Automatic layout and optimization of static CMOS cells

Frédéric Mailhot *Giovanni DeMicheli*

Computer Systems Laboratory
Department of Electrical Engineering
Stanford University
Palo Alto, CA 94305

Abstract

A novel algorithm for generating complex CMOS gates from Boolean factored forms is presented. It uses a hierarchical composition of cells corresponding to the sub-expressions of a Boolean factored form. Composition rules that allow for constructing the gates in linear time are derived. Cell width/height trade-offs are made possible to ease pitch-matching. The algorithm has been coded in a pair of programs, called Castor and Pollux. The programs have been used to generate moderately complex layouts, consisting of circuits having up to a few thousand transistors. They can also be used to automatically generate a library of logic gates.

1 Introduction

Computer-aided synthesis of digital circuits has shown to be a viable way of designing large and complex hardware systems. Logic synthesis techniques allow designers to optimize circuit representations in terms of logic equations and map them into forms amenable for gate implementation. This paper addresses the automatic layout of complex digital gates from their logic specifications. This problem is important in two respects: first as a way of generating parametrized cells for a library and second for the automatic generation of macro-cells from the logic specification of a combinational or sequential logic unit.

Automatic layout techniques require a structured implementation. In MOS technology, logic gates can be implemented by aligning transistors in a linear array. In particular, for static CMOS implementations, a layout strategy was proposed by Uehara and van Cleemput [13], where complementary transistor pairs are organized in an array with polysilicon gates formed by vertical lines and with drain/source connections by abutment (Fig. 1). This structure requires as many transistor pairs as there are inputs to the gate. Ideally, all source/drains of the transistor pairs should connect by abutment, to reduce total area and parasitic capacitance, and hence to improve the gate switching performance. Unfortunately, for a given logic function, it is not always possible to achieve such a connection and diffusion gaps may be necessary. Moreover, the search for an arrangement of the transistor pairs that minimizes the area (or equivalently minimizes the diffusion gaps) is a difficult problem, and very likely intractable. For this reason, Uehara and vanCleemput introduced a simple heuristic algorithm to align the gates.

Nair, Bruss and Reif [11] analyzed the problem in terms of graph theory, and published a linear-time algorithm which finds a transistor pair ordering under the assumption that one exists with no diffusion gaps. Unfortunately their paper does not report on how to solve the problem of finding an optimum, or local optimal solution in the general case, when diffusion gaps do exist. Maziasz proposed a linear-time algorithm for aligning optimally transistor pairs, under the limiting assumption of a fixed topology, i.e. no rearrangement of the series components of the graph is allowed [8]. Hill presented an automatic layout system with an emphasis on the gate matrix structure [5].

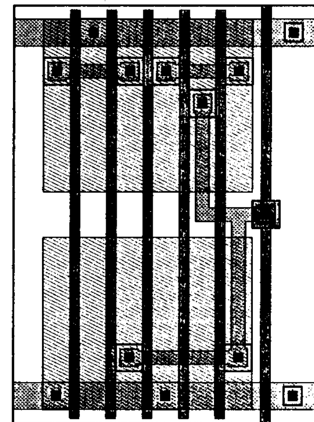

Figure 1: Uehara and vanCleemput's layout strategy

Optimal gate design is much simpler in the case of nMOS (pMOS) or dynamic CMOS technology, because it involves only the chaining of transistors of one type (either N or P), rather than transistor pairs. A heuristic algorithm was presented by McMullen and Otten [10] and linear-time algorithms by Müller [9]. Recently, Wimer, Pinter and Feldman [14] presented a layout structure for static CMOS gates that is reminiscent of Hill's work [5]. It used a routing channel between the N and the P parts of the circuit, therefore simplifying the problem of aligning transistor pairs to that of aligning two independent sets of transistors, at the expense of some area spent in routing.

In this paper we consider the automatic synthesis and optimization of gates implementing logic functions in static CMOS technology. In addition we consider two important implementation issues: bounding the cell height and allowing transistors with non uniform polysilicon gate widths. The first consideration is important for pitch-matching the cells in a standard-cell like fashion, the second for the design of high-performance cells. Note that the previously published algorithms had no control on the height of the cell and that the corresponding gate implementations used transistors with uniform polysilicon gate widths.

