The VLSI-programming language Tangram and its translation into handshake circuits

5 Author(s) K. van Berkel ; J. Kessels ; M. Roncken ; R. Saeijs ; F. Schalij View All Authors

Abstract:
Views VLSI design as a programming activity. VLSI designs are described in the algorithmic programming language Tangram. The paper gives an overview of Tangram, providing sufficient detail to invite the reader to try a small VLSI program himself. Tangram programs can be translated into handshake circuits, networks of elementary components that interact by handshake signaling. The authors have constructed a silicon compiler that automates this translation and converts these handshake circuits into asynchronous circuits and subsequently into VLSI layouts.

Published in: Design Automation. EDAC., Proceedings of the European Conference on

Date of Conference: 25-28 Feb. 1991

Date Added to IEEE *Xplore*: 06 August 2002

INSPEC Accession Number: 4077128

DOI: 10.1109/EDAC.1991.206431

Publisher: IEEE

Conference Location: Amsterdam, Netherlands, Netherlands

Keywords

IEEE Keywords
Very large scale integration, Computer languages, Costs, Silicon compiler, Finite impulse response filter, Programming profession, Laboratories, Algorithm design and analysis, Asynchronous circuits, Reactive power

INSPEC: Controlled Indexing
VLSI, circuit layout CAD, high level languages

INSPEC: Non-Controlled Indexing
VLSI layouts, Tangram, handshake circuits, VLSI designs, algorithmic programming language, elementary components, silicon compiler, asynchronous circuits

Authors

K. van Berkel
Philips Res. Labs., Eindhoven, Netherlands

The VLSI-programming language Tangram and its translation into handshake circuits

Kees van Berkel Joep Kessels Marly Roncken Ronald Saeijs Frits Schalij

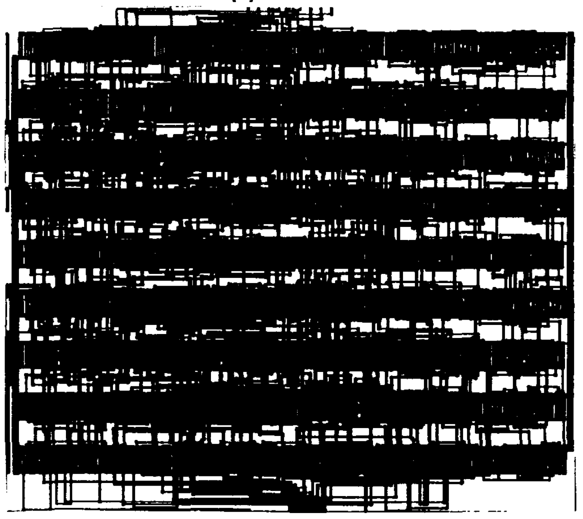
Philips Research Laboratories,
P.O. Box 80.000, 5600 JA Eindhoven, The Netherlands

Abstract

In this paper we view VLSI design as a programming activity. VLSI designs are described in the algorithmic programming language Tangram. The paper gives an overview of Tangram, providing sufficient detail to invite the reader to try a small VLSI program himself. Tangram programs can be translated into handshake circuits, networks of elementary components that interact by handshake signaling. We have constructed a silicon compiler that automates this translation and converts these handshake circuits into asynchronous circuits and subsequently into VLSI layouts.

Introduction

The picture below shows a VLSI layout generated by the Tangram (silicon) compiler. The input for this layout was the Tangram program *BUBBLE*, a cell of a systolic block sorter as described in [2].



For completeness the Tangram program *BUBBLE* is repeated:

```
N= const 64   &   C= type (0..127)
|
(a?C  &  b!C).
|[ x, y: var C
| #[ a?y
  ; #(N-1)[ a?x
          ; case(x>y):[true → b!y; y:=x [] false → b!x]
          ]
  ; b!y
  ]
]|
```

A VLSI program is the description of a VLSI circuit in an algorithmic language. VLSI programming denotes the systematic development of such programs from specifications, such that the corresponding compiled circuits satisfy the specified cost and performance constraints. Our VLSI-programming language Tangram is based on Hoare's work on Communicating Sequential Processes ([5] and [6]). CSP form an attractive basis for a VLSI-programming language:

