Parallel Program and Asynchronous Circuit Design [*]

Jo C. Ebergen
John Segers, Igor Benko

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

Revised March 30, 1994

Abstract

Asynchronous circuit design is a beautiful application area for any formalism that can reason about parallelism. By means of two small, but challenging, exercises we illustrate the similarities and differences between parallel program and asynchronous circuit design. The exercises are simple to state and have many solutions, which are sometimes surprisingly efficient. They all illustrate many aspects of asynchronous circuit design. For each exercise we present several solutions, which are analyzed with respect to delay assumptions, safety, progress, and performance issues. We also mention some open problems.

[*]This work is supported by the Natural Sciences and Engineering Research Council of Canada under grant OGP0041920 and by a grant from the Information Technology Research Centre of Ontario.
These notes are a revision of the lectures presented at the BANFF VII Workshop on Asynchronous Hardware Design, August 28 – Sept. 3, 1993.

Contents

It is quite difficult to think of the code
entirely in abstracto *without any kind of circuit.*
Alan M. Turing [33].

1 Introduction

The design and analysis of asynchronous circuits has witnessed a remarkable upsurge in the past five years. Many researchers have claimed or demonstrated that asynchronous circuits have a great potential for speed, low power consumption, robustness, modular design, and ease of design. Some of these properties, like modular design and ease of design, were already demonstrated in the 60's in the Macromodules project [7]. The recent advances are characterized by many novel design and verification techniques [1, 3, 6, 11, 12, 17, 21, 32] or by the new applications of and improvements in classical approaches [4, 8, 9, 19, 24, 28]. The performance analysis of asynchronous circuits is also beginning to draw wide attention [2, 5, 13, 27, 30, 36]. The novel techniques indicate that the design and analysis of asynchronous circuits have much in common with the design and analysis of parallel programs. The purpose of this paper is to illustrate some techniques in designing asynchronous circuits and in analyzing the performance of these circuits with respect to area complexity, power consumption, and response time. The design of a circuit consists of the derivation of a parallel program for a specification. Wherever possible, we try to give a heuristics for each design decision. The derivations are presented in a format that allows a quick verification of all steps.

Our method for designing asynchronous circuits is based on a simple formalism. Specifications are given by means of guarded commands with input and output actions, a notation inspired by Dijkstra's guarded commands [10] and Hoare's CSP [15]. The formalization of implementation is given by the definition of decomposition. A decomposition of a specification consists of a network of basic components realizing the specified behavior. The notion of a delay-insensitive circuit plays an important role in our implementations. A delay-insensitive circuit is a special type of an asynchronous circuit, which is informally characterized as a network of basic components implementing a specification such that the correctness of the implementation is insensitive to any delays in wire connections or variations in the response time of the basic components.

The modulo-N counter and the up-down N counter are beautiful examples for illustrating both design and analysis techniques for asynchronous circuits. Both components have simple specifications, but admit a surprising variety of implementations. We give detailed derivations of several implementations for the two counters and a performance analysis of the designs. Our final designs for the counters have a bounded response time, a bounded power consumption, and an area complexity logarithmic in N. All bounds are asymptotically optimal. We start by giving a specification for the modulo-N counter. Along the way we explain the rules of the game.

2 The Modulo-N Counter

The goal of the game is to find an efficient decomposition of the modulo-N counter into basic components, for any $N > 1$. The decomposition should be efficient with respect to area complexity, power consumption, and response time.

There are several ways to specify a modulo-N counter. Perhaps the most simple one is to view the modulo-N counter as a component with one input r and two outputs $a0$ and $a1$. After each of the first $N-1$ inputs r, the component may respond with output $a1$, and after the Nth input the component may respond with output $a0$. This behavior then repeats. A formal specification for this component can be formulated as follows.

$$\mathbf{pref} * [\,(r?;\ a1!)^{N-1};\ r?;\ a0!\,]$$

where '?' denotes an input action, '!' denotes an output action, ';' denotes concatenation, 'E^N' denotes N-fold repetition of the command E, '$*[\ ]$' denotes arbitrary repetition of the enclosed command (or Kleene's closure), and $\mathbf{pref}E$ denotes the prefix-closure of E. (A more precise explanation of the notation follows in the next section.)

A higher-level specification can be given if we consider the outputs $a0$ and $a1$ as special implementations of sending the values 0 and 1 on channel a. Channel a is then considered as an output channel of type binary. We denote this by $a! : bin$. Channel r can be considered an input channel of type unary, since each signal on channel r can be seen as the communication of a value that is always the same. If r is an input channel of type unary, we denote this by $r? : un$. In this more abstract view, the modulo-N counter can be specified as follows.

```
  $ModC(N : int,\ r? : un,\ a! : bin\ )$
=     { by definition }
  $|[$ var $n : int$ ::
    initially $n = 0$ ::
    pref $* [\ r?; n := (n+1) \bmod N;$
             if $n \neq 0$ then $a := 1$
              | $n = 0$ then $a := 0$
             fi ; $a!$
             ]
  $]|$
```

where *int* denotes the type integer and an occurrence of $a!$ denotes 'the value of a is sent along its channel.' In order to leave as much freedom as possible in choosing an implementation, in particular in implementing the data types, we use this specification for deriving decompositions.

A property worth remembering of this last specification is that the precondition for each output $a!$ is given by

$$(\#r \bmod N = 0)\ \equiv\ (a = 0)$$

where $\#r$ is the number of inputs r received thus far. For this reason, a postcondition of a corresponding input $a?$ in another component can given by the same assertion. We use this property frequently in the coming derivations.

When trying to find an implementation for the modulo-N counter, we assume that the environment provides the inputs as specified. For example, after providing input r the environment waits for output a before providing the next input. Under this assumption our implementation should provide outputs as specified.

In traditional circuit design, a modulo-N counter is specified by means of a state transition diagram, where states and transitions are given by means of logic values. Implementations are nearly always based on a binary representation of the count and consist of a series of latches and logic gates. Often, these designs have a so-called ripple-carry delay, which makes the response time dependent on N. Furthermore, if every latch is clocked in each clock period, the power consumption of these designs is at least logarithmic in N. Various designs for the modulo-N counter implemented in this way can be found in almost any textbook on switching theory. We demonstrate that modulo-N counters with bounded response time and bounded power consumption can be designed. Our designs are inspired by ideas developed at Philips Research Laboratories [2, 18].

3 Rules of the Game

In the previous section we briefly stated our goal and gave a specification of the modulo-N counter without much explanation. An attentive reader may have lots of questions. For example, what is the formal meaning of our program notation? What do we mean by 'implementation'? What basic components can we use? How can we implement data types? How do we measure area complexity, power consumption, and response time? In short, what are the rules of the game? We try to answer briefly some of these questions in the coming sections.

3.1 Commands

The modulo-N counter is specified by a so-called *guarded command.* A guarded command prescribes the communication behavior of a component by listing all sequences of communication actions that may occur between the component and its environment. We first consider a subset of guarded commands called *commands.*

The semantics of a command is given by a *trace structure*, which is a triple $\langle I, O, T \rangle$. Set I is the *input alphabet* and represents all input terminals of the component; O is the *output alphabet* and represents all output terminals of the component; T is the *trace set* and represents all possible communication behaviors between component and environment. Every trace in T is constructed from symbols in $I \cup O$.

For command E, the notations $\mathbf{i}E$, $\mathbf{o}E$, and $\mathbf{t}E$ stand for the input alphabet, output alphabet, and trace set of the trace structure represented by E respectively. The alphabet of E is denoted by $\mathbf{a}E$ and given by $\mathbf{a}E = \mathbf{i}E \cup \mathbf{o}E$. Equality between commands denotes equality of the trace structures represented by the commands.

