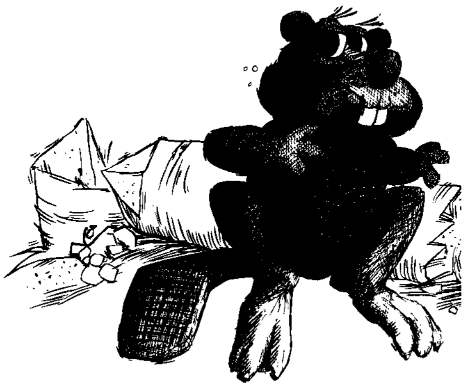


From Functional Specification to a Delay-Insensitive Circuit

Jo C. Ebergen

Research Report
CS-89-44

October, 1989

From Functional Specification to a Delay-Insensitive Circuit

Jo C. Ebergen

Computer Science Department
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1

Abstract

A method for designing and implementing parallel computations in a systematic manner is discussed by means of some examples. The design starts with a functional specification and ends with a circuit realization consisting of a network of basic circuit elements. A formalism and notation for specifying and decomposing communication behaviors of components, including their data communications, is explained in detail. The circuits thus obtained are delay-insensitive, i.e., the correctness of the network is insensitive to delays in the response times of circuit elements and connection wires. It is shown that these circuits lend themselves elegantly for reasoning about the correctness of a circuit design.

Keywords: Delay-insensitive circuit, behavioral specification, micropipeline.

1 Introduction

The purpose of this paper is to demonstrate that the complete design route for many parallel computations, which start with a functional specification and end with a delay-insensitive circuit, can be carried out in a systematic manner. We have divided this design route, inspired by Rem's work [6], into two parts. The first part concerns the *functional* specification and *functional* decomposition of the computation. The second part concerns the *behavioral* specification and *behavioral* decomposition of the computation.

In a functional specification the output values of the computation are specified as a *function* of the input values. Subsequently, in the functional decomposition, this function is decomposed into a number of simpler functions in a manner similar to the design of a functional program.

A behavioral specification prescribes the *order* in which input and output values are communicated between a component and its environment. Such a specification prescribes not only what the component should do, but also how the environment must interact with the component. In a behavioral decomposition, such a component is decomposed into a number of communicating subcomponents that implement the functions obtained in the functional decomposition.

The complete decomposition of a component is obtained by repeatedly decomposing subcomponents until we arrive at basic components. Thus, by stepwise refinement, we derive a decomposition into basic components in a hierarchical way. The basic components correspond to basic circuit elements, and the resulting network of these circuit elements corresponds to a circuit realization of the component.

The circuits designed as described in this paper are *delay-insensitive* [5]. This means that the correctness of the network is insensitive to delays in the response times of the circuit elements and connection wires. The reason for choosing delay-insensitive circuits, as realizations of parallel computations, is their suitability for a structured and formal design discipline. This is largely due to the separation that can be made between mathematical and physical correctness concerns for these circuits [9]: the design of a *network* of basic circuit elements is based on mathematical principles only; the design of the basic circuit elements themselves may also rely on physical properties, such as specific assumptions on delays in certain wires. By maintaining this strict separation in our design approach we can avoid many timing problems, such as those caused by metastability [3] or scaling [9].

Related work on the derivation of delay-insensitive circuits as realizations of parallel computations has been done by Rem [6], van Berkel et al. [11], and Martin et al. [1]. In [6], Rem discusses a method for deriving functional decompositions of parallel computations. These decompositions then give rise to CSP-like programs. In [11] and [1], van Berkel, and Burns and Martin, respectively, have shown how these programs can be transformed into networks of basic circuit elements. Their programs, however, have a different behavioral interpretation. In their programs, each matching pair of an input and an output is synchronized by definition, in accordance with the traditional CSP semantics of a communication action. In the transformation of a program to a circuit realization, this synchronization is achieved by applying a four-phase handshake protocol. In our approach, we require that whenever a component can produce an output, the matching inputs can take place as well. Thus, no handshake protocol is needed to implement the synchronization of matching inputs and outputs, but, on the other hand, we have an additional requirement that must be satisfied. It turns out that this requirement can be met easily for the examples discussed.

The contents of this paper are as follows. Functional specification and decomposition are briefly discussed in Section 2 by means of an example. The theory underlying the behavioral specification and decomposition is presented in Sections 3 through 6. Behavioral specification and decomposition is applied to an example in Sections 7 and 8. Section 9 briefly discusses a data encoding scheme for delay-insensitive circuits and Section 10 describes the general approach applied to some other examples.

2 Functional Specification and Decomposition

In a functional specification of a component the sequences of output values are defined as a function of the sequences of input values. Subsequently, in a functional decomposition this specification is decomposed into a number of basic functions. We demonstrate this briefly by giving a functional specification and decomposition of a simple, but non-trivial, computation, called *bit convolution* [6, 11].

Given is a bit sequence $q = q_0 q_1 .. q_{M-1}$ of length $l(q) > 0$, where $M = l(q)$. We are asked to design a component that computes for an input bit sequence a of infinite length the output bit sequence $a \otimes q$ representing the convolution of the sequences a and q. The sequence $a \otimes q$ is defined by

$$(a \otimes q)_i = \left(\sum j,k : 0 \leq j \ \wedge\ 0 \leq k < l(q) \ \wedge\ j+k = i : a_j \times q_k\right). \tag{1}$$

Here summation and multiplication are taken modulo 2.

