VLSI Complexity, Efficient VLSI Algorithms and the HILL Design System*

by Th. Lengauer [+] K. Mehlhorn [+]

FB 10
Universität des Saarlandes
D-6600 Saarbrücken

A83/03

July 1983

[+] Research partially supported by Deutsche Forschungsgemeinschaft under grant SFB 124, Teilprojekt B2

* This paper is a written summary of a sequence of talks given at the International Professorship in Computer Science: Algorithmics for VLSI, given by the second author. The International Professorship took place at the Université Catholique de Louvain, Nov. 82 - May 83 and was funded by IBM Belgium. The paper will appear in the proceedings of the International Professorship.

0. Introduction

1. A Complexity Theory of VLSI
   1. The VLSI Model
   2. Communication Complexity
   3. Extensions and Related Results

2. Efficient VLSI Algorithms

3. The HILL Design System
   1. HILL Layout Language and Graphics Editor
   2. Compaction in HILL
      1. Efficient Constraint Generation
      2. Efficient Constraint Resolution
      3. Hierarchical Compaction
   3. The HILL Simulator

0. INTRODUCTION

In this paper we discuss the relation of algorithms and VLSI in two ways: algorithms for VLSI design and algorithms implemented in VLSI.

In the third chapter we will discuss the main parts of the HILL design system which is under development at the university of Saarbrücken. The main parts are the layout specification language, the compacter and the switch-level simulator. The HILL layout language provides a convenient way of describing a layout symbolically either by a HILL program or by an interactive graphics session. A HILL program describes a layout at the level of stick diagrams enhanced by extensive means for structuring a design hierarchically. The level of stick diagrams was used previously in systems like CABBAGE, STICKS and MULGA. In contrast to HILL these systems are graphics oriented and can therefore support only limited mechanisms for structuring the design, mainly composition and simple iteration. In contrast to that HILL offers the full power of a high level programming language (HILL is a PASCAL extension); in particular it offers recursion, iteration and fully parameterized designs. Recursion and iteration are central to (software) algorithm design and we believe that they will be equally important for hardware design. We give some concrete examples below to support this claim. The compacter takes a stick diagram and produces compacted mask data from it. Compaction in HILL is constraint based. We will discuss how to extract a minimum system of constraints from the symbolic layout and how to solve constraint systems efficiently. The HILL simulator is a switch-level simulator. We discuss the underlying mathematical model of MOS circuit behavior and show how to derive an efficient simulation algorithm from the model. The simulator is correct with respect to the model.

VLSI design systems are used to implement digital systems, i.e. to realize algorithms in hardware. The second chapter is devoted to efficient VLSI algorithms for the basic arithmetic functions. In particular, we will describe a multiplier for n bit binary numbers which has area $A = O(n^2)$, delay $T = O(\log n)$ and period $P = O(1)$.

Is this a good design? We can infer from the first chapter on VLSI complexity theory that it is. One of the results derived in that chapter and due to [BK 81, V 80] is that $AP^2 = \Omega(n^2)$ for every chip which can multiply. More generally, we will present a theoretical model of VLSI computation and methods for deriving lower bounds on the complexity of concrete problems such as multiplication and addition. Moreover, we will compare the relative efficiency of various modes of computation, namely deterministic vs. randomized, in that model.

I. A COMPLEXITY THEORY FOR VLSI

The first chapter is devoted to a complexity theory for VLSI computations. It is based on a theoretical model of VLSI computations which captures the essential features of the technology, in particular its planarity, but abstracts from the technological details. With respect to this model we derive two types of results

a) lower bounds on the complexity of important functions, e.g. shift and multiplication

b) relations between different models of computation, e.g. deterministic vs. randomized computation, influence of the I/O-convention, influence of the propagation delay assumption.

VLSI complexity theory originates with Thompson's Ph.D. thesis [T 80] which contains theorems 1, 2 and 4 below. A number of researchers later extended his results. Specific references are given below.

1.1 The VLSI Model

Our VLSI model is based on Boolean circuits. This choice is adequate also for modelling more general "multi-directional" VLSI structures, e.g. buses [LM 81].

Definition 1: A *chip* $\chi = (\Gamma, \Lambda, \Delta)$ consists of three structures:

a) The *circuit* Γ: A synchronous Boolean circuit with feedback and unbounded fanin. Formally, this is a directed bipartite graph $\Gamma = (V,E)$ where V is partitioned into a set S of *switches* and a set W of *wires*. Here $S = P \cup G$ where P is a set of *ports* labelled in or out and G is a set of *gates* labelled and, or, nand, or nor. For $s \in W$, if $(s,w) \in E$ then w is called an *output* of s, if $(w,s) \in E$ then w is called an *input* of s. All gates have out-degree 1, all wires have in-degree 1. Each input port has one input and one output. Each output port has two inputs and no output. The "additional" input signal for the ports is an enable signal computed on the chip that activates the port.

b) The *layout* Λ: The layout maps every vertex in the circuit into a compact connected region in the plane. Furthermore, each point in the region lies inside some Cartesian square of side length $\lambda > 0$ that is completely contained in the region. (This provision models the finite resolution of the fabrication process for VLSI chips.) Each point in the plane belongs to the interior of at most $\nu \geq 2$ regions.

(The parameter ν is another fabrication process specific constant representing the number of functional *layers* on the chip. Since $\nu \geq 2$ also non-planar circuits Γ can be laid out.) Two regions touch exactly if the vertices they represent are neighbours in Γ. (We say that regions R_1 and R_2 touch if $R_1 \cap R_2 \neq \emptyset$ but $R_1^o \cap R_2^o = \emptyset$, where R_1^o, R_2^o are the interiors of R_1 resp. R_2.)

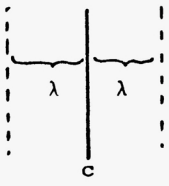
c) The *manual* Δ: The manual is a set of directions for the communication between the chip and is environment. It contains for every input port a sequence of numbers that identify the input bits that enter through this port. The sequence also determines the order in which the input bits enter. When its enable signal is raised the port requests the next input bit in the sequence to enter. Analogously, for each output port the manual contains a sequence of numbers identifying the output bits produced at that port and their order. When its enable signal is raised the port produces the next output bit in the sequence.

After defining all components of the chip we can define the operation of the circuit. To this end we associate with each port a word $w \in B^*$, $B = \{0,1\}$, that we call its *history*. Furthermore, we label each wire with an initial Boolean value from $B \cup \{X\}$. (X stands for the undefined Boolean value, $0 \wedge X = 0$, $1 \vee X = 1$, $0 \vee X = 1 \wedge X = X$.) Such a labelling we call a *state* of the circuit. The initial state is most often the completely undefined state. In the i-th cycle the circuit does the following: Each gate "reads" the values on all its input wires and computes the Boolean operation given by its label. The resulting value is put on its output wire. Each input port puts an X on its output wire if its enable signal is 0, otherwise it puts the next bit from its history on its output wire. Each output port checks if its enable signal is 1, and if so it puts its other input at the end of its history.

Thus input ports consume their histories and output ports produce them. All actions of the gates happen in parallel. Thus a new state is reached on which the (i+1)-st cycle of the computation is started.

A VLSI computation uses up computational resources. We are interested in area A, time T and switching energy E. Area A is the area of the smallest rectangle which encloses layout Λ and T is the number of steps taken by the circuit to produce the desired outputs. Alternative definitions of area and time and a definition of switching energy are discussed in 1.3.

We are now in a position to outline the argument for proving lower bounds on the AT^2 complexity of VLSI chips. Let us consider a chip computing some Boolean function $f : B^n \to B^m$. Let R be a smallest enclosing rectangle, let a, b be the side lengths of R, $a \leq b$. Then $A = a \cdot b \geq a^2$. Let us assume furthermore that the chip has n input ports and that a unique input bit is assigned to each port. Clearly, we can cut the chip into two halves L and R by a line C parallel to the side of length A of the chip, such that about half the input ports lie on either side of the cut. Then C has length $a \leq \sqrt{A}$ and hence at most $(2\nu/\lambda)\sqrt{A}$ circuit components can intersect C. This can be seen as follows. Consider a strip of width 2λ with center C. If a circuit



component intersects C then it has a square of area at least λ^2 in common with the strip. Since any point of the strip belongs to at most ν (regions associated with) circuit com-

ponents we conclude that $h\lambda^2 \leq a\, 2\lambda\, \nu$ where h is the number of circuit components intersecting C. Thus $h \leq 2a\nu/\lambda$. We conclude further (and this is made precise in theorem 1 below) that at most $h \leq 2\nu/\lambda \sqrt{A}$ bits of information can cross C in any clock cycle.

Suppose now that we can show that w bits of information have to cross cut C in order to allow successful computation of f. (We will see below how such a claim can be shown.) Then the computation of f must take at least ω/h clock cycles, i.e. $T \geq \omega/h \geq \omega\lambda/2\nu\sqrt{A}$ or $AT^2 \geq (\lambda^2/4\nu^2)\, \omega^2$.

This concludes the basic lower bound argument. We will fill in the details in the next section.

1.2 Communication Complexity

The lower bound of the preceding section is based on the cost of communication in VLSI computations. We will therefore study communication complexity of Boolean functions in somewhat more detail in this section.

Let $f: X \times Y \to A \times C$ be a function. We consider the following scenario. There are two computing agents L and R. Initially, L knows $x \in X$ and R knows $y \in Y$. They now want to cooperatively compute $f(x,y) = (a,c)$ by exchanging information between each other. More precisely, L sends a bit depending on x to R, R returns a bit depending on y and the bit just received, ... until L knows a and R knows c.

Definition [Yao 79]: A deterministic algorithm is given by two response functions $r_L : X \times B^* \to B$ and $r_R : Y \times B^* \to B$ and two partial output functions $out_L : X \times B^* \to A$, $out_R : Y \times B^* \to C$, where $B = \{0,1\}$. A computation on input x,y is a sequence $w = w_1 w_2 \ldots w_k$ of bits such that

1) $w_{2i+1} = r_L(x, w_1 \ldots w_{2i})$ for $i \geq 0$

2) $w_{2i+2} = r_R(y, w_1 \ldots w_{2i+1})$ for $i \geq 0$

3) $(x, w_1 \ldots w_k) \in \mathrm{dom}(\mathrm{out}_L)$ and

$(y, w_1 \ldots w_k) \in \mathrm{dom}(\mathrm{out}_R)$

4) there is no shorter sequence with this property

k is the length of the computation w and is denoted k(x,y). An algorithm is correct if $f(x,y) = (\mathrm{out}_L(x,w), \mathrm{out}_R(x,w))$, where w is the computation on input x,y, for all $x \in X$, $y \in Y$. The complexity of an algorithm Alg is defined as

$$C(Alg) = \max\{k(x,y); x \in X, y \in Y\}$$

Finally, the complexity of f is defined by

$$C_{def}(f) = \min\{C(Alg); Alg \text{ computes } f\}$$

For the definitions above we assumed that the partition of the inputs and outputs into left and right inputs and outputs is part of the problem specification. This assumption is quite reasonable in applications to distributed computing in general, it is, however, too restrictive for VLSI computations. Note that in VLSI a problem is given as a Boolean function $f: B^n \to B^m$. It is up to the chip designer to fix the locations on the chip where certain inputs are consumed and certain outputs produced. We therefore define:

Definition: Let $f : B^n \to B^m$ and let $\omega \geq 0$. Function f is **ω-separable** if for all balanced partitions X, Y of [1...n], i.e. $n/3 \leq |X|, |Y| \leq 2n/3$, and all partitions A,C of [1...m] we have $C_{def}(f_{X,Y,A,C}) \geq \omega$, where $f_{X,Y,A,C} : B^X \times B^Y \to B^A \times B^C$ is defined by partitioning the input and output bits into left and right input and output bits as given by partitions X,Y and A,C.

□

We can now formalize the first part of the lower bound argument. In the form given here, theorem 1 emerged over a sequence of papers ([T 80, BK 81, V 80, S 81, LS 81, K 82]).

Theorem 1: If $f: B^n \to B^m$ is ω-separable then $AT^2 \geq (R^2/16\, r^2)\, \omega^2$ for every chip computing f.

Proof: Let $\chi = (\Gamma, \Lambda, \Delta)$ be a chip computing f. Let π be an input port and let R_π be its associated region in the layout Λ. Let the input bit x_i enter the chip through input port π. With each such x_i we associate a point p_i in the interior of R_π such that different points are associated with different input bits. For the purposes of the lower bound proof we will consider the bit x_i to enter the chip through point p_i. Similarly, we associate a point q_i with every output bit y_i.

Let Q be a smallest area rectangle enclosing layout Λ. Then Q has side lengths a,b. Assume $a \leq b$. We can cut Q into halves L and R by a cut C parallel to the side of length a such that exactly half of the p_i's lie to the left of cut C. Cut C gives rise to a balanced partition X,Y of the input bits and a partition A, C of the output bits in a natural way. Let $f' = f_{X,Y,A,C}$ be the function induced by these partitions. Since f is ω-separable we conclude that $C_{det}(f') \geq \omega$.

Since cut C has length at most a, at most $h \leq (2r/\lambda)\sqrt{A}$ regions of Λ associated with components of Γ can intersect C. We will now derive from the chip an algorithm Alg for f' with $C(Alg) \leq 2Th$

Consider computation of f by χ. At each cycle we associate two values with C, a left and a right "crossing" value. The left (right) crossing value $v^\ell = (v_1^\ell, \ldots, v_h^\ell)$ $(v^r = (v_1^r, \ldots, v_h^r)$ contains a component $v_i^\ell (v_i^r)$ for each region R_i intersecting C.

If R_i is a (nand,nor,and,or) gate then $v_i^\ell(v_i^r)$ is the (nand,nor, and,or) of all input values during the last cycle whose regions intersect L(R). If R_i is a wire then $v_i^\ell(v_i^r)$ is the above value for its input gate. Ports act as and-gates in this context.

With these definitions the computation of f by χ can be regarded as a deterministic algorithm for computing f' in the sense of the definition above. The information exchanged between L and R are the crossing values. The left crossing values are sent from L to R, and the right crossing values are sent from R to L. The computation is completed when both side have produced their outputs. Since a total of 2k bits are exchanged in every cycle the algorithm described above has complexity 2Th.

Since $2Th \geq C(Alg) \geq C_{det}(\delta') \geq \omega$ we conclude $AT^2 \geq (\lambda^2/16v^2)\omega^2$. □

We will next derive methods for proving lower bounds on the communication complexity of functions. We will discuss the crossing sequence method for multiple output functions and the rank method for single output functions.