The so-called atomic commands are *abort*, *skip*, and $a!$, $a?$, and a for any symbol a. They denote the following trace structures

$$\begin{aligned} abort &= \langle \emptyset, \emptyset, \emptyset \rangle \\ skip &= \langle \emptyset, \emptyset, \{\varepsilon\} \rangle \end{aligned}$$

$$\begin{array}{rcl} a! & = & \langle \emptyset, \{a\}, \{a\} \rangle \\ a? & = & \langle \{a\}, \emptyset, \{a\} \rangle \\ a & = & \langle \{a\}, \{a\}, \{a\} \rangle \end{array}$$

The trace set of a command is constructed in a similar way as the language of a regular expression. The constructions for concatenation ';', selection (or union) '|', N-fold repetition '$()^N$', and arbitrary repetition '$*[\ ]$' are defined as usual and omitted here. The operation *weaving* is used to express a parallel composition with synchronization on common symbols. Formally, the weave $E0 \| E1$ of trace structures $E0$ and $E1$ is defined by

$$\begin{array}{rcl} E0\|E1 & = & \langle\ \mathbf{i}E0 \cup \mathbf{i}E1 \\ & & ,\ \mathbf{o}E0 \cup \mathbf{o}E1 \\ & & ,\ \{t \in (\mathbf{a}E0 \cup \mathbf{a}E1)^* \mid t{\downarrow}\mathbf{a}E0 \in \mathbf{t}E0\ \wedge\ t{\downarrow}\mathbf{a}E1 \in \mathbf{t}E1\} \\ & & \rangle. \end{array}$$

where $t \downarrow \mathbf{a}E0$ stands for the projection of trace t on the alphabet of $E0$. (Recall that $\mathbf{a}E0 = \mathbf{i}E0 \cup \mathbf{o}E0$.) Notice that every trace in the weave $E0\|E1$ must be in accordance with a trace of $E0$, if you only look at symbols from $E0$, and with a trace of $E1$, if you only look at symbols from $E1$. Because the weave $E0\|E1$ consists of all behaviors that are in accordance with behaviors in $E0$ *and* in $E1$, weaving can be considered as the *behavioral conjunction* of $E0$ and $E1$. For this reason, the symbol $\|$ is often pronounced as 'and.' There are two special cases of weaving. If $\mathbf{a}E0 \cap \mathbf{a}E1 = \emptyset$, then weaving $E0$ and $E1$ amounts to the interleaving of the traces of $E0$ and $E1$. If $\mathbf{a}E0 = \mathbf{a}E1$, then weaving $E0$ and $E1$ amounts to taking the intersection of the traces of $E0$ and $E1$.

The *prefix-closure* of command E, denoted by $\mathbf{pref}E$, is a trace structure with the same alphabets as E and with the trace set that consists of all prefixes of all traces in $\mathbf{t}E$. Trace structure E is called *prefix-closed* if $\mathbf{pref}E = E$. The **pref** operation constructs prefix-closed trace structures. Specifications are always given by commands that represent prefix-closed, non-empty trace structures with disjoint input and output alphabets. Trace structure E is called *non-empty* if $\mathbf{t}E \neq \emptyset$.) Whenever we speak of *component* E in the following, E represents a prefix-closed, non-empty trace structure with disjoint input and output alphabets. The domain of prefix-closed, non-empty trace structures is one of the simplest semantic domains [15]. Notice that *abort*, $a?$, and $a!$ do not represent non-empty, prefix-closed trace structures and therefore do not specify components. The command *skip* does represent a prefix-closed, non-empty trace structure and therefore specifies a component: a component that has no input or output terminals and doesn't do anything.

The **pref** operator allows us to rewrite a specification by means of *(un)folding.* For example, we have

$$\begin{array}{ll} & \mathbf{pref} *[\ a?; b!\ ] \\ = & \quad \{\ \text{unfolding}\ \} \\ & \mathbf{pref}(a?; *[\ b!; a?\ ]) \end{array}$$

Finally we mention some laws for *skip* and *abort*

$$\begin{aligned} E;\ \mathit{skip} &= E \\ E \parallel \mathit{skip} &= E \\ E \mid \mathit{abort} &= E \\ \mathbf{t}(E;\ \mathit{abort}) &= \mathbf{t}(\mathit{abort}) \\ \mathbf{t}(E \parallel \mathit{abort}) &= \mathbf{t}(\mathit{abort}) \end{aligned}$$

3.2 Some Specifications and their Interpretations

In order to familiarize ourselves with commands, we specify the basic components WIRE, IWIRE, MERGE, TOGGLE, JOIN, and the 2-by-1 JOIN. Their specifications are given in Table 1. As for the priority of the binary operators, the parallel bar ($\parallel$) has the highest binding power, then the semicolon (;), and finally the choice bar ($\mid$).

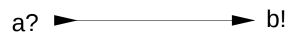
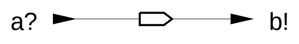
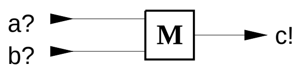
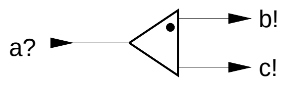
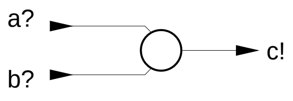
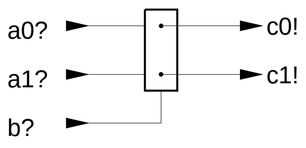
Name	*Specification*	*Schematic*
WIRE	**pref** $*[\ a?; b!\ ]$	
IWIRE	**pref** $*[\ b!; a?\ ]$	
MERGE	**pref** $*[\ (a?\ \mid\ b?)\ ; c!\ ]$	
TOGGLE	**pref** $*[\ a?; b!; a?; c!\ ]$	
JOIN	**pref** $*[\ (a? \parallel b?)\ ; c!\ ]$	
2-by-1 JOIN	**pref** $*[\ (a0? \parallel b?)\ ; c0!$ $\mid (a1? \parallel b?)\ ; c1!$ $]$	

Table 1: Some basic components

In the previous section we stipulated that components are specified by means of commands representing prefix-closed, non-empty trace structures with disjoint input and output alphabets. We interpret such a trace structure in a special *mechanistic* way as opposed to a physical interpretation related to a particular implementation.

In order to understand this mechanistic interpretation, it is important to explain the role of input and output first. In our interpretation, an output may be produced by a component as soon as that output is enabled in the specification. Similarly, an input may be produced by the environment as soon as that input is enabled in the specification. The environment can be considered as a collection of other components.

The mechanistic interpretation consists of two conditions: a condition related to safety and a condition related to progress.

- Safety: no input or output is produced when not allowed by the specification.
- Progress: each trace specified may occur.

Because of our interpretation of inputs and outputs, the safety condition not only prescribes what the component may not do, but also what the environment may not do. A specification is a prescription for both component *and* environment. Because of the environment prescription, we can stipulate the conditions under which correct component behavior must be guaranteed. In this context we can interpret specification E as 'if the environment produces the inputs as prescribed in E, then the component may produce the outputs as prescribed in E.' If the environment violates the prescription, nothing is guaranteed and erroneous behavior may occur. The production of an input or output that violates a specification is also called *computation interference*. This term was first introduced in [31]. In [11], it is called a *choke*. Later, when we consider a network of components as an implementation of a specification one of our proof obligations is to show that computation interference cannot occur.

While the safety condition prescribes what may not happen, the progress condition prescribes to a certain extent what must happen. The progress condition says that, if the environment produces the inputs as specified in E, then the component must behave such that every trace in $\mathbf{t}E$ may occur. Here 'may' should be interpreted as 'is possible, but not guaranteed.' This requirement excludes, for example, implementations of components where some traces cannot occur. On the other hand, this prescription does not require that every trace is guaranteed to occur. The actual occurrence of a trace may depend on nondeterministic choices made during the operation of the implementation: if the right choices are made, then each trace can occur. This progress condition is too weak in a number of cases. On the other hand, it is easy to work with and suffices in many cases. Formulating a general, satisfactory progress condition that is convenient to deal with is still an open problem. We will return to a more formal treatment of these conditions and their problems later when we discuss implementations of specifications.

