Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

J.C. Ebergen

On VLSI design

Department of Computer Science Note CS-N8408 September

ON VLSI DESIGN

J.C. EBERGEN
Centre for Mathematics and Computer Science, Amsterdam

Some of the problems in VLSI design are discussed. A VLSI design method is presented with which these problems may be tackled. An example is provided to illustrate some parts of the design method.

1980 MATHEMATICS SUBJECT CLASSIFICATION: 68C01, 68F05, 94C99.
KEY WORDS & PHRASES: VLSI design, functional specification, regular expressions, layout of a circuit.
NOTE: This report is a slightly revised version of the paper presented at the NGI/SION conference on 16/17 April 1984.

Note CS-N8408
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

1. INTRODUCTION

Since in the last decade the promising possibilities of VLSI systems have become realizable, more and more attention has been paid to efficient and reliable design methods for such systems. Presently, a number of design methods for VLSI systems are being employed. Many of them use computer-aided design tools such as circuit extractors, simulators, and design rule checkers for the automatic verification of parts of the design.

In this article we discuss a somewhat different VLSI design method. In this method the design route of VLSI systems, i.e. the route from specification to finished product, is divided roughly into four levels, i.e.

(1) the formulation of a functional specification;

(2) the design of a program satisfying that specification;

(3) the computation of a layout of a circuit realizing that program;

(4) the computation of a test procedure for that circuit.

When the above design route has been carried out, chips can be fabricated from the layout computed and then tested with the test procedure computed. It is our intention to design VLSI systems, or components as we call them, in the order from (1) through (4) and every end product of a certain level is considered the specification for the end product of the next level. An end product that does not satisfy its specification is considered to be an erroneous implementation of this specification. In this way a component is designed in a top-down manner. In the sequel we address each of the levels (1) through (4) separately. The parts (1) and (2) are dealt with in greater detail than (3) and (4).

In (1) and (2) we have chosen a level of abstraction that ignores physical properties and irrelevant details to such an extent that both functional specifications and programs for a component can be expressed conveniently in a mathematical formalism. Such a formalism should enable us not only to verify the correctness of a program, but also to derive a program for a component. For this formalism we have chosen trace theory. Trace theory can be considered as an extension of a part of language theory. A brief introduction to trace theory is presented in section 3.

In the development of trace theory we have been led by the opinion that ease of programming is to be preferred to ease of implementing. This does not mean that the design of programs is easy. On the contrary, we believe that programming is a difficult task and we have no hope in automating it. It is, however, our hope and ultimate goal that the computation of a layout for a program can be automated. Thus, a so-called silicon compiler can be designed that computes a corresponding layout for a given program. It is also our hope that the computation of a test procedure can be automated. In this way, the design of components is reduced to the design of programs for those components.

One of the major problems, if not the major, in VLSI design is the complexity, no matter which method is used. To design a system with hundreds of thousands of transistors properly interconnected such that a number of constraints, e.g. proper timing, are satisfied is an overwhelmingly complex task. This complexity needs to be bridled. Therefore, we adopt a hierarchical approach in the design task at each level and a greatest possible separation between these design tasks. As for the hierarchical approach, when at a certain level a hierarchical composition is prescribed for a component, this hierarchy is maintained at the succeeding levels. Consequently, if a component can be composed of n identical subcomponents, say, then the design of the layout for that component may be reduced to the design of a layout for one subcomponent, which is copied n times, and a proper interconnection between these layouts. As for the separation of the design tasks, we do not consider specifications that impose real-time or area constraints on a component. In this way we hope to carry out the design tasks as independently as possible.