Method 1 (Crossing Sequences for Multiple Output Functions):

Definition: Let $f: X \times Y \to B^m$ and let I, J be a partition of the output bits of f. f has ω-flow if there is partial input $y \in Y$ such that f restricted to $X \times \{y\}$ in its domain and J in its range has more than $2^{\omega-1}$ different points in its range.

Theorem 2: If f has ω-flow then $C_{det}(f) \geq \omega$.

Proof: Assume that there is a deterministic algorithm computing f that has a communication length of less than ω. Then for two inputs (x_1,y), (x_2,y) generating different output configurations in J the same communication sequence w is generated. Thus for some $j \in J$ $f_j(x_1,y) \neq f_j(x_2,y)$, but the algorithm computes $f_j(x_1,y) = out_{R,j}(y,w) = f_j(x_2,y)$. Here f_j is the j-th bit of function f and

similarly for output function f_R. Thus the algorithm does not compute f correctly, a contradiction. □

[V80] gives an example of a class of functions to which method 1 applies.

Definition 2: Let $f(x_1,\ldots,x_n,s_1\ldots,s_m) = (y_1,\ldots,y_n)$ be a Boolean function. f computes a permutation group G on n elements if for all $g\in G$ there is an assignment $a_1,\ldots,a_m$ to the $s_1,\ldots,s_m$ such that $f(x_1,\ldots,x_n,a_1,\ldots,a_m) = (x_{g(1)},\ldots,x_{g(n)})$ for all choices of $x_1,\ldots,x_n$. We call $x_1,\ldots,x_n$ the permutation inputs and $s_1,\ldots,s_k$ the control inputs. f is called *transitive* of degree n if G is a transitive group, i.e., if for all $i,j=1,1,\ldots,n$ there is a $g\in G$ such that $g(i) = j$.

The most straightforward example of a transitive function of degree n is the cyclic shift function $cs(x_1,\ldots,x_n,s_1,\ldots,s_m) = (y_1,\ldots,y_n)$ where $n = 2^m$ and the $s_1,\ldots,s_m$ encode a number k, $0 \le k < n$ and $y_1 = x_{(i+k) \bmod n}$. cs computes the transitive group of cyclic permutations. Other examples of transitive functions of degree $\Omega(n)$ are the multiplication of n-bit integers, the multiplication of three $\sqrt{n}x\sqrt{n}$ matrices, the sorting of n numbers between 0 and n etc.

Theorem 3: Let $f: B^{n+m} \to B^n$ be transitive of degree n. Then f is n/6-separable.

Proof: Let G be the transitive group computed by f. The equivalence relation $g(i) = h(i)$ for fixed but arbitrary $i\in\{1,\ldots,n\}$ divides G into n equivalence classes of size $|G|/n$. Let A be the set of all permutation input bits and let B be the set of all output bits. Let X, Y be any partition of $A \cup B$ such that $|X|,|Y|\le 4n/3$. For each input bit i in X and output bit j in Y there are $|G|/n$ group elements $g\in G$ such that $g(i) = j$. Let w.l.o.g. X be no greater than Y assume that X contains at least as many input bits

as output bits. (The other cases can be argued similarly.) Let S be the set of input bits in X and S' be the set of output bits in Y. Then

$$|S| \cdot |S| \geq n/3 \cdot n/2 = n^2/6$$

For each of the pairs $(i,j) \in S \times S'$ there are G/n group elements matching them. Since there are only a total of $|G|$ group elements there must be one element $g_o \in G$ realizing at least $n/6$ matchings between inputs in S and outputs in S'. The partial input y realizing the flow sets the control input bits in Y such that together with appropriate assignments to the control input bits in X they encode this element g_o. The other input bits in Y are assigned arbitrarily. □

Theorem 4: ([T 80, BK 81]) There is a constant $c > o$ such that for every chip computing the cyclic shift of n inputs or multiplying n bit binary numbers $AT^2 \geq cn^2$.

Proof: It was shown above that the cyclic shift function is transitive of degree n. Thus the claim follows from theorems 1 and 2. In order to extend the result to multiplication we only have to notice that multiplication by a power of two is a shift. □

Theorem 4 is quite significant because it states a lower bound on AT^2 for two very important functions: cyclic shift and multiplication. The $AT^2 = \Omega(n^2)$ lower bound is a yardstick against which one can measure actual design. This will be done in chapter 2.

We will now turn to the rank method for proving lower bounds on the communication complexity of boolean predicates.

Method 2 (The Rank Lower Bound for Boolean Predicates):

Let $p: X \times Y \to B$ be a predicate. With $A = B$ and $C = \{1\}$ we can

use the definitions above to define $C_{det}(p)$. There are two methods for proving lower bounds on $C_{det}(p)$: the crossing sequence method and the rank method. The former method is older, easier to apply, and similar to the method used to prove theorem 2. Since the rank method is more general we will describe it here. The rank method was developed in [MM 82].

Definition: Let r be a ring and let $r^{(n,m)}$ be the set of n x m matrices over r. The *rank* of $A \in r^{(n,m)}$ over r is the minimum k such that A can be written as $A = C \cdot D$, where $C \in r^{(n,k)}$ and $D \in r^{(k,m)}$. We use N to denote the ring of integers.

If r is a field the above definition coincides with the definition of matrix rank known from linear algebra. Method 2 is based on the following theorem.

Theorem 5: Let $p: X \times Y \to B$ be a boolean predicate, and let P be its associated matrix, i.e. P is an $|X|$ by $|Y|$ matrix with $P_{x,y} = p(x,y)$. Then

$$C_{det}(p) \geq \log \text{rank}_N(P) \geq \log \text{rank}_r(P)$$

where r is any field.

Proof: The second inequality is known from algebra. For the proof of the first inequality we state the following lemma.

Lemma 1: Let r be a ring, $A \in r^{(n,m)}$, $B \in r^{(n,k)}$, $C \in r^{(k,m)}$. Then

$$\text{rank}_r((A\ B)) \leq \text{rank}_r(A) + \text{rank}_r(B)$$

$$\text{rank}_r\left(\begin{pmatrix} A \\ C \end{pmatrix}\right) \leq \text{rank}_r(A) + \text{rank}_r(C)$$

Proof: If $A = D_1 \cdot E_1$ and $B = D_2 \cdot E_2$ then

$$(A\ B) = (D_1\ D_2) \cdot \begin{pmatrix} E_1 & 0 \\ 0 & E_2 \end{pmatrix}.$$

The proof of the second inequality is analogous. □

Now consider any deterministic algorithm for computing p. Inductively on the length of $w \in B^*$ we define the matrix P_w as follows:

$$|w| = 0: \; P_\varepsilon := P$$

$|w| > 0$: If $|w| = 2\ell$ then $P_{w0}(P_{w1})$ is obtained P_w by selecting all rows x with $r_L(x,w) = 0$ $(r_L(x,w)=1)$.
If $|w| = 2\ell + 1$ then $P_{w0}(P_{w1})$ is obtained from P_w by selecting all columns y with $r_R(y,w) = 0$ $(r_R(y,w)=1)$.

By Lemma 1 we have $\max(\text{rank}_N(P_{w0}), \text{rank}_N(P_{w1})) \geq \text{rank}_N(P_w)/2$. Moreover, if $\text{out}_R(x,w)$ is defined then rank $P_w \leq 1$ since P_w must consist of a set of rows which are constant 0 and a set of rows which are constant 1. Thus there are $x \in X$, $y \in Y$ such that the computation of x, y has length at least $\log \text{rank}_{|N}(P)$. □

We will next give two applications of theorem 5.

Theorem 6: Let $X = Y$ and let $p(x,y) = (x=y)$ be the identity predicate. Then $C_{det}(p) \geq \log |X|$.

Proof: Clearly, P is the identity matrix and hence $\text{rank}_{|N}(P) \geq |X|$.

The second example is less trivial and illustrates the fact that randomization helps in distributed computing and in VLSI. In Las Vegas computations computing agents L and R have fair coins available to them. The response of an agent, say L, depends on his argument, on the history of the computation and on the outcome of a toss of the coin. Correctness of an algorithm is defined as above, i.e. the output of the computation must be independent of the outcomes of the coin tosses. The complexity of an algorithm on input x, y is the expected number of bits exchanged. A precise definition is given by:

Definition: A Las Vegas algorithm is given by two response functions $p_L: X \times B^* \times B^t \to B$ and $p_R: Y \times B^* \times B^t \to B$ and the partial output function $a: B^* \to B$. We assume that both L and R first toss t coins to determine the third arguments t_L, t_R in the response functions, and then start a deterministic computation. The computation ends when $(w_1,\ldots,w_{k(x,y,t_L,t_R)}) \in \mathrm{dom}(a)$. Its result is $a(w_1,\ldots,w_k)$.
The Las Vegas communication complexity of p is

$$C_{LV}(p,L\Leftrightarrow R) = \min_{\substack{A \\ LV\text{-alg}}} \sum_{t_L,t_R \in B^t} k(x,y,t_L,t_R)/2^{2t}$$

We consider the following example:

Definition Let $n \in N$ and $X = Y = [0:2^n-1]^n$.
For $x = (x_1,\ldots,x_n) \in X$
and $y = (y_1,\ldots,y_n) \in Y$

let $$p_1(x,y) = \begin{cases} 1 & \text{if } x_i = y_i \text{ for some } i,\ 1 \le i \le n \\ 0 & \text{otherwise} \end{cases}$$

Theorem 7: a) $C_{det}(p_1) \geq n^2$

b) $C_{LV}(p_1) = O(n(\log n))$

Proof: a) Since $P_1 \in B^{2^{(n^2)}}$ we only have to show that $\mathrm{rank}_{GF(2)} P_1 \geq 2^{(n^2)}$, where GF(2) is the field of characteristic 2. Let $\oplus$ denote addition modulo 2. We transform the matrix $\overline{P}_1$ associated with $\overline{p}_1$ into the identity matrix of size $2^{(n^2)} \times 2^{(n^2)}$ by means of linear transformations.

Lemma 2: Let $w_1,\ldots,w_n,y_1,\ldots,y_n \in [0:2^n-1]$. Define

$$g(w_1,\ldots,w_n,y_1,\ldots,y_n) := \sum_{\substack{x_1 \\ x_1 \neq w_1}} \cdots \sum_{\substack{x_n \\ x_n \neq w_n}} \bar{p}_1(x_1,\ldots,x_n,y_1,\ldots,y_n)$$

Then $g(w_1,\ldots,w_n,y_1,\ldots,y_n) = id(w_1,\ldots,w_n;y_1,\ldots,y_n)$.

Proof: Note that

$$g(w_1,\ldots,w_n,y_1,\ldots,y_n) = |\{x_1,\ldots,x_n);x_i \neq w_i, x_i \neq y_i\}| \bmod 2$$

$$= \prod_{i=1}^{n} (2^n - |\{y_i,w_i\}|) \bmod 2$$

$$= 1$$

iff $y_i = w_i$ for all i □

We conclude from lemma 2 that $\text{rank}_{GF(2)} \bar{p}_1 = 2^{(n^2)}$ and hence $C_{det}(p_1) = C_{det}(\bar{p}_1) \geq n^2$

b) The Las Vegas algorithm for p is based on the following simple number-theoretic fact.

Fact: Let $p_1,p_2,\ldots,p_m$ be the set of primes $\leq n$.
Let $0 \leq x, y \leq 2^n - 1$. If $x \neq y$ then
$|\{i;\ x \bmod p_i \neq y \bmod p_i\}| \geq m/2$

The algorithm looks as follows:

```
for i from 1 to n
do for k from 1 to log n
   db  L  selects a prime p_j from the list of
       primes p_1,...,p_m ≤ n  at random and sends
       (p_j, x_i mod p_j)  to  R;
       R computes y_i mod p_j;
       if x_i mod p_j ≠ y_i mod p_j
       then goto nexti fi
   od
```

L sends x_i to R;
if $x_i = y_i$ then halt and output 1;
nexti :
od
halt and output 0;

The algorithm above is clearly correct. Also note that if $x_i = y_i$ then L will send $O((\log n)^2 + n)$ bits to R until this fact is detected. Observe that case $x_i = y_i$ occurs most once. If $x_i \neq y_i$ then $O(k \log n)$ bits are sent from L to R with probability 2^{-k} and an additional n bits are sent with probability $2^{-\log n} = 1/n$. Thus an expected number of $O(\sum_{k \geq 1} k \cdot 2^{-k} \log n + 1/n \cdot n) = O(\log n)$ bits are sent.

Hence $C_{LV}(p_1) = O(n \log n)$ □

Theorem 7 is quite significant. For predicate p_1 randomization provably reduces the amount of communication required by almost a square root. How about chip complexity? Note first, that it is conceivable to incorporate random devices into VLSI chips. Such a device might use statistical physical effects to produce (true?) random sequences. Let us assume that we can build a device which uses area O(1) and produces a random bit in time O(1). A predicate similar to p_1 can be used to show (cf. [MM 82] for details).

Theorem 8 : There is a predicate $p: B^n \to B$ such that $AT^2 \geq cn^2$ and $(AT^2)_{\text{Las Vegas}} \leq cn^{3/2}(\log n)^3$ for some constant c.

3. EXTENSIONS AND RELATED RESULTS

In this section we will briefly mention some extensions and some related results.

3.1 Area

We defined the area A of a chip as the area of the smallest enclosing rectangle. Alternatively and more naturally we might define A as the area of the union of the regions associated with circuit components. Let us call this area the active area. In [LM 81] it is shown that all AT^2 lower bounds are valid with area replaced by active area.

3.2 The Manual and Lower Bounds on Area

The manual is a set of directions for the communication between the chip and its environment. Manuals as defined above were termed strongly where-oblivious manuals in [LM 81]. A more restricted class of manuals are the when- and where-oblivious manuals. In these manuals the location and the time at which a bit enters the chip is independent of the input. We have

Theorem 9: Let f be a transitive function of dgeree n.

a) [V 80] A chip for f has area $A = \Omega(n)$ if the manual is where- and when-oblivious.

b) [LM 81] A chip for f has area $A = \Omega(n^{1/3})$ if the manual is strongly where-oblivious.

3.3 Period

The period P of a chip is the least distance in time between distinct problem instances which can be fed into the chip. Vuillemin [V 80] has shown that the $AT^2 \geq n^2$ lower bound of theorem 4 can be straightened to $AP^2 = \Omega(n^2)$. Baudet [B 81] has shown that $AP = \Omega(n + n \log n/T)$ for every chip realizing binary addition.

