithm that tries to adapt the cell to its estimated environment, while generating its detailed layout. This preliminary environment has to be created on the basis of estimates concerning the area needed by each cell, feasible (rectangular) shapes for it, and the external interconnection structure. The size and the shape of a cell are constrained by the amount and type of circuitry that has to be accommodated in that cell. It is reasonable to expect one dimension of the enclosing rectangle not to increase if the other dimension is allowed to increase. Constraints satisfying that requirement are called *shape constraints*. The precise definition follows.

Definition 1. A *bounding function* is a right-continuous, non-increasing, positive function of one variable defined for all real values not smaller than a given positive constant.

Definition 2. The *bounded area* of a bounding function is f is the set of pairs of real numbers (x, y) such that $f(x)$ is defined and $y \geq f(x)$.

Definition 3. The *inverse* f^{-1} of a bounding function is a bounding function defining the bounded area with exactly those (y, x) for which (x, y) is in the bounded area of f.



2. right continuous
3. monotonously not increasing
(4. Piecewise Linear)

shape constraints:
a ∈ support F
b ≥ F(A)

1. support $f \subseteq R_0^+$, connected, and $\infty \in$ support f

Fig. 4. The definition of a bounding function

The *shape constraint* of a module (or cell) is a bounding function specifying all rectangles that can contain a layout of that module. The bounded area is the set of all dimension pairs of these rectangles.

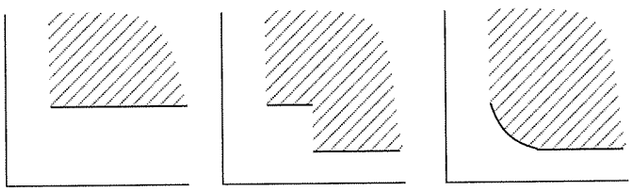


Fig. 5. Some examples of shape constraints

Inset cells have piecewise linear shape constraints. Such constraints can be conveniently represented by a sequential list of their breakpoints. This is not the case for flexible cells, and possibly other cell types occurring in practice. Of course, any shape constraint can be approximated by a piecewise linear bounding function with arbitrary accuracy. From the discussion of flexible cells in section 2.4 it is clear that a piecewise linear approximation with three breakpoints suffices considering the limited accuracy of any area estimation for the given examples.

The shape constraints of the modules in the functional hierarchy can be derived in a straight forward manner from the shape constraints of cells as we will see in section 4.2.

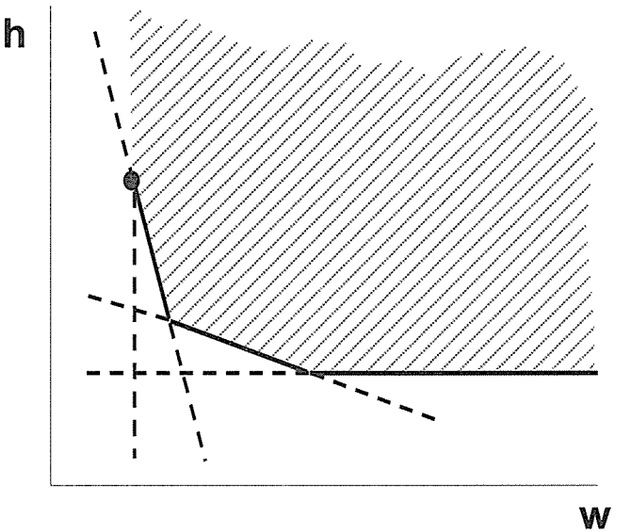


Fig. 6. Piecewise linear approximation of the shape constraint of a flexible cell

4 Placement Versus Floorplan Design

Through the shape constraints, the estimation of the rectangle in which the module is going to be realized, is controlled. Some guidelines for the position of such a rectangle among all the other rectangles are contained in the functional hierarchy, if available. That structure already gives some indication about which modules belong together functionally. Otherwise, or further, we have at least the incidence structures associated with the modules. In the context of layout these incidence structures are often called *net lists*.

Utilizing the data (shape constraints, net lists, and functional hierarchy) the cells have to be arranged in a rectangle. This enclosing rectangle is often desired to be as small as possible, sometimes it is constrained in aspect ratio or completely specified. If the cells were fixed objects this would be the classical placement problem. However, in this context the cells are allowed to take any shape not excluded by its shape constraint. This generalization of placement is called *floorplan design*, and a *floorplan* is a data structure fixing the relative positions of the objects. It does not contain geometrical aspects, although estimates can be generated by performing a suitable floorplan optimization routine (see section 4.2).

Both floorplan design and placement are guided by a number of objectives, not easy to formulate in a single object function. This can be illustrated by the following typical combination of objectives. The first is primarily concerned with the realization of the interconnections. A common figure of merit for it is total wire length, often estimated by summing the perimeters of the rectangles

that enclose all module centers connected to the same net. At the same time it is desirable to give the cells rectangular regions in which they can be efficiently allocated. The first objective is of a rather topological nature, working with concepts such as 'close', 'neighbor' and 'connectivity'. The latter is more a geometrical objective. Major concepts for it are 'deformation', 'dead area', 'aspect ratio', and 'wiring space'. To relate the two objectives an additional refinement step, using an intermediate structure capturing much of the data affecting one of the objectives, might be helpful.

4.1 Floorplan Topologies [17]

It has already been observed that in the final floorplan the modules will be rectangle dissections in which each submodule is either a rectangle dissection itself, or, in the case of cells, a rectangle. Creating a preliminary environment for the cells is essentially generating certain aspects of the rectangle dissection, in which each cell is an undivided rectangle. Since the shape of the cells is not yet known in that stage, the geometrical details of the rectangle dissection cannot be determined. Less restrictive aspects of a rectangle dissection are its neighbor relations, i.e. which cells share a particular line segment in the rectangle dissection. The set of neighbor relations is called the *topology* of the rectangle dissection. This topology is useful information that can be generated at an intermediate stage of the refinement process. Usually, enough freedom is left for the cell assembling procedures after fixing the topology, and such a topology provides useful information about the environment of the cells. Therefore, the first task in designing a floorplan is to determine its topology. A reasonable decomposition of that task, certainly in the light of the discussion of section 2, is to take one module at a time, starting with the root of the functional hierarchy, and progressing downward such that no module is treated before its supermodule is. This translates the functional hierarchies into nested rectangle dissections.

In spite of the constraints accepted so far, the floorplan design problem is still complex. For example, given its topology and the shape constraints of its cells, finding the smallest floorplan is an $\mathcal{NP}$-hard problem. There also is no pseudo-polynomial algorithm for it, since the corresponding decision problem is strongly $\mathcal{NP}$-complete [22]. At this point one may ask whether the class of topologies for which the previous problem, and hopefully several other problems, can be solved in polynomial time, is still large enough to include an efficient floorplan topology for all practical cases. To answer that question that class has to be identified.

A concise way of representing the topology of a rectangle dissection is by its *polar graph*. This is a plane, directed graph without cycles. There are three bijective relations between elements of this graph and its associated dissection: edges correspond one-to-one with undivided rectangles, vertices with the elements of one set of iso-oriented line segments, and inner faces with

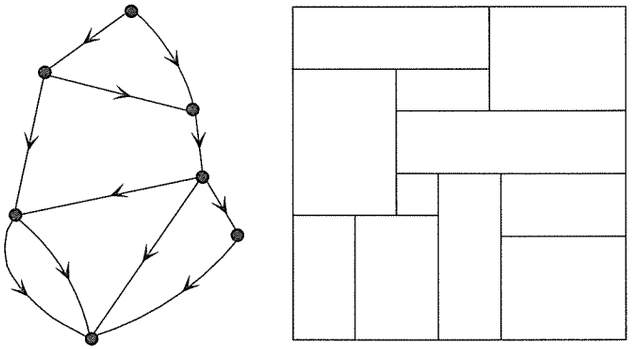


Fig. 7. One of the polar graphs of the given rectangle dissection

the line segments in the other set (Figure 7)[2]. Many floorplans, designed in practice or with any of the successful, more special layout styles, have polar graphs that are two-terminal series-parallel graphs.

A first observation is that, as any two-terminal series-parallel graph, such a topology can be represented by a much easier to handle data structure, namely an ordered tree. By restricting floorplans to such a topologies, it becomes quite natural to maintain data structures in the sense of section 2: by ordering and refining the given functional hierarchy according to one of the rules there described, and the layout structure can be stored consistently.

The tree replacing a two-terminal series-parallel graph is called the *decomposition tree* of that graph. Its leaves correspond with the arcs, and its internal nodes correspond with the two-terminal series-parallel subgraphs of the original graph. Consequently, each leaf represents an undivided rectangle and each internal node represents a rectangle dissection, also with a two-terminal series-parallel graph. The rectangle dissections represented by the endpoints of tree-arcs starting from the same tree-vertex, are placed next to each other in the same order, either from the left to the right, or from the top to the bottom, depending on whether the corresponding two-terminal series-parallel graphs are connected in parallel or in series. A rectangle dissection with a two-terminal series-parallel graph as polar graph is a rectangle dissected by a number of parallel lines into smaller rectangles that might be dissected in the perpendicular direction. Such structures are called *slicing structures* (Figure 8) and the associated tree a *slicing tree*. Each vertex represents a *slice*. Each slice either contains only one cell, or is a juxtaposition of its *child slices*. In the latter case that slice is said to be the *parent slice* of its child slices, and these child slices are the *sibling slices* of each other. The

[2] Although polar graphs are fully general in representing topologies of rectangle dissections, they cannot handle so-called *empty spaces* in a flexible way. Recently, *sequence pairs* have been introduced. These can handle empty spaces.

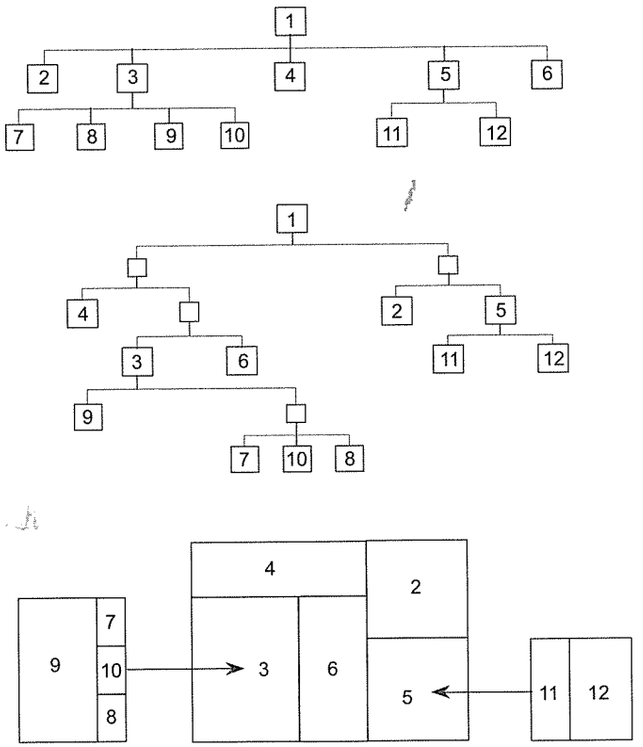


Fig. 8. Slicing

sibling slices are ordered according to their position in the parent slice (for example, left to right and top to bottom).

There are several ways of obtaining slicing structures. A well-known method is the min-cut algorithm [13]. If applied in its pure form it leads to binary slicing trees, but one clearly can extend it to produce general slicing trees. It does not use an intermediate structure that captures globally a large part of the topological aspects of the input, as suggested in section 4.1. Each dissection divides the problem into smaller problems, and it is difficult to take into account decisions in one part when handling the other parts.

Methods using an intermediate structure are also known. One such structure is a point configuration in which the topological properties of the input are somehow translated into a closely related geometrical concept, namely distances, and since the configuration will be embedded in a plane, more particular distances in the two dimensional euclidean space. High connectivity is reflected in relatively short distances. The size of the modules and the number of pins (requiring a certain perimeter) also may influence the relative

distances. The preferred distance metric is often Minkowski-1, because of the orthogonal artwork required by many lithography techniques.