- It is one of the best studied approaches to concurrency and communication in computing science. A body of theory exists to assist in the specification and systematic development of programs (e.g. [15]).
- CSP is a general purpose language. The most diverse VLSI-systems have been described in CSP-based programming languages, including: mutual exclusion on a ring [12], various systolic arrays [15], a block sorter (as part of a source-encoder of a digital video recording application) and FIR filters [2], a graphics processor [4], a microprocessor [14], a systolic rank-order filter [8], various types of queues [11], various Compact Disc data-processing elements [10] and a regular-expression recognizer [9].
- For a given functional specification it is often possible to develop a range of programs, such that the corresponding compiled circuits cover a significant part of the low-cost to high-performance spectrum of VLSI implementations [2]. Simple rules allow the VLSI programmer to predict the cost/performance

2130/91/0000/0384$01.00 © 1991 IEEE

properties of the compiled circuit on the basis of his program.

- Programs can be translated fully automatically into efficient VLSI circuits (Cf. [13] and [3]).

Tangram is a simple general-purpose VLSI programming language. It enables system designers without in-depth knowledge of VLSI circuits and IC technology to approach the design of complex VLSI systems as a programming task. We expect that VLSI programming and silicon compilation will result in considerable savings in design costs as well as design time.

The work presented here is a continuation of the work reported in [2] and [3]. Significant progress has been made with respect to simplification, generalization, formalization and implementation.

The paper addresses the principles behind the silicon compiler in four sections

- the VLSI-programming language Tangram,
- the intermediate representation *handshake circuits*,
- the translation of Tangram programs into handshake circuits,
- some implementation aspects, including a sketch of the implementation of handshake circuits as asynchronous VLSI circuits.

Since May 1990 we have been compiling Tangram programs automatically into CMOS standard-cell layouts.

Tangram[1]

The language Tangram is based on CSP [6]. In Tangram the programmer can indicate whether commands (statements) are to be executed sequentially or concurrently. Mutually concurrent commands can interact by synchronized communications along so-called channels. We distinguish directed channels along which values can be communicated from undirected channels, which are merely used for synchronization. A directed channel connects one output port to one or more input ports (broadcasting is allowed). A communication along a channel requires the simultaneous participation of all communication commands involved.

Tangram contains several elementary commands; the most important ones are discussed below.

- Assignment command $x{:=}E$.
  The value of expression E is assigned to variable x.

[1]Tangram is an ancient Chinese puzzle. It consists of seven elementary pieces (five triangles, one square and one rhomboid) and a simple compostion rule (pieces may not overlap). It allows the construction of a nearly endless variety of fascinating shapes.

- Output command $a!E$.
  The value of expression E is output through output port a.
- Input command $a?x$.
  A value is input through input port a and stored in variable x. The types of port a and variable x have to be the same.
- Synchronization command $b^{\sim}$.
  Such a command achieves a synchronization among all commands connected to channel b.

There are several ways to construct new commands from existing ones, say T and U.

- Sequential composition $T\ ;\ U$.
  First T is executed and subsequently U. Ports occurring in both commands must have the same direction.
- Concurrent composition $T \,||\, U$.
  Commands T and U are executed concurrently. The compound command terminates when both subcommands are terminated. Both commands can only interact by communication along channels. When one command contains an assignment (or input) to a variable, the other command may not access this variable. The || operator binds more then the ; operator.
- Infinite repetition $\#[T]$.
  Command T is repeated infinitely often and, therefore, this command never terminates.
- Finite repetition $\#N[T]$
  Command T is executed N times, with N a natural constant.
- The case command allows the selection of one command out of a set depending on the value of an expression. For instance, the value of boolean expression B in
  case B : $[\textbf{true} \rightarrow T \,[\!]\, \textbf{false} \rightarrow U]$
  determines whether T or U is executed.
- Block command $|[L \mid T]|$,
  where L is a (possibly empty) list of definitions and declarations. All variables and channels used in a command have to be declared in a surrounding block. The language allows the definitions of, for instance, constants and types. Several definitions and declarations can be combined into a list using the separator symbol &. For example, the list
  bit = **type**(0..1) & ***b*** : **var** ***bit***
  defines a type ***bit*** and declares a variable ***b*** of that type.

A Tangram program has the form $(P).T$, where P is a list of ports and T a command. The port list P contains all external ports, i.e. the ports through which the corresponding VLSI circuit communicates with its environment. For each port it is indicated whether it is an input port, output port or undirected port. For each directed port its type is also indicated. The command T describes the behavior of the circuit. An alternative form of a Tangram program is $L \mid (P).T$, where L is a list of definitions that may be used in P and T.