3.4 Energy

Switching energy E is another important computational resource. We assume that every unit of active chip area consumes one unit of switching energy each time it changes its state from 0 to 1 or vice versa.

This complexity measure is closely related to the energy dissipated when charging a wire. In technologies without high d.c. currents the switching energy dominates the total energy dissipation on the chip. We have

Theorem 10 [LM 81]: Let f be a transitive function of degree n. Let E be the worst case switching energy consumed by any chip computing f. Let A be the (active) chip area and let T be the worst case computing time. Then

$$c_1 AT^2 \geq ET \geq \frac{c_2 n^2}{\log \frac{c_3 AT^2}{n^2}} \geq 0$$

for appropriate constants $c_1, c_2, c_3 > 0$.

3.5 Propagation Delay and the Notion of Time

We defined time T as the number of clock cycles spent on the computation. Note that this is not a "physical" measure of time, since the length of a clock cycle may itself vary with the size of the chip. However, as long as the delay of signal propagation along wires is not significantly longer than the delay of the switching elements the number of clock cycles gives a good representation of the time spent, i.e., is asymtotically accurate.

As the size of chips increases this ceases to be the case, at least if the driving capacity of transistors driving long wires is not increased appropriately. [CM 81] introduce a physical time measure T by measuring time in seconds under the assumption that signals are propagated along wires at a constant speed. They get dramatically different lower bounds on circuit complexity. Not only their lower bounds are larger, as we expect, but the optimal chips according to their complexity measures differ significantly from the optimal chips if T is measured in clock cycles. This is, because [CM 81] pay a penalty for long wires across which communication is expensive. The time measure of [CM 81] may become technologically significant eventually, as the speed of light becomes the limiting factor in signal propagation on VLSI chips. In the meantime, however, other physical time measures may be more appropriate, such as the one introduced in [MC 80] that is based on the capacitive properties of VLSI structures. In this model the switching time of a transistor is given by the ratio of the capacity to be driven and the size of the driving transistor. No non-trivial lower bound results have been shown for this model as of today. However, in many cases optimal designs for the unit delay model carry over to the capacity model.

II. EFFICIENT VLSI ALGORITHMS

In chapter I we derived lower bounds on the complexity of the basic arithmetic functions. In particular,

$AP^2 = \Omega(n^2)$, $T \geq P$ for binary multiplication,

$AP = \Omega(n + n \log n/T)$, $T \geq P$ for binary addition.

In both cases there are designs whose performance matches the lower bound.

Theorem 1: a) ([PV 81]) For $\sqrt{n} \geq T \geq (\log n)^2$ there is a chip for binary multiplication with $AT^2 = O(n^2)$, $P = T$

b) ([BK 81]) For $n \geq T \geq \log n$ there is a chip for binary addition with $AP = O(n + n \log n/T)$.

We will not go into these constructions here. Rather we will describe a fast $T = O(\log n)$ and area efficient multiplier. More precisely, we describe

a) a chip with $A = O(n^2)$, $T = O(\log n)$, $P = O(1)$, i.e. $AP^2 = O(n^2)$

b) a chip with $P = T = O(\log n)$ and $A = O(n^2/\log n)$.

Previously, only designs with $A = O(n^2 \log n)$, $T = O(\log n)$ and $P = O(1)$ were known ([V 83], [B 82]).

Our design is based on the following well-known identity due to Karatsuba. Let $a = a_1 2^{n/2} + a_2$, $b = b_1 2^{n/2} + b_2$ be two n bit integers; a_1, a_2, b_1, b_2 are n/2 bit integers. Let $h = (a_1+a_2)(b_1+b_2)$. Then we have $c = a \cdot b = a_1 b_1 2^n + (h - a_1b_1 - a_2b_2)2^{n/2} + a_2b_2$. Thus multiplication of 2 n-bit numbers can be reduced to 3 multiplications of n/2 bit numbers and a few additions.

Let $T(n)$ be the delay of a network based on the identity described above. We have

$$T(n) = T(n/2) + O(A(n))$$

where $A(n)$ is the delay of the adder used. If one uses a fast adder, e.g. a carry lookahead adder, then $A(n) = O(\log n)$ and hence $T(n) = O((\log n)^2)$. A design based on these principles can be found in [L 81].

We add one more idea to Luk's design: a redundant number of representation. Then $A(n) = O(1)$ and hence $T(n) = O(\log n)$ if

the representation is chosen appropriately. Redundant representations were also used in [B 82, V 83].

For $a_0,\ldots,a_{n-1} \in \{-1,0,1\}$ let $\text{num}(a_{n-1},\ldots,a_0) = \sum_{i=0}^{n-i} a_i 2^i$;

i.e. we represent nunbers with digits -1, 0, +1. We will next show how to add two numbers in this respresentation in time O(1).

Let $a_0,\ldots,a_{n-1},b_0,\ldots,b_{n-1} \in \{-1,0,1\}$ and let $a = \text{num}(a_{n-1},\ldots,a_0)$, $b = \text{num}(b_{n-1},\ldots,b_0)$ and $c = a + b$.

Note first that $a_i + b_i \in \{-2,-1,0,+1,+2\}$.

Next write $a_i + b_i$ as $2n_i + s_i$ where n_i and s_i are given by the following table:

a_i+b_i	n_i	s_i	
2	1	0	
1	0	1	if $a_{i-1} + b_{i-1} < 0$
	1	-1	if $a_{i-1} + b_{i-1} \geq 0$
0	0	0	
-1	0	-1	if $a_{i-1} + b_{i-1} > 0$
	-1	+1	if $a_{i-1} + b_{i-1} \leq 0$
-2	-1	0	

Finally, define $c_i = n_{i-1} + s_i$, $0 \leq i \leq n$.

Lemma: a) $c_i \in \{-1,0,+1\}$

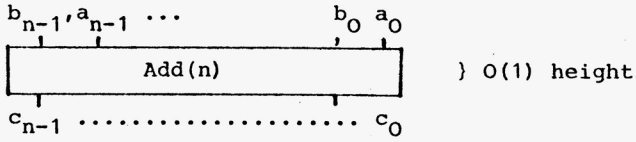
b) $c = a + b = \text{num}(c_n\ c_{n-1} \ldots c_0)$

Proof: a) Note first that n_{i-1}, $s_i \in \{-1,0,1\}$. If $a_i + b_i \neq \pm 1$ then $s_i = 0$ and hence $c_i \in \{-1,0,1\}$. So let us assume that $a_i + b_i = +1$, the case $a_i + b_i = -1$ being symmetric. If $a_{i-1} + b_{i-1} < 0$ then $u_{i-1} \leq 0$ and $s_i = 1$. Hence $c_i \in \{0,1\}$.

If $a_{i-1} + b_{i-1} \geq 0$ then $u_{i-1} \in \{0,1\}$ and $s_i = -1$. Hence $c_i \in \{-1,0\}$. In either case we have shown that $-1 \leq c_i \leq 1$.

b) obvious □

We represent digits in $\{-1,0,+1\}$ by two bits. More precisely, we use representation in 1-complement, i.e. encoding $(+1) = (0,1)$, encoding $(0) = (0,0)$, encoding $(-1) = (1,0)$. Then it is easy to design an adder cell for n digit numbers with height O(1) and width O(n)



Using this adder cell we can build up a multiplier as shown in the diagram below. A conventional multiplier is used for small n, say $n \leq 8$. Let H(n) (W(n)) be the vertical (horizontal) extension of that layout. Then

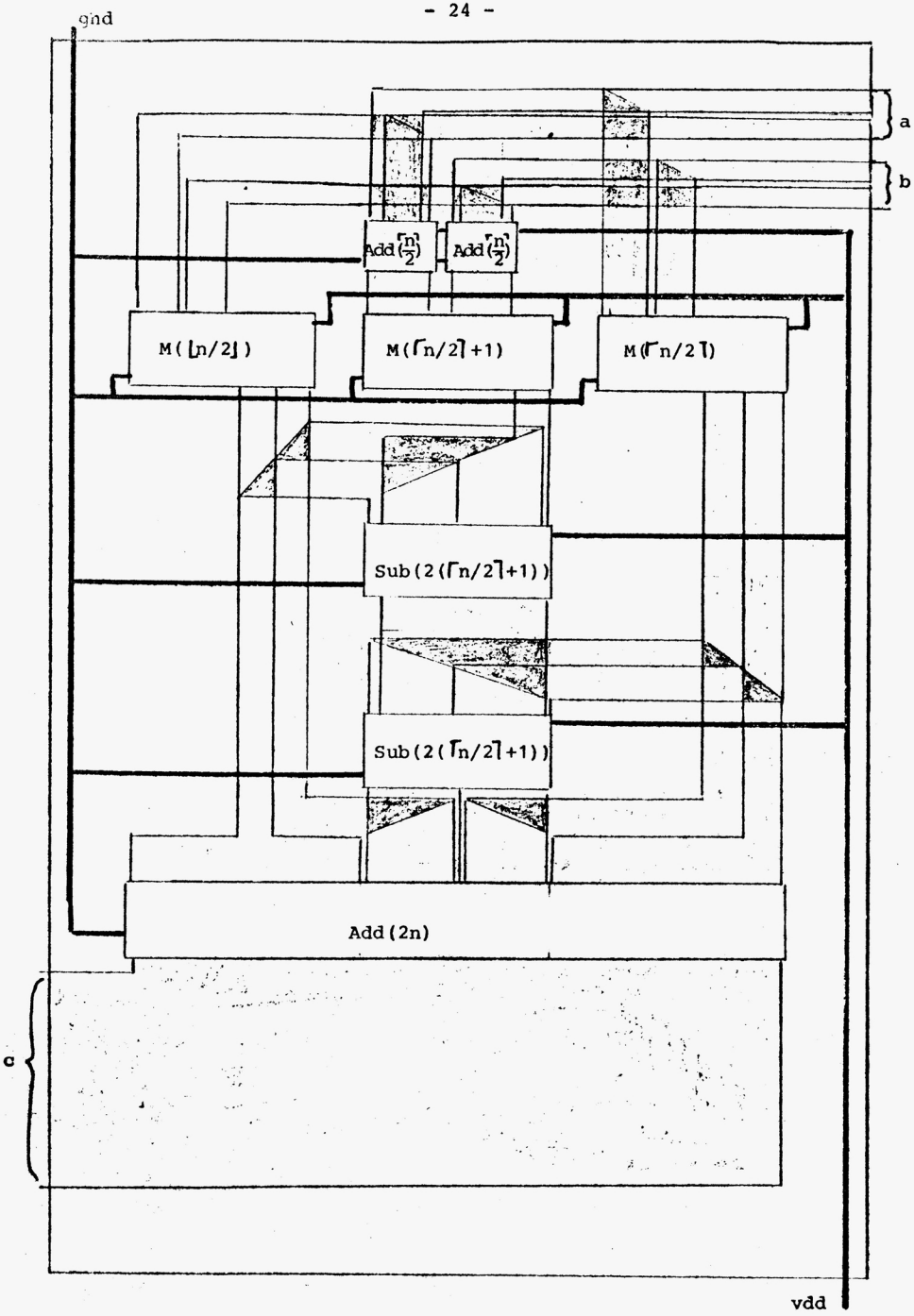
$$H(n) = O(n) + 3W(\lceil n/2 \rceil + 1) \quad \text{and}$$

$$W(n) = O(n) + H(\lceil n/2 \rceil + 1)$$

also $H(n) = W(n) = cn$ for $n \leq 8$ and some appropriate constant c.

We conclude $H(n) = O(n) + 3H(\lceil (\lceil n/2 \rceil + 1)/2 \rceil + 1)$ which has solution $H(n) = O(n)$. Thus $W(n) = O(n)$ and hence $A(n) = H(n) \cdot W(n) = O(n^2)$.

Theorem 2: The multiplication network described above has area $A = O(n^2)$, delay $T = O(\log n)$ and period $P = O(1)$. In particular, $AP^2 = O(n^2)$ which is optimal.


gnd
a
b
Add(⌈n/2⌉)
Add(⌈n/2⌉)
M(⌊n/2⌋)
M(⌈n/2⌉+1)
M(⌈n/2⌉)
Sub(2(⌈n/2⌉+1))
Sub(2(⌈n/2⌉+1))
Add(2n)
c
vdd

Proof: The circuit described above computes the product of two n bit numbers in redundant representation. It is easy to see how to use a $T = O(\log n)$ adder to convert from redundant representation to standard binary representation. This proves $T = O(\log n)$ and $A = O(n^2)$. Finally, observe that the circuit above is synchronous and hence can be used in a pipelined fashion. Thus $P = O(1)$. □

We finally describe how to reduce the area. Let a, b be two n bit numbers. Let $k = \sqrt{\log n}$. Divide a, b into $t = n/k$ pieces of length k each. Then

$$a = \sum_{i=o}^{t} a_i 2^{ki}, \quad b = \sum_{i=o}^{t} b_i 2^{ki}$$

and each a_i, b_i is a k bit binary number. Also

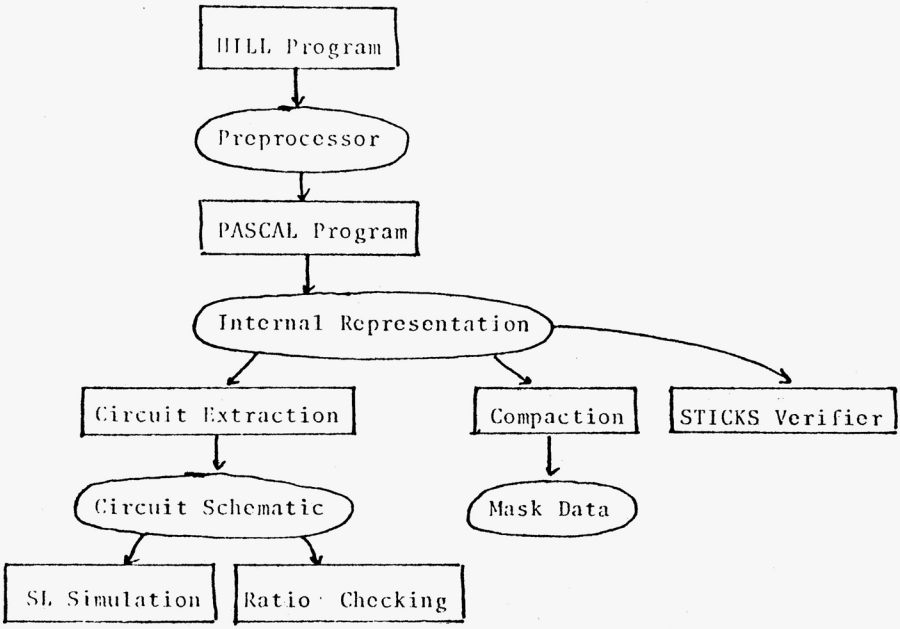
$$c = a \cdot b = \sum_{i=o}^{t} \sum_{j=o}^{t} a_i b_j 2^{(i+j)k}$$. We use a k bit multiplier to compute the t^2 products $a_i b_j$. Since the multiplier can be pipelined this takes time $O(\log n + t^2) = O(\log n)$. Also the t^2 products can clearly be added in time $O(t^2) = O(\log n)$ if redundant number representation is used. We summarize in

Theorem 3: There is an $A = O(n^2/\log n)$, $P = T = O(\log n)$ multiplier for n bit binary numbers.