4.2 Floorplan Optimization

Properties of the final rectangle dissection have to be derived then from such an intermediate structure as a point configuration and the shape constraints. This is called *floorplan optimization.* The topological considerations should be taken into account by preserving relative positions in the point configuration and keeping modules close together if they are represented by points with short distances between each other. The geometrical aspects should be taken care of by keeping track of, for example, deformation implied by the dissections. Another, often applicable, guideline is the area distribution in balanced designs such as those built out of columns of cells with one dimension fixed.

First we develop the mechanisms for manipulating shape constraints. To use this for floorplan optimization with a given slicing tree is then straight forward, but we will also show that we can obtained the "best" slicing structure compatible with a given two dimensional point configuration. Of course, we have to say what we mean by "best". A quite general and often adequate objective is to minimize a *contour score*

Definition 4. A *contour score* c is a function of two variables, defined for a convex subset Γ of the pairs of positive real numbers, which is quasi-concave and monotonously non-decreasing in its two arguments, i.e.

$$\forall_{x_o,x_1,y_o.y_1\in\Gamma}\,[x_1 \geq x_o \wedge y_1 \geq y_o \Longrightarrow c(x_1,y_1) \geq c(x_o,y_o)]$$

and

$$\forall_{\mathbf{x}_1,\mathbf{x}_2\in\Gamma}\forall_{0\leq\lambda\leq 1}\,[c(\mathbf{x}_1) \leq c(\mathbf{x}_2) \Longrightarrow c(\mathbf{x}_1) \leq c((1-\lambda)\mathbf{x}_1 + \lambda\mathbf{x}_2)]$$

Area and perimeter are examples of contour scores. Therefore, if we can minimize contour scores under compatibility and shape constraints, we can construct the smallest compatible rectangle dissection. Also , the smallest rectangle with a given aspect ratio, or with a lower and upper bound on the aspect ratio can then be produced. If we can do that in polynomial time for slicing structures we have identified the class we were looking for, since even for special contour scores and shape constraints the problem has been shown to be $\mathcal{NP}$-hard for more general dissections.

The shape constraint of a compound slice can be derived from the shape constraints of its child slices as is illustrated in Figure 9.. In the final configuration these child slices have to have the same longitudinal dimension, which is the latitudinal dimension of their parent. The inverse of the compound's shape constraint is only defined on the intersection of the intervals on which the shape constraints of its children are defined. lts smallest possible longitudinal dimension for a given feasible latitudinal dimension x is the sum of the

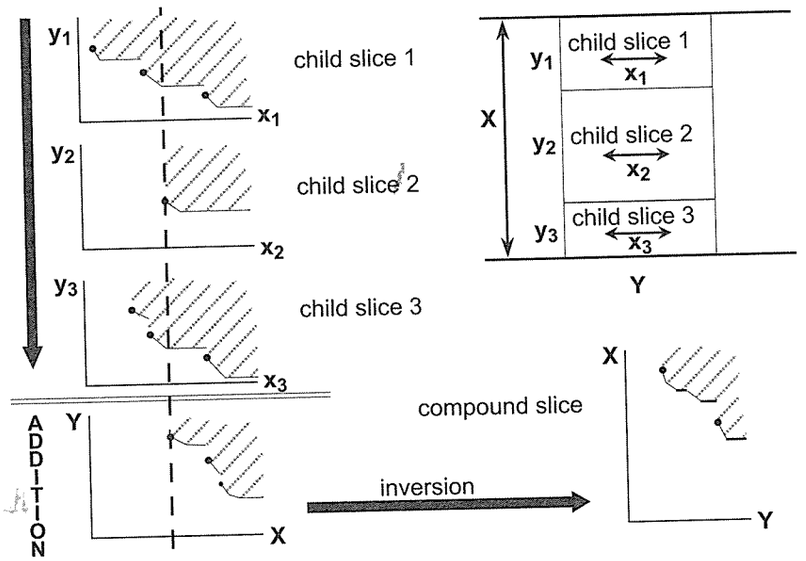


Fig. 9. Shape constraint addition and inversion

values of the shape constraints of the children at x. So, the shape constraint of a compound is obtained by the addition of the shape constraints of its children in the interval where they are all defined, and determining the inverse of the resulting bounding function. These operations are easy for piecewise linear shape constraints, represented by a list of their breakpoints ordered according to the respective longitudinal dimensions. For each breakpoint of any child of which the first coordinate x is in the mentioned intersection, the shape constraints of all the children have to be evaluated and added. If the result is y, then (y, x) is a breakpoint of the parent's shape constraint. Ordering all these new breakpoints according to the y-value yields a consistent representation for the shape constraint of the corresponding compound slice.

The ability to obtain the shape constraints of a slice by adding the shape constraints of the child slices and inverting the result, enables us to obtain the shape constraint of the enveloping rectangle. The bounded area of that shape constraint is the set of all possible outer dimensions of the total configuration. A contour score always assumes its minimum value over the bounded area at the boundary determined by associated shape constraint. This is a consequence of the monotonicity of shape constraints and contour scores. For piecewise linear shape constraints that minimum will be assumed at at least one of its breakpoints, because of the quasi-concavity of the contour score. So, to find an optimum pair of dimensions for the common ancestor slice the contour score only has to be evaluated at the breakpoints of its shape constraint in the convex set of permissible pairs.

Theorem 1. *Given a slicing tree and shape constraints for all its leaves (cells) the shape constraints of all modules can be determined by traversing the tree bottom-up (e.g. in a depth-first manner).*

Corollary 1. *Given a slicing tree and shape constraints for all its leaves (cells) the shape constraints of the chip can be determined by traversing the tree bottom-up (e.g. in a depth-first manner).*

Theorem 2. *Given a slicing tree and piece-wise linear shape constraints for all its leaves (cells), the optimum shape of the chip under a given contour score is represented by at least one of the breakpoints in its shape constraint.*

Given the longitudinal dimension of a slice and its shape constraint, its latitudinal dimension can be found by evaluating its shape constraint for the given longitudinal dimension. After deriving the shape constraint for the common ancestor and determining a dimension pair for which the contour score assumes a minimum, the longitudinal dimensions of its children are known. So, for each of them the latitudinal dimension, which in turn is the longitudinal dimension of its children, can be calculated. Continuing in this way will finally yield the dimensions of all slices in the configuration. If the shape constraint has a zero right derivative at the point where it has to be evaluated, some slack area might have to be included, i.e. the slice can be realized in a smaller rectangle without affecting its environment. In order to have the wiring channels connecting to other wiring channels at both ends this slack should be taken up by slices containing only one cell.

Theorem 3. *Given a slicing tree and shape constraints for all its leaves (cells) and feasible dimensions for the chip (that is, contained in the bounded area of the chip's shape constraint), feasible dimensions for all modules and cells can be found in a top-down traversal of the tree.*

Corollary 2. *Given a contour score, a slicing tree and shape constraints for all its leaves (cells), feasible dimensions of all modules and cells in an optimal chip with respect to the given contour score, can be obtained in two tree traversals, one bottom-up followed by one top-down..*

The algorithm consists of three parts:

1. Visit the nodes of the slicing tree in depth-first order, and just before returning to the parent determine the shape constraint by adding the shape constraints of its children and inverting the result.
2. Evaluate the contour score for each of the breakpoints of the shape constraint of the common ancestor, and select a dimension pair for which the smallest value of the contour score has been found.
3. Visit the nodes of the slicing tree in depth-first order, and before going to any of its children determine the latitudinal dimension by evaluating its shape constraint for the inherited longitudinal dimension.

Clearly, when in the first step the same procedure has been applied to all children of a certain slice the shape constraints of these child slices are known and combining them in the way described yields the shape constraint of their parent. The process will end with determining the shape constraint of the common ancestor slice, and then the shape constraints of all slices are known. As explained earlier the contour score will be evaluated at each of its breakpoints. The dimensions associated with the minimum value will become the dimensions of the enveloping rectangle. This means that after completing the second step the longitudinal dimension of the primogenitive (and all the other children) of the common ancestor is known. Together with the shape constraints this is enough information to begin the process of the third step. At the beginning of a visit to a node in the structure tree, representing a certain slice, the latitudinal dimension of that slice can be determined by evaluating its shape constraint at the value of the dimension that it inherits from its parent slice.

So, completing all three steps yields the dimensions of all slices in an optimum configuration for the given floorplan and cell shape constraints. To determine the position coordinates of the slices from these dimensions and the floorplan is straight forward. Also easy is to determine what orientation the inset cells can have in this optimum configuration.

The traversals themselves are linear in the size of the trees, but the sorting of breakpoints is superlinear. The number of breakpoints is linear in the number cells if the shape constraint of the cells have a limited number of breakpoints. The exact worst case depends also on the tree (balancing helps), but in any case we have:

Theorem 4. *The floorplan optimization problem is efficiently solvable under any given contour score for slicing structures with a given tree and with piecewise linear shape constraints for the cells.*

So far we assumed that the slicing tree was obtained by refinement and ordering operations on an initial hierarchy. This is not a completely satisfactory answer, because it is not obvious how these operations have to be carried out for modules with a large number of submodules, or when hierarchy is not accpeted as a constraint in layout synthesis. As mentioned, there are several techniques that produce intermediate structures, and point configurations play a dominant role among them. So, we also want to answer the question whether we can obtain optimal slicing structures *compatible* with a given point configuration as coordinates in a cartesian system.

Here, compatible means that we can draw line segments parallel to the axes that form a slicing structure with exactly one point in each elementary rectangle. These rectangles do not have to be feasible with respect to the shape constraint associated with the contained point. Of course many slicing structures can be drawn in such a way. Each such structure has an optimal dissection with respect to a given contour score, while allowing only

feasible dimensions for each elementary rectangle. We want among all those compatible slicing structures one that has the lowest score.

First we note that all slices in a compatible slicing structure correspond to *rectangular sets* in the point configuration. These are subsets of points that are enclosed by four lines parallel with the axes (see Figure 10). Of course, many different compatible slicing substructures are possible with such a rectangular set, but we are only interested in the ones that can be part of the optimal one, or even only in the the space they may take in the final optimal structure. In other words we want the shape constraint of such a rectangular set. If we would have the shape constraint of every possible compatible slicing structure of that set, the desired shape constraint of that set is the "minimum" of all these shape constraints. Certain shape constraints are totally dominated by others and have no effect on set's shape constraint. Some shape constraints determine part of the set's shape constraint. It requires a new operation on shape constraints: *taking the minimum of two shape constraints.*

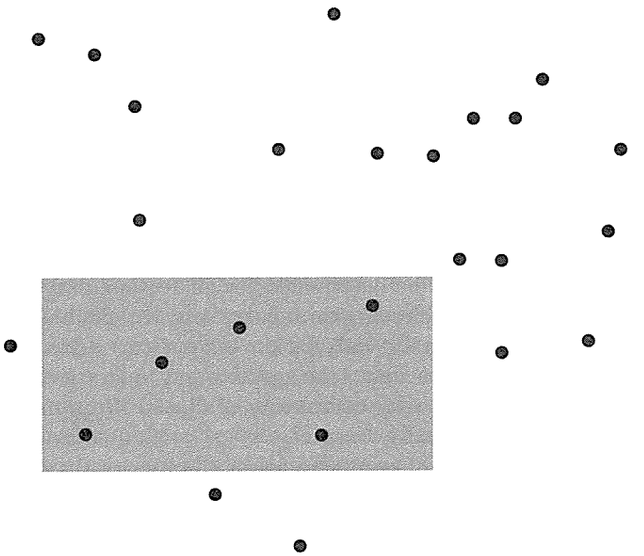


Fig. 10. Rectangular sets

The idea of dynamic programming suggests itself: in a systematic way we calculate the shape constraints of all rectangular sets in order of their cardinality and end up with the shape constraint of the total point configuration and identify an optimal solution under a given contour score in the same way as before. The number of rectangular sets in a point configuration is polynomial in the number of points, namely $\mathcal{O}(m^4)$. Candidates for having

their shape constraints determined consist only of two neighboring sets that have already shape constraints. To quickly retrieve existing shape constraints hashing of the sets is necessary to keep computational efforts low. The power set of points has namely exponentially many elements (2^m) of which only $\mathcal{O}(m^4)$ have to be addressed in worst case!

