Algorithmic Layout of Gate Macros*

Daniel D. Gajski
Avinoam Bilgory
Joseph Luhukay

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

The rapid advancement of VLSI technology necessitates new implementation methodologies with design automation capabilities. Existing implementation styles such as master slice, programmable logic arrays and custom design with cell library do not achieve the best tradeoffs between circuit density and chip development cycle time. The implementation methodology based on register-transfer building blocks called gate macros can be used to drastically cut down the design time. Furthermore, the gate macros which generally represent functional entities like registers, adders, busses, logic units etc. are subjective to algorithmic or totally automatic layout [Verg80], [Joha79].

This paper describes the basic modules of a gate-to-silicon compiler which accepts as its input a high level description of gate macros and generates a layout that satisfies particular technology (NMOS, for example) and environmental parameters (layout area or time delay, for example). The input to the gate-to-silicon compiler are the set of

* This work was supported in part by the NSF under grant No. US NSF MCS80-01561

macros generated at the register transfer level. High-level language constructs like DO loops and IF statements are allowed in the input language. However, only Boolean scalars, vectors and strings are allowed. For example, a 16-bit binary adder can be described as follows:

```
S1:     C(0) = CIN
      DO I = 1,16
S2:     C(I) = A(I)*B(I) + (A(I) + B(I))*C(I-1)
S3:     S(I) = A(I) ⊕ B(I) ⊕ C(I-1)
      END
S4:     COUT = C(16)
```

The above description can be used for variety of implementation styles. For example, if the delay time specified is relatively slow with respect to technology used the 32-bit adder will be implemented as a ripple-carry adder. If a faster version is required the look-ahead-carry adder will be used. For different delay times different number of bits will be looked ahead. Similarly, different layouts will be produced for different time delays.

The compiler consists basically of four modules (Figure 1):

1. Boolean Analyzer partitions the input description into blocks with easily recognizable structure. For example, the statements S_1 and S_2 will be recognized as a recurrence system while the statement S_3 is detected to be a vector operation. Statement S_4 is detected as a scalar operation. Furthermore, the Boolean Analyzer generates the dependence graph with statements as vertices and dependences as edges. The dependence graph represents the internal structure of the gate macro. It