III. THE HILL DESIGN SYSTEM

The HILL design system is currently under development at the University of Saarbrücken. It is part of a larger VLSI project which is sponsored by the DFG (Deutsche Forschungsgemeinschaft). A major other project is the CADIC system which is developed under the direction of G. Hotz.

An overview over the HILL system is given by the diagram below:



At present the HILL system has three major ports:

a) HILL layout specification language and graphics editor (discussed in 3.1)

b) HILL Compacter (discussed in 3.2)

c) HILL Simulator (discussed in 3.3)

3.1 HILL layout language and graphics editor

HILL is a tool for single chip development. The main focus of HILL is layout generation and verification. HILL aims at supporting the designer who has a comprehensive global image of his circuit. HILL provides a convenient way of describing a layout symbolically either by a HILL program or during an interactive session at the graphics terminal. Even though the layout is specified in a symbolic manner the designer has many means for exerting direct influence on the quality of the resulting mask data.

HILL is a system which combines convenient circuit description with efficiency of the implementation. We aim for efficiency in three respects: Human design time, chip area and delay, and computational resources.

Human Resources: In HILL, integrated circuits are described at the level of stick diagrams enhanced by extensive means for structuring a design hierarchically. We have chosen the level of stick diagrams because on the one hand it still allows the designer to express his insights about the topology of the circuit and on the other hand, it frees the designer from the tedious and error-prone task of specifying his circuit on the mask level. The stick diagram level has been used successfully in systems like [CABBAGE], [STICKS], and [MULGA].

Even though it certainly is no good practise to specify a whole large scale circuit with one great stick diagram, if enhanced by extensive means for hierarchical structuring, especially with a powerful cell concept, stick diagrams become a convenient symbolic representation of even large scale layouts.

In the HILL report [LM 83] this thesis is exemplified by a number of examples. We will give only a small example below.

Hierarchical structuring exploits the regularity of a layout. Of course, there are always irregular parts of a layout and therefore HILL also allows graphical layout specification. The graphics editor is similar to existing ones and is therefore not discussed here. Previous stick diagram systems used to be totally or almost totally graphics oriented. However, graphics alone cannot support powerful mechanisms for structuring layouts. Essentially, it can only support composition and simple forms of iteration. However, it cannot support recursion, and the full power of iteration and parameter passing.

Recursion and iteration are central to (software) algorithm design, and they have already proven to be powerful concepts in hardware design [GV 82, MC 80]. Only a high level programming language provides the flexibility needed, and so HILL is designed as a PASCAL extension, which interacts gracefully with a graphics editor.

A good example is provided by the multiplier designed in the previous section. The layout given there is regular, however, the regularity can certainly not be captured in a pure graphics system. Rather, powerful descriptive tools, such as recursion, iteration and parameterized cells are needed to capture the regularity.

Chip area and delay: Experience with existing systems [CABBAGE, STICKS, MULGA, HILL 82] suggests that automatic compaction can yield small layouts which come very close to hand compacted layouts. The stick diagram level is close enough to the silicon to allow the designer to incorporate performance aspects into his specification, and the compacter supports chip performance with his knowledge of the fabrication process. Finally, the case of circuit description and the flexibility of the algorithms used in the system allow the designer to try several approaches to his circuit and select the one he likes best.

Computational Resources: Most existing systems have definite short-comings in this respect. In all cases no theoretical analysis of the running time is given, often algorithmic concepts enter the system only scantily. Computational experience with the systems suggests that the running time is highly non-linear. For example, it is reported that CABBAGE takes time $O(n^{1.2})$ to compact a circuit with n transistors and wires. In the HILL prototype we improved upon this; an O(n log n) compaction algorithm is described in [L 83]. However, even such an algorithm will not do for large scale circuits, mainly because it also has an $O(\sqrt{n})$ space requirement. The solution to this problem is to compact hierarchically, see [L 82b]. The compaction algorithms used in HILL are described in section 3.2.

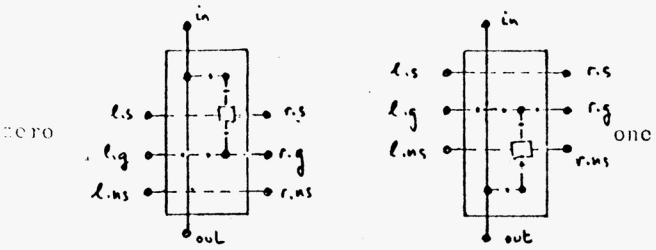
The main structuring device in HILL is that of a "cell". A cell is the specification of a subcircuit of the chip to be designed. This subcircuit will in general function as a modul in the chip that communicates with its surroundings through relatively few connections (pins) and performs a specific subfunction of the chip function. It is rectangular in shape with the connection pins arranged on its boundary. It is very much reminiscent of a function in a sequential programming language like PASCAL. As procedure parameters form the (up to side effects exclusive) interface between the procedure and its call environment, the pins of a cell facilitate the interface between the cell and the circuitry around its location of "placement" on the chip. The only way to contact to a cell is through on of its pins. Like procedure cells can be compiled separately and defined externally. For "instantiating" a cell only a description of its rectangular boundary, its so-called "template" has to be given. The template contains no (electrical or topological) information about any of the inner workings of the cell. However, it contains both electrical and topological information of the cell boundary. Pins can be related to each other electrically in the template and they have to be "placed" in order to specify the order in which they occur around the cell boundary.

In addition to specifying in which order the pins appear on each side of the cell HILL also requires to specify the relative positions of pins on opposite sides of a cell. This is in marked contrast to other symbolic layout systems, e.g. ALI, and has the following advantage. A layout (even a symbolic one) restricts the relative positions of pins; e.g., in the symbolic layout for the multiplier given in the previous section the output pins must be to the right of the input pins. In HILL these restrictions are part of the template and hence can be taken into account by a designer when he uses a cell.

We close this section with an example which highlights many of the features of the HILL layout language. For a complete description the reader is referred to [LM 83].

We specify the logical part of a decoder. The decoder takes n inputs (horizontally) and 2^n inputs at the top. It produces 2^n outputs at the bottom all but one of which are zero. One of the inputs, namely the one selected by the horizontal inputs, is fed through.

For the design we need two leaf cells zero and one. The layouts of these cells are given below:



(The horizontal diffusion line carries gnd; lines s and ns carry a signal and its complement. Cell zero lets the signal fed in at the top pass through iff the signal is zero. Otherwise it grounds the signal.) Cells zero and one are most naturally defined graphically and then entered into the cell library. They can then be included into a HILL program by

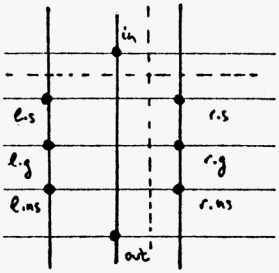
```
aggregate bus = record  s: poly;  g: diff sig = gnd;
                                  ns : poly end;
cell zero = temp pins in, out : metal;
                  l,r : bus;
            order implicit all;
            sides top in;
                  bottom out;
                  left l;
                  right r;
      pmet;
      external;
```

In these definitions we first introduce a bus consisting of a poly line followed by a diff line carrying the signal gnd followed by a poly line. In the cell definition of cell zero we first define the template. In the pin section we introduce the names and the sorts of the 8 pins in, out, l.s., l.g., l.ns, r.s, r.g, r.ns of the cell, for instance, the pin in has sort metal. We then specify the distribution of the pins over the four sides of the cell and their relative position. The phrase order implicit all states that the ordering is given by the sides section, i.e. the top pin on the left side is aligned with the top pin of the right side, and so on.

An alternative to a graphical definition of cell zero would be to specify the layout within HILL. We would then replace keyword external by

```
layout
components pd: t(2,y);
begin  place pd on (in right create 1, l.s);
       route l.s. to pd to r.s;
       route l.g  to r.g;
       route l.ns to r. ns;
       route layer diff pd up create 1 left to in;
       route pd down to l.g.;
       route in to out;
end
```

This program assembles the layout of cell zero on a rectangular grid. This grid can grow dynamically by the use of create; it is given initially by the template (the solid grid lines in the diagram below).



In order to assemble the layout we first declare the components needed, here a transistor pd of channel length 2 and width 4.

We place pd on a new vertical grid line one to the right of in and the horizontal grid line given by l.s. Then we route a number of wires. In most of these route statements the layer of the wire can be deduced from the sort of its terminals. E.g. the wire from l.s. to t must run in the poly layer, since l.s and the gate of a transistor both exist only on the poly layer. In the next to last route statement we route a diffusion wire from the top terminal of the transistor up to a new grid line and then left to the vertical grid line determined by in. Since it hits the metal line running down from in to out there an md contact is created automatically.

We will next describe how to build up the entire layout for the decoder from cells zero and one. Cells zero and one are to be placed into a rectangular array. For n = 3 we want to create the following pattern

```
0 1 0 1 0 1 0 1
0 0 1 1 0 0 1 1
0 0 0 0 1 1 1 1
```

In HILL there are two modes for specifying layouts. In composition mode one assembles layouts from smaller cells by abutment without explicit routing. Since HILL cells are stretchable abutment (composition) is a method of construction which is frequently used. In layout mode one assembles the layout on a rectangular grid. Components are placed on this grid by place statements and wires are routed between them by route statements.

In composition mode the decoder is defined by

```
cell decode (n: int) =
    temp pins in, out: array[0..2^n-1] of metal;
                l,r: array[0..n-1] of bus;
```

```
        order implicit all;
        sides top in;
        bottom out;
        left ℓ;
        right r;
    pmet;
composed
    var  i, j : int; leaf : pcell;
        begin decode := nil;
            for  i  from  0  to  2^n-1
            do begin leaf := nil
                for  j  from  0  to  n-1 do
                begin if (i div 2^j) mod 2 = 0
                        then leaf := leaf yy zero
                        else leaf := leaf yy one
                end ;
            decode := decode xx leaf
               end
        end
```

In this specification we build up the layout column by column. In each column we compute the column from top to bottom. After computing the type of the next cell to be abutted we add it to the partially constructed column by leaf := leaf yy zero or leaf := leaf yy one respectively.

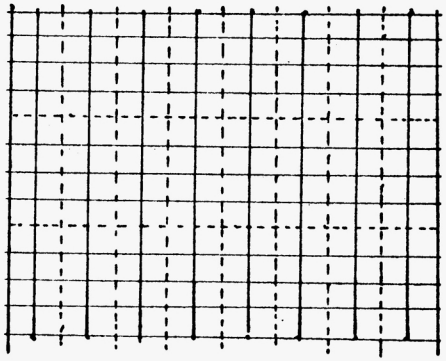
In layout mode we assemble the layout on a square grid. The use of layout mode for the present example is less elegant

than the use of composition mode; however, layout mode is a must in many cases, e.g. for leaf cells and the multiplier cell of chapter II. Even for the present example layout mode has two advantages. Firstly, layout mode is a "graphical" mode and hence the system gives better error messages in layout mode, secondly, layout mode provides us with explicit names for pins of subcells which can be accessed by the simulator. For the following example we use logical to denote the common template of cells zero and one.

```
cell decode (n:int);
temp pins tin, bout : array [0..2^n-1] of poly;
          lb, rb : array [0..n-1] of bus;
     order implicit all;
     sides top tin;
           bottom bout;
           left lb;
           right rb;
pmet
layout var  i,j : int ;
            xstretch : xlines;
            ystretch : ylines;
     components leaf : array [0..2^n-1,0..n-1] of logical;
begin  for  i := 0 to 2^n-2 do
            xstretch := tin[i] create 1;
       for  j := 0 to n-2 do
            ystretch := lb[i] create 1;
       xstretch := allx;
       ystretch := ally;
```

```
for  i := 0 to  2^n-1 do
        for  j := 0 to  n-1  do
begin
place leaf [i,j] on  (xstretch[2*i, 2*i+2],
                      ystretch[ 4*j, 4*j+4]);
if (i div 2**j) mod 2 = 0 then
        fill leaf [i,j] with zero
else  fill leaf [i,j] with one
end
end
```

This specification has to be interpreted as follows. We start out with a grid as given by the template (solid lines in diagram)



In the components section we declare an array leaf of rectangles which all have template logical. These rectangles are placed on the grid and then filled with cells zero or one as appropriate. In order to place the rectangles we first create new grid lines (broken lines in the above figure), on which we can place the pins of the leaf cell array.

This is done with the create operator. Then we place the leaf cells on the grid. For each leaf cell we specify the vertical and horizontal grid lines that the template of the cell is placed upon. For this specification we use the notation xstretch[i,j] where xstretch is a list of gridlines and xstretch[i,j] is the sublist of xstretch consisting of the i-th to j-th element. The variables allx and ally denote the list of all gridlines in the vertical resp. horizontal direction.

We close this section with a short remark about computational experience with the HILL prototype system. It suggests that the system works quite efficiently and that layouts with up to 5000 components will be processed (this includes compaction) in less than 60 sec on a Siemens 7760. Since running time of the system is O(n log n) as we will see in the next section it is safe to predict that layouts with 50 000 components will take about 10 min.

3.2 Compaction in HILL

Execution of a HILL program yields (the hierarchical representation of) a stick diagram. The compaction program takes this stick diagram and squeezes it whilst observing the design rules dictated by the fabrication process. This approach is taken by all stick diagram based systems [CABBAGE, FLOSS, MULGA, SLIM, STICKS, TRICKY].