Again, the algorithm consists of two phases. During the first phase, the shape constraint for each candidate slice is computed, and stored in a global data structure for retrieval and the second phase. From the shape constraint of the entire point configuration, the optimal shape is chosen. The second phase then traces back the computations of the first phase that led to this shape. While it does that, it slices the point placement, and assigns dimensions to the slices.

For reasons of complexity, general piecewise constant functions cannot be used. If piecewise constant functions were used, the number of line sections could grow exponentially with the number of elementary rectangles in a slice. Only (small) integer values are permitted therefore as rectangle dimensions which means that the shape constraints are *integer stair-case functions*. Because all the discontinuities are then at integer coordinates, the discontinuities in different functions will often coincide at the same coordinate. Therefore, the number of discontinuities will not grow exponentially. Using integer stair-case functions, the number of sections is limited by the maximum dimensions of the slice. The shape constraints can be implemented as arrays of integers, with indexing by the argument of the shape constraint. For integer stair case functions, this is the most efficient implementation. Addition, minimization and inverting a shape constraint can all be implemented as simple "for" loops.

The complexity of the algorithm is polynomial, although a rather high polynomial. Shape functions have to be determined for all rectangular sets. There can be up to $m^2(m+1)^2/4$ different rectangular sets, a tight bound. Considering further that each constraint calculation takes $\mathcal{O}(m)$ additions and minimizations, which themselves may take up as many operations as there are breakpoints, we arrive at a time complexity of $\mathcal{O}(m^6)$ if the maximum dimensions are bound by a (small) constant.

The second stage of the algorithm has a much lower complexity then the first stage. It only recomputes the shape constraints of the slices that are actually used in the slicing structure. The complexity of the algorithm is determined by the complexity of the first stage.

Theorem 5. *Given a point configuration with coordinates in a cartesian system, the optimum compatible slicing structure under a given contour score can be found in polynomial time.*

Actually, only the sequences along the two axes are used! But that does not help to lower the worst case complexity which is high. Fortunately, in practice, the computation time evolves at a much lower rate, as can be seen in Figure 11.

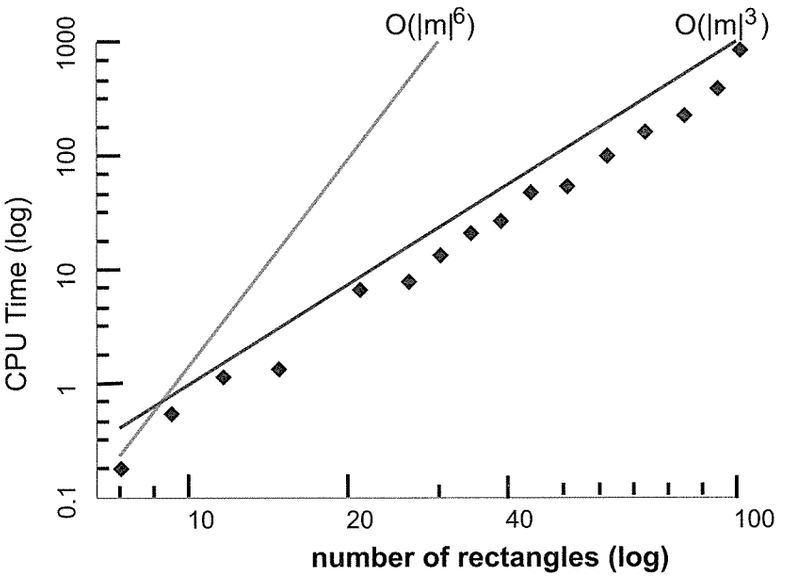


Fig. 11. The time complexity of the optimizations

4.3 Net Assignment

The more complex the circuit to be integrated, the more dominant the wiring is in the final layout, as can easily be learnt by examining existing integrated circuits. Today much of the wiring can be realized on top of active devices, particularly when there are many metal layers, Still considerable portion of the chip is used exclusively for the realization of the incidence structures of the modules. That part of the chip area is called the wiring space. If the cells are realized in rectangular regions, the wiring space can be seen as the union of nonoverlapping rectangles. The selection of the rectangles that together form the wiring space affects the efficiency of the wiring procedures. It determines the sequence in which the wiring can be generated, the algorithms to be used for this generation, and the number of different algorithms to perform that task.

Slicing structures have, again, considerable advantages over general rectangle dissections. Firstly, because they imply a decomposition of the wiring space into the minimum number of rectangles. These rectangles are in a one-to-one correspondence with the slicing lines. To distinguish these undivided rectangles from the ones that correspond with the cells in the functional hierarchy (in the slicing tree both kinds are represented by leaves), they are called *junction cells*. Secondly, feasible sequences for generating the wiring can be easily derived from the slicing tree. A possible rule here is to do the junction cells in a sequence based on the length of the path from the leaf

representing the junction cell to the root of the slicing tree, such that the longer this path, the earlier the wiring in that cell has to be generated. And thirdly, all these rectangles can be wired by using the same kind of algorithm, usually called a channel router, though not necessarily in the strictest sense [5,9]

Since the wiring can consume quite a high percentage of the total area, it would be useful to have early estimates for this space, so that the sequence of floorplan calls can take these estimates into account when designing the nested floorplans. Several objectives may be important in realizing the interconnections, and many of these are directly related to the size of these nets. This immediately raises two problems. The first one is a consequence of the interpretation of a floorplan as a topology rather than a geometrical configuration. Yet, in order to measure the size of a net, some metric is necessary. The second problem is the need for an ambience in which the wide diversity of objectives can be formulated and optimized.

An often used structure for approaching these problems is the plane graph determined by the rectangle boundaries in the rectangle dissection. Each rectangle corner is a vertex, and the line segments between them are the edges. This graph depends on the geometry of the rectangle dissection, and this geometry is not known in the floorplan design stage. A closely related graph can be defined for slicing structures. Then the vertices are the intersections between junction cells. Further, there is an edge between each pair of intersections that involves the same junction cell. This structure does not depend on the geometry. Reasonable estimates for the distances between the intersections can be obtained by deriving a preliminary geometry from the topology and data known about the environment. This can be done very fast for slicing structures as we saw in section 4.2.

If minimization of the total wire length is the important objective, a steiner tree on the above structure is wanted. Though slicing yields a considerable saving in computation time, this problem still requires exponential worst-case time (if $\mathcal{P} \neq \mathcal{NP}$). Therefore, a heuristic is required. This heuristic program has to assign each net to a number of segments of junction cells. After finishing this for all nets the densities in the junction cells can be determined, and on that basis a fairly accurate estimation of the wiring space is possible. In addition to obtaining accurate estimates for the shapes of the rectangles, the net assignment also yields information about the location of external nets for the floorplans and cells to be designed later.

4.4 Assembling Cells

The refinement steps described in the previous section determine a topology for every compound of the functional hierarchy. If these topologies are restricted to slicing structures, floorplan design replaces each subtree of which the vertices represent a certain module and all its submodules by another

tree of which the root represents the selected module, and the leaves represent its submodules. This is in accordance with one of the rules suggested at the end of section 2. The other type of refinement considered there was the replacement of a leaf in the functional hierarchy (a function cell) by a tree decomposition. The reasons for not having this decomposition in the initial tree can be quite diverse. For example, the decomposition suitable for the functional design may be far from optimal for layout design. In that case, such a hierarchy is pruned. Clearly, a data base problem has to be resolved when this happens. It might also happen that the decomposition is suitable for using in the layout program, but more specialized programs are needed than the general floorplanning scheme described. Most often, however, there is no need for further decomposition from the functional design point of view, but flexibility is increased if the layout design procedures use some inherent decomposition.

Algorithms for designing cells, possibly using such a decomposition, are called cell assemblers. The task of a cell assembler is to determine the internal layout of its cell (with respect to a reference point in that cell's region) on the basis of a suitable specification and data about its environment. There may be quite a diversity of cell assemblers in a silicon compiler system. The application range of the silicon compiler is highly dependent on the set of implemented cell assemblers. Whereas most of the decisions during floorplan design are to a high degree technology independent, cell design is dominated by the possibilities and limitations of the target technology. The numeric design rules are stored as numbers of which the value is assigned to certain variables in the cell assembler. The structural rules are to be incorporated in the algorithms, if possible in the form of case statements, so that a variety of rules can be satisfied.

The layout of a slice is obtained by first obtaining the layout of all its child slices except junction cells, and then calling the appropriate assemblers for the junction cells. Visiting the slicing tree in depth first order, and performing the above operations when returning to the parent slice enables the program to determine the chip's coordinates (coordinates with respect to a unique point on the chip) of all layout elements in the parent slice before leaving the corresponding vertex.

The translation of that result into the rectangles of the various masks is also performed at this point. This translation is straight forward. In the remaining part of this section some types of cell assemblers are described.

Function cells The task of a cell assembler is extremely simple for inset cells of which the internal layout is stored in a library. From the topology determined in the floorplan design process, and the shape constraints of all cells, including the junction cells, an estimate for the rectangle that is going to accommodate that module, can be derived. Reasonably accurate data about the position of the nets to be connected to that cell is generated by the net

assignment process. On the basis of that data the assembler has to decide which orientation has to be given to the inset cell, and how it has to be aligned with its sibling slices.

For function cells of which the internal layout is not stored in a library, it may still be implied by the specification. For example, if a cell is going to be realized as a programmed logic array, the specification is either a personality matrix, or a set of boolean expressions. In the former case the assembler does nothing else than performing a straight forward translation and handling the result in the same way as the stored inset cells. If the array is specified by a set of boolean expressions, the assembler must have a so-called pla-generator. The shape of the resulting array is still difficult to control, but the pin positions can be adapted to the results of the net assignment, performed during the floorplan design stage. Some sophisticated pla-generators use techniques such as row and column folding to make the area of the array smaller. This constrains the choice in pin positions considerably, and might lead to a higher area consumption, because of the complex wiring around that array. A pla-generator in a cell assembler should at least be able to take the results of the net assignment into account.

Regular arrays such as programmed logic arrays and memories, have an obvious decomposition into array cells with no or only slight variations in their dimensions. Their positions in the array are heavily constrained, and this decomposition can therefore not be used to manipulate the shape of the array. There also are cells that have a natural or given decomposition that can be used for that purpose. These cells are called macros. They are decomposed into circuits that either are selected from a pre-specified catalogue, or can be designed with a simple algorithm from a function specification. The reason for keeping such a macro from the floorplan design stage is that the circuits have certain properties that make special layouts very efficient. For example, the catalogue, or the simple algorithm, may have a constraint that gives all cells in the macro the same width and the same positions for general supply pins, such as power supply and clock pins. In that case a pluricell layout style is suitable for the macro. It forces the cells to be distributed over columns, but the number of columns can be chosen freely. Therefore, the aspect ratio of the macro can be influenced. Also the pin distribution around the periphery can be prescribed on the basis of data about the environment.

Decompositions like in macros occur very often, but for special cells that are frequently used, it sometimes is worthwhile to implement a special algorithm producing a highly optimized layout. In word-organized digital systems these special cells often process a number of bit vectors. The layout as a whole may benefit from aligning these cells so that the buses carrying these bit vectors do not have to be matched to the pitch of each individual cell. Also buses that pass over such a cell without making any contact have to be accommodated. These requirements imply a certain kind of flexibility, such as stretchability and variable bus pitches. If possible, such an algorithm must

be able to produce these highly optimized cells for several bus dimensions and a range of performance requirements.