2. THE FORMULATION OF A FUNCTIONAL SPECIFICATION

If we want to give a functional specification of a component, we have to ask ourselves how we characterize a component formally. For this formal characterization we have chosen a trace structure. A trace structure R is an ordered pair $<B,X>$ such that $B \subseteq U$ and $X \subseteq B^*$. Throughout this article U denotes a finite, but sufficiently large, set of symbols and B^* the set of all finite sequences of symbols from B. A finite sequence of symbols is also called a trace. The empty trace is denoted by ϵ. B is called the alphabet of the trace structure R and is also denoted by αR. X is called the trace set of R and is also denoted by tR. The alphabet αR characterizes all communication actions a component is capable of, where actions are to be considered instantaneous and indivisible. The trace set X characterizes all finite sequences of communication actions in which the component can engage.

Henceforth, we give a functional specification of a component by giving a specification for its trace structure. The functional specifications that we use in this article are based on predicates on traces. Before we present this specification method, we introduce some preliminary definitions. Let P be a predicate on traces and B a set of symbols with $B \subseteq U$. We say that P is an invariant of a trace t iff P holds for every prefix of t. P is called an invariant of a trace structure R iff P is an invariant of every trace of tR. The pair (B,P) is called a specification iff $\delta P \subseteq B$, where δP, the dependence alphabet of P, is defined by

$$b \in \delta P \equiv (\exists r,s :: P(rbs) \not\equiv P(rs))$$

for all $b \in U$ and traces r and s. The condition $\delta P \subseteq B$ expresses that the value of P depends on symbols from B only. Finally, we say that R satisfies the specification (B,P) iff $\alpha R = B$ and P is an invariant of R.

With the above definitions we can specify a trace structure R uniquely by specifying R as the greatest trace structure that satisfies the specification (B,P). Operationally speaking we, thus, require not only that any behavior of the component keeps a certain relation invariant, but also that the component can exhibit any behavior that keeps that relation invariant.

To illustrate this specification method we give a functional specification of a simple component which we call a k-counter. Let B and P be defined by

$$B = \{x,y\}$$
$$P(t) \equiv 0 \leqslant tNx - tNy \leqslant k$$

for all traces t, where tNx denotes the number of occurrences of symbol x in trace t and $k \geqslant 1$. Notice that $\delta P = \{x,y\}$ and that, consequently, (B,P) is a specification. The component $count_k(x,y)$ is specified as the greatest trace structure that satisfies (B,P). Operationally, one can interpret the component $count_k(x,y)$ as a component that records an integer value. The symbol x denotes an increment by 1 and the symbol y denotes a decrement by 1 of this value. The component $count_k(x,y)$ can engage in any sequence of increments and decrements as long as its value stays within the bounds 0 and k. (Notice that its initial value is 0.)

3. TRACE THEORY

When we have a specification for a component we compose that component from other, preferably simpler, components. Since a component is formally characterized by a trace structure, we, consequently, need composition methods for trace structures. In this section the basic operations on trace structures are introduced. For a further introduction to trace theory the reader is referred to [1,6]. The reader may also find interesting material, which differs slightly from [1,6], in [3].

The first binary operation on trace structures that we describe is called weaving. Weaving characterizes the simultaneous cooperation of two components. While cooperating these components can communicate through common actions, i.e. actions that are characterized by $\alpha R \cap \alpha S$, where R and S are the trace structures of the two components. We consider a communication between

components as a simultaneous mutual participation in a common action. Accordingly, common actions take place upon mutual agreement and, consequently, through common actions components synchronize their behaviors. Thus, the behavior of the simultaneous cooperation of two components can be any behavior, restricted to the set of actions $aR \cup aS$, that it is in accordance with the behaviors of the components in isolation. The weave of R and S, denoted by $R \,,, S$, is defined by

$$R \,,, S = \langle aR \cup aS \,, \{t \mid t \in (aR \cup aS)^* \wedge t \lceil aR \in tR \wedge t \lceil aS \in tS \}\rangle$$

where $t \lceil B$ denotes the projection of t on B, i.e. the trace that remains when all symbols outside B are deleted from t.