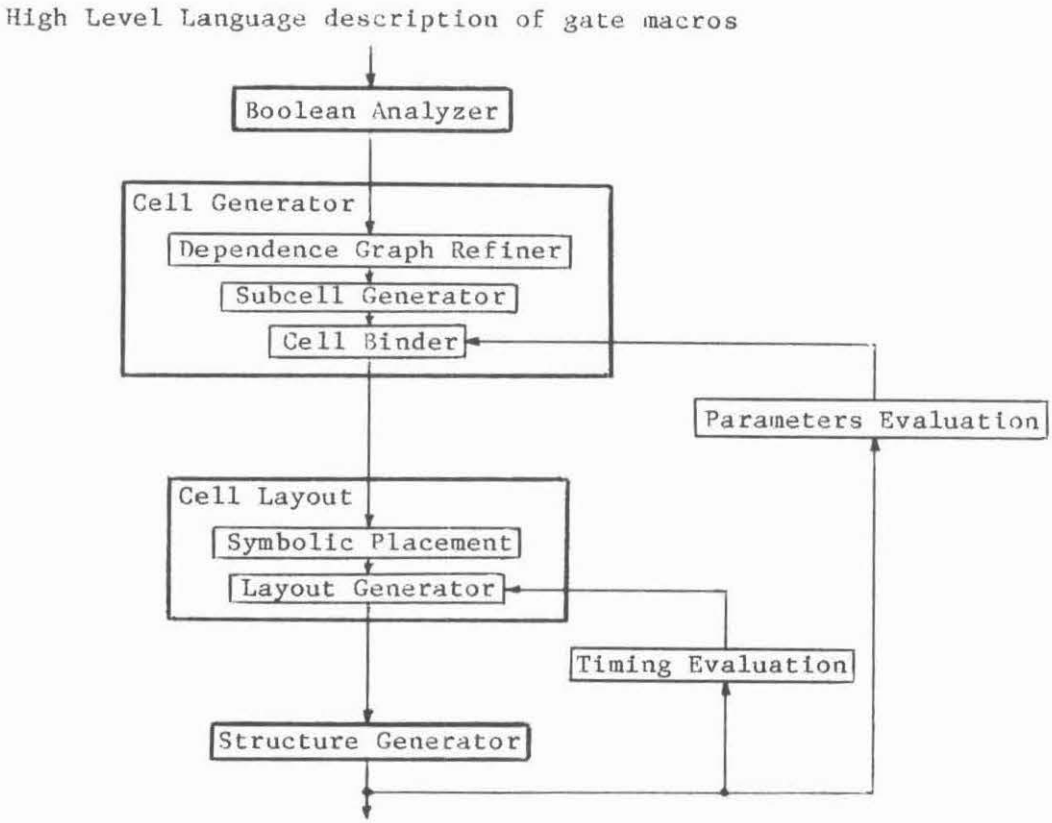


Figure 1. Block diagram of a gate-to-silicon compiler.

indicates the critical time delay and cell structure of the future layout alternatives.

2. Cell Generator modules consist of Dependence Graph Refiner, Subcell Generator and Cell Binder.

The Dependence Graph Refiner tries to break each of the dependence graph nodes into as many nodes as possible. The resulting dependence

graph is more detailed, which allows the Cell Binder more flexibility in optimization. Since statements S_1 and S_4 are scalar operations without operators their layout area and time delay are $O(\varepsilon)$ where ε is a small value, so they are left untouched. Statement S_2 is a recurrence with maximum O(n log n) layout area and minimum O(log n) time delay where n is the recurrence length. Since the recurrence node will be broken into three or more different types of subcells, its decomposition is left to the Subcell Generator. Statement S_3 has an O(n) layout area and O(1) time delay. Since the EXCLUSIVE-OR operation is associative, statement S_3 can be dissolved into S_{3a} and S_{3b}. Using the above approximation the original program is distributed as shown below.

```
S1:     C(0) = CIN
        DO I = 1,16
S2:       C(I) = A(I)*B(I) + (A(I) + B(I))*C(I-1)
        END
        DO I = 1,16
S3a:      T(I) = A(I) ⊕ B(I)
        END
        DO I = 1,16
S3b:      S(I) = T(I) ⊕ C(I-1)
        END
S4:     COUT = C(16)
```

The new dependence graph is shown in Figure 2.

The Subcell Generator consists of several submodules, each for one type of a block recognized by the Boolean Analyzer. Each submodule generates the functional description of the basic subcells used to synthesize the given block. The recurrence statements S_1 and S_2 generate

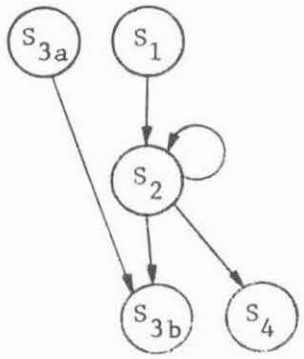


Figure 2. Dependence graph of distributed program.

four types of subcells:

type 2.1 subcell: $G = A*B$

type 2.2 subcell: $P = A + B$

type 2.3 subcell: $G = G_1 + G_2*P_1, \quad P = P_1*P_2$

type 2.4 subcell: $C = G + P*C_0$

A description of cell generation for recurrence structures is found in [BiGa80]. Statements S_{3a} and S_{3b} generate one type of subcell each, called type 3a and 3b subcells, respectively.