CH2643-5/88/0000/0180$01.00 © 1988 IEEE

We then present a heuristic algorithm to compute a near optimal ordering of the transistor pairs and show the layout implementation in three different styles. We conclude this paper by relating the use of gate synthesis to the OLYMPUS synthesis system at Stanford University.

2 CMOS gate implementation styles

Gates implemented by linear transistor pair arrays in CMOS technology can use different layout styles. We consider here three of these styles. The first style uses one level of metal and is the one proposed by Uehara and van Cleemput (Fig. 1). The gate personality is determined by the sequence of the transistor pairs (possibly including diffusion gaps) and by metal stripes contacting the diffusion areas. Power and ground lines run horizontally in metal at the top and the bottom of the cell respectively.

The second style uses two levels of metals. One level is used to strap the diffusion area to reduce the parasitic resistance while the other level of metal is used to connect the first one to personalize the gate (Fig. 2). This style allows efficient gate implementations with wider gates.

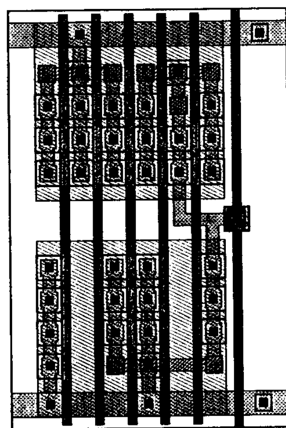

Figure 2: Layout with two levels of metal

The third style has power and ground busses close to the center of the cells. Transistors may have polysilicon gates with arbitrarily large widths, extending in the external direction (Fig. 3). With this layout style, transistors with variable widths may be used within the same gate. The transistors that are larger than a certain limit are broken into smaller parallel ones, as done in [5] and in the CLEO layout system [3].

3 Algorithms for cell layout

The three implementation styles can be abstracted by the same structural model. Each gate instance may be represented by a symbolic layout, representing the transistor pair sequence, the diffusion gaps and the metal interconnections. Such a symbolic layout is then mapped automatically into a geometric layout in any of the three implementation styles.

3.1 Optimal input order (width minimization)

While previous work [13] [11] was related to transforming an electric diagram of a circuit into a layout, we consider here as a starting point a representation in terms of logic equations. The reason for our choice is that synthesis systems are today displacing schematic capture programs

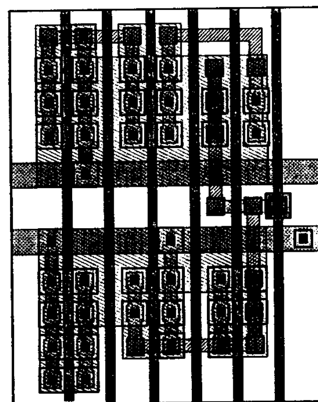

Figure 3: Layout with central Vdd and Gnd lines

for digital design specification. The right-hand side of each logic equation consists of a Boolean factored form (Bff), which can be described as follows using a Backus-Naur form (BNf) [1]:

$$Bff \rightarrow literal \mid literal' \mid (Bff) \mid Bff + Bff \mid Bff * Bff$$ [1]

e.g.: A', $A' + B$, $A + C(B + D(E + F))$

The BNf description shows the Boolean factored form has the property of being generated by successive reductions [1] applied on literals (Fig. 4).