For example, for $l(q) = 3$, we have

$$\begin{array}{lcl}
(a \otimes q)_0 & = & a_0 \times q_0 \\
(a \otimes q)_1 & = & a_1 \times q_0 + a_0 \times q_1 \\
(a \otimes q)_2 & = & a_2 \times q_0 + a_1 \times q_1 + a_0 \times q_2 \\
(a \otimes q)_3 & = & a_3 \times q_0 + a_2 \times q_1 + a_1 \times q_2 \\
\vdots & &
\end{array}$$

Let the output sequence $a \otimes q$ be denoted by b and the component we have to design by $CONV(q)$ (cf. Figure 1). The input elements $a_i, i \geq 0$ are

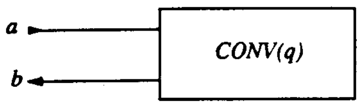


Figure 1: Component $CONV(q)$.

provided in order of increasing index and the output elements b_i are produced in order of increasing index.

In order to avoid calculating (1) from scratch each time when a new input is provided, we decompose (1) into a number of simple calculations like binary additions and multiplications. By induction on the length of q we derive the following properties. (For a complete derivation we refer to [6].)

$$\text{If } l(q) = 1, \quad (a \otimes q)_i = a_i \times q_0 \quad \text{for } i \geq 0 \tag{2}$$

$$\text{If } l(q) > 1, \quad (a \otimes q)_i = \begin{cases} a_i \times q_0 & \text{for } i = 0 \\ a_i \times q_0 + (a \otimes (tl(q))_{i-1} & \text{for } i > 0, \end{cases} \tag{3}$$

where $tl(q) = q_1 q_2 .. q_{M-1}$. We now have decomposed the convolution $a \otimes q$ into a simple multiplication, an addition, and a convolution of sequence a with a smaller sequence, viz., $tl(q)$. Carrying the induction through, it follows that $a \otimes q$ can be decomposed into a series of simple multiplications and additions.

The functional decomposition (3) suggests a behavioral decomposition into a subcomponent $CONV(tl(q))$ and a component $E(q_0)$ as depicted in Figure 2, where $E(q_0)$ performs a simple multiplication and addition. In this behavioral decomposition we have to specify in what order and which values are communicated on the so-called *channels* a, b, c, and d such that the component $CONV(q)$ is realized by this decomposition. For this purpose, we first introduce a formalism and notation to give behavioral specifications of components.

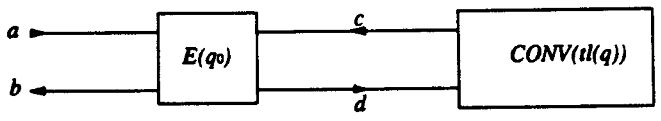


Figure 2: Tentative decomposition of $CONV(q)$.

3 Behavioral Specifications

In this section we present a brief introduction to a formalism for specifying communication behaviors which was developed at Eindhoven University of Technology [7, 12].

A behavioral specification of a component is given in a program notation called *commands*. As an example of a command, we give a specification of a communication behavior for $CONV(q)$. From (1) it follows that b_i only depends on the input values a_j, $0 \leq j \leq i$. Therefore we propose as a possible communication behavior an alternation of inputs and outputs, starting with an input, such that after each receipt of input a_i output b_i is produced by the component. This communication behavior can be specified by the command

$$\mathbf{pref}[a?;b!],$$

where $a?$ denotes an input action at channel a, $b!$ denotes an output action at channel b, ';' denotes concatenation, '[ ]' denotes arbitrary repetition of the enclosed, and **pref** denotes prefix-closure.

Formally, a command defines a *directed trace structure*. A directed trace structure is a triple $<I,O,T>$, where I is the input alphabet, O is the output alphabet, and T is a set of traces constructed from symbols in $I \cup O$. The trace ϵ represents the empty trace. A trace structure is called *regular* when its trace set is a regular set. Regular directed trace structures can be represented by commands, which are in many ways similar to regular expressions.

Commands are defined inductively. The atomic commands $\epsilon, a?, a!$, and $!a?$ represent the atomic trace structures $<\emptyset,\emptyset,\{\epsilon\}>$, $<\{a\},\emptyset,\{a\}>$, $<\emptyset,\{a\},\{a\}>$, and $<\{a\},\{a\},\{a\}>$ respectively. For commands E, $E0$, and $E1$, the expressions $E0;E1$ (concatenation), $E0|E1$ (union), $[E]$ (repetition), and $\mathbf{pref}E$ (prefix-closure) are also commands. Let $\mathbf{i}E, \mathbf{o}E$, and $\mathbf{t}E$ denote the input alphabet, output alphabet, and trace set of the directed trace structure represented by the command E respectively. The directed trace structures represented by $E0;E1$, $E0|E1$, $[E]$, and $\mathbf{pref}E$ are defined by

$$\begin{aligned}
E0;E1 &= <\mathbf{i}E0 \cup \mathbf{i}E1, \mathbf{o}E0 \cup \mathbf{o}E1, (\mathbf{t}E0)(\mathbf{t}E1)>,\\
E0|E1 &= <\mathbf{i}E0 \cup \mathbf{i}E1, \mathbf{o}E0 \cup \mathbf{o}E1, \mathbf{t}E0 \cup \mathbf{t}E1>,\\
[E] &= <\mathbf{i}E, \mathbf{o}E, (\mathbf{t}E)^*>,\\
\mathbf{pref}E &= <\mathbf{i}E, \mathbf{o}E, \{t_0 | (\exists t_1 :: t_0 t_1 \in \mathbf{t}E)\}>,
\end{aligned}$$

where concatenation of sets is denoted by juxtaposition and * denotes Kleene's closure. (For reasons of brevity, we use the same notation for the command and the language defined by the command.) For example, we have

$$\mathbf{pref}[a?;b!] = <\{a\},\{b\},\{\epsilon, a, ab, aba, abab, ..\}> .$$

(Since we only use directed trace structures in this paper, we omit the word 'directed' from now on.)