The Cell Binder combines subcells to form larger cells. The subcells to be combined are selected according to the constraints posed by the dependence graph. Since type 2.1 and 2.2 subcells (generated for the recurrence) perform vector operation as well as type 3a subcell, the three can be combined to form one cell called type 1 cell. Type 2.4 and 3b subcells can also be combined into one cell, but it was not done in this example, so type 2.3, 2.4 and 3b subcells will each be assigned one

type of cell and renamed as type 2, 3 and 4 cells, respectively. The layout occupies minimum area when all the cell types have similar widths. So, if the Structure Generator finds, for example, type 1 cell to be too large, a separate cell type may be dedicated to subcell 3a. Since S_{3a} is not on a critical path, this cell can be positioned almost anywhere in the layout in that case.

3. Cell Layout modules consist of Symbolic Placement and Layout Generator.

The Symbolic Placement module generates a two-dimensional array of symbolic transistors and their connections. Compaction is done automatically when this two-dimensional array is translated by the Layout Generator into a complete mask description in compliance with layout design rules of the chosen technology.

Each cell can be manually designed if so desired, leaving the placement and routing to be automatically performed by the system. The manual cell design presents one extreme of the provided layout design space [MeCo80]. However, the overall aim is to have an automatic layout system, where a manual cell design or a cell library is replaced by the library of algorithms in which one or more algorithms for automatic generation of layout specifications are available for each cell model supplied by the Cell Generator module. It then follows that the algorithmic layout is the other extreme of the layout design spectrum.

For example, an obvious approach would be to implement each cell with a small programmable logic array. The MOS and I^2L technologies are well adaptable to automatic synthesis as shown in [SOHT80] for one-dimensional gate arrays. We have chosen a two-dimensional array approach as described below.

The Symbolic Placement module is based upon a grid system of tracks - or channels - on different layers of the integrated circuit structure. Interaction among the layers is governed by the technology, and as a result, geometric relationship among the tracks is determined by the technology's layout design rules. Figure 3 shows a grid system which is used for silicon-gate MOS.

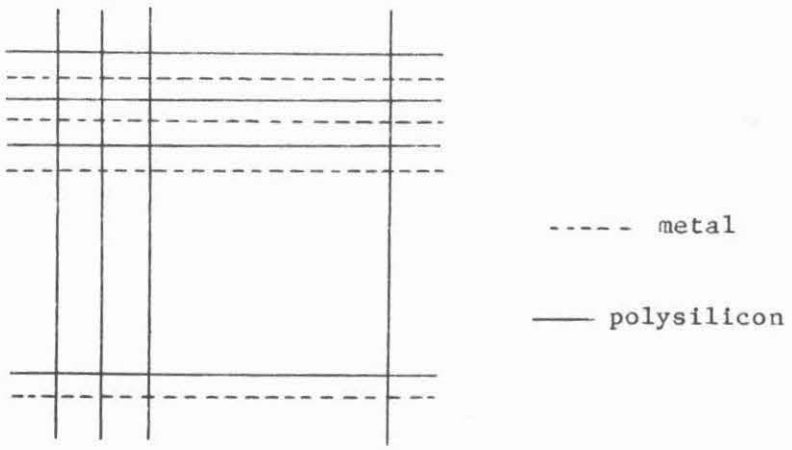


Figure 3. Sample grid system for MOS technology.

Here the metal layer is more or less independent of the polysilicon and the diffusion layers, whereas polysilicon and diffusion interact strongly with each other. Hence polysilicon and diffusion tracks can be "hidden" underneath metal tracks. Using this grid as a base, synthesis procedures have been developed. For example, using a metal and polysilicon grid like in Figure 3, two-dimensional arrays can be formed by manipulating the diffusion to form the necessary devices, interconnected such as to build the required circuit.