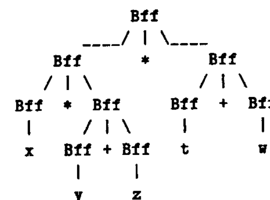

Figure 4: Successive reductions[2] for $x(y + z)(t + w)$

The algorithm presented here is based on that property. There is a close analogy between these reductions and the way a logic gate is constructed: whenever two Bff are reduced together, two groups of transistors (called cells thereafter) have to be combined. For nMOS transistors, a sum (Bff + Bff) is implemented by putting the two cells in parallel, and a product (Bff Bff) by putting them in series (pMOS transistors require the opposite). For static CMOS implementations, a cell is an array of transistor pairs (N and P) whose gates are aligned. The simplest component of a Bff, a literal, is implemented by a cell having two transistors (P and N) on top of each other. In general a Bff is implemented by a series/parallel connection of N and P type transistors. We assume, as in [13], that the graphs abstracting the interconnection of the N and the P parts are the dual of each other, and are series/parallel reducible. The corresponding decomposition tree is an alternative representation of the BNf. Therefore a general cell can be constructed by combining basic cells in a similar way as the Bff is reduced from literals. Then the problem of constructing a logic gate

[1] The * is optional in a product of Bff
[2] The reductions $(Bff) \rightarrow Bff$ have been omitted for the sake of simplicity

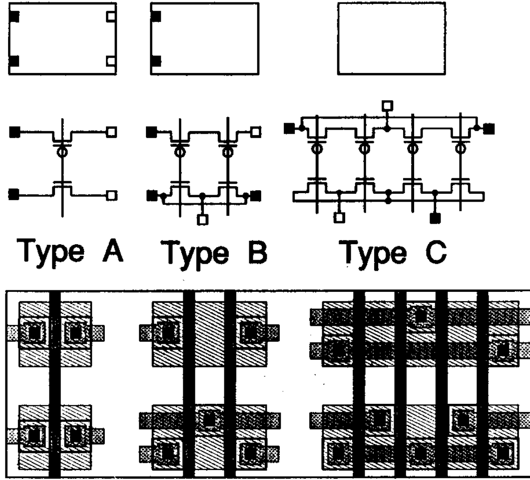

Figure 5: Cell abstraction

can be simplified to the one of finding its component cells for each level of the BNf hierarchy, and the way they interact with each others.

Adjacent cells can be combined in two ways. They can abut directly (providing direct connection in the diffusion layer between the source/drain of the transistors), or they can be separated by a diffusion gap. Ideally no diffusion gap should be used, to achieve minimum cell length. Unfortunately, there are BFfs with no corresponding linear alignment of transistors, or equivalently decomposition trees with no corresponding dual Euler paths, as formally shown in [11]. It is then the purpose of our method to minimize the number of diffusion gaps. For this reason, an appropriate way of classifying the cells is to explore the way they connect to neighbors. A cell can *possibly* be abutted to from two sides. Hence there are three different types of cells (Fig. 5), with a simple set of composition rules defining how they can be reduced together:

Type **A**: A cell that can be abutted to other cells from both sides.
Type **B**: A cell that can be abutted to other cells on one side only.
Type **C**: A cell that cannot be abutted to others (needs a cut).

The type of a cell is determined by the position of its terminal connections, i.e. the diffusion corresponding to the source and the sink of the series/parallel graphs representing the N and P transistor networks.

Type **A** cells have all terminal connections on the right and the left side of the cell.
e.g.: *literal, odd number of literals in a logic sum or product.*

Type **B** cells have only one pair of terminals (N and P) on one side of the cell.
e.g.: *an even number of literals in a logic sum or product.*

Type **C** cells are the remaining ones.
e.g.: $(a+b)c(d+e)$

The composition rules the 3 types follow are the following (where $\|$ means joining two adjacent cells):

$$\mathbf{A} \,\|\, \mathbf{A} \,\|\, \cdots \,\|\, \mathbf{A} = \mathbf{A} \text{ (odd number of A's)} \quad (1)$$
$$= \mathbf{B} \text{ (even number of A's)} \quad (2)$$
$$\mathbf{A} \,\|\, \mathbf{B} = \mathbf{B} \quad (3)$$
$$\mathbf{A} \,\|\, \mathbf{C} = \mathbf{B} \quad (4)$$
$$\mathbf{B} \,\|\, \mathbf{B} = \mathbf{C} \quad (5)$$
$$\mathbf{B} \,\|\, \mathbf{C} = \mathbf{B} \quad (6)$$
$$\mathbf{C} \,\|\, \mathbf{C} = \mathbf{C} \quad (7)$$