The abstract mechanistic interpretation allows for several physical implementations, like a mechanical, optical, or electrical one. The usual electrical implementation is that each symbol in the alphabet is associated with a terminal of a circuit. Each occurrence of a symbol in a trace corresponds to a voltage transition at that terminal. There is no distinction between rising and falling transitions: both transitions are denoted by the same symbol. This type of signaling is called *transition signaling* [32]. Outputs are transitions caused by the circuit and inputs are transitions caused by the environment.

With the above mechanistic interpretation in mind, we can explain the behavior of the basic components as follows. A WIRE has one input and one output terminal. Each communication behavior is an alternation of inputs and outputs, starting with an input, if any. (A WIRE can be implemented by a physical wire.) Notice that the safety condition prescribes that the environment is not allowed to provide two inputs in a row, nor is the component allowed to produce initially an output, or to produce two outputs in a row. The progress condition, on the other hand, prevents implementations where an output is guaranteed never to occur.

The IWIRE also has one input and one output terminal. Each communication behavior is an alternation of inputs and outputs, starting with an output, if any.

The MERGE has two input terminals and one output terminal. Each communication behavior is an alternation of inputs and outputs: the environment may choose to produce one of the two inputs (not both), the component may then produce an output. (The MERGE can be implemented by an XOR gate.)

The TOGGLE has one input and two output terminals. Each communication behavior is also an alternation of inputs and outputs starting with an input, if any. After each input has been received, an output may be produced. The outputs are produced at alternating terminals.

The JOIN has two input terminals and one output terminal. Each communication behavior is an alternation of two inputs and an output. After both inputs have been received, the JOIN may produce an output. The JOIN can be implemented by a Muller C-ELEMENT [23]. The Muller C-ELEMENT has a more general specification:

$$\textbf{pref} * [\ a?^2 \mid b?^2 \mid (a?\|b?;\ c!)\ ]$$

Notice that the behaviors of the JOIN are a proper subset of the behaviors of the C-ELEMENT. For example, for the C-ELEMENT the environment is allowed to produce an input a and then immediately withdraw that input. The specification of the JOIN prescribes that the environment is not allowed to withdraw an input. Knowing what the environment may do and what it may not do will be essential later when we give our correctness criteria for an implementation. For these reasons we distinguish between the JOIN and the C-ELEMENT. (The JOIN is also known as the RENDEZ-VOUS element in the Macromodules project [7].)

The 2-by-1 JOIN has three input and two output terminals. Each communication behavior is an alternation of two inputs followed by an output. Either $a0$ and b are received in parallel and then a $c0$ may be produced, or $a1$ and b are received in parallel and then a $c1$ may be produced. We have listed the 2-by-1 JOIN only. Other n-by-m JOINs for $n, m > 0$ are specified in a similar way. Notice that the standard JOIN is a special n-by-m JOIN, viz., a 1-by-1 JOIN. (n-by-m JOINs for $m, n > 1$, are called DECISION-WAIT modules in the Macromodules Project [7].)

3.3 Guarded Commands

If we extend our command language with variables, channels, and guarded selection we get the language of guarded commands with input and output actions. A variable n of type T is declared by **var** $n : T$. Type T denotes the set of values n can take. For example, the type binary is defined by

$$\textbf{type}\ bin = \{0, 1\}$$

If the initial value of a variable n is val then this can be indicated in the program by

$$\textbf{initially}\ n = val$$

The scope of variables is delineated by means of the scoping brackets $|[$ and $]|$. The formal meaning of delineating the scope of a list of variables V is given by hiding those actions that involve changes to those variables. For example, we have

```
|[ var $n : bin$ ::
   initially $n = 0$ ::
   pref $*[\ r?;\ n := (n+1) \bmod 2;\ a!1;\ r?;\ n := (n+1) \bmod 2;\ a!0\ ]$
]|
=     { hiding local variable $n$ }
   pref $*[\ r?;\ a!1;\ r?;\ a!0\ ]$
```

In any parallel composition components are not allowed to share variables; components can only communicate through message passing via channels. For each component a channel is either an input channel or an output channel. An input channel r of type T is declared by $r? : T$ in the specification of the component. The occurrence of $r?$ in the program text denotes an input action and means 'r receives a value on its channel.' If *val* is a value of type T, then $r?val$ means 'receive value *val* on channel r.' An output channel b of type T is declared by $b! : T$. The occurrence of $b!$ in the program text denotes an output action and means 'the value of b is sent along its channel.' Similarly, $a!val$ means 'send value *val* along channel a.' Local channels in a parallel composition are declared as follows.

$$|[\ \mathbf{chan}\ b : T ::\ \ C0(a?, b!) \parallel C1(b?, c!)\ ]|$$

where components $C0(a?, b!)$ and $C1(b?, c!)$ have been specified elsewhere with channel b of type T. If channels have types associated with them, then the formal meaning of a component is given by a trace structure where symbols are pairs (b, val) with b representing the name of the channel and *val* representing the value that is communicated. (See for example [27].)

The last extension to our language is guarded selection. A guarded selection has the following form.

```
if $B0$ then $S0$
| $B1$ then $S1$
fi
```

Here $B0$ and $B1$ are guards of the commands $S0$ and $S1$ respectively. A guard is a Boolean expression. Informally, the meaning of this guarded command is the selection of the commands for which the guards evaluate to true. If no guard evaluates to true, the command is trace-equivalent to *abort*. (Recall that *abort* is the identity of selection.) So the formal meaning is

```
   if $B0$ then $S0$ | $B1$ then $S1$ fi
=     { by definition }
   $(S0' \mid S1')$
```

where the alphabets of $S0$ and $S0'$ are the same and

$$\mathbf{t}(S0') = \begin{cases} \mathbf{t}(S0) & \text{if } B0 \text{ holds} \\ \mathbf{t}(abort) & \text{otherwise} \end{cases}$$

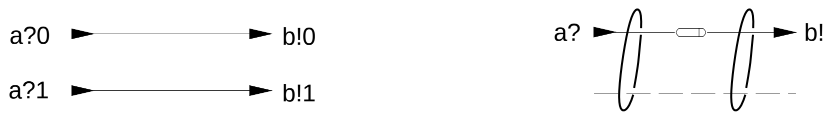
A similar meaning applies to $S1'$.

3.4 Two Implementations for Data Communications

If a channel is of type unary, then implementing a data communication along that channel is straightforward: the channel consists of a single wire connection, and sending a single transition along that connection implements a data communication. What if the data types are not unary? For example how do we implement a data communication along a binary channel? We consider two types of implementations: an implementation using a dual-rail transition encoding and an implementation using a single-rail data-bundling encoding. As an example we use a component with a binary input channel a, binary output channel b, and communication behavior given by

$$\textbf{pref} * [\ a?;\ b := a;\ b!\ ]$$

With a dual-rail transition encoding, each bit is implemented using two wires, one wire for the value zero and one wire for the value one. Sending a transition along the '0' wire implements the communication of a zero, and sending a transition along the '1' wire implements the communication of a one. With single-rail data bundling, each bit is implemented using a single wire. The level on that wire indicates which binary value is communicated. These wires are called the data wires. Besides the data wires, which indicate *what* value is communicated, there is one wire that is used to indicate *when* the data is communicated. This wire is called the data-valid wire. Whenever a transition occurs at the connection, the data wires are assumed to have reached a stable value. This assumption imposes a delay constraint between the arrival of the transition on the data-valid wire and the 'validness' of the data wires. This constraint is called the data-bundling constraint. In order to satisfy the data-bundling constraint, delays may have to be inserted in the data-valid wire to compensate for any possible delays in data wires. In Figure 1 these delays are indicated by long ovals. As a convention, we indicate the



(a) Dual-rail transition encoding (b) Single-rail data bundling

Figure 1: Two data implementations for data types

data-valid wires by solid lines. The data wires in the data part are indicated by dashed lines. The data wires and the data-valid wire that belong together are 'bundled' by drawing a circle around them.