With this brief overview of Tangram the reader should be able to understand the program of *BUBBLE* in the introduction. The additional constructs include guarded commands, arrays, tuples, arithmetic operators, procedures and functions.

Tangram is similar to Occam [7]. Most differences can be understood from the differences between the target media: VLSI circuits vs. Transputer networks. E.g. Tangram supports broadcast, finer data types and sharing of functions, but does not provide facilities for assigning procedures and channels to physical processors and physical channels.

Handshake circuits

The synchronization primitive of CSP is sometimes referred to as the "CSP handshake". Indeed, this synchronization can be implemented by handshake signaling (Cf. [16]). Although most practical implementations are based on a 4-phase handshake, a 2-phase protocol simplifies the further presentation and suffices to capture the essence of handshake signaling.

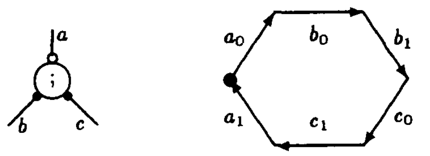
In its simplest form 2-phase handshake signaling is described for a channel connecting exactly two processes. One process, the *active* one, may issue a "request" signal along that channel (phase 1). The other process, the *passive* one, may respond to that signal by issuing an "acknowledge" signal along the same channel (phase 2); the reception of this signal by the former process completes the handshake. For channel a the two signals will be denoted by a_0 and a_1 respectively.

In the case of directed communications the data may be encoded in either the request or the acknowledge (bi-directional data transfer can also be considered, but is not part of Tangram).

When we adopt handshake signaling for the implementation of Tangram synchronization, communicating Tangram processes will be implemented as networks of "components" (one for each process) that interact by handshake signaling. When we carry this line of thinking one step further, one may wonder if such a Tangram process can be decomposed into a network of such "handshake components" drawn from a limited set of different components. Indeed, this is possible. Such networks will be called *handshake circuits* (formerly called abstract circuits, cf. [2]). The implementation of Tangram requires a relatively small set of different handshake components, basically one for each primitive concept of the language.

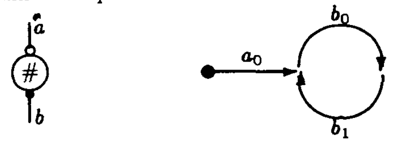
The interface of a handshake component to the external world consists of a set of named *ports*. Ports are either passive or active, depending on their role during a handshake. Ports may be undirected ("synchronization only") or directed (input/output) as in Tangram. Likewise, directed ports are typed.

Example. A *sequencer* is a handshake component with one passive port a and two active ports b and c. All three ports are undirected. Once activated along a it will complete handshakes along b and c *sequentially*, before completing the handshake along a. Its behavior is depicted by a state diagram below, together with a symbol for the component. (Passive/active ports are depicted by open/closed circles at the periphery of the component. The initial state has been marked with a fat dot).

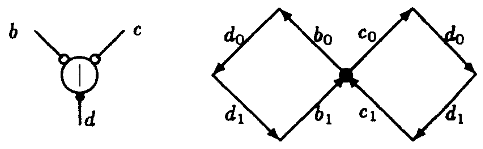


The drawing convention is that the active ports of the sequencer are served counter clockwise, starting from its passive port.

Example. A *repeater* is a handshake component with a passive port a and an active port b. Once activated along a it will repeatedly handshake along b; completion of the handshake along a will never occur. Symbol and state diagram are depicted below.



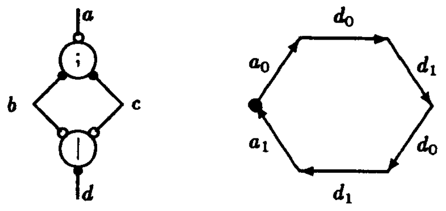
Example. A *mixer* is a handshake component with two passive ports b and c and one active port d. A handshake along d is enclosed by either a handshake along b or one along c; the choice is left to the environment. Handshakes along b and c must be strictly sequential. The Tangram compiler guarantees the correct usage of mixers. Symbol and state diagram are depicted below.