4 Component and Environment Prescriptions

Communication behaviors of components are specified by non-empty, prefix-closed trace structures with disjoint input and output alphabets. Such trace structures are often represented by commands. The alphabets of the trace structure represent the terminals or channels through which a component can communicate with its *environment*, where the environment of the component is constituted by all other components with which the component communicates. The trace set of the trace structure stipulates all possible communication behaviors of a component with its environment. The initial state is represented by the empty trace ϵ. Since the trace structure is prefix-closed, every prefix of a trace is contained in the trace set. Moreover, since the trace set is non-empty as well, ϵ is always contained in the trace set.

A behavioral specification not only prescribes the behavior of the component, i.e., when the component may produce an output, but also of the environment, i.e., when the environment may produce an input. For example, the command $\mathbf{pref}[a?; b!]$ specifies that the environment may produce an input initially and each time after the component has produced an output b. If the environment satisfies these prescriptions then the component may produce an output b each time after it has received an input a. Consequently, on the one hand a specification can be read by the 'user' of a component (i.e. the environment) as how this component should be operated, and, on the other hand, a specification can be read by an 'implementer' as what should be realized provided the component is used properly.

If the environment does not obey the specifications nothing is guaranteed. So for example, if for a specification $\mathbf{pref}[a?; b!]$ the environment would provide two a's in a row without the component having been able to produce an output b, clearly the environment has violated its prescription and nothing is guaranteed anymore. A violation of the environment prescription is also called *computation interference* [12], because the environment interferes with the computation performed by the component. In order to guarantee absence of computation interference we have to show that whenever a component is able to produce an output, also the matching inputs can take place in the receiving components. Absence of computation interference will be our main concern in decomposing a component into a network of basic components.

There are many other safety (or liveness) concerns that may be considered for the correctness of a decomposition, such as absence of deadlock or livelock. In this paper, we deal with absence of computation interference only. For more extensive discussions on other topics we refer to [4, 14].

One way to avoid computation interference in communications between components is to make sure that the communication behaviors of the respective components are each other's *reflection*. The reflection of a specification E, denoted by $\overline{E}$, is defined by

$$\overline{E} = < \mathbf{o}E, \mathbf{i}E, \mathbf{t}E > .$$

For example, we have $\overline{\mathbf{pref}[a?; b!]} = \mathbf{pref}[a!; b?]$. By reflecting we interchange the prescriptions of component and environment, i.e., of 'implementer' and 'user'. Obviously, when E and $\overline{E}$ are connected to one another no computation interference can occur: if E can produce an output, this output can also be accepted by $\overline{E}$ as an input and *vice versa.*

5 Parallelism and Synchronization

In order to specify communication behaviors that allow for parallelism and hiding of internal actions, we introduce two operations: *weaving* and *projection.*

A *weave* of two behavioral specifications represents all behaviors that are in accordance with the two specifications in isolation. For this reason it is sometimes also called the 'conjunction' of two behavioral specifications.

Formally, the weave $E0\|E1$ of two trace structures represented by the commands $E0$ and $E1$ is defined by

$$\begin{aligned} E0\|E1 &= \ < \mathbf{i}E0 \cup \mathbf{i}E1,\ \mathbf{o}E0 \cup \mathbf{o}E1, \\ &\quad \{t \in (\mathbf{a}E0 \cup \mathbf{a}E1)^* \mid t{\downarrow}\mathbf{a}E0 \in \mathbf{t}E0\ \wedge\ t{\downarrow}\mathbf{a}E1 \in \mathbf{t}E1\} > . \end{aligned}$$

Here, $\mathbf{a}E$ denotes the alphabet of E and $\mathbf{a}E = \mathbf{i}E \cup \mathbf{o}E$. The notation $t \downarrow A$ denotes the trace t projected on the alphabet A, i.e. the trace t from which all symbols not in A have been deleted. As a small example of weaving, we have

$$\mathbf{pref}[a?;c!] \parallel \mathbf{pref}[b?;c!] = \mathbf{pref}[a?\|b?\,;\,c!],$$

where the priority of the operations is as follows: '$\|$' has highest priority, then ';' (, and then '|'). Notice that, in a weave, common symbols must match. One could also say that weaving expresses 'parallelism with synchronization on common symbols.' In the above example, the symbol c is the only common symbol of the two commands in the weave on which synchronization has to take place.

There are two special cases of weaving. If $\mathbf{a}E0 \cap \mathbf{a}E1 = \emptyset$, then the weave $E0\|E1$ represents the interleaving (or shuffle) of $E0$ and $E1$. If $\mathbf{a}E0 = \mathbf{a}E1$, then weaving $E0$ and $E1$ amounts to taking the intersections of the trace sets.

The *projection* of E on an alphabet A, denoted by $E{\downarrow}A$, is formally defined by

$$E{\downarrow}A = < \mathbf{i}E \cap A, \mathbf{o}E \cap A, \{t{\downarrow}A \mid t \in \mathbf{t}E\} > .$$

As an example of weaving and projection, we consider the command

$$(\mathbf{pref}[a?;!s?;b!] \parallel \mathbf{pref}[c?;!s?;d!]) \downarrow \{a,b,c,d\}. \tag{4}$$

Notice that now synchronization has to take place on the common symbol s. Using the definitions of weaving, projection, and the other operations, we can formally derive that the above command defines the same trace structure as

$$\mathbf{pref}(a?\|c?;[(b!;a?)\|(d!;c?)]). \tag{5}$$

Operationally speaking, (4) expresses a communication behavior in which initially inputs a and c are accepted and then, repeatedly, a 'hidden' synchronization on s occurs followed by $b!; a?$ in parallel with $d!; c?$.