Besides the projection of a trace t on an alphabet B, we also define the projection of a trace structure R on an alphabet B, denoted by $R \lceil B$, by

$$R \lceil B = \langle aR \cap B \,, \{t \lceil B \mid t \in tR \}\rangle.$$

By projection we can 'hide' internal communications between components, i.e. we can eliminate common symbols from the weave of two trace structures and, consequently, describe the net effect of a composition. This operation is called blending. The blend of R and S, denoted by $R \,, S$, is defined by

$$R \,, S = (R \,,, S) \lceil (aR \div aS)$$

where $\div$ denotes symmetric set difference. In contrast to weaving, blending is in general not associative. We do have that if $aR \cap aS \cap aT = \varnothing$, then $(R \,, S) \,, T = R \,, (S \,, T)$.

A component that can engage in a finite sequence of actions, can also engage, operationally speaking, in any prefix of that sequence of actions. We, therefore, define the prefix of a trace structure R, denoted by ***pref*** R, by

$$\textit{pref}\, R = \langle aR \,, \{t \mid (\exists s :: ts \in tR)\}\rangle.$$

We say that a trace structure is prefix-closed iff $\textit{pref}\, R = R$. For each component initially the sequence of communication actions performed is the empty sequence ϵ. Therefore, we are especially interested in trace structures, as characterizations of components, that are not only prefix-closed but also non-empty. For prefix-closed and non-empty trace structures we have that their weave and their blend is again prefix-closed and non-empty. Furthermore, if R is specified as the greatest trace structure that satisfies (B, P), then R is prefix-closed. If $P(\epsilon)$ holds as well, then R is also non-empty.

Concatenation, union, fixed repetition, and arbitrary, but finite, repetition are usually defined for sets. We define these operations for trace structures as well:

$$R ; S = \langle aR \cup aS \,, tR \, tS \rangle,$$

$$R \mid S = \langle aR \cup aS \,, tR \cup tS \rangle,$$

$$R^0 = \langle \varnothing, \{\epsilon\}\rangle \text{ and } R^{n+1} = R^n ; R \text{ , and}$$

$$[R] = \langle (\cup n : n \geq 0 : aR^n), (\cup n : n \geq 0 : tR^n)\rangle,$$

where concatenation of sets is denoted by their juxtaposition.

In order for a component to be realizable in an integrated circuit, it has to have, among others, a finite number of states. From language theory we know that we can translate this constraint into the condition that the component's trace structure has to be regular, i.e. its trace set has to be a regular set. We have that the set of regular trace structures is closed under weaving, blending, taking prefixes, concatenation, union, fixed repetition, and arbitrary repetition. Trace structures of the form $\langle \{b\}, \{b\}\rangle$, $b \in U$, and $\langle \varnothing, \{\epsilon\}\rangle$ are called atomic trace structures. An expression for a trace structure built up from atomic trace structures and the operators ,, , ; , | , and [ ] is called a command. (A command is similar to a regular expression in language theory.)

In order to condense lengthy commands, we use the following notational conventions. If no confusion can arise the trace structures $<\{b\},\{b\}>$ and $<\varnothing,\{\epsilon\}>$ may be abbreviated to b and ϵ respectively – notice that „ , ; , | , and [ ] are defined on trace structures only. As for the priority rules, „ has highest binding power, then ; , and then | , i.e. the smaller character has a higher binding power.

4. THE DESIGN OF A PROGRAM

It is our aim to implement a prefix-closed, non-empty, regular trace structure as an integrated circuit. Furthermore, we want to implement a blend of two such trace structures by an interconnection of the implementations of the trace structures. We, therefore, express a component as a blend of a number of prefix-closed, non-empty, regular trace structures.

For example the component $count_k(x,y)$, $k \geq 1$, as specified in section 2, can be expressed in a number of ways. The following equations are given without proof. They can, however, be derived by means of a derivation method based on the functional specifications defined in section 2. This derivation method is currently under investigation.