Two handshake components are *connectable* when common port names refer to ports of complementary activity, and (in the case of directed ports) of complementary direction and identical types. The sequencer and the mixer above (with the given port names) are connectable. A pair of ports with identical names is called a *channel.*

A *handshake circuit* is a pairwise connectable set of handshake components. Consequently, a port name may occur at most once for a passive port and once for an active port. Note that this excludes broadcasting among handshake components. (The broadcast of Tangram is implemented by "fork" components.) The *external* ports of a handshake circuits are the ones whose name occurs only once.

Example. A handshake circuit consisting of the sequencer and mixer has passive a and active d as external ports. The combined behavior (as observable at the external ports) is that of a *duplicator*: once activated along a it will complete *two* handshakes along d, before completing the handshake along a.



Translation of Tangram programs into handshake circuits

The translation of Tangram programs into VLSI-circuit layouts is divided into two steps with handshake circuits as an intermediary. In this section we address the first step: the translation of a Tangram program into an *equivalent* handshake circuit. This equivalence has two aspects.

Firstly, the external ports of a handshake circuit must match the Tangram ports in direction and type. We shall make these ports active (this is *not* essential, but simplifies the presentation of the translation). Furthermore, we introduce one additional undirected, passive port named $\surd$. $\surd_0$ activates the circuit (starts the execution of the Tangram program) and $\surd_1$ concludes the activity in the circuit (signals the termination of the Tangram program). Of course, for non-terminating programs, $\surd_1$ will never occur.

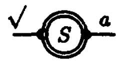
Secondly, the behavior of the handshake circuit as observable at its external ports must be identical to the behavior of the Tangram program, when taking the handshake-signaling conventions into account.

We view the translation as a function C from the domain of all Tangram programs to the domain of handshake circuits. Although the translation method differs considerably from [3] it yields essentially the same handshake circuits; albeit simpler, more amenable to formal analysis and easier to extend to larger source languages.

Here we confine the description of C to that portion of Tangram that deals with undirected communication.

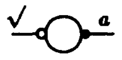
The definition of C is based on the syntax of Tangram: it describes a translation rule for each production rule of the syntax. More technically, C is defined by induction over the syntax of Tangram. Here we shall adopt a pictorial description of C. The application of C to a Tangram program $(P).S$ is depicted by enclosing the command S by two circles, and one handshake port for $\surd$ and for each element of P. For example, the program $(a^{\sim}).S$ yields

$(a^{\sim}).S \quad \rightarrow$



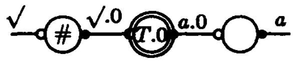
When S is simply a synchronization on port a, the corresponding handshake circuit consists of a single *connector.* A connector merely encloses a handshake on its active port by each handshake that occurs on its passive port.

$a^{\sim} \quad \rightarrow$



In the examples that follow we assume that all subcommands have a synchronization port a. When S is of the form $\#[T]$ the translation introduces a repeater.

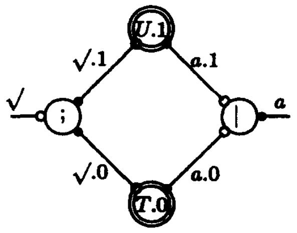
$\#[T] \quad \rightarrow$



A simple scheme is used to name the newly introduced channels, such as $\surd.0$ above. $T.0$ denotes T with ".0" appended to *all* its names. This simple scheme necessitates the introduction of connectors for the external ports of T, as exemplified by port a. These connectors can be removed by appropriate renaming of ports. (In an implementation of C this introduction of such connectors can be avoided).

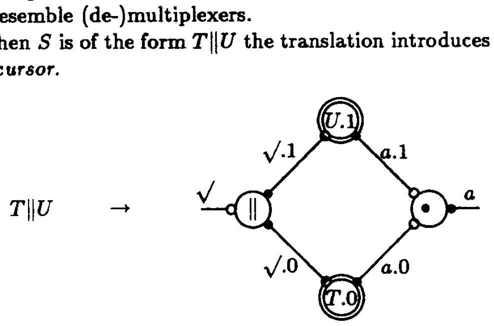
When S is of the form $T; U$ the translation introduces a sequencer.

$T; U \quad \rightarrow$



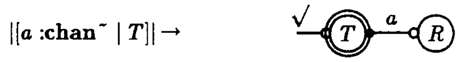
For port names and variable names that occur in both T and U some "glue" components have to be introduced as well: a mixer for each undirected port. For common directed ports we introduce generalizations of the mixer that resemble (de-)multiplexers.