Only compositions involving type **C** cells (rules 4, 6 and 7) involve the introduction of a diffusion gap. Since $\mathbf{A} \,\|\, \mathbf{B} = \mathbf{B}$, then in our algorithm rule 3 has the priority over rules 1 and 2. Therefore merging any number of type **A** cells with one type **B** cell results finally into one single type **B** cell. A composition example for $x(y+z)(t+w)$ is shown next, using the BNf reductions shown in Fig. 4. An associated schematic is also shown in Fig. 6. Notice that the final cell generates the complement of the initial equation.

x	:	**A** type cell (rule 1)
$(y+z)$	:	$\mathbf{A} \,\|\, \mathbf{A} = \mathbf{B}$ type cell (rule 2)
$(t+w)$	:	$\mathbf{A} \,\|\, \mathbf{A} = \mathbf{B}$ type cell (rule 2)
$(y+z)\ \ x$	:	$\mathbf{B} \,\|\, \mathbf{A} = \mathbf{B}$ type cell (rule 3)
$(y+z)x\ \ (t+w)$	:	$\mathbf{B} \,\|\, \mathbf{B} = \mathbf{C}$ type cell (rule 5)

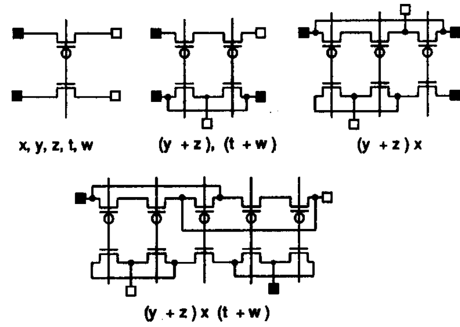

Figure 6: Schematics associated with the composition example

The way we defined the BFf lends itself very easily to LR parsing [1]. We therefore assume a well formed parse tree (which follows very closely the Backus-Naur Form description) is available before applying the algorithm. The logic information that is read into a tree is readily available from logic minimization programs like MIS [2] or Minerva [6]. The algorithm visits the tree in depth-first search and constructs the symbolic layout bottom-up, by combining the symbolic layouts corresponding to the children of each cell. The algorithm is detailed in Fig. 7.

```
build_hierarchically(father)
{
    if the parent cell has no child (literal cell) {
            create a type A cell as the father }
    else {
        for each child of the parent cell {
                build_hierarchically(child);
        }
        merge_children(father); }
}
```

Figure 7: Pseudo code for cell building

The sequence in which cells are combined affects the type of the resulting cell. Since the overall goal is to minimize the number of diffusion gaps, the algorithm discriminates between the three cell types to achieve better parent cells. The *merge_children* procedure merges type **A** cells first, then type **B**, and finally type **C** as shown in Fig. 8.

Proposition 3.1 *The number of gaps required by the cell sequence generated in procedure* merge_children *is minimal with respect to all permutations of the children.*

Proof: Let us assume a cell has x type **A**, y type **B** and z type **C** children (subcells). By definition, a diffusion gap is needed whenever a type **C** cell is merged with another cell. Therefore $z-1$ gaps are needed if there are only type **C** cells, and at least z gaps if there are other types of cells. Also by definition, a gap is required whenever a pair of type **B** cells are combined with type **B** or **C** cells. Therefore, at least $z + \lfloor \frac{y-1}{2} \rfloor$ cuts are needed.

Let us compare this result with the number of diffusion gaps generated by procedure *merge_children*. This number depends on the total number of type **C** cells, including those generated by merging type **A** and type **B** cells (composition rules 3 and 5). Hence we need to know how many additional type **C** cells are generated while combining type **A** and **B** cells.

If there are no type **B** cells initially ($y = 0$), then type **A** cells are combined together (using composition rules 1 or 2), into one single cell, namely one type **B** cell if x is even or one type **A** cell if x is odd. Therefore z cuts are required if $x \neq 0$ and $z-1$ if $x = 0$.