The symbol s is hidden, or projected 'away'. Therefore, s is called an *internal* symbol. Internal symbols are denoted by prefixing and postfixing them with the marks '!' and '?' respectively. The external symbols are a, b, c, and d.

Remark. One may be tempted to consider weaving as representing a parallel composition of components. We emphasize, however, that weaving should be interpreted as an operation to express parallel behavior (with possible synchronization) of *one* component only. □

6 Including Data Communications

By means of a command we specify the *order* in which the communication actions take place. We can also describe which *data values* are communicated along the channels by including the data communications in a command. For each channel a we introduce a Boolean variable va. The additional programming primitives are assignments to variables, $a?va$, and $a!va$, where $a?va$ denotes 'receipt of value at channel a which is assigned to va' and $a!va$ denotes 'output via channel a the value va.'

As an example, we introduce in the command $\mathbf{pref}[a?\|b?\,;\,c!]$ data communications of type Boolean. Consider the command E defined by

$$E = \mathbf{pref}[a?va \,\|\, b?vb \;; vc := va \vee vb \;;\; c!vc], \tag{6}$$

Accordingly, E specifies a component that repeatedly receives, in parallel, Boolean values on channels a and b and outputs the disjunction of these values on channel c. Instead of command (6), we often write the shorter version

$$E = \mathbf{pref}[a?va \,\|\, b?vb\,;\, c!(va \vee vb)].$$

Component E calculates the disjunction of the input values and is therefore called the *disjunction* or *DIS component.* See Figure 3 for a schematic.

In a similar way as we introduced data communications in $\mathbf{pref}[a?\|b?; c!]$, we can introduce data communications in the command of (4). In (7) and (8), two different data communications have been introduced in the command of (4).

$$(\mathbf{pref}[a?va; !s?(vb := va); b!vb] \,\|\, \mathbf{pref}[c?vc; !s?(vd := vc); d!vd]) \downarrow \{a, b, c, d\} \tag{7}$$

$$(\mathbf{pref}[a?va; !s?(vb := vc); b!vb] \,\|\, \mathbf{pref}[c?vc; !s?(vd := va); d!vd]) \downarrow \{a, b, c, d\} \tag{8}$$

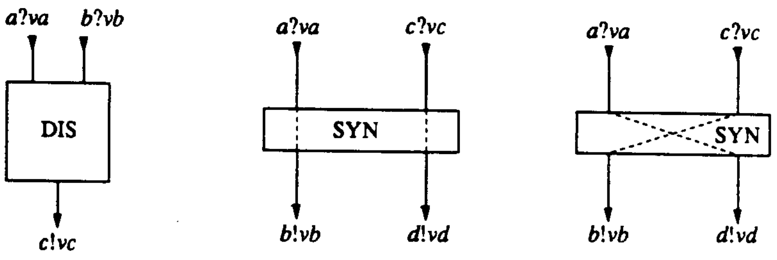


Figure 3: Components with data communications: DIS and SYN.

This time we have extended our commands by postfixing occurrences of the internal symbol s with assignment statements. The left-hand side of an assignment statement is a variable of an output channel. The right-hand sides is a variable of an input channel. The assignments are 'executed' at each occurrence of the internal symbol s in a trace of the weave. Notice that at each such occurrence of s the input variables have stable values. In command (7), we have expressed that the data received at channel a is transferred to channel b and the data received at channel c is transferred to channel d. These transfers are synchronized by means of the (internal) synchronization symbol s.

Command (8) expresses a different data transfer. Here, data received at channel a is transferred to channel d, denoted by the assignment $vd := va$, and data received at channel c is transferred to channel b, denoted by the assignment $vb := vc$. These transfers also are synchronized on the occurrence of the internal symbol s. We call the above components, which synchronize data transfers, *SYN components*. For schematics of the SYN component, see Figure 3.

In the DIS and SYN component we have specified data communications in their most primitive form, viz., computing an output value as a simple function of the input values and a synchronization of data transfers on separate channels. In the next section we introduce a combination of these primitive forms explained by means of our example of Section 2.

7 An Example

We illustrate the notations introduced in the previous sections by giving a behavioral specification and decomposition of the convolution component $CONV(q)$.

First, we give a behavioral specification of the component $CONV(q_0)$, i.e., in case $l(q) = 1$. The relation between the input and the output values was given by

$$b_i = a_i \times q_0 \quad \text{for } i \geq 0.$$

Obviously, the following communication behavior establishes this relation.

$$\mathbf{pref}[a?va\,;b!(va \times q_0)].$$

We proceed with the decomposition of $CONV(q)$ in case $l(q) > 1$. Led by the functional decomposition (3) of $CONV(q)$, we try to find a behavioral decomposition of $CONV(q)$ into a component $E(q_0)$ and $CONV(tl(q))$. The tentative decomposition is depicted in Figure 4. The communication behavior

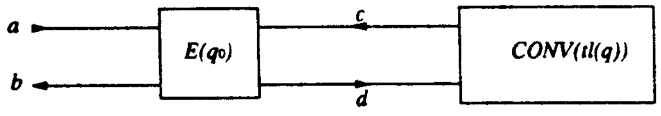


Figure 4: Tentative decomposition of $CONV(q)$.

of $CONV(tl(q))$ is similar to that of $CONV(q)$, namely $\mathbf{pref}[d?;c!]$, where $c_i = (d \otimes tl(q))_i$ for $i \geq 0$. In order to avoid computation interference, the communication behavior of $E(q_0)$ with respect to $CONV(tl(q))$ is given by the reflection of $\mathbf{pref}[d?;c!]$, i.e., $\mathbf{pref}[d!;c?]$. Consequently, the complete communication behavior of component $E(q_0)$ is a (parallel) composition, or weave, of the behaviors $\mathbf{pref}[a?;b!]$ and $\mathbf{pref}[d!;c?]$.