Compaction is done in a number of phases p_1, p_2,...,p_k. During the odd numbered phases the extend of the layout in the x-direction is reduced by applying a squeeze in horizontal direction to the layout. The y-coordinates of the layout components are not changed in odd numbered phases. Analogously the even numbered phases squeeze the layout in y-direction. At present, the HILL compactor treats phases independently. The reader

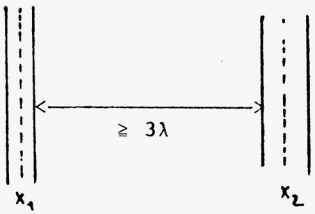
should consult [SLIM, GW 82] for attempts to relate compaction in x- and y-direction.

Within a phase there are two approaches to compaction. We concentrate on compaction in x-direction in the sequel. In one approach compression ridges are run through the layout that mark areas of the layout containing excess space. This space is then removed. This process is iterated until no more compression ridges are found. [MULGA, SLIM]. The second approach is graph-theoretic in nature. It is taken in HILL and also in [CABBAGE, FLOSS, STICKS, TRICKY, GW 82].

We associate a real variable with every layout component, i.e. with every wire, contact or transistor. The value of this variable represents the x-coordinate of this component. Linear inequlities (and equalities) between these variables are used to express the constraints on the x-coordinates of different layout components. There are three types of constraints.

1) Minimum distance constraints, $x_i - x_j \geq a_{ij} > 0$.
   Constraints of this form ensure minium separation requirements dictated by the fabrication process.

   Example: Consider two parallel diffusion wires of width w_1 and w_2. Let x_1, x_2 denote the x-coordinates of the center

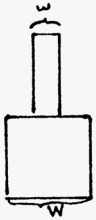


   lines of these wires. Then $x_2 \geq x_1 + 3\lambda + (w_1 + w_2)/2$ in the process described in [MC 80].

2) Alignment constraints, $-a_{ij} \leq x_i - x_j \leq a_{ij}$.

Constraints of this form encode contact rules, i.e. they solder layout components together that should contact each other.

Example: Consider a square contact of side length W and a line of width w. If x_1, x_2 are the x-coordinates of the centers of the contact and the wire respectively, then



$$|x_1 - x_2| \leq (W - w)/2 \quad \text{or}$$

$$-(W - w)/2 \leq x_1 - x_2 \leq (W - w)/2$$

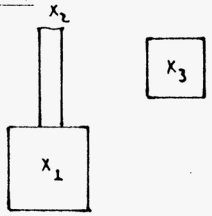
3) Maximum distance constraints $x_i - x_j \leq a_{ij}$.

Maximum distance constraints encode requirements given by the designer explicitly. For example, he might wish to keep two components together because of wire delay considerations. In HILL keep-statements are used to introduce user-defined constraints.

In section 3.2.1 below we discuss efficient methods for generating a sufficient set of linear inequalities. Suppose now that we have a set of linear inequalities describing the layout. We then want to find values $\bar{x}_i$ for variables x_i, i = 0, 1, ... such that max $\bar{x}_i$ - min $\bar{x}_i$ is minimal (see also [L 82a]). This problem is easily formulated as a shortest path problem as follows. Generate a graph with nodes x_i, i = 0, 1, ... If $x_j \leq x_i + a_{ij}$ is a constraint (note that $x_j \geq x_i + a_{ij}$ is equivalent to $x_i \leq x_j - a_{ij}$ and hence all constraints are of this form) generate an edge $x_i \xrightarrow{a_{ij}} x_j$ from x_i to x_j of cost a_{ij}. Also augment the graph by an additional node s and edges $s \to x_i$ from s to x_i, i = 0, 1, ..., of length 0.

Then solve the single source least cost path problem with source s on this graph. Let $\mu(s,x)$ the cost of a least cost path from s to x. Then $\bar{x}_i = \mu(s,x_i)$, $i = 0, 1, 2, \ldots$ is an optimal solution, i.e. a solution for which $\max \bar{x}_i - \min \bar{x}_i$ is minimal.

Example:

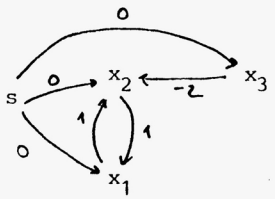


The layout shown might give rise to the following system of inequalities

$$|x_1 - x_2| \leq 1$$

$$x_3 - x_2 \geq 2$$

From this system we obtain the following least cost path problem.



$$\mu(s,x_1) = -2$$

$$\mu(s,x_2) = -1$$

$$\mu(s,x_3) = 0$$

Choosing $\bar{x}_1 = -2$, $\bar{x}_2 = 1$, $\bar{x}_3 = 0$ gives an optimal solution of the above system of inequalities. □

In general, the correctness of this approach can be seen as follows:

1.) Note first that $\bar{x}_i = \mu(s,x_i)$ is a solution. This follows from the observation that $\mu(s,x_j) \leq \mu(s,x_i) + a_{ij}$ whenever there is an edge of cost a_{ij} from x_i to x_j, i.e. whenever there is an inequality $x_j \leq x_i + a_{ij}$.

2.) Suppose now that $\hat{x}_i$ is a minimal solution. We may assume w.l.o.g. that $\max \hat{x}_i = 0$. Let j be arbitrary and let p be a minimal cost path from s to x_j, say $p = \xrightarrow{a_1} v_o \xrightarrow{a_2} \ldots \xrightarrow{a_k}$

with $s = v_o$, $v_k = x_j$. Then $\mu(s,x_j) = a_1 + a_2 + \ldots a_k$.

Also $v_{i+1} \leq v_i + a_{i+1}$ is a constraint for $0 \leq i < h$ and hence $\hat{x}_j \leq \hat{v}_1 + a_1 + \ldots + a_k \leq \mu(s,x_j)$. Finally, observe $\mu(s,x_i) \leq 0$ for all i since there is an edge of cost 0 from s to x_i. Thus $\max \mu(s,x_i) \leq 0 = \max \hat{x}_i$ and $\min \hat{x}_i \leq \min \mu(s,x_i)$ and hence $\max \bar{x}_i - \min \bar{x}_i \leq \max \hat{x}_i - \min \hat{x}_i$.

In section 3.2.2 we discuss algorithms for solving least cost shortest path problems. In section 3.2.3 we briefly discuss hierarchical compaction.

3.2.1 Efficient Constraint Generation

Recall that we discuss compaction in x-direction. Efficient constraint generation is a major problem for most existing stick-diagram based systems in a two-fold sense. First, constraint generation has large running time, and second, it produces a large number of constraints which in turn influences the complexity of the shortest path algorithms used to solve the constraint system. For example, CABBAGE may produce as many as $O(n^{1.5})$ constraints and typically produces about $O(n^{1.2})$. Of course, running time is at least that much.

Already in the HILL prototype we overcame this deficiency and used an O(n log n) algorithm for constraint generation. The algorithm is based on a left to right plane sweep and used quite complicated data structures. In the spring of 83, I. Cnop (after hearing the lectures at Louvain), R. Reischuk and Th. Lengauer independently suggested to use a top down plane sweep instead. The following presentation follows [L 83].

Note first that alignment constraints are easily generated in time O(n). Also maximum distance constraints are user defined and therefore are of no concern here. This leaves minimum distance constraints (type I).

THE INTERVAL GRAPH

A straightforward way to generate the Type II constraints would be to look at each pair of layout components in turn and use the design rule table to generate the appropriate inequality. However, this would result in $\Omega(n^2)$ inequalities, far too many to be practical.

In fact, many of these inequalities are redundant. Mostly, minimum distances are small (a few microns) such that layout components that are far apart from each other will be assured sufficient separation by the constraint that already exist in each of their neighbourhoods. Therefore only a few of the constraints are actually necessary. We will discuss how to generate such "minimal" constraint systems. To this end we formulate the following graph-theoretic problem.

Definition 1: a) Let L be a set of n vertical intervals in the plane. Each interval is a triple (x,y_ℓ,y_h) where x is the x-coordinate and y_ℓ and y_h are the y-coordinates of the low and high endpoints of the interval.

b) Let $(L,<)$ be the total ordering that orders L w.r.t. the x-coordinate of each interval. Ties are broken arbitrarily but fixed.

c) Intervals I_1 and I_2 are said to "overlap", if

$$I_1 \lambda I_2 :<=> I_1<I_2 \wedge y_{\ell,1}<y_{h,2} \wedge y_{\ell,2}<y_{h,1}$$

d) The following set is called the set of intervals "between" I_1 and I_2:

$$\beta(I_1,I_2) := \{I \in L;\ I_1 < I < I_2 \quad I_1 \lambda\ I\ \lambda\ I_2\}.$$

L represents the layout geometry. Specifically, each interval represents a layout component. For a layout to be correct w.r.t. a set of design rules we assume that overlapping intervals have to be separated by certain minimum distances in the x-direction. The exact amounts are of no concern here. Non-overlapping intervals are assumed to be sufficiently separated in the y-direction, such that no constraint in the x-direction has to be generated. This can always be ensured by slightly enlarging the interval.

We will define several ways for L to induce a so-called constraint graph G=(L,E). G is a directed acyclic graph with each edge $e=(I_1,I_2)$ representing an inequality of the form $x_2-x_1 \geq a_{21} > 0$. Here x_1 and x_2 are the x-coordinates of I_1 and I_2 in the compacted layout. The constraint graph G is almost exactly the graph analyzed in [L82]. The redundancy of some constraints can now be formulated as the following axiom.

PROCESS AXIOM: If G=(L,E) represents an inequality system that ensures all design rules to be met -we call such a system "admissible"- then the transitive reduction ρ of G also represents such a system.

The process axiom allows us to neglect all inequalities that form "short-cuts" in the constraint graph. This is a realistic assumption because design rules are typically of a highly local nature. The process axiom is a powerful and also essentially the only existing tool for reducing the size of the set of constraints for compaction.

Clearly there are many ways of extracting a constraint graph from layout L. Here is the simplest one.

Definition 2: Let $G_0=G_0(L)=(L,E_0)$ be defined as follows:

$$(I_1,I_2) \in E_0 \text{ iff } I_1\ \lambda\ I_2.$$

Clearly the undirected graph underlying G_0 is an interval graph [G80], and its interval representation is given by L, if all x-coordinates are set to zero. Thus the question, how to compute the transitive reduction ρ_0 of G_0 asks for algorithms to efficiently compute the transitive reduction of such interval dags. Obviously the following is true:

Fact 1: $(I_1,I_2)\in\rho_0 \iff I_1\lambda I_2 \wedge \beta(I_1,I_2)=\emptyset$

Using this characterization ρ_0 can be computed in time $O(n \log n)$ with the following algorithm Gen_0. Gen_0 uses a top-down plane sweep. Thus the algorithm scans a horizontal sweep line across the layout L from top to bottom. During the sweep a data structure D is maintained that stores information about all intervals that currently intersect the sweep-line. D is a leaf-chained balanced tree that keeps the intervals in sorted order according to $(L,<)$. Furthermore with each interval $I\in D$ two pointers to other intervals in L are associated. The pointer left(I) points to nil or an interval less than I. The pointer right(I) points to nil or an interval greater than I. Both pointers represent edges between I and the interval pointed to that are candidates for ρ_0 but whose membership in ρ_0 has not been decided yet. W.l.o.g. we assume that the y-coordinates for all $I\in L$ are pairwise distinct. Then during the sweep we encounter two kinds of events the algorithm has to deal with, namely the insertion of an interval into the sweep line and the deletion of an interval from the sweep line. Upon these the algorithm does the following:

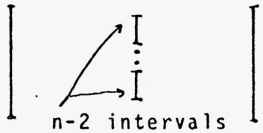
```
Insert(I) : Insert I into D;
            Find the left and right neighbour I_ℓ and I_r of
            I in D;
            left(I):=I_ℓ; right(I):=I_r;
            left(I_r):=right(I_ℓ):=I;

Delete(I) : if left(I) ≠ nil and left(I)∈D then
              append (left(I),I) to ρ_0;
            if right(I) ≠ nil and right(I)∈D then
              append (I,right(I)) to ρ_0;
```

delete I from D;

Theorem 1: a) Gen_0 computes ρ_0 in time $O(n \log n)$

b) $|\rho_0| \leq \begin{cases} 0 & \text{if } n=1 \\ 1 & \text{if } n=2 \\ 2n-4 & \text{if } n>2 \end{cases}$

The upper bound of Theorem 1b is tight, as the following layout shows:



Several systems attempt to find ρ_0 [CABBAGE,SLIM]. CABBAGE may produce as many as $O(n^{1.5})$ constraints and typically produces about $O(n^{1.2})$. SLIM comes closer, but it also produces more than the transitive closure, since a constraint is generated between each pair of intervals that are visible from each other at least in part.

THE LAYER APPROACH

While ρ_0 is always an admissible constraint system for compaction it is in general not the best one. It does not allow to change the layout toplogy during compaction, because there is a constraint between an interval and all of its neighbours. [CABBAGE] states this problem without offering a solution. Within our framework we are able to generate different admissible constraint systems that entail more topological freedom.

To this end we define a symmetric and reflexive binary compatibility relation $\pi \subset L \times L$. We call two intervals I_1 and I_2 compatible if $I_1 \pi I_2$. Intuitively two intervals should be compatible if the associated components do not have to meet any minimum distance constraint. Obviously no edge has to exist in the constraint graph between any pair of compatible intervals. Therefore we define:

Definition 3: Let $G_\pi = G_\pi(L) = (L, E_\pi)$ be defined as follows:

$$(I_1, I_2) \in E_\pi \;:\Leftrightarrow\; I_1 \lambda I_2 \wedge \neg I_1 \pi I_2$$

We make the reasonable assumption that it can be decided in time O(1) if $I_1 \pi I_2$. One possibility to define π is to realize that VLSI circuits are typically laid out on several, say ℓ layers that are insulated from each other, except for contact holes that provide connections between the layers. Therefore we can define: $I_1 \pi I_2$ if the layout components associated with I_1 and I_2 exist on different layers that are insulated from each other. The resulting graph G_π is in general no interval dag and its transitive reduction ρ_π may be hard to compute. But we can efficiently compute a supergraph of ρ_π with few edges.