(4.0) $count_1(x,y) = \mathbf{pref}\ [x;y]$

(4.1) $count_{k+l}(x,z) = count_k(x,y) \,„\, count_l(y,z)$

(4.2) $count_{k+1}(x,y) = s.count_k(x,y) \,„\, \mathbf{pref}\ [x;s.x \mid x;y \mid s.y;y]$

(4.3) $count_{2k+1}(x,y) = s.count_k(x,y) \,„\, \mathbf{pref}\ [(x \mid s.y;y)\,;(s.x;x \mid y)]$

where $k \geq 1$, $l \geq 1$, and $s.count_k(x,y)$ denotes the trace structure in which every symbol $b \in \{x,y\}$ is replaced by $s.b$. From these equations we can deduce a number of compositions for the component $count_k(x,y)$, $k \geq 1$, that are all blendings of a number of prefix-closed, non-empty, regular trace structures.

A composition for a component is expressed in a program. In general a program has the following form.

com $C(B)$:

(0) sub $s_0{:}C_0(B_0)$, $s_1{:}C_1(B_1)$, $\cdots$, $s_{n-1}{:}C_{n-1}(B_{n-1})$

(1) $a_0{=}b_0$, $a_1{=}b_1$, $\cdots$, $a_{m-1}{=}b_{m-1}$

(2) S

moc

In line (0) n, $n \geq 0$, subcomponents are listed. We say that subcomponent s_i is of type $C_i(B_i)$. The trace structure of component s_i is $s_i.C_i(B_i)$. In line (1) m, $m \geq 0$, equalities are listed. The symbols constituting an equality are considered to be the same symbol. In line (2) a non-empty, regular trace structure is given in the form of a command. This command may be absent, in that case we take $S = <\varnothing,\{\epsilon\}>$. The above program defines the trace structure $C(B)$ by

(4.4) $C(B) = s_0.C_0(B_0) \,„\, s_1.C_1(B_1) \,„\, \cdots \,„\, s_n.C_n(B_n) \,„\, \mathbf{pref}\ S.$

Definition (4.4) is consistent if the alphabets of the right-hand side and the left-hand side of (4.4) are equal and blending is associative in (4.4). Therefore, we impose some conditions on the above program. Let A be defined by $A = B \cup s_0.B_0 \cup s_1.B_1 \cup \cdots \cup aS$. We require that

- each symbol in A occurs either in two distinct alphabets constituting A, or in one equality; and
- for each equality, its symbols belong to two distinct alphabets constituting A.

From (4.4) the correctness of a program can be derived if $C(B)$, the subcomponents $C_i(B_i)$, $0 \leqslant i < n$, and the command S are known.

For example, we can derive the following programs for $count_k(x,y)$, $k \geqslant 1$, from the equations (4.0) through (4.3).

(4.5) **com** $count_1(x,y)$: $[x;y]$ **moc**

(4.6) **com** $count_{k+l}(x,y)$:
sub $s:count_k(x,y)$, $t:count_l(x,y)$
$x = s.x$, $s.y = t.x$, $t.y = y$
moc

(4.7) **com** $count_{k+1}(x,y)$:
sub $s:count_k(x,y)$
$[x;s.x \mid x;y \mid s.y;y]$
moc

(4.8) **com** $count_{2k+1}(x,y)$:
sub $s:count_k(x,y)$
$[(x \mid s.y;y);(s.x;x \mid y)]$
moc

for $k \geqslant 1$ and $l \geqslant 1$.

We observe that there may be several programs for the same component. Consequently, we may choose the program that suits our particular needs, e.g. with respect to time efficiency, area efficiency, or ease of layout computation. Furthermore, if a command can be implemented in a fixed area and the interconnections of these areas do not need disproportionately more area, then we can also derive some properties of the area complexity of a component. A program for $count_k(x,y)$ similar to (4.6) or (4.7), for example, has an area complexity of $O(k)$. The area complexity of a program for this component similar to (4.8), however, is $O(\log k)$. For a further introduction to the design of programs for components the reader is referred to [4].