In order to find the proper synchronization between the two behaviors, we consider (3) once more. We try to establish

$$d_i = a_i \qquad \text{for } i \geq 0, \text{ and} \tag{9}$$

$$b_i = \begin{cases} a_i \times q_0 & \text{for } i = 0, \text{ and} \\ a_i \times q_0 + c_{i-1} & \text{for } i > 0. \end{cases} \tag{10}$$

Since then, we have, by definition of $CONV(tl(q))$, that $c_i = (a \otimes tl(q))_i$ and subsequently, by (3), $b_i = (a \otimes q)_i$ for $i \geq 0$.

Based on (9) and (10) we propose the following program for $E(q_0)$.

$$\begin{aligned} E(q_0) \;=\; & (\,\mathbf{pref}[a?va\,;!s?(vb := va \times q_0 + vc)\,;b!vb] \\ & \|\ \mathbf{pref}(vc := 0;[!s?(vd := va)\,;d!vd\,;c?vc]) \\ &)\downarrow\{a,b,c,d\}. \end{aligned}$$

Let us first examine this program by excluding all information about data communications. We are then left with the command

$$\begin{aligned} & (\,\mathbf{pref}[a?\,;!s?\,;b!] \\ & \|\ \mathbf{pref}[!s?\,;d!\,;c?] \\ &)\downarrow\{a,b,c,d\}. \end{aligned}$$

In this reduced version of $E(q_0)$, we have expressed a synchronization between the two communication behaviors $\mathbf{pref}[a?;b!]$ and $\mathbf{pref}[d!;c?]$, by introducing the common internal symbol s between an input and an output in the two sequential commands. (Notice that, since s precedes d, the output d can be generated only after synchronization on s has taken place.) In this way, component $E(q_0)$ is ready to accept the next input a as soon as output b is produced. A similar reasoning applies for the input c and output d. Consequently, computation interference does not occur in the decomposition.

Second, we examine the data communications. At each occurrence $i, i \geq 0$, of symbol s, the invariant

$$P: \quad va = a_i \ \wedge \ vc = c_{i-1}$$

holds, where $c_{-1} = 0$, because of the initialization $vc := 0$. By means of the assignment $vd := va$, condition (9) is established, i.e., the value received on channel a is passed on to channel d. By means of the assignment $vb := va \times q_0 + vc$, condition (10) is established, i.e. from the value received on channel a (i.e. a_i) and the value received on channel c (i.e. c_{i-1}) the output value for channel b (i.e. b_i) is calculated. Consequently, the correct data are communicated. Accordingly, $CONV(q)$ can be decomposed into $E(q_0)$ and $CONV(tl(q))$ for $l(q) > 0$.

8 Decomposition Continued

Subsequently, we try to decompose $E(q_0)$ and $CONV(q_0)$ into more primitive components. In the next section we will then see how these components can be realized using a particular data encoding scheme.

Component $CONV(q_0)$ is specified by $\mathbf{pref}[a?va; b!(va \times q_0)]$. Considering the cases $q_0 = 0$ and $q_0 = 1$ separately and recalling that multiplication is taken modulo 2, this specification boils down to

$$\begin{array}{rl} \mathbf{pref}[a?va; b!va] & \text{if } q_0 = 1 \text{ and} \\ \mathbf{pref}[a?va; b!0] & \text{if } q_0 = 0. \end{array}$$

For $q_0 = 1$, $CONV(q_0)$ simply copies the value received at the input to the output. For $q_0 = 0$, $CONV(q_0)$ produces after receipt of any binary value at the input the value 0 at the output channel.

For the decomposition of component $E(q_0)$ we also distinguish the cases $q_0 = 0$ and $q_0 = 1$. For $q_0 = 0$, the assignment $vb := va \times q_0 + vc$ in $E(q_0)$ boils down to $vb := vc$. Notice that in this case $E(q_0)$ is equivalent to a SYN component, except for the right initialization. The right initialization can be established by a component specified by $\mathbf{pref}(f!0; [c?vc; f!vc])$. This component starts with producing a 0, and then, repeatedly, copies at the output channel the value received at the input channel. For the moment, we call this an INIT component. For $q_0 = 0$, we have that $E(q_0)$ can be decomposed into a SYN and INIT component. This decomposition is depicted in Figure 5.

For $q_0 = 1$, the assignment $vb := va \times q_0 + vc$ in $E(q_0)$ boils down to $vb := va + vc$. Since addition is modulo 2, we insert a disjunction component in decomposition of $E(0)$. to obtain the decomposition for $E(1)$, i.e. $E(1)$ can be decomposed into a SYN, INIT, and DIS component. This decomposition is also depicted in Figure 5. Notice that also in these decompositions the

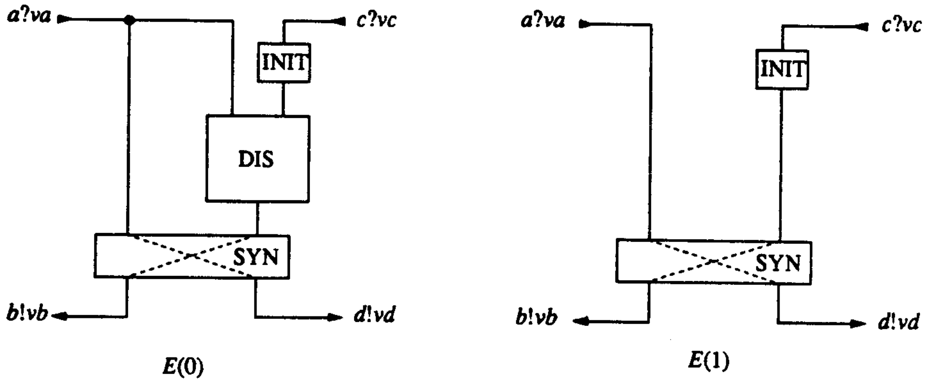


Figure 5: Decompositions for $E(q_0)$.

environment prescription for each component is not violated.