Junction cells Junction cell assemblers are closely related to channel routers [5,9], because of the way they are isolated and the moments on which they are called. The junction cell is a rectangular area of which the latitudinal sides are parts of the longitudinal sides of junction cells that are represented in the slicing tree by vertices closer to the root. When the assembler is called for a certain junction cell, the longitudinal coordinates of the entry points of the nets are known. There are several ways a net may enter the junction cell: from the longitudinal sides of that cell, from a higher metal layer, from the latitudinal sides, and perhaps in still other ways. The task of the assembler is to realize all the required interconnections in a rectangle with an as small as possible latitudinal dimension. The longitudinal dimension has to be commensurate with the latitudinal dimension of the parent slice. Increasing the longitudinal dimension of the channel should therefore be avoided if possible.

5 Global Wires

5.1 Hierarchical Design

With some hierarchy maintained throughout layout stages the definition of what is a global wire seemed easy: any wire that connected different blocks in the actual hierarchy level was considered global. They were treated special in that global routing routines first assign them to restricted regions, and the result was used to further update the estimates concerning area usage, their consequences for shapes in the floorplan and congestion analysis. After detailed placement, the final outcome of global routing can also be used in the preparation for detailed routing.

With the number of wiring layers restricted to two to four a large part of the effective resources for interconnections was where there were no active devices. The assignment was often to "channels": areas between the blocks identifiable even in a floorplan. Although a constraint, it provided a convenient decomposition of the total wire problem into a sequence of channel routing problems (the best understood problem in layout synthesis) and, if not slicing, switch box problems. Nowadays, channels have lost most of their effectiveness due to the progress in technology dedicated wiring spaces are no longer necessary (which does not imply that the algorithmic techniques of channel routing cannot be used anymore!).

The longest wires in a hierarchical design are expected to be found among the so-called global wires at the higher levels of the hierarchy. If interconnect delay becomes the dominant bottleneck in achieving higher performance, we either have to avoid global wires altogether (that is to abandon the hierarchical design style as a constraint), or find ways to reduce that delay. There are several methods for reducing the delay:

repeaters: since the wire delay grows quadratically with wire length, repeaters splitting the wire do help as long as the gain by summing squares of shorter lengths is not absorbed by the additional delay for restoring by the repeater.

swing reduction: regenerative reaction to smaller voltage changes at the end of a line will speed up communication, but noise is limiting the maximum gain that can be obtained.

shape optimization: tapering wires improves wire delay (in theory even limitless), but creates unsolved layout problems when applied freely.

All these methods have their fundamental limitations, often reached well before maximum performance has been achieved. In those cases layout synthesis under the classical separation from functional synthesis, is powerless.

5.2 Interconnect Modelling

In recent years many sophisticated models for interconnect delay have been developed [19]. The complexity of these models and/or the size of the look-up tables used inhibits their use during synthesis, when the geometry of the interconnect is unknown, and when only estimates of length and topology are available. In these early stages only simple models such as Elmore's first moment matching can be used effectively. This model is the basis for analyzing almost all methods for reducing delay in point-to-point interconnection with unidirectional signal flow. The most common reduction method is to split the wire into segments buffered by inverters, and that will also be the choice in this section. What is the the optimum segmentation, and what is the optimum buffer? The answers have interesting implications, and most importantly, they will point us towards a decomposition independent global wire concept.

5.3 Critical Lengths and Critical Delay

We use a first order model for a generic restoring buffer (called a *repeater* although in current technologies it will be inverting) driving a capacitive load through a homogeneous line of length l given in Figure 12. No resistance after branching, no slope dependency, no transition differentiation, no holding and internal charging effects are assumed. Just a point-to-point connection and we are interested in questions such as optimum segmentation and buffering.

The repeater is represented as a voltage source controlled by the voltage v_{st} at the input capacitance. This voltage source switches instantaneously when the fraction denoted by x, $0 \leq x \leq 1$, of the total swing has been reached. The switching at the voltage source is a perfect voltage step (Figure 13).

The *parasitic capacitance* C_p is, in the case of static cmos circuits, mainly composed of the drain capacitance of the transistors. It complicates the

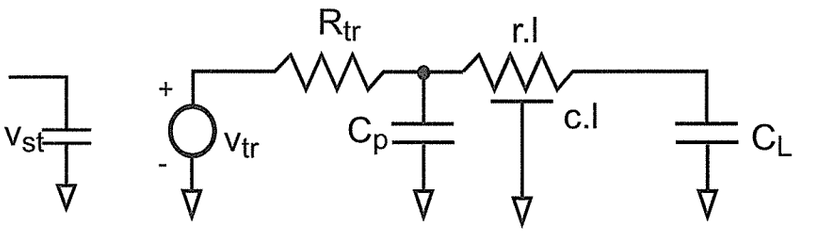


Fig. 12. Generic restoring buffer model

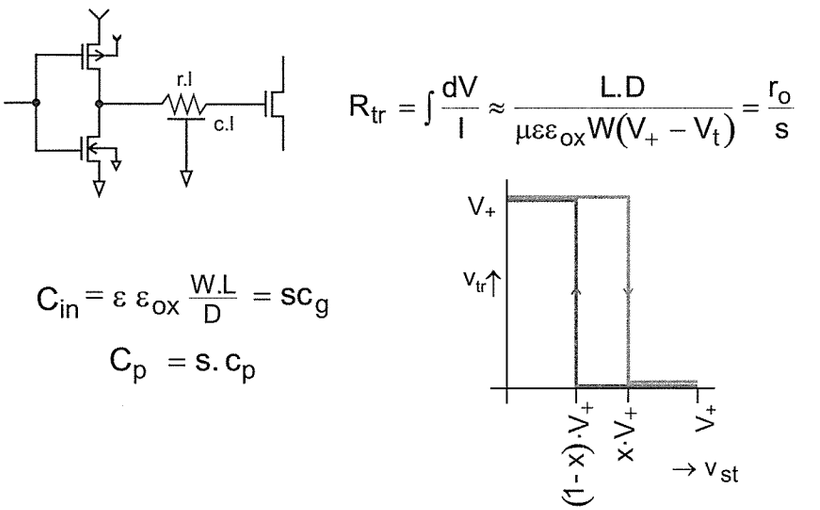


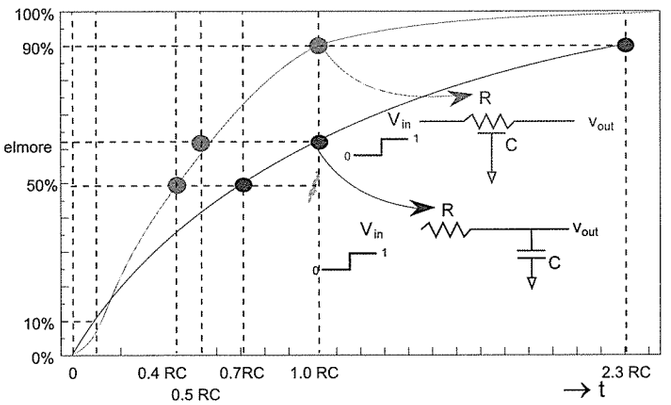
Fig. 13. Repeater model assuming inversion. s is a sizing factor

derivation a bit without affecting the conclusions. The numerical results, however, are considerably different when these parasitics are neglected.

The line is assumed uniform. To justify this assumption in practice special constraints have to be accepted in layout. We will address this point in section 5.4. Of course, more than one gate may require the same signal. The validity of the model is restricted to cases where the resistance after branching is negligible. That is, either the line has a unique receiver, or all receivers are so close to each other that representing them by a single lumped capacitor (C_L in Figure 12) is justified.

Starting from this simple model, a general formula for the delay between the switching of the buffer and completing the x fraction of the swing at the end of the line can be derived [21]:

$$\tau = b(x)R_{tr}(C_L + C_p) + b(x)(cR_{tr} + rC_L)l + a(x)rcl^2. \qquad (1)$$



	0 → 90% $\tau_{90\%}$	0 → 63% τ_E	0 → 50% $\tau_{50\%}$	τ
distributed line:	1.0 RC	0.5RC	0.4RC	**a RC**
single RC-section:	2.3 RC	1.0RC	0.7RC	**b RC**

Fig. 14. Model constants depending on swing

R_{tr} is the equivalent transistor resistance. The constants a and b depend on the switching model of the repeater, that is on x. In [21] several values for a and b are reported, and the table in Figure 14 gives some values (see also [3]).

If $x = 0.9$ (90%-swing) then $a = 1.0$, $b = 2.3$. The elmore delay with $x = 1 - 1/e \approx 0.63$ has the well-known result with $a = 0.5$ and $b = 1.0$. Mostly, in situations where circuits are chained and total delay of the chain is to be calculated, ($x = 0.5$) is used, yielding $a = 0.4$, $b = 0.7$. This is also our case, but rather simulation should be used to obtain values for a and b so that, when we divide the line in n equal parts by inverters, the delays of the sections can be added.

Dividing the line by inserting inverters may decrease the total delay, because the last term in equation 1 indicates a quadratic growth with length. A reduction in delay is possible if the gain is not offset by the delay of the inserted inverter. Obviously, there is an optimum segmentation of such a line by identical inverters. To formulate the optimization problem we give the size of the inverters in multiples (s) of the minimum size inverter. This makes $R_{tr} = r_o/s$, $C_L = s.c_o$ and $C_p = sc_p$. The initial driver of the line is assumed to have the same size, possibly after cascading up from smaller initial drivers for optimum speed (see Figure 15). The total delay for n such sections of length l/n is

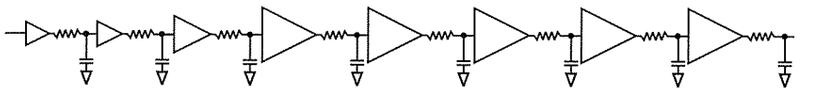


Fig. 15. The segmented line model

$$T = n.\tau = n\left[br_o(c_o + c_p) + b(c\frac{r_o}{s} + rc_o s)\frac{l}{n} + arc\frac{l^2}{n^2}\right] =$$

$$= br_o(c_o + c_p)n + b(c\frac{r_o}{s} + rc_o s)l + arc\frac{l^2}{n} \qquad (2)$$

Now we can ask for the values of s and n which give the minimum delay. For too small n the quadratic contribution of the line delay will dominate, while increasing the number of buffers will cause a large restoring delay. Obviously, T as a function of n has a minimum for positive n:

$$\frac{\partial T}{\partial n} = br_o(c_o + c_p) - arc\frac{l^2}{n^2} = 0$$

or the optimum length of each section is

$$l_{crit} = \frac{l}{n_{opt}} = \sqrt{\frac{br_o\,(c_o + c_p)}{arc}} = \frac{P}{\sqrt{rc}}. \qquad (3)$$

Accepting that $r_o c_o$ and $r_o c_p$ are process constants makes the optimum distance between inverters only dependent on the rc product per unit length of the wire.

Theorem 6. *The length of a section in an optimally segmented* [3] *line is inversely proportional to* $\sqrt{rc}$.

P depends on the process and the delay model (x) only. Since r and c differ from layer to layer, these distances also differ from layer to layer.