If there is at least one type **B** cell initially ($y > 0$), then all type **A** cells are merged with one of the existing type **B** cells (composition rule 3). Then, by composition rule 5, procedure *merge_children* combines these type **B** cells into $\lfloor \frac{y}{2} \rfloor$ type **C** cells. Let $z' = z + \lfloor \frac{y}{2} \rfloor$ be the resulting number of type **C** cells , to be combined with an additional type **B** cell if y is odd. Then the total number of diffusion gaps generated in the parent cell is $z + \frac{y}{2} - 1$ if y is even and $z + \lfloor \frac{y}{2} \rfloor$ if y is odd, i.e. $z + \lfloor \frac{y-1}{2} \rfloor$. ◇

Therefore, procedure *merge_children* minimizes the number of diffusion gaps introduced by reordering one level of subcells. However, this is not sufficient to state that the algorithm finds a global minimum number of gaps. Consider for example the logic function $(a+b)(c+d(e+f)(g+h))$. The algorithm finds $abcefdgh$ as the best ordering of the inputs. This order requires one diffusion gap between the polysilicon gates implementing the c and the e literal. A better ordering would be $efdcabgh$, because it needs no diffusion gap. However, to find such an order, the permutations over more than one level of the Bff hierarchy have to be considered at the same time. Such a search, possibly of exponential complexity, is not considered by our present approach because of computing time and because the results obtained by the present algorithm have shown to be close to minimum.

```
merge_children(father)
{
    for each child of the parent cell: {
        Check the type of the child cell (A, B, or C)
        If type = A
            Add the child to the list of type A cells
        If type = B
            Add the child to the list of type B cells
        If type = C
            Add the child to the list of type C cells
    }
    if the list of type B cells is not empty {
        take one type B cell as the starting cell
    }
    for each cell in the list of type A cells:
    {
        Merge the cell with the ones already merged
        (If it is the first cell to be merged, take it as the starting cell)
    }
    for each cell in the list of type B cells: {
        Merge the cell with the ones already merged
    }
    for each cell in the list of type C cells: {
        Merge the cell with the ones already merged
    }
}
```

Figure 8: Pseudo code for cell merging

The algorithm merges the cells bottom up to construct a symbolic layout of the root cell. For a n-input gate, there are always $(n-1)$ cells processed in *merge_children* (in between n elements there are $(n-1)$ spaces). Therefore, the algorithm has linear time complexity.

3.2 Optimal track assignment (heigth minimization)

The geometric layout of the cell is derived from the symbolic one. The geometric cell length depends on the number of inputs as well as diffusion gaps. The geometric cell height depends on the number of metal interconnection stripes needed.

Proposition 3.2 *The number of metal interconnections (rows) required in the cell layout corresponding to a Bff is at most* $4(D-2)+7$ *if* $D > 1$ *and* 4 *if* $D = 1$, *where* D *is the factorization depth of the Bff. This number includes power and ground.*

Proof: At each level of factorization of the Bff, there is a series/parallel connection on the N and the P part of the subcells, or vice versa. Let us first look only at what happens on one side of the cell (either N or P). When combining two or more subcells, the resulting number of rows is at least equal to the maximum number needed in any of the subcells before the joining. A parallel connection may require two additional rows. Suppose N children cells are combined, all having two terminals, say a left and a rigth one. Then, in the parallel connection, a metal stripe contacts the left terminals of the children cells in the odd position and the right terminals of the children cells in the even position. The second metal stipe contacts the remaining terminals. A series connection requires also two additional rows, in the general case when the terminals are inside the children cells (e.g. type **C** cells). In the special case of a series connection of type **A** subcells (e.g. cells corresponding to literals), then the series connection requires no additional row. In the case that the children cells are of type **A** and **B** only, at most one additional row is needed. These two special cases apply to the first level of factorization.