As an example of a complete decomposition for $CONV(q)$ into the components SYN, INIT, and DIS, we have chosen the sequence $q = 1101$. The decomposition for this particular q is depicted in Figure 6.

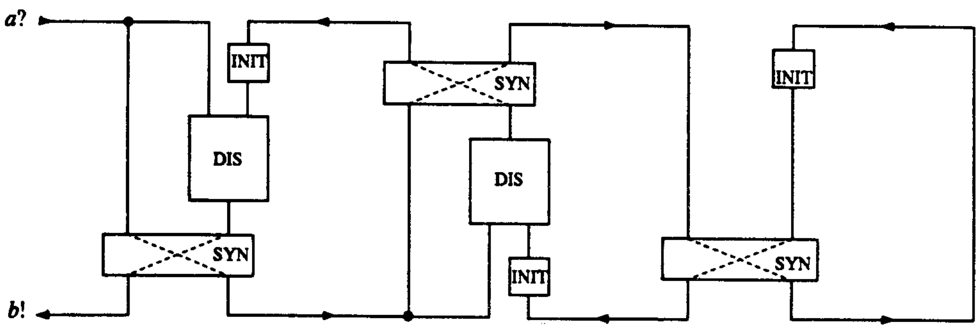


Figure 6: A decomposition for $CONV(1101)$.

9 Data Encoding

In order to decompose the components $CONV(q_0)$ and $E(q_0)$ further into basic components corresponding to circuit elements, we have to apply a *data encoding scheme.* Our data encoding scheme encodes data communications of type Boolean into communications of so-called *signals*, i.e. unary values. Components communicating signals only correspond in a specific way with circuit elements. In order to explain this correspondence first, we give some specifications of basic components communicating signals only in Figure 7. The first column lists the names of the components, the second column the specifications, and the third column the schematics.

WIRE	**pref**$[a?; b!]$
IWIRE	**pref**$[b!; a?]$
MERGE	**pref**$[a?; c! \mid b?; c!]$
SOURCE(a)	$< \emptyset, \{a\}, \{\epsilon\} >$

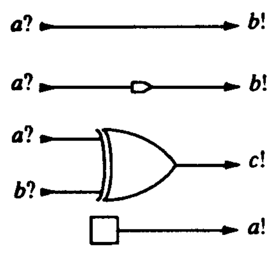


Figure 7: Specifications of some basic components.

The basic components of Figure 7 can be related to basic circuit elements in the following way. Each symbol in the input or output alphabet corresponds to an input or output terminal of the circuit respectively. Each occurrence of a symbol in a trace corresponds to a voltage transition at that terminal. (We assume that initially all voltage levels are low.) No distinction is made between a high-going and a low-going transition. This type of signaling is therefore also called *transition signaling* [10].

The WIRE and IWIRE components describe the transmission of a signal from terminal to terminal. Notice that both components have the same behavior except for a difference in initial states. The WIRE component initially receives an input a. The IWIRE component initially produces an output b. This difference in initial states (or the production of initial symbols) is depicted by an open arrow head in the schematic.

The MERGE component 'merges' two inputs into one output. Notice that for this component the environment produces either an input a or an input b. In both cases the component will then produce an output c, after which the environment may produce a next input. (The MERGE component can be implemented by an XOR gate.)

The SOURCE component has one output only, but does not produce any transition on this output. SOURCE components are connected to dangling inputs.

In order to show how a command expressing data communications of type Boolean can be transformed into commands where the only communications are signals, i.e., unary values, we use a *two-rail, two-cycle signaling encoding scheme* [9]. This encoding scheme is used to obtain delay-insensitive circuits as realizations of our computations. Other encoding schemes for delay-insensitive circuits are possible, and for a more extensive discussion of these we refer to [13].

Another possibility would be to use a more conventional level encoding scheme with additional control wires signaling the validity of the data [2]. Such an encoding scheme, however, does not render completely delay-insensitive circuits. There are certain delay constraints, also referred to as the *data bundling constraints*, that have to be satisfied. In [10] an example of an implementation using the data bundling constraint is given. Here, we shall restrict ourselves to delay-insensitive encoding schemes.

We briefly illustrate the encoding scheme by giving a transformation for the command of the disjunction component. A similar transformation can be applied to the other components communicating data of type Boolean. Applying a two-rail, two-cycle signaling encoding to command E in (6), we obtain the command

$$E1 = \mathbf{pref}[a0?||b0?;c0! \mid a1?||b0?;c1! \mid a0?||b1?;c1! \mid a1?||b1?;c1!].$$

Two-rail encoding means that two new symbols are introduced for each channel communicating data of type Boolean. For example, for channel a we introduced the symbols $a0$ and $a1$. *Two-cycle signaling* means that there are two cycles for the communication of the input values and the corresponding output value: in the first cycle the symbols corresponding to the input values are received and in the second cycle the symbol corresponding to the output value is produced. Notice that for each combination of input values we have an alternative of the form $<$ *inputs* $>;<$ *output* $>$.