Substituting n_{opt} in (2) yields

$$T(n_{opt}) = \left(2\sqrt{abrcr_o\,(c_o + c_p)} + \frac{bcr_o}{s} + brc_o s\right) l$$

which shows

Theorem 7. *The delay of a line that is optimally segmented is linear in its length.*

The optimum repeater size is obtained as

$$\frac{\partial T}{\partial s} = b(rc_o - \frac{r_o c}{s^2})l = 0 \Rightarrow s_{opt} = \sqrt{\frac{r_o\,c}{c_o\,r}}$$

[3] We call it an optimally segmented line rather than an optimally buffered line because this length is independent of the buffer size s.

which is independent of n, the number of inverters used. By substituting the optimum repeater size and the optimum number of sections into (2) we find the delay of the line to be

$$T(l) = 2l\sqrt{rcr_o c_o}\left(b + \sqrt{ab\left(1+\frac{c_p}{c_o}\right)}\right)$$

which of course is also linear in l. More surprisingly, substituting the critical length shows that the delay of a section of critical length does not depend on the line resistance and capacitance:

$$\tau_{crit} = 2br_o c_o\left(1+\sqrt{\frac{b}{a}\left(1+\frac{c_p}{c_o}\right)}\right) = 2br_o c_o\left(1+\frac{P}{\sqrt{r_o c_o}}\right) \tag{4}$$

and therefore only depends on the process (and the model), but not on the wiring layer.

Theorem 8. *The delay of a section in an optimally buffered line is the same for all layers.*

Note that all derivations were made on a chain of inverters driving an uniform wire. Using this in more general networks, with different fanouts and branch-off geometries is therefore at best an approximation, which can only be made more accurate if isolation techniques are used to offset fanout effects.

5.4 Model Justification

Since we use the results only for point-to-point connections (that is without branching) between restoring circuits, first moment matching is accurate enough. However, some remarks concerning the model parameters are in order.

The via resistance shows up as a resistor in series with the R_{tr} in Figure 12. It is reasonable that this scales with the size of the inverter[4], and hence can be absorbed in r_o and the formulas do not change. Although $r_o c_o$ is no longer a process constant and a layer dependence is introduced in the critical delay, experiments show that the via resistance, even to the top layer, is negligible and has hardly any effect on the wave forms.

The line was assumed to have constant capacitance per unit length. For advanced technologies, this is dominated by capacitance to parts of other interconnections, especially neighboring sections in the same layer. Since these may undergo voltage changes, the value is not even constant. The latter effect may cause variations in the effective capacitance by up to a factor of 3.

[4] The contact resistance (the largest part) will scale if the contact area grows with buffer size, and also the cross section of the via is likely to scale then.

To make use of the derivations in the previous section, before the geometry of the wiring is known, requires the enforcement of a routing style which produces a time-invariant homogeneous line. (One (possibly drastic) way of achieving this is to shield each signal line with neighboring lines tied to fixed voltages. In addition to reliable characterization, this style eliminates most cross-coupling noise problems.) In addition, its resistance and capacitance should be known a priori.

The remaining problem with the model is the determination of the effective transistor resistance. It is reasonable that such a resistance exists if we only consider one waveform and a fixed x. The most practical way to obtain useful parameters is to simulate a ring of an odd number of buffer sections with large transistors (100 times minimum size) after extracting c very accurately. Then we optimize speed by varying the value of l for each section to obtain l_{crit}. This will give P of equation 3 since r is known quite accurately. With the length of each section fixed at l_{crit} the ring is optimized next for speed once more, now by varying s. This will yield τ_{crit} and by equation 4 therefore $r_o c_o$. Since we can accurately calculate c_o from the transistor geometry[5], we get

$$r_o = \frac{1}{4c_o}\left(\sqrt{P^2+\frac{2\tau_{crit}}{b}}-P\right)^2.$$

5.5 Numerical Data

To quantify what this all means we performed some calculations using a fictitious, but well reviewed technology file based on [20], and an extraction program [1] for solving exact 3-dimensional field problems. The critical lengths and the critical delay are given in Table 1[6]. Each layer has its own critical length, the higher layers having longer critical lengths than the lower ones. The values are pairwise close. Such a pair is called a *tier*. With a bit of process tuning the critical lengths within a tier can be made almost the same where the difference is mainly in the "between layers" capacitance.

Note that the critical length, measured in the feature size, changes much less than proportional with the feature size! This may come as a surprise, but is mainly due to velocity saturation effects, and therefore represents a trend that will affect smaller feature sizes even more. Other recent studies also indicate that even with scaling down in the logic blocks, the gate delay will continue to dominate the performance [6].

[5] Theoretically, we don't have to do this since we obtained s_{opt} in the second optimization and $c_o = r_o c/(r s_{opt}^2)$. However, since τ_{crit} is likely to be insensitive to s at the optimum, the value of s_{opt} is probably not very accurate, even though τ_{crit} is accurate.

[6] Table 1 is by Amit Mehrotra of UC Berkeley with independent corroboration by studies by Lixin Su, Sunil Khatri, and Dennis Sylvester.

critical parameter	feature size 0.25 μ	0.10 μ
$l_{crit}(m1)$	10440	6757
$l_{crit}(m2)$	10600	7162
$l_{crit}(m3)$	36000	43446
$l_{crit}(m4)$	38400	45135
$l_{crit}(m5)$	63200	64932
$l_{crit}(m6)$	62000	56892
$l_{crit}(m7)$		97581
$l_{crit}(m8)$		93378
$T(l_{crit})$	205ps	80ps

Table 1. Critical wire lengths measured in feature size units

Today's synthesis is capable of handling blocks with up to 10000 gates. A square with side lengths of $l_{crit}(m1 - m2)$ can contain on the order of a hundred of these blocks in a 0.1μ technology. So, even with careful extrapolation, this means that fairly complex blocks can still be designed while mainly controlling gate delay.

6 Wire Planning

The term *wire planning* was coined more than ten years ago "to describe an approach that first focuses on determining an optimal plan for global wiring" [4]. In its original context its task was mainly to identify groups of nets each connecting to (almost) the same set of modules. The most common example are *buses*: when identified routing complexity can be reduced considerably by handling bus wires as groups inside *data path generators*.

Another, new task for wire planning is indicated by the computations and derivations of Section 5. Although complex modules can still be designed using present day logic synthesis methodologies, controlling mainly the gate delays to achieve performance, at the chip level, future technologies will involve many (hundreds to thousands) of such complex modules, and at this level wire delay begins to dominate. Wire planning should produce a location for these modules with the main concern that timing constraints on input/output paths are met. This requires knowledge of the global interconnection structure and the performance implications of the functionality of the modules. In the early "conceptual" stages of a design there is not much more than an awareness of size-speed trade-offs. This accuracy depends completely on the design experience in the team. In general, all that can be said is that these interpolated delay-area relations appear convex. Delay in synchronous systems is often measured in terms of the number of clock cycles to complete the computation of the module. Obviously, a module that takes more cycles to do its computation never requires more area than one that takes

fewer cycles. Although defining speed of modules in general is not possible, the reasoning will always be similar, whether speed in a given technology is obtained by sizing, parallelism, or other means.

Knowing module functionality and interconnection structures implies a decomposition. Initially, such decompositions emerge solely on the basis of functional considerations, with little regard for their impact on both the performance of the product as well as the efficiency of synthesis steps later on. Therefore, while the design evolves into a hierarchical description acceptable for behavioral synthesis, wire planning tools should aid in quick analyses and proposals for function duplication, absorption and decomposition as well as module (re)locations and (partial) pad planning.

Examples of wire planning tasks are establishing the existence of a module placement in which no path from input to output has to make detours, assigning time budgets to modules such that area is minimized, establishing the existence of a valid retiming and producing a valid minimal-area retiming, assigning wire sections to layers so that feasible time budgets are preserved, and encouraging floorplans that lead to efficient optimizations in later stages of the design.

The final result of conceptual design aided by wire planning is a composition of a network of blocks and interconnections along with well established time budgets and delays. Considering the data concerning critical lengths, blocks will be small compared to these critical "units". They can be treated without internal distributed delays, and their wiring is mostly realized in the lower levels of metal. The tools can aid in creating subsets of regular grids with blocks at grid points and predefined wire segments on the grid lines. The latter enables good characterization of these segments, and routing consists of "using the available segment" rather than "placing segments". By the time that synthesis begins to create the gate and net lists, the delay on the "global" wires is quite well established and therefore also the timing budget that remains for the blocks.

6.1 Monotonic Wire Plans

Consider a high level description of a design described as a *functional network* modeled by as a directed acyclic graphs with primary inputs as sources, primary outputs as sinks and "functions" on the other nodes. There is an arc from one node to another if the "result" of the former is used as an argument in the latter. If a primary output depends on a primary input, there must be a path connecting them, possibly passing through other blocks, and possibly sharing some with other paths. Total delay is the sum of the delay in the blocks and the delay in the wires. If wires are composed entirely out of sections with critical delay, the total wire delay on a path is a multiple of the critical delay, and is invariant with respect to how the functional units are distributed over the restoring sites (*end points of critical sections*). If a functional block is placed at each "grid point" along a path then no repeaters

are necessary. A *wire plan* in this context is a position for all the nodes in the functional network and a pin assignment for all primary inputs and outputs. Such a wire plan is called *monotonic* if all interconnections can be made so that the $\mathcal{L}_1$-length ("Manhattan length") of each input/output path is equal to the $\mathcal{L}_1$-distance ("Manhattan distance") between the two associated i/o pins. Under the model this is the fastest possible wire plan for a functional network with that pin assignment[7] having its wires in a given tier.

For a given pin assignment a monotonic wire plan may not exist. This existence question has been answered in [7] as follows. The *support* of a node is the set of primary inputs connected to that node by a directed path. The *range* of a node is the set of primary outputs connected to it by a directed path The *inbox* of a node is the smallest iso-rectangle containing its support, and the *outbox* is the smallest iso-rectangle containing its range. A *bridge* of a node is a minimum $\mathcal{L}_2$-length line connecting its inbox with its outbox. Using these ideas and and working out a few special cases leads to Theorem 9:

Theorem 9. *Every node in a monotonic wire plan must be placed within the smallest iso-rectangle containing its bridge.*

A simple proof by induction then yields:

Theorem 10. *A functional network has a monotonic wire plan with respect to a given pin assignment if and only if every node has a unique bridge.*

This makes it very easy to find out whether such a wire plan exists: we only have to check on a node by node basis whether each node in the network has a unique bridge. Such a check is extremely simple since

Theorem 11. *A node has a unique bridge if*

1. *the support or the range contains a single pin, or*
2. *the range is contained in an iso-line while the support is on a single line perpendicular to that, or*
3. *the output box is in the "projection" of the input box, that is the two boxes have disjoint support in both axes, except for at most one point.*

Note that a placement conformant to Theorem 9 is not necessarily a monotonic wire plan. A valid placement, but possibly having nodes at the same position, is assigning each node the point which the output box has in common with the bridge[8].

Of course, certain deviations from strict monotonicity may be necessary or desirable, because of availability of space or for sharing functionality with

[7] Under a model where interconnections have capacitance but negligible resistance, a monotonic wireplan has the minimum total wire capacitance. This can be useful when power is a major concern and may be relevant for logic synthesis when a pin assignment is given [7].

[8] Also the points that input boxes have in common with the bridge is a feasible set.

other paths. However, deviation from monotonicity can only be allowed if the timing requirements are not violated. Note that monotonicity can always be obtained by duplicating functionality, synthesizing faster blocks, and absorbing functions in their fanout. In the extreme, a monotonic wire plan always exists if each output is produced by a single node.

Once the wireplan for a functional network has been determined, which means that the delay on the arcs of this network is known, the remaining time budgets have to be distributed over the function nodes. If the same graph is a suitable model for this task, and the sources and sinks have arrival times and required times assigned to them, a simple (quasi-)convex optimization problem can be used to answer questions such as "what is the smallest network that does not violate any timing constraints?" Size is in this case the sum of the areas assigned to each node according to its area-delay trade-off.

6.2 Valid retiming

Wires with a delay and synchronisation at the end of the line are functionally equivalent with a series of latches in number equal to the ceiling of the delay divided by the clock period. It would be advantageous if a wire plan is such that a synchronous equivalent design exists with that many (or more) latches at the interconnections. The wire plan is said to have a *valid retiming* in that case. Since a wire plan is only a point placement, the delay over a connection is unknown until it has a layer assigned to it, and its geometry is determined. A lower bound for the delay follows from assuming that the fastest layer (usually the highest tier) is used and a detour free geometry is realised. Let the ceiling of the quotient between that lower bound and the clock period be denoted k_{ij} for the interconnection from module i and module j. A more formal characterization reads then:

A retiming r is *valid* when

$$\forall_{(i,j)\in E}\left[w_{ij}^r \geq k_{ij}\right]$$

where E is the set of connections in the wire plan, and $w_i^r j$ represents the number of latches at that connections after retiming r.