Figure 4 shows the processes implemented by the Cell Layout modules. Input to the Symbolic Placement module consists of functional description of a cell (or a set of cells), in the form of a set of AND-OR-INVERT Boolean equations. In addition to this, basic topological information about the cell is also given, which comprises assignment of topological attributes to the input-output nodes of the cell. For example, the cell shown in Figure 5(b) was specified with $\bar{G}_1$, $\bar{P}_1$ and $\bar{T}$ (ordered from left to right) as top-inputs coming in polysilicon, $\bar{G}_2$ and $\bar{P}_2$ (ordered from top to bottom) as right-inputs coming in metal, G, P and $\bar{T}$ (ordered from left to right) as bottom-outputs going out in polysilicon, and G and P (ordered from top to bottom) as left-outputs going out in metal. The functional description specified for the cell is: $G = \overline{\bar{G}_1 * \bar{P}_1 + \bar{G}_1 * \bar{G}_2}$; $P = \overline{\bar{P}_1 + \bar{P}_2}$ and $\bar{T} = \bar{T}$.

If the I/O nodes ordering along the cell boundaries is fixed, such as in our case, then the Symbolic Placement module will start by ordering product-terms within an AND-OR-INVERT function, and also of the drive-transistors within a product-term. Otherwise, the module will first generate a symbolic placement of the functions themselves. The ordering's goal is to minimize the cell's height by reducing the number of horizontal tracks needed to lay out the cell. In our example, we need to place the product-terms of function G ($\bar{G}_1 * \bar{P}_1$ and $\bar{G}_1 * \bar{G}_2$), function P ($\bar{P}_1$ and $\bar{P}_2$) and function $\bar{T}$ ($\bar{T}$), such that $\bar{G}_1$, $\bar{P}_1$ and $\bar{T}$ - which come in polysilicon - need not traverse any unnecessary vertical diffusion tracks. This is done by identifying the polysilicon input variable shared by both functions (here: $\bar{P}_1$) and ordering the product terms such that metal crossovers for the polysilicon input variables (to get over diffusion tracks) are minimized. The following table shows how this process is done:

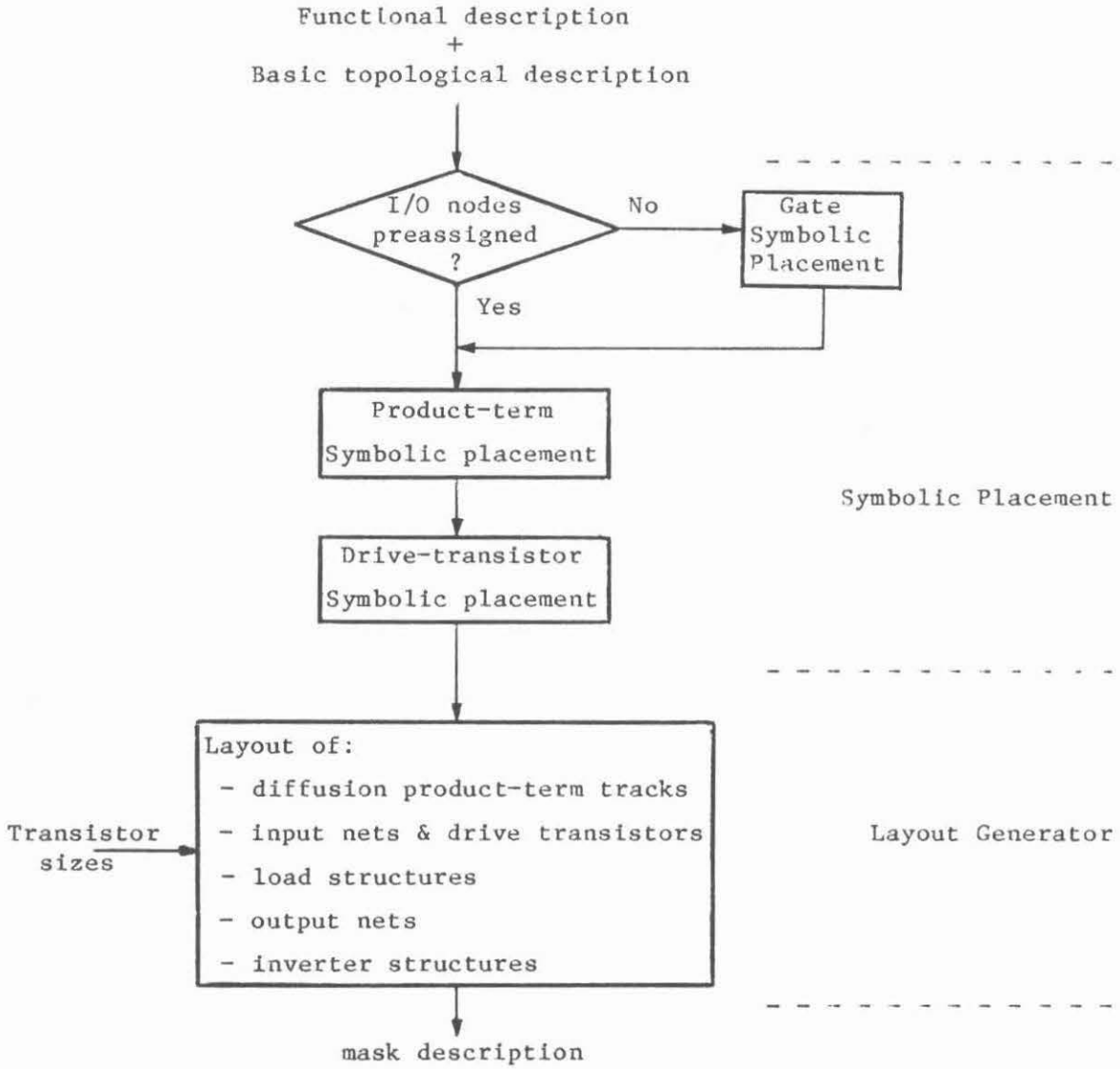


Figure 4. Block diagram of the Cell Layout modules.

		$\bar{G}_1$	$\bar{P}_1$	$\bar{T}$
G:	$\bar{G}_1 * \bar{P}_1$	1	1	0
	$\bar{G}_1 * \bar{G}_2$	1	0	0
P:	$\bar{P}_1$	0	1	0
	$\bar{P}_2$	0	0	0
$\bar{T}$:	$\bar{T}$	0	0	1

Before ordering

		$\bar{G}_1$	$\bar{P}_1$	$\bar{T}$
G:	$\bar{G}_1 * \bar{G}_2$	1	0	0
	$\bar{G}_1 * \bar{P}_1$	1	1	0
P:	$\bar{P}_1$	0	1	0
	$\bar{P}_2$	0	0	0
$\bar{T}$:	$\bar{T}$	0	0	1

After ordering

The output of the Symbolic Placement module is a table denoting relative placement of transistors on the reference grid system, and net-lists for the inputs and outputs. For our example, the table will be as follows:

$$\begin{array}{cccc} \bar{G}_1 & \bar{G}_1 & - & \bar{P}_2 \\ \bar{G}_2 & \bar{P}_1 & \bar{P}_1 & - \end{array}$$

where columns denote vertical diffusion tracks, and rows denote horizontal polysilicon tracks.

The Layout Generator uses the symbolic placement data to generate the masks, described in an intermediate form like the CIF [MeCo80]. It generates the rectangles necessary to lay out the masks: diffusion product-term "tracks", input nets and drive transistors, load structures, output nets, and inverter structures. Figure 5 shows the simulated layout of four types of cells used in our example.

Rather than predefining device parameters and then laying them out using a placement and routing scheme, the circuits are first laid out in an array-like structure with minimum device sizes. The electrical and geometrical parameters are passed on to the next module. Iteration of