Operationally speaking, in a two-rail encoding scheme we introduce two (wire) terminals for the communication of Boolean values, one for each value. With two-cycle signaling, the communication of a Boolean value is signaled by a *transition* on the terminal corresponding to that value.

Using the above components, we can give some further decompositions of $CONV(q_0)$ and INIT. Their respective specifications are given below.

$$\begin{aligned} CONV(0) &= \mathbf{pref}[a?va;b!0] \\ CONV(1) &= \mathbf{pref}[a?va;b!va] \\ \text{INIT} &= \mathbf{pref}(b!0;[a?va;b!va]) \end{aligned}$$

Applying a two-rail, two-cycle signaling scheme, these components can be decomposed as depicted in Figure 8. Notice that $CONV(0)$ produces after a receipt of a transition at either of the two input terminals a transition at the $b0$ output terminal (and never a transition at the $b1$ terminal).

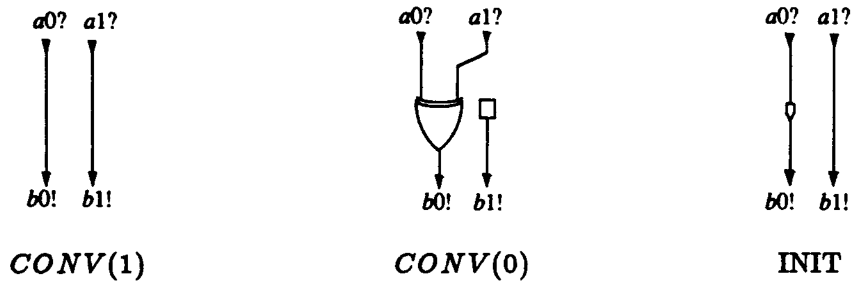


Figure 8: Decompositions for $CONV(q_0)$ and INIT.

10 Generalizations

The design that we have derived for the bit convolution depicted in Figure 6 can be seen as a special form of a micropipeline [10]. Instead of data flowing in only one direction, here data is flowing in both directions through the stages of the pipeline. In each stage, computations are performed on the data in the 'function' blocks and the transfers between the stages are controlled by the SYN components. A general schematic of such a stage is given in Figure 9. Depending on the particular communication behavior, the variables va, vb, vc,

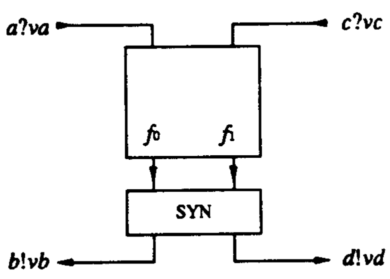
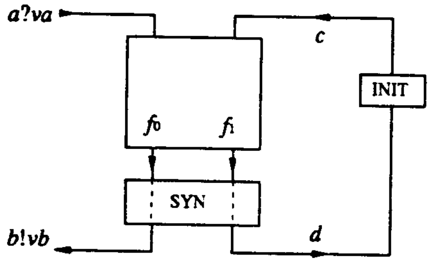
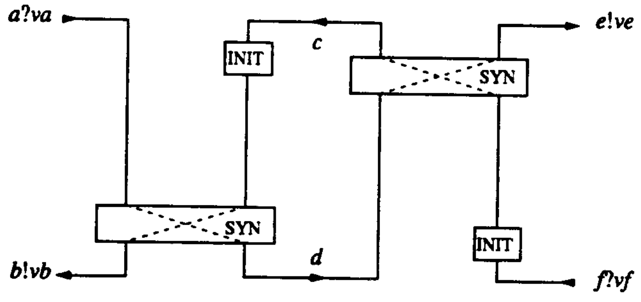


Figure 9: A schematic of a stage in a micropipeline

and vd may have to be initialized by means of INIT components. The function block implements the functions f_0 and f_1 and their function values are subsequently transferred to the channels d and b, or b and d, by a SYN component. The functions f_0 and f_1 are determined by the functional decomposition of the computation, such as (9) and (10) in case of the bit convolution. Two other special cases of this micropipeline are an implementation of a finite state machine and a buffer. Schematics of both designs are given in Figure 10.



(a) Finite State Machine



(b) Buffer

Figure 10: Some other special cases.

For the implementation of the finite state machine, the functions f_0 and f_1 calculate the ***next output symbol vb*** and the ***next state vd***, respectively, from the current input symbol ***va*** and the current state ***vc***. The command for this design reads

$$\begin{array}{ll} (& \mathbf{pref}[a?va;\, !s?(vb := f_0(va, vc));\, b!vb] \\ \| & \mathbf{pref}[c?vc;\, !s?(vd := f_1(va, vc));\, d!vd] \\ ) & \downarrow \{a, b, c, d\}. \end{array}$$

In each cycle the environment provides an input symbol and a current state after which the component produces the next output symbol and the next state. The next state produced by the component is fed back to the input as a current state by the INIT component

$$\mathbf{pref}(vd := init;\, [c!vd;\, d?vd]),$$

which also establishes the initial state *init*.

In the case of the buffer the functions f_0 and f_1 simply copy the input values. If the left side is the input side and the right side is the output side of the buffer, then the data communicated at the a, d, and e channels are of the type for which the buffer is designed. The data communicated on the f, c, and b channels can be reduced to signals, i.e., unary values. The command that specifies the communication behavior with respect to the channels a, b, c, and d reads

$$\begin{array}{ll} (& \mathbf{pref}[a?va;\, !s?;\, b!] \\ \| & \mathbf{pref}[!s?(vd := va);\, d!vd;\, c?] \\ ) \downarrow \{a,b,c,d\}. & \end{array}$$

The value received at the a channel is transferred to the d channel (, and subsequently to the e channel). The signals f, c, and b serve as an acknowledge signals, conveying the information that the next value can be transferred. The INIT components (which can be replaced by IWIREs if signals are communicated) in the c and f channels denote that initially the d and e channel can receive their first value. Since three values can be stored in this buffer (viz., va, vd, and ve), we call this a three-place buffer. Notice that the SYN components serve as separators between the three places.