Therefore when both the N and the P parts of two subcells are joined, at most 4 additional rows are required in the general case, 3 if the factorization depth is 2, and 2 if the depth is 1. A type **A** cell implementing a literal needs 2 metal stripes. Then the maximum cell heigth is bounded by $2+2+3+(D-2)(4) = 4(D-2)+7$ if $D > 2$, $2+2+3 = 7$ if $D = 2$, and $2+2 = 4$ if $D = 1$. ◇

Note that proposition 3.2 states only a bound on the number of stripes. In practice, fewer stripes may be required. For this reason, a line packing procedure [4] is applied to minimize the final heigth of the cell. The line packing is done independently on the N and the P part of the gate, once the ordering of the inputs is obtained and a netlist of metal stripes connecting sources and drain is derived.

For each part (N or P) of the gate, there are two special stripes, corresponding to the power contact (Vdd or Gnd) and the output contact. In the graph representation of the transistors network, theses two stripes are represented by the source and sink. All the other metal stripes are simply intermediate equipotentials. Since the power contact and the output contact are interchangeable, they are not distinguished initially. The algorithm first orders the metal stripes by the coordinate of their left end. It then iteratively chooses the leftmost stripe that fits on a track without overlapping stripes that were already placed. When no more stripes are found to fit on a certain track, a new track is begun. The power and output stripes need special attention in that process: the power stripe takes one entire track, and the output stripe has to be placed closest to the dual (P or N) output stripe. A pseudo-code description of the line packing algorithm is shown in Fig. 9.

Boolean factored forms are optimized in terms of literals in the logic synthesis stage, to reduce the cell length. Therefore each Bff is initially factored as much as possible. If the corresponding cell height does not satisfy the design requirements (e.g. pitch matching), then the Boolean factored form can be reduced in depth until the desired height is achieved. Because the number of metal tracks needed is bounded by the deepest factor in the Bff, then the expansion should always be done on the factor with the largest depth. This allows to trade-off cell length for cell height.

```
stripe_compaction()
{
    for each metal stripe {
        put the stripe in the to_place list
        find the left end coordinate
        find the right end coordinate
        }

    sort the stripes by their left end coordinate
    set the right limit to zero
    reset the chosen_list

    while there is a stripe in the to_place list {
        take the stripe with the leftmost left end that is greater
        or equal to the right limit

        if a stripe was chosen {
            if the stripe is the source or sink {
                if there is no source or sink already in the chosen_list {
                    put the stripe in the chosen_list
                    mark the chosen_list as containing the source or sink
                    set the right limit to the right end of the stripe
                    }
                else { leave the stripe in the to_place list
                    }
            }
            else { put the stripe in the chosen_list
                set the right limit to the right end of the stripe
                }
        }
        else (if no stripe was chosen) {
            if the chosen_list has the source or the drain in it {
                if there is no other stripe in the list {
                    mark the list as the power line
                    }
                else { mark the list as the output line
                    if the power line is not defined yet {
                        get the remaining source or sink line
                        mark it as the power line
                        put it in the completed_lines list
                        }
                    }
            }
            copy the chosen_list into a completed_lines list
            reset the chosen_list
        }
    }
}
```

Figure 9: Pseudo code for stripe compaction

4 Layout system implementation

The software implementation consists of two programs. The first one, named **Castor**, parses the logic description and creates a symbolic layout and an interconnection netlist. The second one, **Pollux**, constructs the geometric layout, using a user-defined implementation style.

The software is designed to generate a macro-cell made of combinational and/or sequential circuits. Special cell generators construct the geometric layout of registers and tristates elements. Combinational gates are constructed for each Boolean factored form. To save computing time, each form (or equivalently each BNf) is ranked using a technique similar to the one presented in [12]. The ranking allows to generate one master cell for each given type and to instantiate it if multiple occurrences are needed. Inverters are inserted whenever a signal needs to be complemented. Table 1 shows run times and results for single logic equations (coming from existing publications). The required number of gaps , optimum number of gaps, and heigth (number of horizontal tracks required) of the generated gates are shown. Table 2 shows run times for circuits consisting of multiple equations, latches and tristates. Program **Castor** was used on a MicroVax II for these tests.