A given plan may or may not possess a valid retiming, but if it does it probably has many different valid retimings. Among those, the ones with smaller area may be preferred. The more cycles are "retimed" into a functional node the smaller the area required by the module represented by that node. The problem can be formulated by modifying the network in the following way: duplicate each node while assigning all of its inputs to one node and all of its outputs to the other node; add an arc from the node with the inputs to the node with the outputs. For the new arcs the value of k is unbounded ($k = \infty$), but it has a function $\alpha : \mathbb{Z}_o^+ \rightarrow \mathbb{R}^+$ associated with it. It is the area-delay trade-off curve that maps each number of cycles on the area of the node. The optimization problem is

minimize

$$\sum_{a \in A} \alpha_a(w_a^r)$$

subject to

$$\forall_{(i,j) \in E} \left[w_{ij}^r \geq k_{ij} \right]$$

where A is the set of new arcs, one for each functional node.

An efficient solution to an approximation of this problem is in [25]. The area-delay trade-off curve α is there not only defined for all non-negative integers, but for all non-negative real numbers. It is a piece-wise linear functions where the slope of the pieces may not increase with the delay: that is the trade-off curve must be convex. They observed that the problem is very similar to the classical minimum area retiming problem, only the optimization criterion now is really area, and not simply the number of registers, and the cost-contributions are from convex area-delay trade-offs, and not constants. Combinatorial delay is neglected.

What is missing in the formulations is how the assignment of wires to layers plays a role. In the above formulation, only the top level wire type is considered. The fact that a wire can be placed on a lower level of wire and still meet its timing obligation is not considered. A possible answer is to modify the total area cost function, to penalize wires that are put on higher layers.

6.3 Layer assignment

The purpose of *layer assignment* is to assign every wire or wire segment in a wire plan to a given layer, or rather to perform a quick analysis whether a layer assignment for the given wire plan is likely to exist. Each layer is only distinguished by its critical length. Several layers may have critical lengths that do not differ significantly. The assignment of wires is only to m "classes" of layers, where layers in the same class have the same critical length. These are sorted in ascending order and denoted as,

$$l_1 < l_2 < \cdots < l_m$$

The classes are dictated by technology. The layout style must be such that any pair of points of the chip can be connected by wire segments from the same class.

A wire segment between (x_i, y_i) and (x_j, y_j) that is assigned to class k adds a $C_k((x_i, y_i), (x_j, y_j))$ to the *cost* of the assignment. The contribution of a segment depends on the class k as well as on the position of the segment on the chip. It is likely to be more costly if its class is on a higher layer (a scarcer resource) or if its rectangle is nearer the center of the chip (an area more apt to congestion). The total cost of the assignment is to be kept as low as possible:

minimize

$$\sum_{wires} C_k((x_i, y_i), (x_j, y_j))$$

Now given a register transfer level description of the design with timing information for the interconnections and a wire plan. The assignment is determined in two stages. First each connection $i \rightarrow j$ gets a delay number δ_{ij}, in such a way that all delays from chip input or latch output to chip output or latch input can be made in the assigned number of clock periods for that connection. If a path is allowed a delay longer than a single clock period then it is divided into an appropriate number of pseudo-nodes. Thus, every combinational path must be within a single clock period. Let π be such a path. A fraction q of the clock period is assigned to this path, reserving the other part for the gates on the path. So

$$\sum_{ij \in \pi} \delta_{ij} = q < 1$$

where δ_{ij} will be the delay assigned to connection $i \rightarrow j$. The *zero-slack distribution algorithm* [14] is used to assign the delay numbers such that all paths satisfy the above equations. Now let the Manhattan length of wire $i \rightarrow j$ in the wire plan be denoted by d_{ij}, then connection $i \rightarrow j$ is assigned to the k^{th} class fif

$$\frac{d_{ij}}{l_k} \leq \delta_{ij} \leq \frac{d_{ij}}{l_{k+1}}$$

Thus wire $i \rightarrow j$ will be assigned to a class where the delay on the wire, $\frac{d_{ij}}{l_k} \leq \delta_{ij}$. Therefore for any path π we have

$$\sum_{ij \in \pi} \frac{d_{ij}}{l_k} \leq q$$

If a solution exists, then all wire segments can be assigned a fixed wire delay, δ_{ij}. Note that for the number of available wires of class k are only indirected accounted for by assigning a wire to the least level which gives the required delay. This assumes that wires at the lower levels are the more plentiful. The cost of a wiring rectangle is weighted by its relative overlap with the center of the chip, but the number of wires in the wiring rectangle is not accounted for in this formulation.

A solution may not exist if there is no placement where all the delays can be met. In that case, we may have to return to higher levels wire planning and even possibly alter the chip latency.

7 Gate Sizing

To complement an approach based on wire planning, layout synthesis should realize the functional blocks in such a way that the delays in the blocks do not exceed their timing budgets, or rather keep them right on target. Since logic

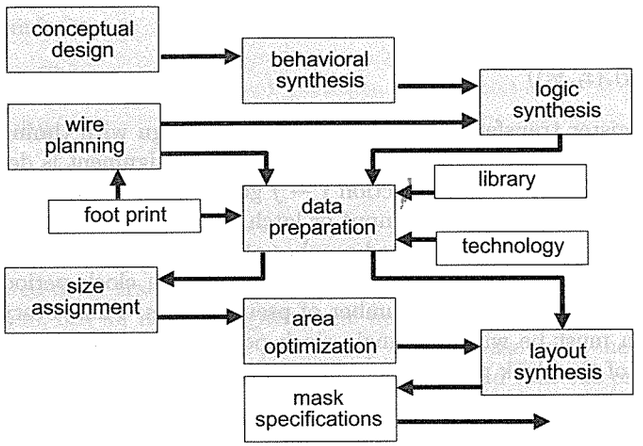


Fig. 16. Constant delay flow

synthesis knows the budgets after wire planning and the range of available gates, it should deliver a gate list with an assigned specified delay for every gate. Layout synthesis should produce a network in which each gate causes exactly that delay. This is called *constant delay synthesis* [8]. Given a fixed delay for a gate, its size becomes a function of the output capacitance.

This of course is not without consequences for the back-end tools. Sizes now have to be assigned according to the results of logic synthesis, and scale only with the imposed capacitances on the outside. Timing optimization is out of the question: buffer insertion can only serve as an area reduction trick. New cell libraries have to be developed to adequately reflect the demands of delay-based requirements. And finally, the layout generation must be capable of handling a variety of cell sizes, and absorb changes in sizes efficiently.

7.1 Sutherland Delay

Again starting from the model of Figure 12, but not including the wire, leads to the configuration of Figure 17 and a delay formula which is the sum of two terms, the *effort delay* and the *parasitic delay* [23,24]:

$$\tau = bR_{tr}C_L + bR_{tr}C_p = br_o c_o \frac{C_L}{C_{in}} + br_o c_p = \frac{g}{f} + p. \qquad (5)$$

The right-hand expression is called *sutherland delay*. The parasitic delay $p = br_o c_p$ is independent of size. The effort delay g/f is a product of *computing effort* $g = br_o c_o$, and *restoring effort*

$\frac{1}{f} = \frac{C_L}{C_{in}}$

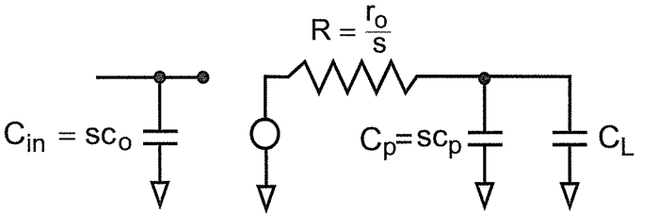


Fig. 17. Gate model for obtaining a size independent delay expression

The computing effort is also size independent, but in general depends on the function, topology and relative transistor dimensioning of the gate type. The important observation is that τ can be kept constant by fixing $f = C_{in}/C_L$. This leads to a new paradigm in synthesis [8,18]: any delay imposed by synthesis can be realized, provided that the sizes of the gates can be continuously adjusted, and the imposed delay exceeds the parasitic delay. Note however that the derivation replaced the gate by a single linear "effective" resistance and a linear input and drain capacitance.

For more general charge or discharge networks of mos-transistors a single sizing factor can be derived [8]:

Theorem 12. *If*

1. *each transistor can be modelled by an effective resistance inversely proportional to the device-width,*
2. *each node i in the (dis)charge network can be modeled by a linear capacitance C_i composed of a constant part and device dependent parts, and*
3. *the gate delay can be approximated by a summation over all nodes of $R_i C_i$ where R_i represents the total resistance between node i and the output node [28],*

then the delay of the gate remains constant if all device widths scale linearly with with the load capacitance C_L

The third condition reflects the elmore intuition of approximating delay by adding the time constants of single rc-sections, each consisting of a node capacitance and the total resistance through which it is charged or discharged. Of course, the delay is an approximation for an already idealized network (linear "effective" components, lumped capacitances, etc), but experiments[9] support the stated fact extremely well [8]:

[9] The same experiments also supported the validity of the sutherland model by showing the invariance of the socalled *self loading*: $\frac{c_p}{c_p + c_L}$.

Theorem 13. *The delay of a logic gate can be kept constant by scaling the devices linearly with the (external) load.*

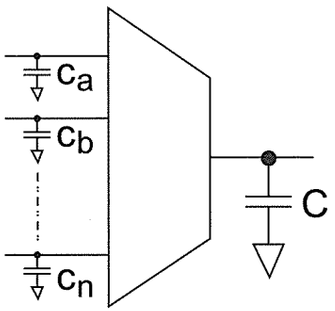


Fig. 18. Fixed relation of each input c with output c

The derivation of theorem 12 also implies that under constant delay all input capacitances of a gate scale linearly with the load. This leads to an analog "*restoring effort*" as in the sutherland delay for a buffer, that is, a single factor f. So, with reference to Figure 18, for every input x we have $c_x \propto fC$ where the proportionality constant can be different for different inputs of the gate (but these constants do not change with the restoring effort, and therefore not with f!). It is reasonable to assume that the size of a gate is proportional to the sum of its transistors, and therefore proportional to the sum of the input capacitances:

$$\text{gatesize} \propto \textstyle\sum_{x\in\{a,b\cdots n\}} c_x \ .$$

Since each c_x is proportional with fC, the size of a gate is proportional to fC as well, and this proportionality constant is called the *area sensitivity* of that gate type: afC. The area sensitivity of the gate equals the gate area if the restoring effort is 1 and at the output there is a unit capacitive load[10].