11 Concluding Remarks

We discussed a method for the design and implementation of parallel computations. This method starts with a functional specification and decomposition which, subsequently, may lead to a behavioral specification and decomposition. The formalism and notation we introduced is particularly suited for expressing parallel behaviors and synchronizations of those behaviors.

The decompositions we derived were all part of a general decomposition pattern consisting of a number of function blocks, which implement the functions obtained in the functional decomposition, and SYN components, which synchronize the transfer of the data between the function blocks. These designs show a strong similarity with synchronous circuit design. In synchronous circuit design, the functions are implemented by combinational logic blocks and, instead of SYN components, clocked registers are used. There are some important differences, however. Reasoning about the correctness of a clocked design also includes reasoning about delays, in order to determine the clock period, the clock distribution, and the clock skew. In the above designs this is not needed. Reasoning about delays is applied in the design of the basic components only.

Although we have not explicitly reasoned about time, it is also possible to determine the response time of a design in a formal way. For example, one can argue that the design for the convolution has a constant response time whatever the length of the pipeline [6].

One of the disadvantages of the approach presented is that the implementations of the basic components, like the disjunction and SYN component, can be become rather large compared to realizations with more traditional encoding schemes. These basic components, however, are still small enough so that their physical design does not present major difficulties. They can be implemented directly using conventional components or by methods as discussed in [8].

For reasons of simplicity and brevity, we have refrained from giving a formal characterization of a delay-insensitive circuit and proving formally that the final designs represent delay-insensitive circuits. For a more formal discussion on delay-insensitive circuits and methods to prove that the circuits obtained are indeed delay-insensitive, we refer to [4].

The clear separation of mathematical and physical correctness concerns is one of the attractive aspects of the design of delay-insensitive circuits. It allows us to reason about circuit design, in almost every step of the design route, in the same manner as we can reason about program design, i.e. based on mathematical principles. Furthermore, it confines the reasoning about physical correctness concerns, like timing, to one of the last steps of the complete design route, viz., the design of the basic components. We believe that the full exploitation of this design approach will lead to more manageable and reliable designs.

Acknowledgements

Acknowledgements are due to Martin Rem, whose work was a major inspiration for this article, and to the Eindhoven VLSI Club, in particular Huub Schols and Tom Verhoeff, for their many constructive remarks.

References

[1] Burns, S.M. and A. J. Martin, Syntax-Directed Translation of Concurrent Programs into Self-Timed Circuits, in: J. Allen and F.T. Leighton, eds., *Advanced Research in VLSI, Proceedings of the Fifth MIT Conference*, (MIT Press, 1988) pp. 35-50.

[2] Clark, W.A., et al., Macromodular Computer Systems, in: *Proceedings of the Spring Joint Computer Conference*, AFIPS, (1967), pp. 335-393.

[3] Chaney, T.J. and C.E. Molnar, Anomalous Behavior of Synchronizer and Arbiter Circuits, *IEEE Transactions on Computers*, C-22 (1973) pp. 421-422.

[4] Ebergen, J.C. *Translating Programs into Delay-Insensitive Circuits*, CWI Tract 56, Centre for Mathematics and Computer Science, Amsterdam, 1989.

[5] Molnar, C.E., T.P. Fang and F.U. Rosenberger, Synthesis of Delay-Insensitive Modules, in: H. Fuchs, ed., *Proceedings 1985 Chapel Hill Conference on VLSI*, (Computer Science Press, 1985) pp. 67-86.

[6] Rem, M., Trace Theory and Systolic Computations, in: J.W. de Bakker, A.J. Nijman and P.C. Treleaven, eds., *Proceedings PARLE, Parallel Architectures and Languages Europe*, Vol. 1, (Springer-Verlag, 1987) pp. 14-34.

[7] Rem, M., J.L.A. van de Snepscheut, and J.T. Udding, Trace Theory and the Definition of Hierarchical Components, in: R. Bryant, ed., *Proceedings of the Third Caltech Conference on VLSI*, (Computer Science Press, 1983) pp. 225-241.

[8] Rosenberger, F.U., C.E. Molnar, T.J. Chaney, T.-P. Fang, Q-Modules: Internally Clocked Delay-Insensitive Modules, *IEEE Transactions on Computers*, **37** (1988) pp. 1005-1018.

[9] Seitz, C.L., System Timing, in: Carver Mead and Lynn Conway, eds., *Introduction to VLSI Systems*, (Addison-Wesley, 1980) pp. 218-262.

[10] Sutherland, I.E., Micropipelines, *Communications of the ACM*, **32** (6) (1989) pp. 720-738.

[11] Van Berkel, C., C. Niessen, M. Rem, and R. Saeijs, VLSI Programming and Silicon Compilation: a Novel Approach from Philips Research, in: *Proceedings of IEEE International Conference on Computer Design 1988*, (1988).

[12] Van de Snepscheut, J.L.A., *Trace Theory and VLSI Design*, Lecture Notes in Computer Science 200, (Springer-Verlag, 1985).

[13] Verhoeff, T., Delay-Insensitive Codes - An Overview, *Distributed Computing*, **3**, (1988) pp. 1-8.

[14] Zwaan, G., *Parallel Computations*, Ph. D. Thesis, Eindhoven University of Technology, 1989.