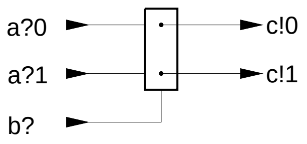
As a further illustration, we discuss some implementations of two other components.

$$C(a? : bin,\ b? : un,\ c! : bin)$$

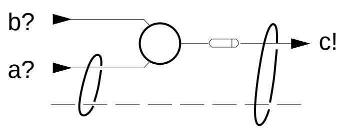
$$= \quad \{ \text{ by definition } \}$$

$$\textbf{pref} * [a?\|b?;\ c := a;\ c!\ ]$$

If we forget the data types, the behavior of component C is the same as that of the JOIN. If we use a dual-rail transition encoding, an implementation consists of a 2-by-1 JOIN. The implementation using single-rail data bundling consists of a JOIN, a delay, and a data wire as given in Figure 2.



(a) Dual-rail transition encoding



(b) Single-rail data bundling

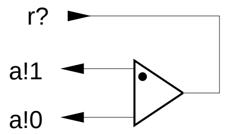
Figure 2: Two implementations for C

Finally we discuss two implementations for the modulo-2 counter $ModC(2, r?, a!)$. There are several ways in which the specification of the modulo-2 counter can be written. Here are two of them.

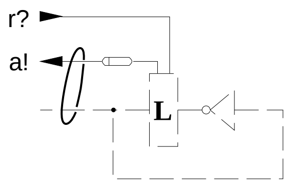
```
|[ var $n : bin$ ::
   initially $n = 0$ ::
   pref $* [r?;\ n := (n+1) \bmod 2;\ a := n;\ a!]$
]|
```

$=$ { hide variable n }

pref $* [\ r?;\ a!1;\ r?;\ a!0\ ]$



(a) Using dual-rail transition encoding



(b) Using single-rail data bundling

Figure 3: Two implementations for $ModC(2, r?, a!)$

When using dual-rail transition encoding, the modulo-2 counter can be implemented by a TOGGLE. This implementation resembles the last specification. The implementation using single-rail data bundling resembles the first specification. It consists of a transition latch marked L, a delay, and a feedback wire with an inverter. When the transition latch receives an input transition, it latches the input data —which becomes the new output data— and then sends the data-valid signal for the output data. Normally, each set of data wires implementing a variable should have a data-valid wire associated with it to compensate for possible delays in the data wires. We have not done so here for the feedback wire with the inverter, since the local delays in this wire can easily be compensated for by a delay in the data-valid wire for the output. In the following we apply a similar optimization for local feedback wires.

The generation of the data-valid signals can be seen as just another 'clocking' strategy. Unlike conventional clocking strategies, where the clock signals are generated at regular, fixed intervals and sent to all storage devices, the generation of the data-valid signals depends on the occurrence of a data communication, and the route of a data-valid signal may depend on the value of the data. A convenient component to route the data-valid signal depending on the value of a binary variable is the SELECT. The SELECT can be seen as a demultiplexer for the data valid signal. Its specification is given in Figure 4. Each input t will propagate to either

$$\mathbf{pref} * [\ *[\ t?;\ q0!];\ b?;\ *[t?;\ q1!];\ b?\ ]$$

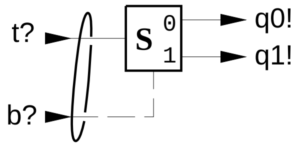


Figure 4: Specification of the SELECT

output $q0$ or output $q1$. The level of the input b will determine whether output $q0$ or $q1$ will be produced. If the level is 0 output $q0$ will be produced, and if the level is 1, output $q1$ will be produced. Initially the level of b is 0. It is assumed that b has reached a stable level before a transition on t occurs. This constraint is in fact a data bundling constraint. For this reason we have encircled the t and b wires in Figure 4.

Implementing data communications using data bundling has been employed extensively in the Macromodules project [7]. One of the advantages of employing data bundling is that conventional logic design can be used for the data part, which often results in smaller circuits. A disadvantage is the implementation of the data-bundling constraints, which results in a worst-case behavior and makes the design sensitive to certain delay variations.

In a data-bundling implementation, the circuit consisting of the data-valid wires is the control circuit. This circuit can be designed separately from the data part. When combining the control part and the data part, the data-bundling constraints are met by inserting appropriate delays in the (data-valid) wires of the control part. Of course, such a method will only work if the control part is a delay-insensitive circuit. That is, the correctness of the circuit for the control part should be insensitive to adding arbitrary delays in the wire connections. Accordingly, delay-insensitive circuits play an important role in the design of data-bundling implementations. In the next sections we explain what exactly is a delay-insensitive circuit.

3.5 Decomposition versus Parallel Composition

Expressing a specification as a parallel composition (i.e., a weave) of smaller specifications is a first step towards finding a network implementation. Unfortunately, it is not the case that each parallel composition can be interpreted as a network of components that implements the specification. The problem is that that our parallel composition expresses a synchronization on common symbols irrespective of whether these symbols are input or output symbols. In general, a component in a network will produce an output as soon as that output is locally enabled. It will not wait for other components, where that same symbol is an input for example, to synchronize on the production of that symbol. In order to transform a parallel composition into a network implementation and avoid these synchronization failures there are several things we can do. One solution is to enforce the proper synchronizations by introducing handshakes for each communication on each channel. This method is proposed by Martin [21] and is also used by Van Berkel [1]. Another solution is to design your parallel composition in such a way that no handshakes are needed: whenever a component can produce an output, the components for which that symbol is an input are waiting for that input to happen. We need to formalize this property more precisely. We first discuss what our correctness criteria are for a network implementation of a specification.

The definition of *decomposition* formalizes the idea of 'implementing a specification by a network of components.' In this section we discuss four conditions that have to be satisfied in order to call a network a decomposition of a specification. These conditions are based on our abstract mechanistic interpretations of specifications.

Let us briefly recall the mechanistic interpretation of a specification. Our interpretation is based on the distinction between the component and the environment and on two conditions, the safety and the progress condition. Both conditions prescribe the behavior of the component and the environment. The safety condition says that no input or output may be produced when not allowed by the specification. The progress condition says that every trace specified may occur. It is important to distinguish the component's prescriptions and the environment's prescriptions in a specification. When implementing a specification, we try to implement the component's prescriptions by a network of components assuming that the environment's prescriptions are satisfied.

We formulate the conditions for a decomposition of component E into components $E_1, \ldots, E_n$, $n > 0$. The network of components E_1 through E_n is denoted by $(E_1, \ldots, E_n)$. The property that E can be decomposed into the network consisting of E_1 through E_n is denoted by $E \rightarrow (E_1, \ldots, E_n)$.

First, we take into account the behavior of the environment with respect to the network $(E_1, \ldots, E_n)$. The environment's prescriptions for the network are given in E. In order to consider the production of an input by this environment as the production of an output by a component, we consider the *reflection* of E, denoted by $\overline{E}$ and defined by

$$\overline{E} = \langle \mathbf{o}E, \mathbf{i}E, \mathbf{t}E \rangle$$

Consequently, $\mathbf{i}\overline{E} = \mathbf{o}E$, $\mathbf{o}\overline{E} = \mathbf{i}E$, and $\mathbf{t}\overline{E} = \mathbf{t}E$. By reflecting E, we interchange the prescriptions for the component and the environment. Instead of considering $\overline{E}$ and network $(E_1, \ldots, E_n)$, we now consider the network $(E_0, \ldots, E_n)$, where $E_0 = \overline{E}$.