When S is of the form $T \| U$ the translation introduces a *concursor*.

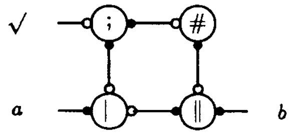


Once activated along $\surd$ the concursor will complete handshakes along $\surd.0$ and $\surd.1$ *concurrently*, before completing the handshake along $\surd$. For port names common to T and U a synchronizer is introduced. The synchronizer encloses each handshake on a by handshakes on *both* $a.0$ and $a.1$. The situation becomes slightly more complicated when we also consider read access to common variables, where synchronization is undesirable.

The last command form that we shall discuss is the block command $|[a$ **:chan**$^{\sim}$ $| \ T]|$. Its translation introduces a *run* component, i.e. a component that is always ready to engage in a handshake along its passive port.



Example. A systematic application of C, as introduced so far, to the Tangram program for a ternary semaphore $(a^{\sim}, b^{\sim})$. $a^{\sim}; \#[b^{\sim} \| a^{\sim}]$ yields the following handshake circuit (the connectors have been removed).

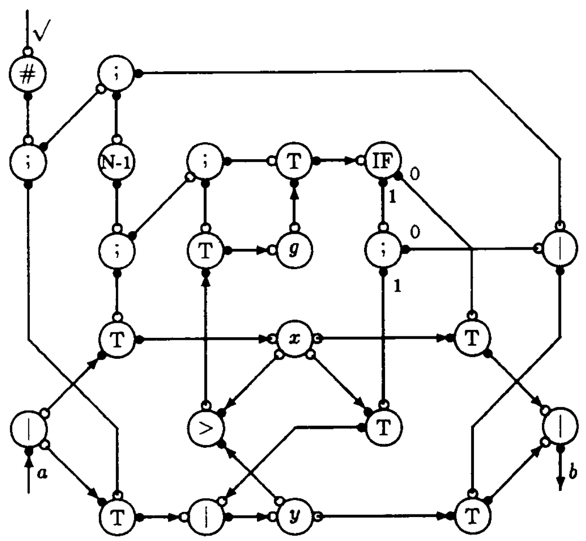


The translation of Tangram programs into handshake circuits is clearly *syntax directed*. It is relatively straightforward to extend C to *all* other production rules of Tangram. They require the introduction of a few more handshake components such as the *variable* and the *transferrer*

A variable is a component with a passive write port (input) and one or more passive read ports (output), all of the same type. The environment may choose to send a value to its write port, or may request a value from one of its read ports. Reading on multiple ports may occur concurrently and independently. The symbol for a variable is a circle with its name in it.

A transferrer has a passive undirected port and two active ports, one for input and one for output. A handshake on its passive port, encloses an active fetch of a message and a subsequent delivery of that message through its output port. It transfers a message on request. The symbol for a transferrer is a circle with a T in it.

The handshake circuit of the *BUBBLE* program is depicted below. The arrows are channels directed according to direction of data transport. We invite the reader to compare this handshake circuit with the Tangram program of the introduction; the structural similarities are quite clear. (The condition $x > y$ is stored in an auxiliary boolean variable g, since its value can be modified during the execution of the command.)



The C function can easily be adapted for all Tangram constructs presented in the section on the language.

Due to its syntax-directed nature, the translation is highly transparent. Given this transparency and some cost/performance data of the handshake components, some simple rules can be devised to reason about costs and performance at Tangram level.

Efficiency at the handshake-circuit level may be enhanced by refining C to consider special cases, and/or by applying some "peep hole" optimizations afterwards. Such improvements may include the replacement of expensive subcircuits by cheaper components (Cf. [3]).

Note that this translation scheme may yield handshake circuits of arbitrary complexity. Nevertheless, clock signals are absent: *all* synchronization is by means of handshakes. Moreover, handshake circuits are *delay-insensitive*: their correct operation is independent of any assumption on delays in handshake components and channels, except that the delays be finite [17].

The Tangram compiler

The silicon compiler implements the C function adapted for *four-phase* handshake circuits. This adaptation affects the C function in a few minor details, because more optimizations may be considered.

The current version of the compiler implements a major portion of Tangram, including all the commands discussed in the section on Tangram. The implementation of C greatly benefited from the compiler-construction tool Elegant [1].