5. THE COMPUTATION OF A LAYOUT

Once a program is obtained for a component we compute a layout for this program. In computing a layout we exploit the hierarchy that is defined in the program. First, we compute layouts for the subcomponents and the command listed in the program and then we compute a suitable interconnection for these layouts. The layouts of the subcomponents are, of course, computed in the same way, consequently the layout of the component is computed in a bottom-up manner. If some commands or subcomponents are used repeatedly in the program we, obviously, copy their layouts, thus reducing the computation time for the layout even more. For further discussions on the computation of a layout for a command the reader is referred to [2,7].

One of the major problems in the computation of layouts for VLSI systems are timing problems. For certain layouts the delays incurred in the interconnection wires can cause an incorrect operation of the circuit composed. This can especially occur in circuits in which signals have to be distributed over long distances within a certain time period, as for example can occur in synchronous VLSI systems. These timing problems may become even more serious when one scales down the feature sizes of a chip. Therefore, it is attractive when layouts are such that the generated circuit is insensitive for any delays occurring in the interconnection wires. Such VLSI systems, also called self-timed systems, are described in [1,5,8].

To conclude this section we give a possible schematic of the program

com $count_4(x,y)$:

sub $s_0, s_1, s_2, s_3 : count_1(x,y)$

$x = s_0.x,\ s_0.y = s_1.x,\ s_1.y = s_2.x,\ s_2.y = s_3.x,\ s_3.y = y$

moc

The translation of programs into schematics has not been formalized yet. The example may, nevertheless, give an idea in what manner we want to solve this problem.

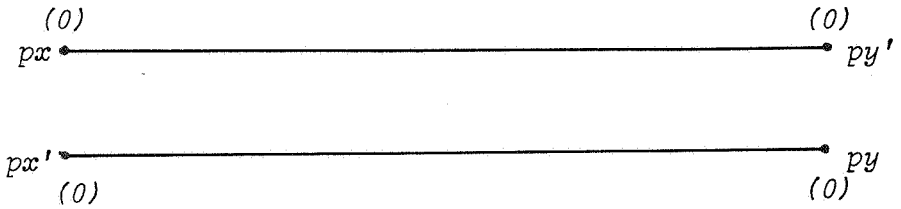

fig.1

First, we give a possible translation of $count_1(x,y)$ into a schematic. Making a circuit self-timed requires a kind of 'hand-shake' signalling in the communication protocol. Therefore, the symbols x and y are mapped onto the request-acknowledge pairs px,px' and py,py' respectively, where px,px',py, and py' represent voltage transitions in the marked points on the wires in fig. 1. We stipulate that the environment of this circuit behaves in such a way that requests and acknowledgements alternate and that the first occurrences, if any, for the pairs px,px' and py,py' are px and py' respectively. The zeroes in the figure denote the initial values at these points. The reader is encouraged to check that under the described stipulations this component can engage in exactly all finite sequences t of actions from $\{px,px',py,py'\}$ for which $0 \leqslant tNpx - tNpy \leqslant 1$ holds, irrespective of any delays that the propagation of transitions can incur in the interconnection wires.

Now consider fig. 2. The component $count_4(x,y)$ is blended of four subcomponents of type $count_1(x,y)$. By means of blending the subcomponents are synchronized. Synchronization takes place in this composition between the subcomponents s_0 and s_1, s_1 and s_2, and s_2 and s_3, i.e. in three places. In circuits synchronization can be realized by C-elements. The behaviour of a C-element in fig. 2 is as follows. The output of the C-element becomes 0 or 1 if both inputs are 0 or 1 respectively, and otherwise the output stays in its current state. Under the same stipulations as described for the component of fig. 1 the component can engage in exactly all finite seqeunces t of actions from $\{px,px',py,py'\}$ for which $0 \leqslant tNpx - tNpy \leqslant 4$ holds, irrespective of any delays that the propagation of transitions can incur in the interconnection wires.