In order for E to be decomposable into the network $(E_1, \ldots, E_n)$, four conditions have to hold for the network $(E_0, \ldots, E_n)$. Two conditions concern the so-called structure of the network and two conditions concern the behavior of the network. The behavioral conditions are related to the safety and the progress condition in our interpretation. We discuss these four conditions below.

- **Closed network**
  In the network $(E_0, \ldots, E_n)$ there are no dangling inputs and outputs: every input is connected to an output and every output is connected to an input.

- **No output interference**
  Outputs of distinct components are not connected to each other. In other words, the output alphabets of the components in the networks are pairwise disjoint. The first two conditions guarantee that each symbol is an output of exactly one component and an input of at least one component. (Notice that an output may be connected to multiple inputs.)

- **No computation interference** (Safety)
  The third condition states that the environment's prescriptions for every specification in the closed network may not be violated. In other words, no output may be produced by any component when not allowed by the specifications for which this symbol is an input. Or, to put it positively, every output that can be produced by a component can be accepted as input by the receiving component. This condition can be checked by constructing all possible network behaviors and verifying that no environment prescription is violated. The trace set T of all possible network behaviors can be constructed as follows. Initially, $T = \{\varepsilon\}$. We repeatedly enlarge trace set T with extensions of traces in T as indicated below until T can no longer be enlarged. The rule for extending T is as follows. Choose a trace $t \in T$, symbol z, and component E_i, such that after network behavior t, component E_i may produce output z. Since component E_i cannot be prevented from producing output z after behavior t, tz is also a possible network behavior, and we add tz to T. Our third condition states that any possible network behavior must be in accordance with every specification, which can be formulated by

  $$T \downarrow \mathbf{a}E_i \subseteq \mathbf{t}E_i \quad \text{for all } i = 0, \ldots, n. \tag{1}$$

  where T represents the trace set of all networks behaviors. If this condition holds, we say that the network is *free of computation interference.*

- **Completeness with respect to specification** (Progress)
  The fourth condition states that every trace of the specification E may also occur in the network behaviors. This condition is formulated as

  $$T \downarrow \mathbf{a}E = \mathbf{t}E \tag{2}$$

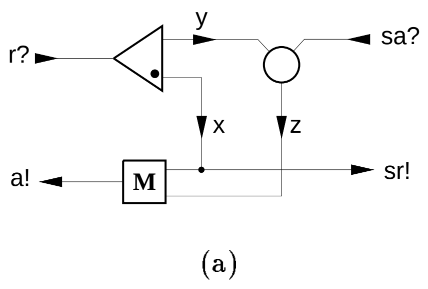
  where again T represents the trace set of all network behaviors. If this condition holds we say that the network behaviors are *complete with respect to the specification.*

3.6 Some Examples

We consider some examples to familiarize ourselves with this definition of decomposition. We are given component E specified by

$$E = \mathbf{pref} * [\, r?;\ (a!;\ r?) \parallel (sr!;\ sa?);\ a!\,]$$

For the moment we consider all channels to be unary. Notice that for every four communications on r and a there are two communications on sr and sa. For this reason, this component can be called a four-phase-to-two-phase converter. In Figure 5 two decompositions for E are given. Both networks contain a TOGGLE, JOIN, and a MERGE. Network (b) also contains an IWIRE. It



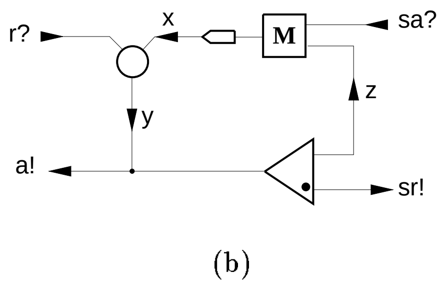


Figure 5: Two decompositions for a four-to-two-phase converter E

is easy to verify that both networks, *including* the environment, form a closed network without output interference. Notice that in networks (a) and (b) there are multiple input connections at terminals x and y respectively. If we construct the network behaviors for network (a), again including the environment, we find

$$T = \mathbf{t}\left(\mathbf{pref} * \left[\, r?;\ x;\ ((a!;\ r?;\ y) \parallel (sr!;\ sa?));\ z;\ a! \,\right]\right)$$

If we project trace set T on each of the component's alphabets we obtain a subset of the trace set of the component. That is,

$$T \downarrow \mathbf{a}E_i \subseteq \mathbf{t}E_i$$

where E_i is a specification of a component. We even have $T \downarrow \mathbf{a}E = \mathbf{t}E$. From these observations we can conclude that network (a) is indeed a decomposition of the four-to-two-phase converter.

The network behaviors for network (b), including the environment, are given by

$$T = \mathbf{t}\left(\mathbf{pref}\left(r?\|x;\ *\left[\, y;\ ((a!;\ r?) \parallel (sr!;\ sa?;\ x));\ y;\ ((a!;\ r?) \parallel (z;\ x)) \,\right]\right)\right)$$

If we project T on each of the component's alphabets, we obtain subsets of the trace sets of the components: no specification is violated. Furthermore, we have that $T \downarrow \mathbf{a}E = \mathbf{t}E$. (Notice that by deleting x, y, and z and after some folding we get specification E again.) Consequently, network (b) is also a decomposition of the four-to-two-phase converter.

As an example where the safety condition is violated, we can take network (a) with the WIRE between terminal x and sr replaced by an IWIRE. Since initially this network can produce sr, the set of network behaviors would now include trace sr. According to the specification E, sr may not occur initially, so the safety condition is violated.

As an example where the progress condition is violated, we take network (b) with the IWIRE replaced by a WIRE. The network behaviors are then given by $T = \mathbf{t}\,(\mathbf{pref}\ r?)$. After the initial input r, nothing will ever be produced. Obviously, not every trace from the specification E may occur, so the progress condition is violated.

The above decompositions only apply if the channels are unary channels. If the channels are not unary, we may get a different decomposition depending on how we implement data communications. If data communications are implemented using a single-rail data bundling scheme, these decompositions may be used as a control circuit for directing the data-valid signals. When the control part is combined with the data part, delays may have to be inserted in some of the wire connections. The important question then is whether these delay insertions will affect the correctness of the decomposition. In other words, do these networks represent delay-insensitive circuits? This question is answered in the next section.

3.7 Modularity and Delay-Insensitivity

With the definition of decomposition we can now state precisely what it means that a circuit can be designed in a modular way and what the differences are between a speed-independent and a delay-insensitive circuit.

The modularity of our design method is based on the Substitution Theorem [12]. The Substitution Theorem applies to problems of the following kind. Suppose that component E_0 can be decomposed into a number of components of which F is one such component. Suppose, moreover, that F can be decomposed further into a number of components. Under what conditions can the decomposition of F be substituted for F in the decomposition of E_0?

Theorem 1 (Substitution Theorem) *For components E_0, E_1, E_2, E_3, and F we have*

$$\begin{array}{rcll} \textit{if} & E_0 & \rightarrow & (\ E_1,\ F\ ) \\ \textit{and} & F & \rightarrow & (\ E_2,\ E_3\ ) \\ \textit{then} & E_0 & \rightarrow & (\ E_1,\ E_2,\ E_3\ ), \end{array}$$

if the following condition is satisfied.

$$(\mathbf{a}E_0 \cup \mathbf{a}E_1) \cap (\mathbf{a}E_2 \cup \mathbf{a}E_3) = \mathbf{a}F. \qquad (3)$$

Condition (3) states that the only symbols that the decompositions of E_0 and F have in common are symbols from $\mathbf{a}F$. It is essentially a trivial condition, since, by an appropriate renaming of the internal symbols in the decomposition of F, this condition can always be satisfied. The internal symbols of the decomposition of F are given by $(\mathbf{a}E_2 \cup \mathbf{a}E_3)\backslash\mathbf{a}F$, where '\' means set deletion.

The theorem above considers decompositions into two components only. The generalization of this theorem to decompositions into more than two components is straightforward.