Corollary 3. *The size of a gate is proportional to the capacitance at the input which under the constant delay paradigm equals the capacitance at the output multiplied by f as imposed by synthesis.*

In the context of a network the implications of corollary 3 can be worked out by writing the expression for the total capacitance at a single node (see

[10] That there is a single factor f for each gate means that by relative dimensioning of the pull-up and pull-down networks, the transfer behavior can be manipulated and constant delay synthesis will not change that. However, if each input of a gate requires a separate relation to the output capacitance, it would not complicate the formulation

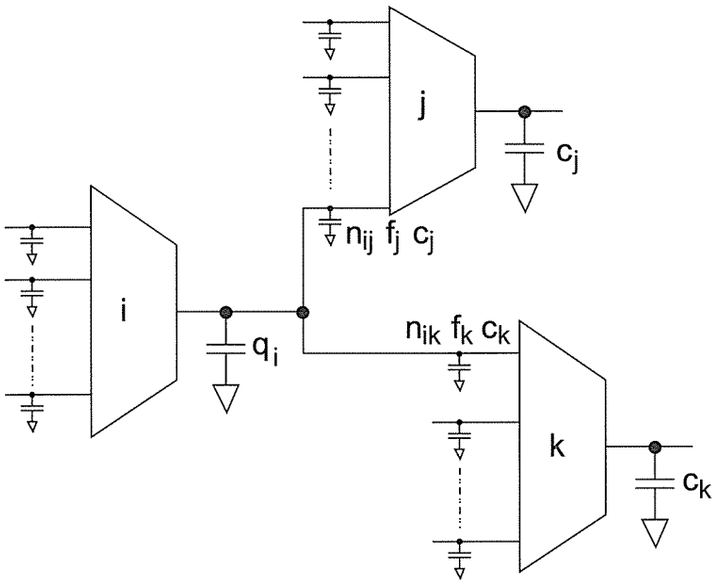


Fig. 19. Effort relations in a network

Figure 19) and then collect them in vector form as follows. The capacitance at node i is the imposed capacitance q_i at that node (this can be the external capacitance to be driven by the network or the wiring capacitance, but lumped and without wire resistance) and the (scaled) input capacitances in its fanout:

$$c_i = q_i + \textstyle\sum_{j\in fanout(i)} n_{ij} f_j \, c_j .$$

In Figure 19 the summation for node i contains two terms: $c_i = q_i + n_{ij} f_j \, c_j + n_{ik} f_k \, c_k$. If we make $n_{ij} = 0$ whenever gate j is not in the fanout of gate i, and equal to the proportionality constant that comes with the gate type of gate j (accounting for function, topology, and relative sizing) we can write in general

$$c_i = q_i + \textstyle\sum_{j=1}^{n} n_{ij} f_j \, c_j$$

Collecting the imposed capacitances in a vector $\mathbf{q}$, the capacitances at the output in a vector $\mathbf{c}$ and the reciprocals of restoring effort in a diagonal matrix $\mathbf{f^D}$, yields the following relation:

$$\mathbf{c} = \mathbf{q} + \mathbf{N}\,\mathbf{f^D}\mathbf{c}$$

or

$$\left(\mathbf{I} - \mathbf{N}\mathbf{f^D}\right)\mathbf{c} = \mathbf{q} \qquad (6)$$

The matrix $\mathbf{N}$ has the zero/non-zero pattern of the incidence matrix of the (directed) network, and contain the relative sizing of the transistors in the values of the non-zeros. Both, the pattern of $\mathbf{N}$ and $\mathbf{f}$ are imposed by logic synthesis, and the entries come from the library. They should be such that equation 6 has a positive real solution.

Corollary 4. *The node capacitances are related to the imposed capacitances by a linear transformation.*

Once all node-capacitances are known, that is the vector $\mathbf{c}$, the area of the network can be written as

$$(\mathbf{a}\cdot\mathbf{f})^{\mathbf{T}}\mathbf{c} = (\mathbf{a}\cdot\mathbf{f})^{\mathbf{T}}\left(\mathbf{I}-\mathbf{N}\mathbf{f}^{\mathbf{D}}\right)^{-1}\mathbf{q}$$

where the dot indicates componentwise multiplication. Based on this relation one can determine whether inserting buffers can save area.[11] Note however that although logic synthesis is to produce a network of gates with restoring effort assigned to each gate, the area of the gates and the total network is not known, because the capacitances at the nodes are not known.

Solving the size equation Synthesis has to come up with the vector $\mathbf{f}$. This is of course impossible if the parasitic delay on a path exceeds already the timing requirement. Moreover, it is not unlikely that gates are only available in discrete sizes, while the theory assumes continuous sizability. The selection of available sizes [2], algorithms that use these limited libraries [12], and the discretization problem in synthesis [10] deserve careful analysis, but the problems seem to be certainly surmountable. That is, if a solution exists under the assumption of continuous sizability, these studies and experience up to now show that effective solutions exist for reasonably rich discrete libraries, or adequate sets of cell generators. For establishing existence conditions for continuous solutions, we look at three different cases: acyclic networks, strongly connected networks and general networks.

In case of *acyclic networks*, that is no memory elements and no cycles, etc., the network can be represented by an acyclic directed graph. Consequently, there exists a topological ordering of the nodes. Using that to order the equations in the set 6 yields a lower triangular matrix that can be solved by backsubstitution.

If the network is strongly connected, the matrix $\mathbf{N}$ is irreducible[12], and therefore $\mathbf{N}\,\mathbf{f}^{\mathbf{D}}$ as well. The question whether the equation system 6 has positive solutions ($\mathbf{c}>\mathbf{0}$) for any $\mathbf{q}>\mathbf{0}$ has been studied extensively. It is known as an *open leontief system*. The main result from this setting is

[11] Buffers can only be inserted if they do not cause violations in the timing constraints; buffers are allowed to introduce additional delay locally only if some slack time is available there.

[12] The usual definitions of irreducibility using powers of matrices or matrix decomposition are equivalent, but in the present context indirect and not so useful.

Theorem 14. *The equations*

$$\left(\mathbf{I}-\mathbf{N}\mathbf{f}^{\mathbf{D}}\right)\mathbf{c}=\mathbf{q}$$

with $\mathbf{N}$ *an nonnegative irreducible matrix and* $\mathbf{f}$ *a strictly positive vector, has a solution* $\mathbf{c}$, $\mathbf{c}\geq\mathbf{0}$, $\mathbf{c}\not\equiv\mathbf{0}$ *for any* $\mathbf{q}\geq\mathbf{0}$, $\mathbf{q}\not\equiv\mathbf{0}$ *fif* λ, *the perron-frobenius root (that is the dominant eigenvalue) of* $\mathbf{N}\,\mathbf{f}^{\mathbf{D}}$, *is smaller than* 1.
In that case, there is only one solution $\mathbf{c}$, *which is strictly positive and given by*

$$\mathbf{c}=\left(\mathbf{I}-\mathbf{N}\mathbf{f}^{\mathbf{D}}\right)^{-1}\mathbf{q}.$$

A number of useful corollaries easily follow from that result. We mention

Corollary 5. *If it exists,* $\left(\mathbf{I}-\mathbf{N}\mathbf{f}^{\mathbf{D}}\right)^{-1}>0$ *fif* $\lambda<1$.

Corollary 6. *If none of the row sums of* $\mathbf{N}\mathbf{f}^{\mathbf{D}}$ *exceeds unity, and at least one is less than unity, then* $\lambda<1$.

A condition equivalent to $\lambda<1$ and avoiding the solving of eigenproblems is from the following theorem[13]

Theorem 15. $\lambda<1$ *fif all leading principal minors of* $\left(\mathbf{I}-\mathbf{N}\mathbf{f}^{\mathbf{D}}\right)^{-1}$ *are positive, where the i-th leading principal minor is calculated from the first i rows and columns of* $\left(\mathbf{I}-\mathbf{N}\mathbf{f}^{\mathbf{D}}\right)^{-1}$.

Under the stronger (than $\lambda<1$) condition of corollary 6 one can derive that when increasing the imposed capacitance at one node, the gate driving that node gets the greatest absolute increase in input capacitance. This does not imply the greatest increase in area, because that also depends on the area sensitivities. About relative changes something can be said under the minimal conditions of theorem 14:

Theorem 16. *If the components of the vector* $\mathbf{q}$, $\mathbf{q}\geq\mathbf{0}$ *and* $\mathbf{q}\not\equiv\mathbf{0}$ *change by amounts of* $\Delta\mathbf{q}$ *such that* $\mathbf{q}+\Delta\mathbf{q}\geq\mathbf{0}$ *and* $\mathbf{q}+\Delta\mathbf{q}\not\equiv\mathbf{0}$, *while* $\lambda<1$, *then for each i*

$$\min\left\{(0,\min_{\{j|\Delta q_j<0\}}\tfrac{\Delta c_j}{c_j}\right\}\leq\tfrac{\Delta c_i}{c_i}\leq\max\left\{0,\max_{\{j|\Delta q_j>0\}}\tfrac{\Delta c_j}{c_j}\right\}.$$

This implies that if only the imposed capacitance at a particular node changes while all other imposed capacitances remain the same, the gate driving that node changes by the greatest percentage (both, its input capacitance and its size, regardless of its area sensitivity).

Finally, if the network is not strongly connected, and consequently $\mathbf{N}\mathbf{f}^{\mathbf{D}}$ is reducible, we have to resort to weaker results of the perron-frobenius theory.

[13] The condition of this theorem is known as the hawkins-simon condition

The strict positivity in corollary 5 has to be replaced by non-negativity, that is

$$\lambda > 1 \Longrightarrow \left(\mathbf{I} - \mathbf{N}\mathbf{f}^{\mathbf{D}}\right)^{-1} \geq 0,$$

but theorem 15 is still valid, also for networks that are not strongly connected. Non-negative solutions for the components of vector $\mathbf{c}$ are therefore still ensured, but some may be equal to 0.

Numerical solution of the size equations An iteration equation for solving the equation set 6 presents itself in a natural way:

$$\mathbf{c}(k+1) = \mathbf{N}\,\mathbf{f}^{\mathbf{D}}\mathbf{c}(k) + \mathbf{q}.$$

In fact, it is the well-known *jacobi* method and $\mathbf{N}\mathbf{f}^{\mathbf{D}}$ is the socalled *jacobi matrix* of the system. However, it is just one out of many ways of *splitting* the coefficient matrix for iteratively solving the equations. Another familiar splitting leads to the *gauss-seidel* iteration:

$$\mathbf{c}(k+1) = (\mathbf{I} - \mathbf{L})^{-1}\mathbf{U}\,\mathbf{c}(k) + (\mathbf{I} - \mathbf{L})^{-1}\mathbf{q}.$$

where $\mathbf{L}$ is the strictly lower triangular part of $\mathbf{N}\mathbf{f}^{\mathbf{D}}$ and $\mathbf{U}$ the strictly lower triangular part[14]. The inverse obviously exists (and is actually equal to $\sum_{i=1}^{n-1}\mathbf{L}^i$ with n the number of gates in the network). Let us denote the dominant (non-negative) eigenvalue of the *gauss-seidel* matrix $(\mathbf{I} - \mathbf{L})^{-1}\mathbf{U}$ with λ_{GS} (the gauss-seidel matrix is reducible). The well-known stein-rosenberg theorem then implies

Theorem 17. *If $\lambda < 1$ then $0 < \lambda_{GS} < \lambda$*

Corollary 7. *Both the jacobi and the gauss-seidel method do converge when $\lambda < 1$, and the latter converges asymptotically more quickly.*

Other iteration schemes are possible, and may be faster. *Successive over-relaxation* is such a candidate. But to achieve this computational advantage in convergence, more knowledge about the eigensolutions, and the dominant eigenvalues is needed. To do the analysis, or calculating useful bounds might not pay of, and therefore the preferred approach is *gauss-seidel iteration.*

Area recovery Inserting buffers might decrease the total area of a module. However, time critical paths should not get buffers inserted, because they cause additional delay. Only when there is a certain amount of slack, buffers can be inserted if they provide a decrease in area. We will show that all potential insertion points can be determined at synthesis time, that is before the node capacitances, and thus the gate sizes are known.

[14] $\mathbf{N}$ has only zeros on the diagonal because the input and output of driving gates are never directly connected (except for generating the inversion voltage as for fast sensing, but that is outside the present application).