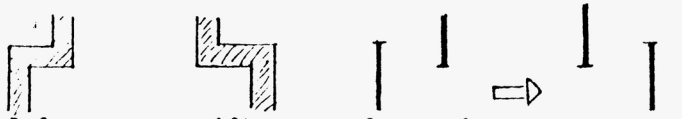
Theorem 2: Let L be a layout such that a subset M(I) of a set $\{1,\ldots,\ell\}$ of layers is associated with each interval $I \in L$. (We say that I exists on the layers in M(I).) Assume that $|M(I)| \leq d$ for all $I \in L$. Let $I_1 \pi I_2$ iff $M(I_1) \cap M(I_2) = \emptyset$. Then we can in time O(dn log n) compute a graph R such that $\rho_\pi \subset R \subset G_\pi$ and $|R| \leq 2dn-4$.

[FLOSS] applies this kind of layer separation to achieve some topological freedom during compaction.

SWITCHING THE POSITIONS OF COMPONENTS WITHIN A LAYER

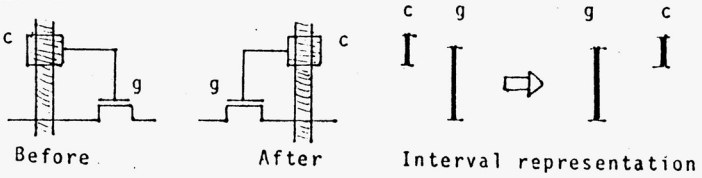
While ρ_π provides for more topological flexibility by handling each layer of the circuit separately there are still desirable transformations that it does not allow. We give two examples:

Example 1: Jog-flipping of wires:



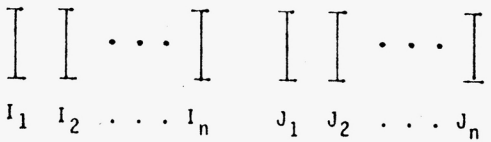
Here the intervals overlap, although slightly. Since they are on the same layer they are incompatible with respect to the above relation π and cannot exchange their positions during compaction. CABBAGE solves this problem by adjusting, specifically for such jog flips, the lengths of the intervals temporarily such that they do not overlap. This solution is ad-hoc, however, and it does not solve the following problem.

Example 2: Transistor flipping:



Here the vertical bold wire is on a top layer and all other structures are on the bottom layer. The contact c connects between the two layers. In this case a simple minded adjustment of interval lengths will not do. Therefore we extend π by also allowing $I_1 \pi I_2$ if I_1 and I_2 exist on the same layer and carry the same electrical signal. Such an extension is desirable, since the above example transformations will reduce the area of many layouts significantly. But now ρ_π can become large:

Example 3:



Let $I_i \pi I_j$ and $J_i \pi J_j$ for $1 \leq i < j \leq n$, but $\neg I_i \pi J_j$ for $1 \leq i,j \leq n$. Then ρ_π is the complete bipartite graph $K_{n,n}$.

Indeed one can show that if one just assumes π to be reflexive and symmetric one cannot hope to find an encoding for the path information contained in G_π that has size $o(n^2)$. Thus G_π is not the appropriate constraint graph. We therefore define a new constraint graph G_1 such that $G_0 \supset G_1 \supset G_\pi$ and G_1

allows the transformations discussed above.

Definition 4: Let $G_1=G_1(L)=(L,E_1)$ be defined as follows:

$$(I_1,I_2)\in E_1 :<=> I_1 \lambda I_2 \wedge (\sim I_1 \pi I_2 \vee \beta(I_1,I_2)\neq\emptyset)$$

Thus E_1 can be obtained from E_0 by deleting all edges in ρ_0 that connect compatible intervals. This allows only exchanges of the positions of neighboring elements during compaction. However, both example transformations are included. The transitive reduction ρ_1 of G_1 can be characterized as follows:

Lemma 1: $(I_1,I_2)\in\rho_1 <=> I_1 \lambda I_2 \wedge (\beta(I_1,I_2)=\emptyset => \sim I_1 \pi I_2)$

$$\wedge (\beta(I_1,I_2)\neq\emptyset => \forall I\in\beta(I_1,I_2): [(I_1,I)\in\rho_0 \wedge I_1 \pi I] \vee [(I,I_2)\in\rho_0 \wedge I \pi I_2])$$

Lemma 1 provides the basis for an efficient algorithm for computing ρ_1. The following lemma shows that information has to be updated only locally during the algorithm.

Lemma 2: Consider an arbitrary position of a horizontal line through the layout L that does not touch the endpoints of any interval in L. Let $\rho_{1,t}$ be the subgraph that is induced from ρ_1 by all intervals intersecting the line.

a) If $(I_1,I_2)\in\rho_{1,t}$ then there are at most two intervals I,I' intersecting the line such that $I_1<I<I'<I_2$.

b) The maximum in- and out-degree of any vertex in $\rho_{1,t}$ is 2.

There are examples that show that we cannot hope to find a simple one-pass algorithm for computing ρ_1 using plane sweep methods. Thus the following algorithm Gen_1 computing ρ_1 is a two-pass algorithm. Here is a description of algorithm Gen_1.

Pass 1: Run algorithm Gen_0 on L. However, output only edges $(I_1,I_2)\in\rho_0$ such that $I_1\pi I_2$. Organize the edges in a linear list Σ of sets, each set being the collection of edges output during one delete operation.

Pass 2: Make a plane sweep bottom-up. Again maintain a balanced tree D, however this time allow for two pointers to the left and two pointers to the right of each interval to

to store candidate edges. Furthermore allow for one ρ_0-pointer to the left and right to store edges from Σ.

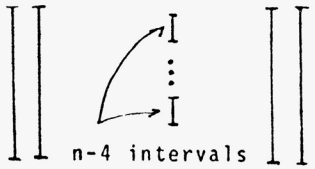
Insert(I) : Insert I into D;
Fetch from the back of Σ all edges that have been output upon the deletion of I in Pass 1, and store them in the ρ_0-pointers. Maintain the set of candidate edges between the intervals at most 3 to the right or left of I in D according to Lemma 1, i.e., delete a candidate edge if the newly fetched edges from Σ show by Lemma 1 that the edge is not a candidate any more;

Delete(I) : Output all candidate edges that have I as an endpoint and an interval crossing the sweep line as the other;
Delete I from D;

Theorem 3: a) Gen_1 produces ρ_1 in time $O(n \log n)$
b) $|\rho_1| \leq 4n$

The following example shows that there are layouts such that $|\rho_1| \geq 4n-16$

Example 5:



The storage requirement of both algorithms Gen_0 and Gen_1 is $O(m)$ where m is the maximum number of intervals intersecting the sweep line at any time. Since layouts can be expected to be roughly quadratic with uniform distribution of the layout components we can expect $m=O(\sqrt{n})$. Here we assume that the output of Pass 1 in algorithm Gen_1 is on a sequential access storage device and not in main memory. (Otherwise the storage requirement would be linear.) Both algorithms are simple enough to expect that they perform well in practice.

Additional passes can be made across the layout to generate

the transitive reductions ρ_i of constraint graphs $G_i=(L,E_i)$ where

$$(I_1,I_2)\in E_i \iff I_1 \lambda\ I_2 \wedge (\sim I_1 \pi I_2 \vee \exists I\in\mathcal{B}(I_1,I_2): (I_1,I),(I,I_2)\in E_{i-1})$$

Then $G_0 \supset G_1 \supset G_2 \supset \ldots$ After at most n iterations the sequence stabilizes in a graph G_n with the following properties:

$$(I_1,I_2)\in E_n \iff I_1 \lambda\ I_2 \wedge (I_1,I_2)\in G_\pi^{\,T}.$$

Here $G_\pi^{\,T}$ is the transitive closure of G_π. Thus $\rho_n=\rho_\pi$. Unfortunately the passes to compute ρ_i become increasingly complex such that this is not a good way to compute ρ_π.

3.3.2 Efficient Constraint Resolution

Constraint resolution is tantamount to solving a single source least cost path problem. Least cost path problems have received a lot of attention in the literature and are fairly well understood. A detailed discussion can be found in [M 83].

Let $G = (V,E)$ be a directed graph, $s \in V$ a special node and let $c: E \to \mathbb{R}$ be a cost function. Let $n = |V|$ and let $e = |E|$. In constraint graphs we have $e = O(n) + m$ where m is the number of user-defined constraints. Of course, m is very small in general, i.e. $m << n$.

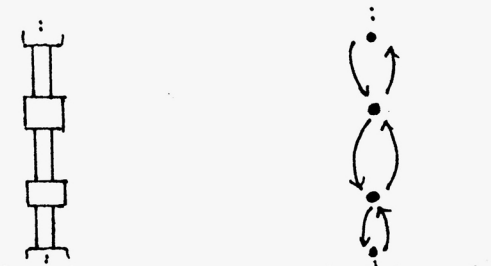
Let us consider a special case first: There are no maximum distance constraints and all alignment constraints are of the form $x_i = x_j$. Then constraint graph G is acyclic and hence the least cost path problem can be solved in time O(e) by topolog-

ical sorting. This algorithm is well-known and is used in [CABBAGE].

Let us now return to the general case. The best algorithm known for the general case is due to Bellman/Ford and runs in time $O(ne) (=O(n^2)$ in our case). Naive use of this algorithm is out of the question because of the large size of constraint graphs. We use (or intend to use) three modifications:

1.) The constraint graph is preprocessed and its strongly connected components (SCCs) are computed. This takes time $O(e)$ by depth first search. A description of these algorithms can be found in most textbooks on algorithms. Then the least cost path problem is solved by the Bellman/Ford algorithm on the strongly connected components and the fast algorithm for the acyclic case between components. If SCCs are small this modification reduces running time considerably. More precisely, the running time is $O(e + \sum_i n_i e_i)$ where $n_i (e_i)$ is the number of nodes (edges) in the i-th SCC, i = 0, 1, 2, ... This algorithm is implemented in the HILL prototype and works quite well.

2.) In the absence of maximum distance constraints the SCCs are of a very special form. SCCs arise from alignment constraints between layout components which are connected vertically.



For graphs of this special form the least cost path problem can be solved in linear time. Suppose that the nodes are numbered $v_1, v_2, v_3, \ldots, v_k$ from top to bottom and that $dist[v_i]$ is the least cost of a path from s to v_i which does not pass through any other node of the SCC shown above. Values $dist[v_i]$ are already computed when the modified algorithm reaches the SCC under consideration.

Then

```
for i from 2 to k
do dist[v_i] ← min(dist[v_i], dist[v_{i-1}] + a_{i-1,i}) od;
for i from k-1 to 1
do dist[v_i] ← min(dist[v_i], dist[v_{i+1}] + a_{i+1,i}) od
```

updates the dist-values correctly under consideration of the edges of the SCC. We thus have: In the absence of maximum distance rules constraint resolution takes linear time O(e).

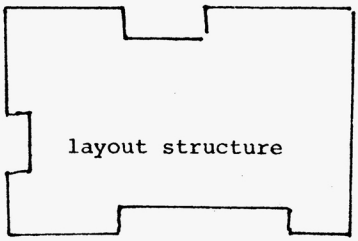
In [Sch 83] this approach is extended to a larger class of graphs, i.e. more general (but of course not completely general) SCCs are considered and linear running time is maintained.

3.) In the presence of maximum distance constraints the work of [M Sch] might be of some help. They show that least cost path problems on planar graphs can be solved in time $O(n^{3/2}\log n)$ instead of time $O(n^2)$ as for the Bellman/Ford algorithm. Constraint graphs as generated in the previous section are almost planar and hence this algorithm might be applicable.

3.3.3 Hierarchical Compaction

In HILL layouts are defined hierarchically. However, no use of the hierarchy is made in the compaction process. Although we presented efficient methods for constraint generation and constraint resolution in the previous sections, the approach might run into difficulties as chips become more complex. In particular the space requirement might become prohibitive. It is therefore worth-while to consider hierarchical compaction. The ideas presented about hierarchical compaction are preliminary.

Suppose, that we enclose every cell into a polygon with horizontal and vertical edges. We might take as



this polygon the boundary of the layout structure in the interior of the cell or a smoothing of this polygon (see below). For compaction in x-direction we associate a variable with every vertical edge of the boundary polygon. We can now set up a set of inequalities relating the boundary and the interior of the cell.

For every occurrence of the cell as a subcell in a larger cell we conceptually replace the cell by its boundary polygon. Mathematically, we introduce a new copy of the boundary variables and use this copy of the boundary variables when we set up the inequality system for the larger cell. In general, if there are

k occurrences of a cell then there are k+1 occurrences of the boundary variables. One copy is used "within" the cell and one copy is used for each occurrence of the cell.

As before we can interpret the inequality system as a least cost path problem. The difference is that we deal with a hierarchically specified graph. In [L 82b] it was shown how to solve such systems by dynamic programming. More precisely, one proceeds as follows:

1. Phase One is a Bottom-up Phase.
   Starting at the leaf cells one solves all pair shortest path problems. When a cell is looked at all subcells have been considered already. The subcells are replaced by complete graphs on the (copies of the) boundary variables of the subcells. The cost of an edge of the complete graph is given by the solution for the subcell.

   At the end of phase one an optimal solution to the constraint system has been found, more precisely, the spread (extension of the layout in x-direction) has been found.

2. Phase Two is a Top-down phase and actually determines the solution as follows: In phase one we determined the solution for the root cell. This fixes the values of the boundary variables for every occurrence of a direct subcell of the root, call such a fixing an instance of the subcell. In general, there will be more than one instance but <u>less</u> than the number of occurrences. For every instance we add the values of the boundary variables as additional constraints to the constraint set of the cell and then solve the least cost path problem again. This fixes the values of the boundary variables for every occurrence of a direct subcell of a direct subcell of the root, ...

The running time of the algorithm outlined above is $\sum_{i=1}^{m} s_i \, n_i^3$

where n_i is the number of layout components in the i-th cell and s_i is the number of different ways cell i is compacted. Of course, s_i might be very large.

One way of partially controlling s_i is to give the enclosing polygon a small number of edges. An extreme case is to make the enclosing polygon a rectangle. This choice guarantees a small number of boundary variables (2 + the number of pins at top and bottom) but might yield poor compaction results.

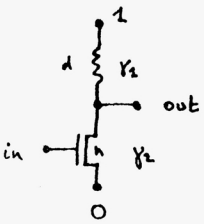
A compromise is to smooth the smallest enclosing rectangle a little but not all the way to a rectangle. For example, we might require that each edge of the polygon has length at least m units or that there are at most m edges. In the former case, [No 83] shows how to compute an optimal (= minimal wasted area) approximation in time O(nm).