Some of the main attractions that are often mentioned in connection with speed-independent and delay-insensitive circuit design are the 'modular design approach,' the 'hierarchical design approach,' 'design by stepwise refinement,' or the 'possibility for incremental improvements.' All these characteristics refer to the same property in our formalism and are symbolized in the Substitution Theorem.

If a network is a decomposition of a specification, it represents a *speed-independent circuit.* Therefore, a decomposition is also called an *SI decomposition.* A speed-independent circuit, however, is not necessarily a delay-insensitive circuit. The reason is that decompositions are invariant under variations in response times of components, but not under the insertion of communication delays in the connections. If the correctness of the circuit is invariant under the insertion of communication delays in the connections as well, then we call such a circuit a *delay-insensitive circuit.* While a speed-independent circuit is formally described by means of (SI) decomposition, a delay-insensitive circuit is formally described by means of *DI decomposition.* A DI decomposition is a decomposition in which all communication delays between the components are taken into account. Formally, these communication delays are represented by WIREs.

We say that the network $(E_1, \ldots, E_n)$ forms a DI decomposition of component E, denoted by $E \overset{DI}{\rightarrow} (E_1, \ldots, E_n)$ if and only if

$$E \rightarrow (\ ren(E_1),\ Wires(E_1), \ldots, ren(E_n),\ Wires(E_n)\ ).$$

where $ren(E_i)$ is a renaming of E_i to a 'localized' version and $Wires(E_i)$ is the collection of WIRE components connecting $ren(E_i)$ with its original terminals $\mathbf{a}E_i$.

In general, DI decompositions are more difficult to derive and verify than SI decompositions, because of all the connection WIREs. By the Substitution Theorem, it follows that a SI decomposition is a DI decomposition, if all constituent components are DI components. A component E is called a DI component, if

$$E \rightarrow (ren(E),\ Wires(E)).$$

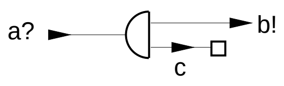
The DI property formalizes that the communication behavior between component and environment is insensitive to the insertion of communication delays. In other words, specification E is invariant under any extension with WIREs at its input or output terminals. All basic components of Figure 1 are DI components. Consequently, the decompositions of Figure 5 are DI decompositions. The C-ELEMENT and the SELECT, however, are not DI components.

The idea of formalizing delay-insensitivity using a characterization of a DI component originates from Molnar [22]. Udding was the first to give a rigorous formulation of the DI property in terms of trace structures [34]. More information on SI decomposition, DI decomposition, and DI components, can be found in [12].

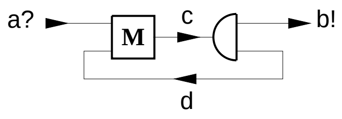
3.8 Limitations of Decomposition

Our correctness conditions for decomposition have certain limitations, in particular the progress condition. The safety condition guarantees that whenever an output can be produced, the production causes no problems. It does not guarantee that every output *will* be produced. Therefore we need a progress condition. Our progress condition excludes, for example, decompositions into so-called 'accept-everything-do-nothing' modules: components that accept every possible input but never produce any output. On the other hand, although the progress condition requires that each trace in $\mathbf{t}E$ *may* occur in the network behaviors, in general it does not require that some traces, or outputs, are guaranteed to occur. The occurrence of a trace, or output, may depend on nondeterministic choices made by some components. Figure 6 illustrates this property by means of two decompositions of a WIRE. The component depicted by the left half of a circle is a SELECTOR. Its specification is given by

$$\mathbf{pref} * [\ a?;\ (b!|c!)\ ]$$



(a) Deadlock



(b) Livelock

Figure 6: Examples of deadlock and livelock

After each receipt of input a, either output b or output c may be produced. The output is chosen nondeterministically. The component depicted by an open square is a SINK. It only accepts inputs. Decompositions (a) and (b) satisfy all four conditions for decomposition. Decomposition (a), however, exhibits a behavior that could be characterized by deadlock: once output c is chosen by the SELECTOR, output b will never be produced. Decomposition (b) exhibits a behavior that could be characterized as livelock: if the SELECTOR makes enough 'bad' choices, there can be an unbounded number of internal communications on d and c, and output b may not be produced. Obviously, if we design an implementation, we do not want deadlock or livelock to occur. So, in formulating a better progress condition we want to exclude implementations with deadlock and livelock. This is easier said than done. Do there exist 'proper' definitions for deadlock and livelock (or of progress) in the context of our formalism? If so, what are they? If not, how much should our formalism be extended to accommodate proper definitions? For example, an obvious definition for livelock might be that in a trace an unbounded number of internal symbols can occur between two external symbols. With this definition in mind consider the decomposition of a 'one-shot' WIRE, **pref**$(a?;\ b!)$, in Figure 7. If we look to the traces of this decomposition, then

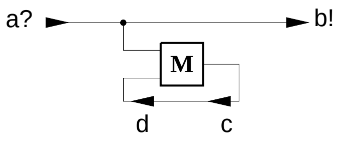


Figure 7: Livelock?

after input a an unbounded number of internal symbols c and d can occur before output b is produced. According to our tentative definition, we should conclude that livelock can occur. On the other hand, from experience we know that output b will eventually occur, if the WIREs are implemented by physical wires. So in our context, this tentative definition may not be a 'proper' definition of livelock. Finding definitions of deadlock and livelock (or of progress) that are simple enough to state, have a proper justification in the context of speed-independent circuit design, are easy to verify, and satisfy a property like the Substitution Theorem is still an open problem.

4 How to Play the Game

Knowing what the rules of the game are, there can be many ways of playing the game. For instance, we can simply apply a trial-and-error process: propose a decomposition and verify all conditions. If the conditions are not satisfied, we try another decomposition. Such a style is rather unsatisfactory, since it does not give much guidance in how one could derive a decomposition in a systematic way.

We explore a different way. It is based on a few simple principles. We first try to obtain a parallel decomposition of our specification E. For example, we try to express E as

$$E = |[\ \mathbf{chan}\, V\ ::\ \ F \parallel G\ \ ]|$$

by using techniques from parallel program design. The obvious candidate for a decomposition then is

$$E \to (\ F\ ,\ G\ )$$

The conditions for decomposition can be verified easily by inspecting the syntax of E, F, and G. For example, the conditions for a closed network and absence of output interference are easily verified by checking the alphabets. Absence of computation interference can be verified by checking that as soon as an output can be produced by a component locally, then the receiving components are waiting for that output to happen. With a little care in specifying the components F and G, this condition can often be satisfied. Finally, we mention without proof that the progress condition (that is, completeness of the network behaviors) is automatically satisfied if we already have $E = |[\ \mathbf{chan}\, V\ ::\ \ F \parallel G\ \ ]|$.