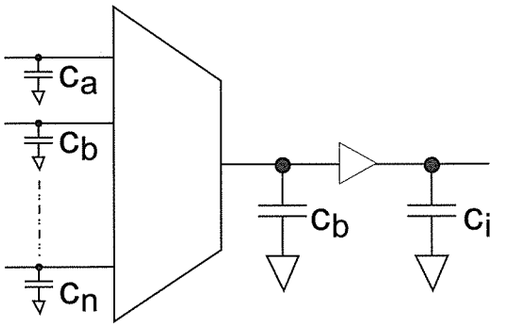


Fig. 20. Inserting a buffer

The delay of a buffer is given by equation 5, that is $\tau_s = g_b/f_s + p_b$ with g_b and p_b as library constants. Area decreases by the insertion only when

$$a_i f_i C_i > a_i f_i c_b + a_b f_s C_i \quad \text{or} \quad a_i f_i C_i > a_i f_i \tfrac{c_b}{C_i} + a_b f_s.$$

Note however that $\frac{c_b}{C_i}$ is precisely equal to f_s, so that

$$a_i f_i > (a_i f_i + a_b) f_s$$

The condition for area recovery is therefore

$$\tfrac{1}{f_s} > 1 + \tfrac{a_b}{a_i f_i}$$

Substituting this in the delay equation of the buffer shows that the added delay by inserting a buffer is at least

$$p_b + g_b\left(1 + \tfrac{a_b}{a_i f_i}\right)$$

and this has to be compared with the available slack at that point. Clearly, all variables are either chosen by synthesis or library constants, and consequently, the comparison can be performed before sizing.

Theorem 18. *A network with restoring efforts assigned in order to meet the timing requirements, can be reduced in size by buffer insertion fif there is a gate in the network with area sensitivity a_i and restoring effort f_i^{-1} such that*

$$p_b + g_b\left(1 + \frac{a_b}{a_i f_i}\right) \tag{7}$$

where g_b, p_b, and a_b are the computing effort, the parasitic delay and the area sensitivity of the buffer to be inserted.

This means that the locations for potential area recovery can be determined at synthesis time. For it is synthesis that creates the network (that is

the matrix $\mathbf{N}$), selects the gates and assigns restoring effort to them. However, inserting a buffer at a certain node changes the slacks on all paths containing that node. Each insertion can invalidate many other potential area recovery points. And since sizing still has to take place, it is not known at synthesis time how much is gained by inserting a buffer. A generic procedure should therefore have synthesis insert buffers at all the candidate points, and minimize

$$(\mathbf{a} \cdot \mathbf{f})^{\mathbf{T}} \left(\mathbf{I} - \mathbf{N}\mathbf{f}^{\mathbf{D}}\right)^{-1} \mathbf{q}$$

without violating the timing requirements by assigning values to the $\mathbf{f}$-components that belong to the inserted buffers, where the value 0 indicates no buffer in the object function, and $-g_b/p_b$ should be used in the delay equations.

Networks Another question is how synthesis can distribute the delay over the gates, which then is to be translated in a value for $\mathbf{f}$. Sutherland's hypothesis of uniform restoring effort [23] might be helpful here. It states that given a network with an equal number of gates on every path from primary input to primary output, a capacitive load at each primary output, and a driving capability at each primary input, the network is fastest when every stage on all input-output paths has the same effort. This is obviously true for a cascade of inverters, and it can be easily extended to networks with equal fanout in a stage. But counter-examples can be easily constructed. Nevertheless, the principle may be useful.

A more serious criticism is that not both, the capacitances and the delays, can be chosen freely. In a wire planning situation where interconnect between modules are optimally buffered, the input and output capacitances are fixed. This limits the possibilities for speed. More generally, timing closure for networks with non-negligable resistance on interconnection is unsolved!

On the more positive side, a benefit is that technology mapping becomes efficient under this constant delay paradigm. Technology mapping is known to be efficient for trees and so-called convergent networks[15]. It has been shown recently [11] that technology mapping for load independent delay can be solved efficiently for all acyclic networks. Thus the standard first step of technology mapping in logic synthesis of partitioning into trees need not be done and hence the optimum solution can be found.

The possibilities for logic synthesis with fixed delay are certainly not exhausted, and will require further research, but for here it is important that layout synthesis is capable of realizing gates with a priori imposed delays.

[15] These are essentially trees of which the primary inputs may feed several gates, in design automation texts often called leaf-dags.

7.2 Cell Generation and Shape Assignment

Wire plans perform their analyses on point placements or sequences, most likely under the presence of larger blocks that may be pre-placed. This position information along with a possibly partial pin assignment must be preserved during the layout synthesis when the results of size assignment and area optimization become available. This requires efficient and robust floorplan optimization. These qualities heavily depend on the floorplan to be optimized. This can only be achieved by maintaining sliceability throughout the design, from the early wire planning stage down to the determination of the final dissection. This presumed "restriction" is amply offset by the guaranteed optimum in its class.

Cell generation is most likely the big challenge in a constant delay approach. The set of functions can be quite small, but extensive research is necessary to determine which sizes should be made available. Ultimately, a library of cell layout generators seems to be the way to go. In addition, yield is also an issue here.

8 Conclusions

In synthesizing high performance chips using present day design practices, the meeting of timing constraints necessitates an iteration which is not guaranteed to converge. In future technologies, unless these global delays are planned up front, convergence will be even more of a problem and even if convergence is achieved, the answer is likely to be far from optimum. This suggests a shift in design methodology where a global wire plan is put in place beginning at the conceptual stage of the design. We propose an approach in which a wire plan is created before the functionality of the blocks in that plan has been fixed. This allows for better control over the performance of the total design. Inherent to such an approach is that wire delay is accurately known wherever it has impact. This means that "global wires" should be well characterized a priori, which requires a strict layout styles. We have chosen to use a minimum width optimally buffered interconnections with a fairly stable electrically environment. Adherence to this style provides delays linear with distance, and thus invariance over equal length paths. Sharing functional blocks is likely to cause detours in one or more paths, causing additional delay. Creating a wire plan may distribute units all over the chip, thus abandoning the principle of easily recognizable and recoverable blocks, in exchange for exact knowledge of delay on connections and control over delay in the blocks. Enforcing delays in the blocks means that sizes become uncertain, and with uncertainties in size also distances become uncertain. If a block cannot be synthesized with the required delay in the available space, then the wire plan cannot be realized. Thus, reliable predictors in the early stages must be developed to obtain a

non-iterative design flow. Of course, existence of solutions can never be guaranteed under too strict timing requirements, but we postulate that this new methodology can find solutions for a broader range of specifications than the current methods.

Acknowledgment:
Part of this chapter has been presented by the present author and Robert Brayton at the *Design Automation Conference 1998* for an audience of over 800 people. Robert Brayton supplied *valid retiming* and *layer assignment* as additional possibilities for doing quick background analyses during wire planning. His group at University of California at Berkeley, and Philip Chong, Wilsin Gosti, Hiroshi Murata,and Mukul Prasad more in particular was quite involved in developing the wire planning concepts. *Monotonic placements* was developed by Wilsin Gosti, who made additions and modifications to *sis* to have "legal operators" that preserve monotonicity, or rather the existence of monotonic placement for all the gates. Further the Nexsis group at Berkeley, in particular Amit Mehrotra, Sunil Khatri, Subarna Sinha, and Philip Chong, made some of the studies that quantified the critical lengths.

Lukas van Ginneken has always been a source of inspiration. Many of the layout synthesis ideas were developed with him when we were both at Thomas J. Watson Laboratories, IBM Research in Yorktown Heights. Through him I also learned about the methods involving *fixed delays* that he developed together with researchers at Thomas J. Watson Laboratories and Synopsys Inc.

References

1. F. Beeftink, A. J. van Genderen, N. P. van der Meijs, *Accurate and efficient layout-to-circuit extraction for high speed mos and bipolar/bicmos Integrated circuits* ICCD, Oct. 1995
2. F. Beeftink, P. Kudva, D. S. Kung, L. Stok, *Gate size selection for standard cell libraries*, International Conference on Computer Aided Design, San Jose, 1998
3. H. B. Bakoglu, *Circuits, interconnections, and packaging for vlsi*, Addison-Wesley Pub Co, 1990
4. R. K. Brayton, C.-L. Chen, J. A. G. Jess, R. H. J. M. Otten, L. P. P. P. van Ginneken, *Wire planning for stackable designs*, Proceedings 1987 International Symposium on VLSI Technology, Systems and Applications, Taipeh, Taiwan, pp. 269–273, May 1987
5. M. Burstein, *Channel routing*, In: *Layout Design and Verification*, T. Ohtsuki (ed.), chapter 4, pp. 133–168
6. P. D. Fisher, *Clock cycle estimation for future microprocessor generations*, 1998; L. P. P. P. van Ginneken, *The predictor-adaptor paradigm*, PhD thesis, Eindhoevn University of Technology, 1989
7. W. Gosti, *Wire planning in logic synthesis*, 1998

8. J. Grodstein, E. Lehman, H. Harkness, B. Grundmann, Y. Watanabe, *A delay model for logic synthesis of continuously-sized networks*, ICCAD, Nov. 1995
9. P. R. Groeneveld; *Context-driven channel routing*, PhD thesis, Delft University of Technology, 1991
10. P. Kudva, *Continuous optimizations in synthesis: the discretization problem*, Logic Synthesis Workshop, proceedings, pp. 408–418, 1998
11. Y. Kukimoto, R. K. Brayton, P. Sawkar, *Delay-optimal technology mapping by dag covering*, DAC, June 1998
12. D. S. Kung, *A fast fanout optimization algorithm for near-continuous buffer libraries*, Proceedings of the 35th Design Automation Conference, San Francisco, 1998
13. U. Lauther, *A min-cut placement algorithm for general cell assemblies based on a graph representation*, Journal of Digital Systems, Vol. 4. 1980, pp. 21–34.
14. R. Nair, C.L. Berman, P.S. Hauge, E. Yoffa, *Generation of performance constraints for layout* IEEE Transactions on Computer-Aided Design, vol 8, nr 8, pp. 860–874, august 1989.
15. R. H. J. M. Otten, *Complexity and diversity in ic layout design*, Proceedings IEEE International Conference on Circuits and Computers, Port Chester, New York, U.S.A., pp. 764–767, October 1980
16. R. H. J. M. Otten, *Layout compilation*, in *Design systems for vlsi circuits*, edited by G. DeMicheli, A. Sangiovanni-Vincentelli and P. Antognetti, pp. 439–472, Martinus Nijhoff Publishers, 1987
17. R. H. J. M. Otten, *Graphs in floorplan design*, International Journal of Circuit Theory and Applications, vol 16, pp. 391–410, 1988
18. R. H. J. M. Otten, L. P. P. P. van Ginneken, N. V. Shenoy, *Speed: new paradigms in design for performance*, ICCAD, Nov. 1996
19. L. Pileggi, *Delay metrics*, ISPD98
20. Semiconductor Industry Association, *The national technology roadmap for semiconductors: technology needs*, California, U. S. A., 1997
21. T. Sakurai, *Approximation of wiring delay in mosfet lsi*, IEEE Journal of Solid-State Circuits, vol SC-18, pp. 418–426, Aug. 1983
22. L. J. Stockmeyer, *Optimal Orientations of cells in slicing floorplan design,* Information and Control, vol 57, pp. 91–101, 1983
23. I. Sutherland, R. Sproull, *The theory of logical effort: designing for speed on the back of an envelope*, in *Advanced Research in VLSI*, UC Santa Cruz, 1991
24. I. Sutherland, R. Sproull, D. Harris, *Logical effort: designing fast cmos circuits*, Morgan Kaufman Publishers, 1999
25. A. Tabbara, R.K. Brayton, A.R. Newton, *Retiming for DSM with Area-Delay Trade-Offs and Delay Constraints*", Proceedings of the Design Automation Conference, 1999, pp. 725–730
26. N. Wirth, *Program development by stepwise refinement*, Communications of the ACM, vol 14, pp. 221–227, 1971
27. W. Wulf, M. Shaw, *Global variables considered harmful*, Sigplan Notices, February 1973, pp. 28–33
28. J. L. Wyatt Jr, *Signal propagation delay in rc models for interconnect*, chapter 11 (pp. 254–291) in *Circuit analysis, simulation and design, 2*, Elsevier Science Publishers B. V., 1987