The compiler contains a small circuit/layout library. Simple handshake components are realized as single standard cells consisting of a few transistors, e.g.: repeater (6), sequencer (20), mixer (22), concursor (40) and synchronizer (12). Components that deal with data are parameterized for their width (in # bits). Simple generators produce net lists for these components for given parameter values. For the construction of some of these circuits we have applied the method discussed in [12]. The resulting circuits are fully asynchronous. For the layout we used the commercially available standard-cell package Tancell.

Conclusion

The translation of Tangram programs to VLSI circuits is relatively straightforward and transparent. This transparency enables the VLSI programmer to make the appropriate trade-offs between silicon area and performance.

Handshake circuits are an attractive intermediary between Tangram and VLSI circuits: the first translation step may safely ignore all electronic and layout aspects, the second step has been reduced to the generation of a limited number of parameterized circuits and the overall layout.

The overall prospects for applying Tangram to practical VLSI systems look promising.

Acknowledgements

Acknowledgements are due to Lex Augusteijn and Jan Stuyt for providing invaluable assistance in the application of respectively Elegant and Tancell to the implementation of Tangram. We also like to mention the numerous stimulating and inspiring discussions with Martin Rem on VLSI programming, handshake circuits and delay insensitivity.

References

[1] Lex Augusteijn. The Elegant compiler generator system. In *Attribute Grammars and their Applications*, Paris, 1990, 238–254. *Lecture Notes in Computer Science, Vol. 461*, Springer-Verlag.

[2] C.H. (Kees) van Berkel, Martin Rem, and Ronald W.J.J. Saeijs. VLSI Programming. In *Proc. of the 1988 IEEE Int. Conf. on Computer Design: VLSI in Computers & Processors*, 1988, 152–156

[3] C.H. (Kees) van Berkel and Ronald W.J.J. Saeijs. Compilation of Communicating Processes into Delay-Insensitive Circuits. *Ibidem.*, 157–162.

[4] Ronald W.J.J. Saeijs and C.H. (Kees) van Berkel. The Design of the VLSI Image-Generator ZaP *Ibidem.*, 163–166.

[5] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[6] C.A.R. Hoare. *Communicating Sequential Processes. Series in Computer Science*, Prentice-Hall Int., 1985.

[7] INMOS Limited. *Occam Programming Manual. Series in Computer Science*, Prentice-Hall Int., 1984.

[8] Anne Kaldewaij and Martin Rem A derivation of a systolic rank order filter with constant response time. In *Proc. of the Int. Conf. on Mathematics of Program Construction*, Groningen, 1989, 281–296. *Lecture Notes in Computer Science, Vol. 375*, Springer-Verlag.

[9] Anne Kaldewaij and Gerard Zwaan. A Systolic Design for Acceptors or regular Languages. Submitted for *Science of Computer Programming.*

[10] Joep L.W. Kessels and Frits Schalij. VLSI Programming for the Compact Disc Player. Submitted for *Science of Computer Programming.*

[11] Joep L.W. Kessels and Martin Rem. Designing systolic, distributed buffers with bounded response time. *Distributed Computing*, Vol. 4, 1990, 37–44.

[12] Alain J. Martin. The design of a self-timed circuit for distributed mutual exclusion. In Henry Fuchs, editor, *Chapel Hill Conference on VLSI*, pages 245–260, Computer Science Press, 1985.

[13] Alain J. Martin. Syntax-directed Translation of Concurrent Programs into Self-timed Circuits. In *Proc. MIT Conf. on VLSI*, pages 35–50. 1988.

[14] Alain J. Martin. The Design of an Asynchronous Microprocessor. In *Proc. Decennical Caltech Conf. on VLSI*, pages 351–373, C.L. Seitz ed., MIT Press, 1989.

[15] Martin Rem. Trace Theory and Systolic Computations. In J.W. Bakker et. al., ed., *PARLE Parallel Architectures and Languages in Europe*, pp. 14–33, *Lecture Notes in Computer Science, Vol. 258*, Springer-Verlag, 19123.

[16] Charles L. Seitz. System Timing. In C.A. Mead and L.A. Conway, *Introduction to VLSI Systems*, Chapter 7, Addison-Wesley, 1980.

[17] Jan L.A. van de Snepscheut. *Trace Theory and VLSI Design. Lecture Notes in Computer Science, Vol. 200*, Springer-Verlag, 1985.