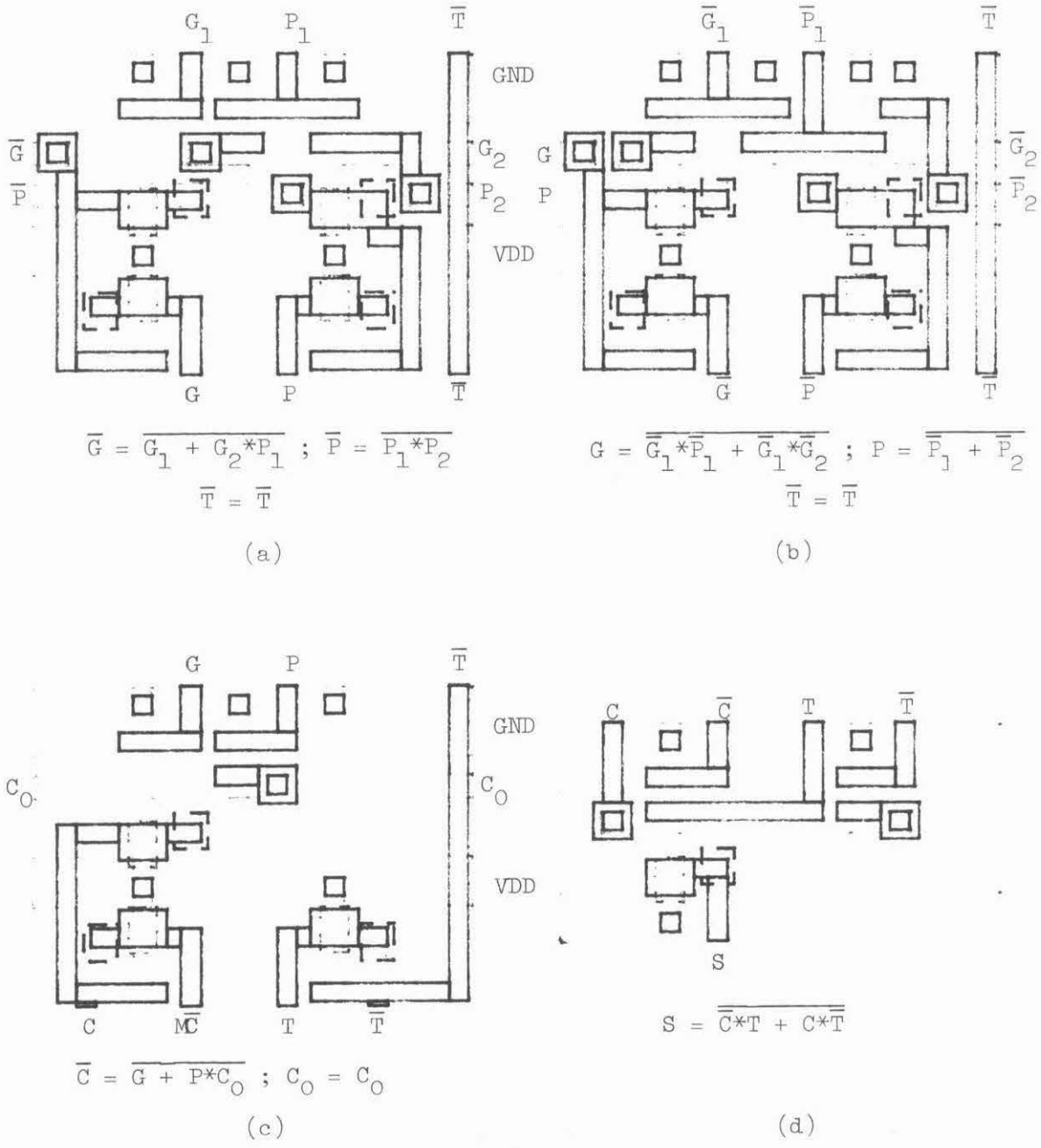


Figure 5. Layout of adder's basic cells:
(a) Type 2a cell; (b) Type 2b cell;
(c) Type 3 cell; (d) Type 4 cell.