5 The Modulo-N Counter Continued

5.1 A First Decomposition: Divide and Conquer

How can we decompose the modulo-N counter into smaller components? An obvious choice is to try a divide-and-conquer approach: decompose the modulo-$2N$ counter into a modulo-N counter and a 'small' subcomponent. 'Small' can be interpreted as the specification has a small number of states, which is independent of N. We derive two such decompositions. The first decomposition is based on the following idea. Keep track of two counts represented by n and k, where n is incremented modulo N each time an input r occurs and k is incremented modulo 2 each time n reaches 0. Initially, both n and k are 0. While incrementing n and k, we maintain the invariant

$$\#r \bmod 2N = kN + n$$

In other words, $\#r \bmod 2N = 0$ is equivalent to $n = 0$ and $k = 0$. This observation leads to the following first derivation step. For $2N > 0$, we have

```
    $ModC(2N, r?, a!)$
=       { $\#r \bmod 2N = kN + n$ }
    |[ var $n : int, k : bin$  ::
       initially $n = 0, k = 0$  ::
       pref $*$ [ $r?;\ n := (n+1) \bmod N$
                ; if $n = 0$  then $k := (k+1) \bmod 2$  |  $n \neq 0$  then $skip$ fi
                ; if $n \neq 0 \ \vee\ k \neq 0$   then $a := 1$
                  |  $n = 0 \ \wedge\ k = 0$   then $a := 0$
                  fi ; $a!$ ]
    ]|
```

Incrementing n modulo N and indicating whether $n \neq 0$ or $n = 0$ can be done by a modulo-N counter. Consequently, in our second step we introduce a modulo-N counter and replace each statement $n := (n + 1) \bmod N$ by the communication actions $sr!;\ sa?$. Since the postcondition for each communication on sa is given by

$$(n = 0) \ \equiv\ (sa = 0)$$

where $n = \#sr \bmod N$, we can replace the conditions $n \neq 0$ and $n = 0$ by $sa \neq 0$ and $sa = 0$ respectively. These observations then lead to the following parallel composition.

```
    $ModC(2N, r?, a!)$
=       { def. of $ModC(N, sr?, sa!)$ }
    |[ chan $sr : un,\ sa : bin$  ::
       $CELL0(r?,\ a!,\ sr!,\ sa?)$ || $ModC(N, sr?, sa!)$
    ]|
```

where $CELL0$ is defined by

```
    $CELL0(r? : un,\ a! : bin,\ sr! : un,\ sa? : bin)$
$=$     { by definition }
    $|[$ **var** $k : bin$ ::
      **initially** $sa = 0,\ k = 0$ ::
      **pref** $*\,[\ r?;\ sr!;\ sa?$
               ; **if** $sa = 0$ **then** $k := (k+1) \bmod 2 \ \mid\ sa \neq 0$ **then** $skip$ **fi**
               ; **if** $sa \neq 0 \lor k \neq 0$   **then** $a := 1$
                 $\mid$  $sa = 0 \land k = 0$   **then** $a := 0$
                 **fi** ; $a!\ ]$
    $]|$
```

The parallel composition above is a candidate for a network implementation of the modulo-$2N$ counter. We have to check if the conditions for decomposition are satisfied. First, we observe that the network of $CELL0$, the modulo-N counter, and the environment is closed and has no output interference. Second, we observe that as soon as a component or the environment can perform an output action, the corresponding input action is also enabled: no computation interference can occur in the communications between $CELL0$, the modulo-N counter, and the environment. We may therefore conclude the following decomposition

```
    $ModC(2N, r?, a!)$
$\rightarrow$     { def. of decomposition }
    $(\ CELL0(r?, a!, sr!, sa?)\ ,\ \ ModC(N, sr?, sa!)\ )$
```

We briefly present two implementations for $CELL0$. We observe that the guarded command for $CELL0$ can also be written as

```
    **pref**$*[\ r?;\ sr!;\ (\ sa?1;\ a!1$
                  $\mid sa?0;\ k := (k+1) \bmod 2;$
                        **if** $k = 1$ **then** $a!1 \ \mid\ k = 0$ **then** $a!0$ **fi** $)$
          $]$
```

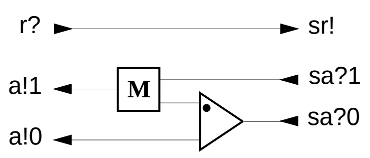
Recalling that counting modulo-2 can be done by a TOGGLE, it is now not difficult to verify that the network given in Figure 8(a) represents an implementation of $CELL0$ using a dual-rail transition encoding. An implementation using single-rail data bundling is given in Figure 8(b).

5.2 A Second Decomposition

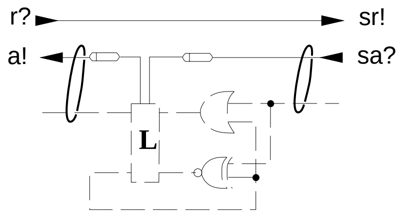
In the first step of our previous decomposition, we can choose a different alternative by interchanging the increments to n and k. That is, we increment k modulo-2 after each input r and increment n modulo N each time k reaches 0. Now the invariant is given by

$$\#r \bmod 2N = 2n + k$$

Based on this observation we obtain the following first derivation step.



(a) Using dual-rail transition encoding



(b) Using single-rail data bundling

Figure 8: Implementations of *CELL*0

```
   $ModC(2N, r?, a!)$
=     { $\#r \bmod 2N = 2n + k$ }
   |[ var $n : int, k : bin$ ::
     initially $n = 0, k = 0$ ::
     pref $*$ [ $r?;\ k := (k+1) \bmod 2$
              ; if $k = 0$ then $n := (n+1) \bmod N$ | $k \neq 0$ then $skip$ fi
              ; if $n \neq 0 \vee k \neq 0$ then $a := 1$
               | $n = 0 \wedge k = 0$ then $a := 0$
               fi ; $a!$ ]
   ]|
```

The next steps are now similar to the previous section. We introduce a modulo-N counter and replace the statement $n := (n+1) \bmod N$ by $sr!;\ sa?$. The conditions $n \neq 0$ and $n = 0$ are replaced by $sa \neq 0$ and $sa = 0$ respectively. If we define *CELL*1 as follows

```
   $CELL1(r? : un,\ a! : bin,\ sr! : un,\ sa? : bin)$
=     { by definition }
   |[ var $k : bin$ ::
    initially $sa = 0,\ k = 0$ ::
    pref $*$ [ $r?;\ k := (k+1) \bmod 2$
              ; if $k = 0$ then $sr!;\ sa?$ | $k \neq 0$ then $skip$ fi
              ; if $sa \neq 0 \vee k \neq 0$ then $a := 1$
               | $sa = 0 \wedge k = 0$ then $a := 0$
               fi ; $a!$ ]
   ]|
```

we obtain the parallel composition and subsequent network decomposition

```
   $ModC(2N, r?, a!)$
=     { def. of weave }
```

$$|[\ \textbf{chan}\ sr : un, sa : bin \ :: \ CELL1(r?, a!, sr!, sa?) \ \| \ ModC(N, sr?, sa!)\ ]|$$

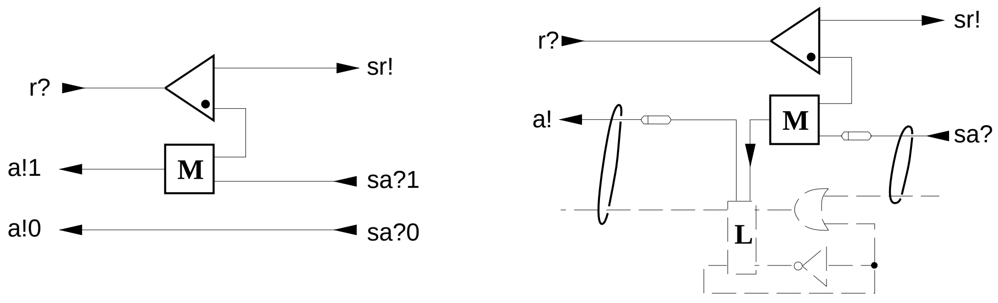
$$\rightarrow \quad \{ \text{ def. of decomposition } \}$$

$$(\ CELL1(r?, a!, sr!, sa?)\ ,\ ModC(N, sr?, sa!)\ )$$

Let us briefly look at some implementations for $CELL1$. First, we observe that $CELL1$, after some simplification, can also be written as

$$\textbf{pref} * [\ r?;\ a!1;\ r?;\ sr!;\ (sa?0;\ a!0 \mid sa?1;\ a!1)\ ]$$

Apparently, we again need a modulo-2 counter to record whether the input $r?$ has to propagate to output $a!1$ or to output $sr!$. This observation then quickly leads to the implementations in Figure 9.