3.4 HILLSIM, a Switch-Level Simulator

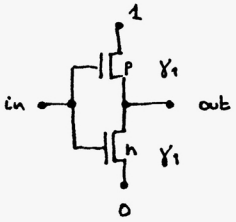
Simulation at the gate level, i.e. logic simulation, has always been very popular with circuit designers. Unfortunately, it is insufficient for MOS integrated circuits because among others it cannot model rationed logic, dynamic memory, charge sharing and bidirectional wires. It seems that switch level simulation plays the role of logic simulation for MOS integrated circuits. A first and very successful switch level simulator, called MOSSIM, was developed by Bryant [B 80]. In later papers [B 81a, B 81b] he gives a theoretical underpinning for the simulator, i.e. he introduces a mathematical model for the behavior of MOS circuits and proves the correctness of the simulator with respect to the model. Unfortunately, the model is quite complicated and at some points inconsistent. Therefore, in [MNN 82] an alternative model is proposed. The model is quite different from Bryant's original model and is

considerably simpler. An equivalent model has recently been proposed independently by Bryant [B 83]. HILLSIM is correct with respect to that model and also very efficient. The simplicity of the model suggests several optimizations which could not have been obtained from Bryant's original model.

We model transistors as voltage controlled switches. An open switch has conductance 0, a closed switch has conductance γ. Here γ is an element of a finite set $\Gamma = \{\gamma_1 < \gamma_2 < \ldots < \gamma_m\}$ of possible conductances. We use three different types of transistors n, p and d. The conductances in Γ are used to model ratioed logic. We use strength (t) to denote the conductance of t in its closed state

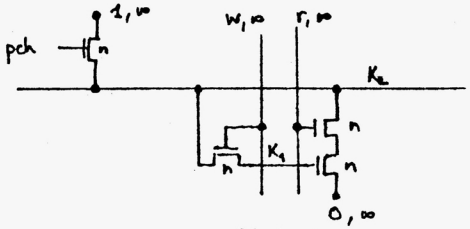


inverter in NMOS



inverter in CMOS

Nodes (wires) are modeled as capacitances. Each node has a capacitance in a finite set $K = \{K_1 < K_2 < \ldots < K_q < \infty\}$ of capacitances. Input nodes have capacity ∞. The capacitances are used to model precharging and dynamic memory. We use cap(k) to denote the capacitance of node k



memory cell in NMOS

The state zk of a network is a function zk: N → {0,1,X} where N is the set of nodes. If zk(k) = X then the state of node k is either unknown (i.e. 0 or 1 but unknown) or undefined (i.e. somewhere between 0 and 1). The state of a transistor is defined by the state of the gate and the type of the transistor according to the following table.

state of gate	n	p	d
0	open	closed	closed
1	closed	open	closed
X	X	X	closed

We can now define the basic simulation algorithm. For the sequel we use N to denote the set of nodes and T to denote the set of transistors of the network. A node state is a mapping zk: N → {0,1,X} and a transistor state is a mapping zt: T → } open, closed, X}.

Basic Simulation Algorithm

Input: a node state zk and stimuli in: I → {0,1,X} where I is a subset of the set N of nodes (the input nodes)

Output: a new stable state settle (2k,in) of the network, if it exists.

$$\text{Let } zk_0(k) = \begin{cases} in(k) & \text{if } k \in I \\ zk(k) & \text{if } k \in N - I \end{cases}$$

i ← 0;

repeat: let zt: T → {open,closed,X} be as defined by node state zk; and the table above, i.e.

zt(t) = δ(type(t),zk(gate(t))

where type (t) $\in$ {n,p,d}, gate(t) $\in$ N is the gate node of transistor t and δ is given by the table above;

$zk_{i+1} \leftarrow Equ(zk_i, zt)$, where function Equ(Equilibrium) is defined below;

$i \leftarrow i+1$

until $zk_i = zk_{i-1}$;

output zk_i ;

It is important to observe at this point that the basic simulation algorithm implements a unit-delay assumption. Note that we first compute the transistor state as given by the node state, and then keep this state fixed in order to compute the equilibrium state on the nodes. Once the equilibrium is reached, we set the transistors to their new states, ... We will come back to the unit-delay assumption at the end of this section.

It remains to define the equilibrium function. We do so in a two step process. We first define Equ(zk,zt) in the case that zt(t) $\in$ {open,closed} for all t $\in$ T and then extend it to arbitrary transistor states.

A transistor state zt is complete if zt(t) $\in$ {open,closed} for all transistors t $\in$ T. Assume now that zt is complete. If zt is complete we define an undirected graph G = (N,E) as follows. The set of vertices is the set of nodes of the network, edges correspond to closed transistors; more precisely

$E = \{(v,w);\ v,w \in N$ and there is $t \in T$ with $z(t) =$ closed and $\{v,w\} = \{drain(t), source(t)\}\}$

Let $V_1, V_2, \ldots, V_m$ be the connected components of this graph.

A connected component V_i is isolated if $V_i \cap I = \emptyset$, i.e. if it contains no input node. For the definition of Equ we will now make a case distinction.

Isolated components: Let V_i be an isolated component. Let $maxcap(V_i) = max\{cap(k); k \in V_i\}$ and let $Equ(zk,zt)(k) = V\{zk(v); v \in V_i \text{ and } cap(v) = maxcap(V_i)\}$ for all $k \in V_i$ where $0v0 = 0$, $1v1 = 1$ and $0vX = Xv0 = 0v1 = 1v1 = 1vX = X$.

This definition captures the following intuition. In isolated components the nodes of maximal capacitance determine the equilibrium state. If all nodes of maximal capacitance carry the same logic value then this signal floods the entire components and X floods the component otherwise.

Example: Consider the NMOS memory cell shown above. Assume precharge = read = 0, write = 1. Then the bus of capacitance k_2 is connected with the memory cell of capacitance k_1 and these nodes form an isolated component. Hence the value on the bus is written into the cell.

Non-isolated components

Let V_i be a non-isolated component. Then every node $k \in V_i$ is connected to at least one input node by a sequence of closed transistors. The logic values carried from the input nodes along these paths will determine the equilibrium logic value of the node. We define:

A path p is a sequence $v_o e_o v_1 e_1 v_2 \cdots e_{k-1} v_k$ of nodes and edges such that e_i connects v_i and v_{i+1}. The strength of a path p is the minimal strength of any edge (= transistor) on

the path, i.e.

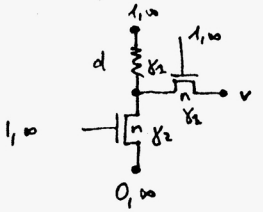
$$\text{strength}(p) = \min\{\text{strength}(e);\ e \text{ is an edge of } p\}.$$

The strength of an empty path is defined as ∞. The strength of a node is the maximal strength of any path connecting it to an input node, i.e.

$$\text{strength}(v) = \max\{\text{strength}(p)\ ;\ p \text{ is a path from a node in } I \text{ to } v\}.$$

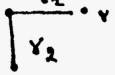
Note that this definition gives input nodes strength ∞.

Example: Consider an inverter with a pass transistor at the output. There are two paths from input nodes to v,



namely

and



both of strength γ_1. Hence v has strength γ_1. □

A path $p = v_o, e_o, v_1, \ldots, e_{k-1}, v_k$ from an input node v_o to a node v_k is <u>essential</u> (for v_k) iff $\text{strength}(v_i) = \min\{\text{strength}(e_o), \ldots, \text{strength}(e_{i-1})\}$ for $1 \leq i \leq k$, i.e. if every initial segment of p supports the strength of its end node. In our example only the second path is essential, because the output node of the inverter has strength γ_2.

The equilibrium logic value of a node is determined by the essential paths ending in that node, i.e. for all $v \in V_1$

$$\text{Equ}(zk,zt)(v) = V\{zk(k);\ k \in I \text{ and there is an essential path from } k \text{ to } v\}$$

In our example the equilibrium logic value of node v is 0. Note that it is crucial that only essential paths are considered in the definition of the equilibrium value.

The definition of equilibrium logic value given above is justified by the following

Theorem: Let $I \subseteq N$ be a set of input nodes, let zk be a node state and let zt be a complete transistor state. Assume also that $zk(i) \in \{0,1\}$ for $i \in I$. For $c \in \mathbb{N}$ construct the following RC-network:

Replace closed transistors of conductance γ_i by resistors of c^{-i} Ohm, replace a node k of capacitance K_i by a capacitor (against ground) of c^i Farad and charge it with $zk(k)c^i$ Coulomb. Finally connect an input node i with a power supply of zk(i) Volt. Let voltage (c,v) be the resulting voltage at node $v \in V$.

Then

$$\lim_{c\to\infty} \text{voltage}(c,v) = \begin{cases} 0 & \text{if } Equ(zk,zt)(v) = 0 \\ 1 & \text{if } Equ(zk,zt)(v) = 1 \\ x & \text{if } Equ(zk,zt)(v) = X \end{cases}$$

where x is some value between 0 and 1 depending on v.

Proof: see [MNN 82]

It remains to extend the definition of Equ to incomplete transistor states. Let zt be a transistor state. Then zt' is a complete extension of zt if zt' is complete and $zt(t) \in \{\text{open,closed}\}$ implies $zt(t) = zt'(t)$, i.e. the state of undefined transistors is changed to open or closed arbitrarily. We define

$$Equ(zk,zt)(v) = \bigvee \{Equ(zk,zt')(v); zt' \text{ is a complete extension of } zt\}$$

This definition captures the intuition that nothing is known about the state of a transistor whose gate has logic value X. Hence all possible complete extension have to be considered.

This completes the definition of Equ and hence finishes the description of the mathematic model of the behavior of MOS circuits. We will now turn to the description of the simulator which efficiently implements this model. The main problem is to compute Equ efficiently. Again we concentrate on complete transistor states first. In this case a simple algorithm works. We explore the network starting at the input nodes by breadth first search. All nodes encountered are entered into a set Active from which nodes are deleted in order of decreasing strength. Hence we will first delete all nodes of strength ∞, then all nodes of strength γ_m, then all nodes of strength $\gamma_{m-1}, \ldots$ from Active. In particular, whenever we remove a node from Active we will have computed its strength. Simultaneously we propagate logic values along essential paths into the network.

For the sequel a signal s is a pair (w,st) with $w \in \{0,1,X\}$ and $st \in \mathbb{K} \cup \Gamma \cup \{\infty\}$. We order $\mathbb{K} \cup \Gamma \cup \{\infty\}$ by $K_1 < \ldots < K_q < \gamma_1 < \ldots < \gamma_m < \infty$
Set $S = \{0,1,X\} \times (\mathbb{K} \cup \Gamma \cup \{\infty\})$ is the set of signals. Define

$o: \Gamma \times S \to S$ by

$$\gamma \circ (w,st) = \begin{cases} (w,st) & \text{if } st \in \mathbb{K} \\ (w,\min(\gamma,st)) & \text{if } st \in \Gamma \cup \{\infty\} \end{cases}$$

and $v: S \times S \to S$ by

$$(w_1,st_1) \vee (w_2,st_2) = \begin{cases} (w_1,st_1) & \text{if } st_1 > st_2 \text{ or } st_1 = st_2 \text{ and } w_1 = w_2 \\ (w_2,st_2) & \text{if } st_1 < st_2 \\ (X_1\ st_1) & \text{if } st_1 = st_2 \text{ and } w_1 \neq w_2 \end{cases}$$

Finally, we order S by $(w_1,st_1) \leq (w_2,st_2)$ if $st_1 < st_2$ or $(st_1 = st_2$ and $(w_1 = w_2$ or $w_2 = X))$.

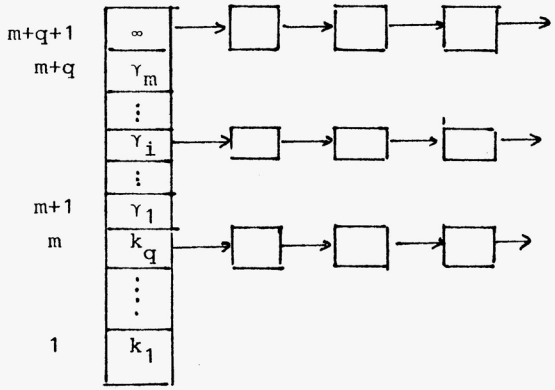
The following algorithm computes Equ in the case of complete transistor states. Let zk be a node state, zt be a complete transistor state and $I \subseteq N$ be an input set. Then zk' = Equ(zk,zt) is computed by

```
(1)  for all i ∈ I do Signal[i] ← (zk(i),∞) od;
(2)  for all k ∈ N - I do Signal[i] ← (zk(k),cap(k)) od;
(3)  Active ← K;
(4)  while Active ≠ ∅
(5)  do select k ∈ Active with maximal signal strength;
(6)     delete k from Active;
(7)     for all closed transistors t with {drain(t), source(t)}
            = {k,h} for some node h
(8)     do  s ← Signal[h] v strength(t) o Signal[k];
(9)         if  s ≠ Signal[h]
(10)        then Signal[h] ← s;
(11)             Active ← Active ∪ {h}
(12)        fi
(13)    od
(14) od
(15) for all v ∈ V do zk'(v) ← w where Signal[v] = (w,st)
                        for some st
                    od
```

We have

Theorem: The algorithm above correctly computes Equ for complete transistor states. Moreover, it can be made to run in time $O(|T|)$.

Proof: For the correctness proof we refer the reader to [MNN 82]. The bound on the running time can be seen as follows. We represent set Active by a bitvector BACTIVE [1..n] with $k \in$ Active iff BACTIVE[k] = true, an array of linear lists



where the i-th list contains all nodes in Active of strength i, a pointer max which points to the top-most non-empty list and an array of pointers of length $|K|$. If $k \in$ Active then P[k] points to the location of node k in the structure of linear lists described above. With these data structures all operations on set Active take time O(1). Whenever the last element of a list is deleted we need to reset pointer max. Because max is never increased (!!) we only need to scan down the array of lists until we find the next non-empty list. Finally observe, that every node is removed from Active at most twice. Thus running time is $O(|K| + |T|) = O(|T|)$ since $K \leq 3T$. □

Extension to incomplete transistor states is quite simple. We use the algorithm above to dynamically compute two complete extensions zt_0 and zt_1 of incomplete transistor state zt. In extension zt_0 we propagate 0 and X are far as possible and in

extension zt_1 we propagate 1 and X as far as possible. We compute zt_o and $zk_o = Equ(zk, zt_o)$ by the algorithm described above with line (7) replaced by

(7) for all transistors t with {drain(t),source(t)} = {k,h} for some h and either zt(t) = closed or zt(t) = X and Signal[h] $\in$ {0,X} x ($\mathbb{N} \cup \{\infty\}$)

(7a) do $zt_o(t) \leftarrow$ closed;

All transistors t with zt(t) = X which are not explicitly closed in line (7a) are open in zt_o. Note that the algorithm above closes a transistor in the X-state only if this helps to propagate an 0 or X. A similar algorithm (replace {0,X} by {1,X} in line 7) is used to compute zt_1 and zk_1. Then

$$Equ(zk,zt)(v) = \begin{cases} zk_o(v) & \text{if } zk_o(v) = zk_1(v) \\ X & \text{otherwise} \end{cases}$$

Theorem: The algorithm above correctly computes the equilibrium state. It runs in time $O(|T|)$.