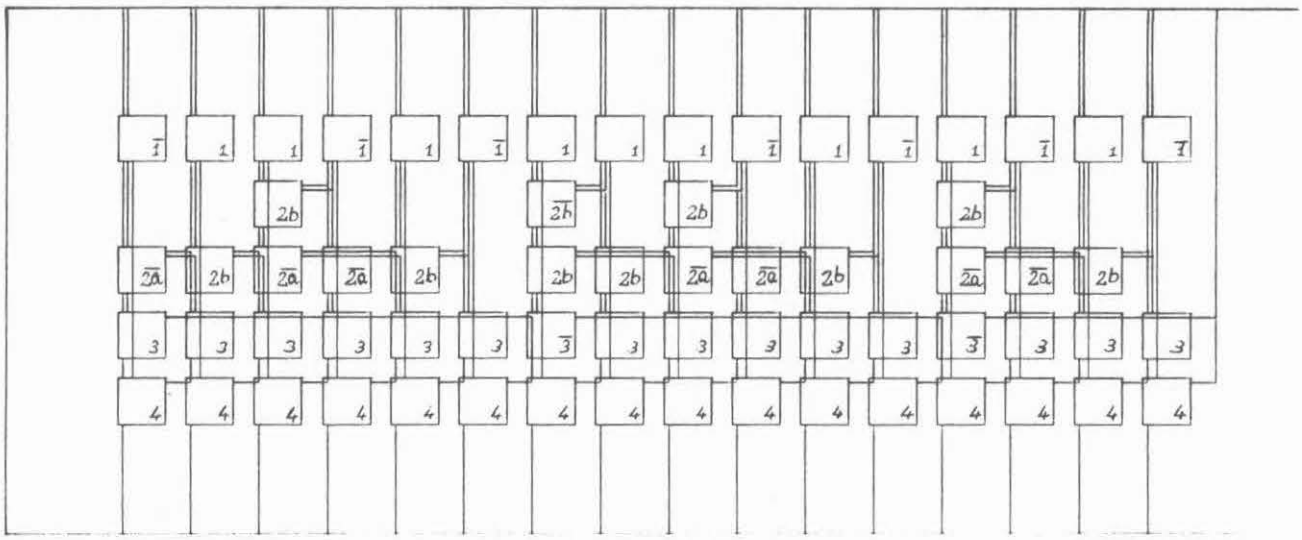
the process will produce the desired circuit with the device sizes necessary to meet the design goals.

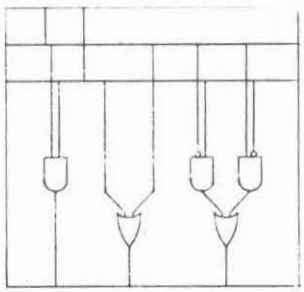
4. Structure Generator attempts to obtain the best possible structure for the given functional description and environmental parameters. It specifies the cell types, the position of each cell in the final layout and the interconnections between the cells.

Figure 6 shows the structure of a 16-bit binary adder. Each cell will be refered to as C[i,j], where i and j are the row and column where the cell is located, respectively, and the top rightmost cell is C[1,1]. Data are flowing only from top to bottom and from right to left. The four types of cells generated by the Cell Generator are located as follows: type 1 cells in the first row, type 2 in the second and third rows, type 3 in the fourth row and type 4 in the fifth row. The second, third and fourth rows perform the carry-look-ahead.

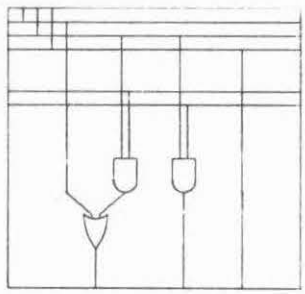
The input carry C(0) is fed into cells C[4,1] through C[4,4] which, together with the cells in the second and third rows above them, function as the carry-look-ahead for carries C(1) through C(4). Then the output of cell C[4,4] (which is C(4)) is fed into cells C[4,5] through C[4,10] that similarly produce the carries C(5) through C(10). Lastly, the output of cell c[4,10] is fed into the cells to its left, so C(11) through C(16) are produced.