The transformation of a symbolic layout into a geometric one is programmed in the GDT 3.1 environment [15]. Four cell generators were written in the L language for the combinational logic gates, latches, tristates and the root macro-cell generator. The root macro-cell uses the three other generators to draw each different master cell. These master cells are then instantiated and assembled using the place-and-route tool available in the GDT environment. An layout generated by the system is given in Fig. 10. It is a four-bit look-ahead carry with latches and tristates on the outputs.

Programs **Castor** and **Pollux** are part of the OLYMPUS synthesis

Equation	Ref.	Cuts (optimum)	Tracks (bound)	Time
$a + bc + de$	[10]	0 (0)	5 (6)	1.2 (s)
$a(b + c(d + e(f + g(h + i))))$	[13]	0 (0)	6 (30)	1.6 (s)
$ab + cd + (e + f)(g + h)$	[8]	1 (0)	6 (10)	1.4 (s)
$(ab + cd)e$	[9]	1 (1)	5 (10)	1.3 (s)

Table 1: Run times for single gate circuits

Circuit	# Equations	# Latches	# Tristates	Time
1	17	0	7	2.7 (s)
2	16	5	5	2.7 (s)
3	42	8	0	4.6 (s)
4	1341	294	0	128.1(s)

Table 2: Run times for circuits of various sizes

system developed at Stanford University. OLYMPUS is an integrated set of synthesis programs. Circuit behavior, described in a high level language, HardwareC [7], is transformed into the corresponding logic view. The logic description is then optimized in terms of delay and/or area. Gate synthesis is used as the back-end of OLYMPUS to map the optimized logic description onto silicon macro-cells.

5 Conclusion

A novel algorithm for complex CMOS gates generation has been presented. By abstracting the problem of building a gate to the one of reducing a Backus-Naur form, simple composition rules were derived. These rules allowed for building the gates in linear time. The implementation allows trading off width for heigth.

The system has been used to generate moderately complex layouts, consisting of circuits having up to a few thousand gates. It can also be used to automatically generate a library of logic gates.

Acknowledgements

This work is supported in part by the National Science Foundation under grant MIP-8710748, by a Stanford CIS seed grant, by the Natural Sciences and Engineering Council of Canada, and by the Quebec Fonds F.C.A.R.

References

[1] A.V. Aho, R. Sethi and J.D. Ullman, *Compilers Principles, Techniques, and Tools,* Addison-Wesley, 796 p., 1986.

[2] R.K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli and A.R. Wang, *MIS: A Multiple-Level Logic Optimization System,* IEEE Transactions on CAD, Vol. CAD-6, Nu. 6, pp.1062-1081, november 1987.

[3] A. Domic, private communication, march 1988.

[4] A. Hashimoto and J. Stevens, *Wire routing by Optimizing Channel Assignment within large aperture,* Proceedings of the 8th Design Automation Workshop, pp. 155-169, 1971.

[5] D.D. Hill, *Sc2: A Hybrid Automatic Layout System,* Proceedings of the ICCAD, Santa Clara, pp. 172-174, 18-21 june 1985.

[6] T. Klein, private communication, may 1988.

[7] D.C. Ku and G. DeMicheli, *Hercules: a system for high level synthesis,* Proceedings of the 25th ACM/IEEE Design Automation Conference, Anaheim, 1988.