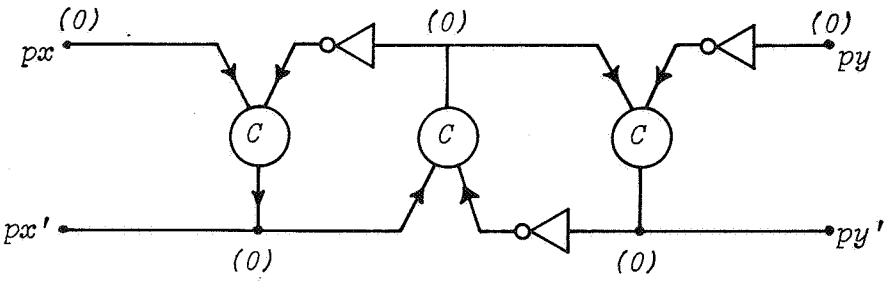

fig.2

6. THE COMPUTATION OF A TEST PROCEDURE

From the layout computed the chips are fabricated and subsequently tested. When we test a chip we test it for fabrication errors only. We do not test a chip for the implementation of the right program or the satisfaction of its functional specification. These verifications should be carried out at higher levels.

To test a chip one may want to test it in every possible state. Thus, the test time needed, however, can be very long since the number of states of the component can be very large. The number of states of the blend of two components with respective numbers of states m and n, is at most mn. Accordingly, the number of states of a component increases at most exponentially in the component's size and, consequently, also the test time for the component increases at most exponentially in its size. If, however, we can test each subcomponent with its interconnections separately the test time needed to test the component is proportional to its size. For example the component $count_k(x,y)$, which has $O(k)$ states, may, when composed with program (4.8), thus be tested in $O(\log k)$ time.

7. CONCLUDING REMARKS

In the preceding sections we have discussed a VLSI design method in which a hierarchical design and a separation of the several design tasks are the main objectives. The ultimate goal is that the design of a component can be reduced to the design of a program. Therefore, special attention is paid to an adequate formalism, trace theory, in which a functional specification and a program for a component can be expressed. With this formalism more than one program may be derived for a component, thereby providing a greater freedom for choosing a suitable program for further implementation. From a program important properties with respect to the area complexity or test time of a component may be derived. For the realization of this design method, however, still a number of problems have to be overcome.

8. ACKNOWLEDGEMENTS

Acknowledgements are due to Martin Rem, Jan van de Snepscheut, and the other members of the Eindhoven VLSI Club.

References

[1] Ebergen, J.C., Trace Theory and Self-timed Systems, M. Sc. Thesis, Dept. of Math. and Computing Sc., Eindhoven University of Technology, (April 1983).

[2] Floyd, R.W. and Ullman, J.D., The Compilation of Regular Expressions into Integrated Circuits, *Journal of the ACM* **29** (1982) 603-622.

[3] Hoare, C.A.R., Notes on Communicating Sequential Processes, Technical Monograph PRG-33, Oxford University Computing Laboratory (Aug. 1983).

[4] Rem, M., Trace Theory and the Design of Concurrent Computations, lecture notes for the colloquium 'Parallelle Computers en Berekeningen', to appear in a C.W.I. syllabus.

[5] Seitz, C.L., System Timing, in: Mead, C. and Conway, L., *Introduction to VLSI Systems,* Addison Wesley (1980).

[6] van de Snepscheut, J.L.A. , Trace Theory and VLSI Design, Ph. D. Thesis, Eindhoven University of Technology (Sept. 1983).

[7] van Lierop, M.L.P., A Flexible Bottom-up Approach for Layout Generation, Internal Report, Dept. of Math. and Computing Sc., Eindhoven University of Technology, To appear.

[8] Udding, J.T., Classification and Composition of Delay-insensitive Circuits, Ph. D. Thesis, Eindhoven University of Technology, To appear.