Let us assume that each type of cell produces its outputs in the same time delay d after its inputs are stable. For this particular adder example it was also given that the sum S(I) has to be available 7d and the input carry C(0) is available 3d after the inputs A(I) and B(I) are stable. Also, the fanout is limited: each cell can drive at most 7 other cells. The structure shown in Figure 6 meets these constraints

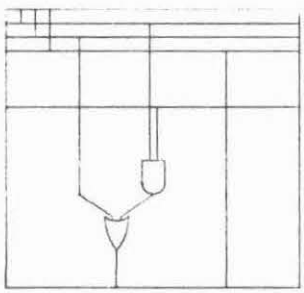




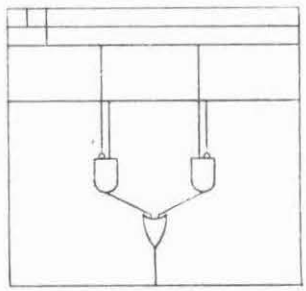
type 1 cell



type 2 cell



type 3 cell



type 4 cell

Figure 6. 16-bit adder structure and types 1, 2, 3 and 4 cells, in AND-OR form.

with a very important feature - it has the minimum number of rows, therefore it occupies minimum chip area (however, this structure is not unique).

Several paths through the structure have the maximum specified delay. They will be called critical paths (e.g C[1,6] → C[3,6] → C[3,8] → C[3,10] → C[4,10] → C[4,12] → C[5,14]). The functions that define each type of cell are evaluated by the Cell Generator in a sum of products form. Since in MOS technology (where this example is implemented) an AND-OR-INVERT logic is implemented more naturally then AND-OR, the complemented outputs are produced by each cell rather than the true ones. Inverting the outputs again is ruled out, since it almost doubles the delay time of each cell. For type 1 and 4 cells the double inversion problem is solved by modifying the functions to fit the complemented outputs. However, for type 2 and 3 cells this solution does not work, since these cells drive cells of the same type. Instead, two different subtypes of type 2 cell are defined: type 2a, which produces complemented outputs from its true inputs and type 2b, which produces true outputs from its complemented inputs. Now, cells along the critical paths are chosen to be of types 2a and 2b alternately. For type 3 cells, inverting the left output of C[4,4] and C[4,10] (that drive other type 3 cells) is unavoidable. Inverters must also be added to few type 1 and 2 cells in order to adjust their outputs to the driven cells. For these cells, only the outputs that drive the cells in the same column are inverted again, while the outputs that drive cells to the left remain unchanged. Since critical paths have already been taken care of, the adder speed does not degrade by these inverters. In Figure 6, cells that contain additional inverters have a bar added above their type number.

Conclusions

We have described the basic ideas behind a gate-to-silicon compiler by walking through a simple and well-known example. The compiler consists of four modules, each of which performs one step of the translation toward silicon level. The first translation is a crude approximation of the final layout, and therefore one or more iterations are needed to achieve a "near optimal" solution.

The novel approach in our compiler is based on (a) the set of synthesis procedures for decomposition of gate macros into small atomic cells and for optimization of obtained cellular structures with respect to environmental and technological parameters, and (b) the set of algorithms for automatic layout of different cell models obtained through decomposition of gate macros.

References

[BiGa80] Bilgory, A. and Gajski, D. D., "Automatic Cell Generation for Recurrence Structures" University of Illinois at Urbana-Champaign, Department of Computer Science, Report UIUCDCS-R-80-1040, November 1980.

[Joha79] Johannsen, D., "Bristle Blocks: A Silicon Compiler," Proc. 16th Design Automation Conf., pp 310-313, 1979.

[MeCo80] Mead, C. A., Conway, L. A., Introduction to VLSI Systems, Addison-Wesley, 1980.

[SOHT80] Shirakawa, I., Okuda, N., Harada, T., Tani, S. and Ozaki, H., "A Layout System for the Random Logic Portion of MOS LSI," Proc. 17th Design Automation Conf., pp 92-99, 1980.

[Verg80] Vergnieres, B., "Macro Generation Algorithms for LSI Custom Chip Design," IBM J. Res. Develop., Vol. 24, pp 612-621, 1980.