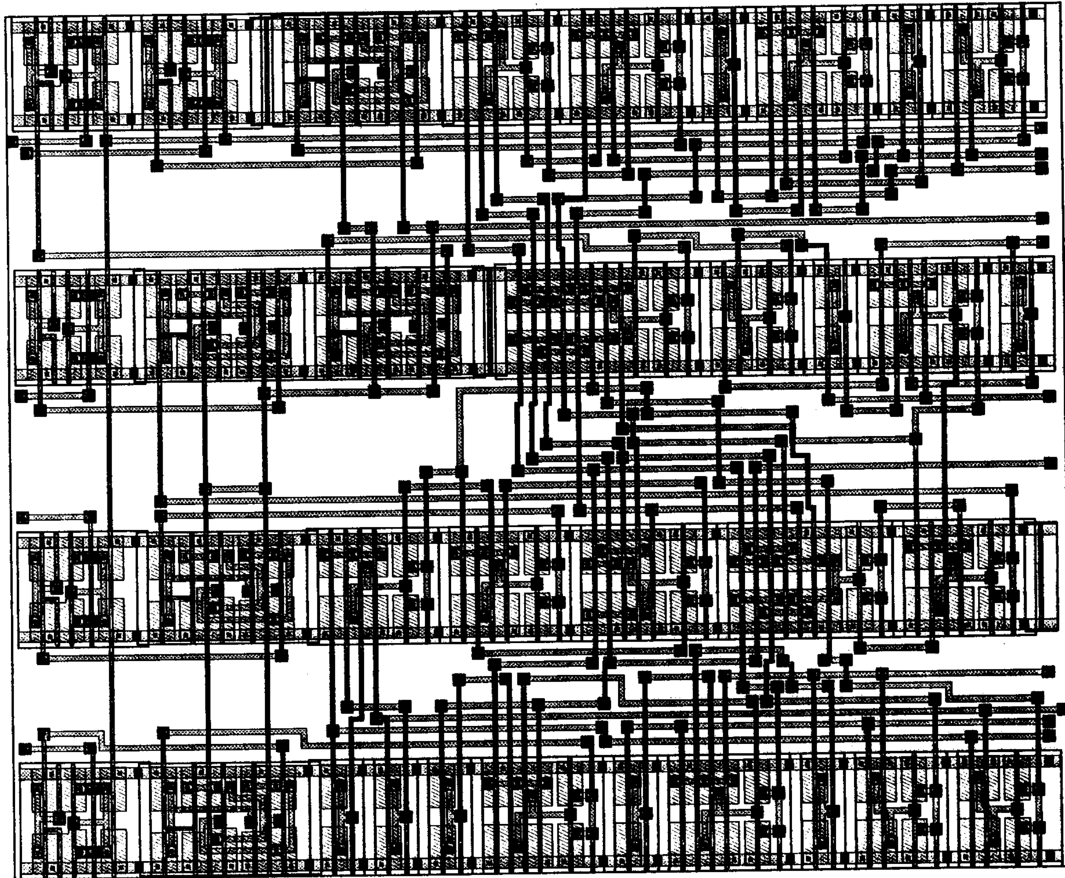

Figure 10: Layout of a four-bit look-ahead adder

[8] R.L. Maziasz and J.P. Hayes, *Layout Optimization of CMOS Functional Cells,* Proceedings of the 24th ACM/IEEE Design Automation Conference, pp. 544-551, 1987.

[9] R. Müller and T. Lengauer, *Linear Algorithms for Two CMOS Layout Problems,* Proceedings of Aegean Workshop on Comput., July 1986.

[10] C.T. McMullen and R.H.J.M. Otten, *Layout Compilation of Linear Transistor Arrays,* Proceedings of the International Symposium on Circuits and Systems, Kyoto, Vol. 1, pp.5-7, 5-7 june 1985.

[11] R. Nair, A. Bruss, and J. Reif, *Linear Time Algorithms for Optimal CMOS Layout,* VLSI: Algorithms and Architectures, P. Bertolazzi and F. Luccio (Editors), Elsevier Science Publishers B.V., pp. 327-38, 1985 New York, December 1983.

[12] C.A. Neff and R. Nair, *A Ranking Algorithm for MOS Circuit Layouts,* IEEE Transactions on Computer-Aided Design, Vol. CAD-6, No. 1, pp.17-21, january 1987.

[13] T. Uehara and W.M. vanCleemput, *Optimal layout of CMOS Functional Arrays,* IEEE Transactions on Computers, Vol. C-30, Nu. 5, pp. 305-12, may 1981

[14] S. Wimer, R.Y. Pinter and J.A. Feldman, *Optimal Chaining of CMOS Transistors in a Functional Cell,* IEEE Transactions on Computer-Aided Design, Vol. CAD-6, No. 5, pp.795-801, september 1987.

[15] Silicon Compiler Systems, *GDT Database and Language Tools Reference,* Version 3:1, December 28, 1987.