Proof: See [MNN 82] □

At this point we arrived at an efficient algorithm for computing the equilibrium state. Thus one clock cycle, i.e. function settle, is simulated in time $O(h \cdot |T|)$ where h is the number of iterations required by the basic simulation algorithm. Note that each iteration takes time $O(|T|)$ by the results above. In general, h is a non-trivial number. For example in the case of combinatorial logic h is the depth of the network.

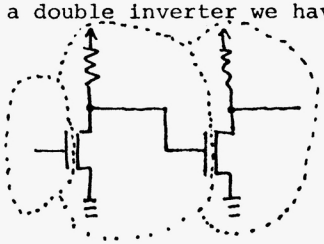
There are several methods for improving the efficiency. We briefly describe two:

1) Simulation in topological order

In networks there is natural direction in which the information flows. In particular, information always flows from the gate of a transistor to its drain and source because the state of the gate determines the state of the transistor which in turn influences the state of its drain and source nodes. In many networks this flow of information is acyclic. We capture this idea in the following definition.

Let zk be a node state and let $I \subseteq N$. Generate a graph $G = (N,E)$ with $E = \{(v,w);\ (v,w) = (\text{drain}(t),\text{source}(t))\}$ for some transistor t and either $\text{gate}(t) \notin I$ or $\text{gate}(t) \in I$ and $\delta(\text{type}(t),zk(\text{gate}(t)) = \text{closed}$, i.e. in graph G we close all transitors which are not known to be open during the entire clock cycle. Let $V_1,V_2,\ldots,V_r$ be the connected components of G. We say that V_i influences V_j if there is a transistor t with $\text{type}(t) \neq d$, whose gate is in V_i and whose drain and source are in V_j. Finally, we define a network to be acyclic (with respect to state zk and input set I), if the influence-relation is acyclic.

Example: In a double inverter we have three connected components



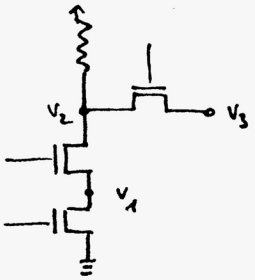
indicated by the dotted boxes. The network is acyclic. □

For acyclic networks one might pursue the following strategy. Order the connected components according to the influence relation. Then simulate the first component as described above until it settles. Note that it will settle immediately, because there is no feedback within a component. Then simulate the second component, ...

Simulation in topological order takes linear time for an entire clock cycle. Unfortunately, this strategy does not yield the same results as the simulation described above in general. The reason for this is that it uses a different timing assumption. We leave it to the reader to find a counter example. However, there is an important subclass of the acyclic networks for which the modified algorithm is equivalent to the original algorithm.

Let $O \subseteq N$ be a set of output nodes. A node is called strong if it is connected to an input node by transistors of type d. Recall that type d transistors model pull-up devices. A node v is inessential if all paths from v to an output node or to the gate of a transistor pass through a strong node. A node v is essential if it is not inessential.

Example: Consider a NMOS nand-gate and assume $v_3 \in O$. Then v_2 is strong, v_3 is essential and v_1, v_2 are inessential. □



Note that in a MOS-circuit every node can serve as a memory cell. We only have to isolate it from the remainder of the circuit. In particular node v_1 in the example above can store a bit. However, the value stored in v_1 does not influence the future computation because v_1 is inessential. This observation is captured in

Lemma 1: Let zk_0, zk_1 be states which agree on all input nodes and all essential nodes. Let $zk_i' = settle(zk_i, I)$, $i = 0,1$, i.e. the network settles in state zk_i' when started in zk_i. Then $zk_0'(v) = zk_1'(v)$ for all essential nodes v.

Proof: Let zk_i^j , $j = 0,1,2,\ldots$ be the sequence of states computed by the basic simulation algorithm when started with zk_i , $i = 0,1$. It is easy to show by induction on j that $zk_0^j(v) = zk_1^j(v)$ for all essential nodes v. For $j = 0$ there is nothing to show. For the induction step observe first that the components defined in the definition of equilibrium are the same because gates of transistors are controlled by essential nodes. This finishes the argument for non-isolated components. For isolated components we only have to observe that either all nodes in the component are inessential or all are essential.

□

From Lemma 1 we obtain

Lemma 2: Let G be an acyclic network, let zk be a node state, let $I \subseteq N$ be an input set. If in zk' = settle(zk,I) there are no isolated essential nodes then settle(zk,I)(v) = settle(zk,I)(v) for all essential nodes where $settle_{top}$ is computed by the modified algorithm described above (simulation in topological order).

Proof: Let $V_1, V_2, V_3, \ldots$ be the components of the control graph sorted according to relation "influences". Assume that the claim is wrong. Let i be minimal such that there is $v \in V_i$, v is essential, non-isolated and $settle(zk,I)(v) \neq settle_{top}(zk,I)(v)$. Since i is minimal and all transistors between nodes in V_i are controlled by essential nodes in V_j, $j < i$, both simulations determine identical transistor states for the transistors connecting nodes in V_i. Hence the same value is computed in both simulations for all non-isolated nodes in V_i, a contradiction.

□

Lemma 2 tells us that simulation in topological order works correctly for a large class of networks. This class of networks includes all combinatorial networks. Sorting in topological order computes the settling state in time $O(|T|)$. This is a significant improvement over the basic algorithm.

2) Local simulation

In many cases changing the value of an input node influences only a small part of the network. This observation can be built into the simulator easily. We only have to initialize Active differently, namely to all nodes which are drain or source of a newly set transistor and all nodes reachable from these nodes by closed or undetermined transistors. For details we refer the reader to [MNN 82].

Computational experience with the simulator is quite favorable. Typically, a clock cycle takes about 0.2 msec per transistor on a Siemens 7760.

We want to close with a short remark about the delay assumption. The basic simulator is based on the unit delay assumption. There have been several proposals to extend switch level simulation such that propagation delays are included ([D], [HHL]). In lemma 2 above we went into the opposite direction. Lemma 2 above states in a certain sense that the settling state is independent of the particular propagation delays. Information of this sort could be quite important in symbolic layout systems. Note that in these systems simulation usually precedes compaction and hence simulation has to be done without precise knowledge of delays. It would therefore be very desirable to have a simulator or a network analyzer which indicates that a network is hazard-free no matter what the propagation delays are. Results in this direction will be reported in [Na 83].

References (for chapters 1 and 2)

[B 81] G.M. Baudet: "On the Area Required by VLSI circuits" CMU Conference of VLSI Systems and Computations, 1981, pp. 100-107

[BB 82] B. Becker: Interner Bericht, FB 10 - Universität des Saarlandes, 1982

[BK 81] R.P. Brent, H.T. Kung: "The Area-Time Complexity of Binary Multiplication", Journal of the ACM, Vol. 28, No. 3 (1981), pp. 521-534

[CM 81] B. Chazelle, L. Monier: "A Model of Computation for VLSI with Related Complexity Results", 13th Annual ACM-STOC Conference (1981), pp. 318-325

[K 82] R. Kolla: "Untere Schranken für VLSI", Diplomarbeit, FB 10, Universität des Saarlandes, Saarbrücken, West Germany (1982)

[LM 81] T. Lengauer, K. Mehlhorn: "The Complexity of VLSI Computations", CMU-Conference on VLSI Systems and Computations (1981), pp. 89-99

[LS 81] R.J. Lipton, R. Sedgewick: "Lower Bounds for VLSI", 13th Annual ACM-STOC Conference (1981), pp. 300-307

[L 81] W.K. Luk: "A Regular Layout for a Multiplier of $O(\log^2 N)$ time", CMU-Conference of VLSI Systems and Computations (1981), pp. 100-107

[MC 80] C. Mead, L. Conway: "Introduction to VLSI Systems", Addison Wesley, Reading, Mass. (1980)

[MM 82] K. Mehlhorn, E. Meinecke-Schmidt: "Las Vegas is better than Determinism in VLSI and Distributed Computing", 14th Annual ACM-STOC Conference (1982), pp. 330-337

[PV 80] F.P. Preparata, J. Vuillemin: "Area-Time Optimal VLSI Networks for Multiplying Matrices", Inf. Proc. Let., Vol. 11, No. 2 (1980), pp. 77-80

[PV 81] F.P. Preparata, J. Vuillemin:"Area-Time Optimal VLSI Networks for Computing Integer Multiplication and Discrete Fourier Transform", 8th Int. Colloq. on Automata Theory Languages and Programming (Springer Lecture Notes in Comp. Sci., No. 115), (1981), pp. 29-40

[T 80] C.D. Thompson: "A Complexity Theory for VLSI, Ph.D. Thesis, Dept. of Comp. Sci., Carnegie-Mellon University (1980)

[V 80] J. Vuillemin: "A Combinatorial Limit to the Computing Power of VLSI Circuits", 21st Annual IEEE-FOCS Symposium (1980), pp. 294-300

[V 83] J. Vuillemin: "A very fast Multiplication Algorithm for VLSI Implementation", INRIA Report 183 (1983)

[Y 79] A.C. Yao: "Some Complexity Questions Related to Distributive Computing", 11th Annual ACM-STOC Conference (1979), pp. 209-213

[Y 81] A.C. Yao: "The Entropic Limitations on VLSI Computations", 13th Annual ACM-STOC Conference (1981), pp. 308-311

References (for chapter 3)

[B 80] R.E. Bryant: "An Algorithm for MOS Logic Simulation", Lamba Magazine (now VLSI Magazine) 1980, Fourth Quarter, pp. 46-53

[B 81a] R.E. Bryant: "A Switch-Level Simulation Model for Integrated Logic Circuits", Ph.D. Thesis, MIT, March 1981, Report MIT/LCS/TR-259

[B 81b] R.E. Bryant: "A Switch-Level Model of MOS Logic Circuits", VLSI Conference, Edinburgh 1981, pp. 329-340

[B 83] R.E. Bryant: "A Switch-Level Model and Simulator for MOS Digital Systems", Caltech, CS Technical Report No. 5065, 1983

[CABBAGE] M.Y. Hsueh: "Symbolic Layout and Compaction of Integrated Circuits", Ph.D. Thesis, EECS Division, University of California, Berkeley, CA (1979)

[D] D. Dumlugöl, H. de Man: "Logmos: A MOS transistor oriented logic simulator with assignable delays", Technical Report, Univ. Louvain

[FLOSS] R.A. Auerbach, B.W. Lin, E.A. Elsayed: "Layouts for the Design of VLSI Circuits", Computer Aided Design, Vol. 13, No. 5 (1981), pp. 271-276

[HHL] M.H. Heydeman, G.D. Hachtel, M.R. Lightner: "Implementation Issues of Multiple Delay Switch Level Simulation", Techn. Report, Dept. of Electr. Eng. and Comp. Sci., University of Colorado, Boulder, Color.

[HILL 82] T. Lengauer, K. Mehlhorn: "HILL - Hierarchical Layout Language, A CAD System for VLSI Design", TR A82-10, FB 10, Univ. d. Saarlandes, Saarbrücken (1982)

[HILL 83] T. Lengauer, K. Mehlhorn: "Report on the HILL Specification Language", TR A 83/05, FB 10 - Informatik, Univ. d. Saarlandes, Saarbr. (1983)

[GW 82] G. Kedem, H. Watanabe: "Optimization Techniques for IC Layout and Compaction", TR 117, Dept. of Comp. Sci., University of Rochester, Rochester, N.Y., (1982)

[G 80] M.C. Golumbic: "Algorithmic Graph Theory and Perfect Graphs", Associated Press (1980)

[L 82a] T. Lengauer: "On the Solution of Inequality Systems Relevant to IC Layout", Proc. of the 8th Workshop on Graphtheoretic Methods in Comp. Sci. (WG 82), Hanser Verlag, München (1982)

[L 82b] T. Lengauer: "The Complexity of Compacting Hierarchically Specified Layouts of Integrated Circuits", 23th FOCS (1982), pp. 358-369

[L 83] T. Lengauer: "Efficient Algorithms for the Constraint Generation for Integrated Circuit Layout Compaction", 23th FOCS(1982), pp. 358-369

[M 83] K. Mehlhorn: "Data Structures and Efficient Algorithms", Springer Verlag, to appear

[MNN 82] K. Mehlhorn, St. Näher, M. Nowak: "HILLSIM: Ein Simulator für MOS-Schaltkreise", TR A 82/08, FB 10, Univ. d. Saarlandes, Saarbrücken (1982)

[MS 83] K. Mehlhorn, B. Schmidt: "A Single Source Shortest Path Problem for Graphs with Separators", Proc. of FCT Conference 1983, LNCS, to appear

[MULGA] N.H.E. Weste: "MULGA - An Interactive Symbolic Layout System for the Design of Integrated Circuits", The Bell System Technical Journal, No. 60, Vol. 6 (1981), pp. 823-857

[Nä 83] St. Näher: Diplomarbeit, Univ. d. Saarlandes, in preparation (1983)

[No 83] M. Nowak: Diplomarbeit, Univ. des Saarlandes, in preparation (1983)

[Sch 83] B. Schmidt: Doktorarbeit, FB 10, Univ. d. Saarlandes, in preparation (1983)

[SLIM] A.E. Dunlop: "SLIM - The Translation of Symbolic Layouts into Mask Data", Proc. of the 17th Design Automation Conference, IEEE (1980), pp. 595-602

[STICKS] J.D. Williams: "STICKS - A Graphical Compiler for High Level LSI Design", Nat. Comp. Conf. (1978), pp. 289-295

[TRICKY] A. Hanczakowski: "TRICKY - Symbolic Layout System for Integrated Circuits", VLSI Spring Compcon (1981)