(a) Using dual-rail transition encoding (b) Using single-rail data bundling

Figure 9: Implementations for $CELL1$

5.3 What about odd N?

So far we have presented two decompositions for even N. What about odd N? Although there are some decompositions that apply to odd N only, we present a decomposition that applies to any $N > 0$. We try to decompose a modulo-$(N + 1)$ counter into a modulo-N counter and a small subcomponent. Again we keep track of two integers n and k, where we maintain the invariant

$$\#r \bmod (N + 1) = n + k$$

Initially $k = 0$ and $n = 0$. If $k = 0$ and an r is received, then k is set to 1. If $k = 1$, then for each r received, n is incremented by one modulo N. When n becomes 0, k is also set to zero. This leads to the following derivation step.

$$ModC(N + 1, r?, a!)$$

$$= \quad \{\ \#r \bmod (N + 1) = n + k\ \}$$

```
$|[$ var $n : int, k : bin$ ::
  initially $n = 0, k = 0$ ::
  pref $*[\, r?;$ if $k = 1$ then $n := (n+1) \bmod N \mid k = 0$ then $skip$ fi
           ; if $k = 0 \lor n \neq 0$ then $k := 1;\ a := 1$
            $\mid$ $k = 1 \land n = 0$ then $k := 0;\ a := 0$
            fi ; $a!\,]$
$]|$
```

Similar to the previous decompositions we introduce a modulo-N counter for incrementing n modulo N and indicating whether $n \neq 0$ or $n = 0$. We then obtain the following parallel composition and network decomposition.

```
   $ModC(N+1, r?, a!)$
=      { intro. of mod-$N$ counter, def. of parallel composition }
   $|[$ chan $sr : un,\ sa : bin$ :: $CELL2(r?, a!, sr!, sa?) \parallel ModC(N, sr?, sa!)\, ]|$
$\rightarrow$      { def. of decomposition }
   $(\ CELL2(r?, a!, sr!, sa?)\ ,\ ModC(N, sr?, sa!)\ )$
```

where *CELL2* is defined by

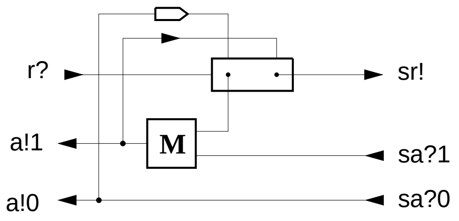
```
   $CELL2(r? : un,\ a! : bin,\ sr! : un,\ sa? : bin)$
=      { by definition }
   $|[$ var $k : bin$ ::
     initially $sa = 0,\ k = 0$ ::
     pref $*[\, r?;$ if $k = 1$ then $sr!;\ sa? \mid k = 0$ then $skip$ fi
              ; if $k = 0 \lor sa \neq 0$ then $k := 1;\ a := 1$
               $\mid$ $k = 1 \land sa = 0$ then $k := 0;\ a := 0$
               fi ; $a!\,]$
   $]|$
```

We give two implementations for *CELL2* without proof. In the case that binary channels are implemented using a dual-rail transition encoding, *CELL2* can be implemented as given in Figure 10(a). (Notice that the 2-by-1 JOIN is used for implementing the binary variable k. An IWIRE is used for the proper initialization.) In the case that single-rail data bundling is used for implementing binary channels, *CELL2* can be implemented as given in Figure 10(b).

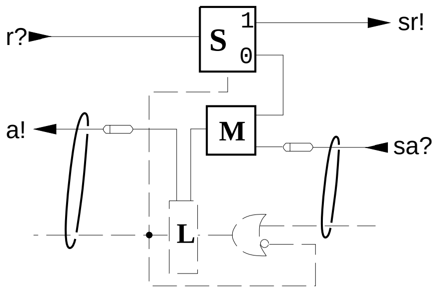
5.4 What about Parallelism?

All our decompositions thus far have a sequential behavior in the sense that all communication actions (and even internal actions) are totally ordered. As such, our programs are not much different from a normal sequential program with procedure calls. How can we derive decompositions that exhibit parallel behavior? For example, would it be possible to introduce some parallel behavior in our sequential decompositions without invalidating their correctness? There are several ways to do this. We discuss two.

Observe our specification of the modulo-N counter once more



(a) Using dual-rail transition encoding



(b) Using single-rail data bundling

Figure 10: Implementations for *CELL2*

```
    $ModC(N : int,\ r? : un,\ a! : bin\ )$
=       { by definition }
    $|[$ var $n : int$ ::
      initially $n = 0$ ::
      pref $*[\ r?;\ n := (n+1) \bmod N;$
              if $n \neq 0$ then $a := 1$ | $n = 0$ then $a := 0$ fi ; $a!$
              ]
    $]|$
```

Notice that the value to be sent on channel a can be computed ahead of time, possibly before r arrives. Only when r arrives the value of a is sent along channel a, and, in parallel, the computation of the next value of a is initiated. In order to compute the value of a before r arrives, we can put component P in front of $ModC(N,\ sr?,\ sa!)$.

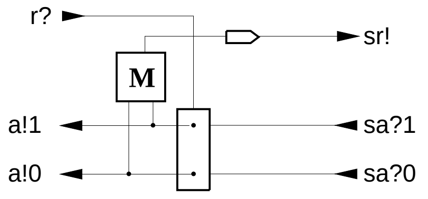
```
    $P(r? : un,\ a! : bin,\ sr! : un,\ sa? : bin)$
=       { by definition }
    pref( $r? \,||\, (sr!;\ sa?);\ *[\ a := sa;\ ((a!; r?) \,||\, (sr!; sa?))\ ]$)
```

Initially, in parallel with receiving input r, the first value of a modulo-N counter is requested and received via channels sr and sa respectively. After both input r and the first value on sa have been received, the value of sa can be sent on channel a. In parallel with sending a and receiving the next request r, the next value from the modulo-N counter is requested and received on channels sr and sa, respectively. This behavior then repeats. It is not difficult to see that the modulo-N counter can be decomposed into a renaming of itself and component P.

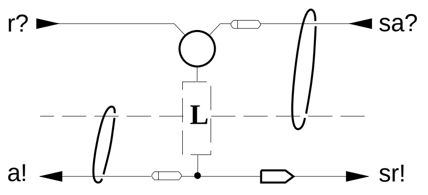
```
    $ModC(N, r?, a!)$
→       { def. of decomposition }
    $(\ P(r?, a!, sr!, sa?)\ ,\ \ ModC(N, sr?, sa!)\ )$
```

Since component P can be put in front of every modulo-N counter, it can also be put in front of every *CELL* without changing the correctness of the decomposition. In this way we can obtain decompositions where many communications may take place in parallel. Notice, however, that the communication behavior on the pair of channels between any two adjacent cells remains invariant when we insert component P between the cells: after the insertion, actions from different pairs of channels can occur in parallel, but actions from the same pair still occur in the same sequential order and the same values are communicated.

If we decide to implement binary channels by means of dual-rail transition encoding, then component P can be implemented by a 2-by-1 JOIN, a MERGE, and an IWIRE. See Figure 11(a). If we decide to implement binary channels by means of single-rail data bundling, then component P can be implemented by a transition latch, a JOIN, and an IWIRE. See Figure 11(b).



(a) Using dual-rail transition encoding



(b) Using single-rail data bundling

Figure 11: Implementations for component P

Another way to introduce parallelism is to consider the specification of each *CELL* in isolation and to try changing or reordering the statements in the guarded command such that communications on the channels r and a can be done in parallel with the communications on channels sr and sa. In doing so we should not destroy the correctness of the decomposition. In other words, if *PCELL* is the specification obtained after changing *CELL*, then *PCELL* should still satisfy the decompositions in which *CELL* is used. In order not to destroy the correctness of the decomposition, we keep the communication behavior on the channels r and a invariant and, for reasons of symmetry, also the communication behavior on the channels sr and sa. Communications from different pairs of channels, however, may overlap.

Based on these observations we try to specify the communication behavior of *PCELLi* as follows.

```
  $PCELLi(r? : un,\ a! : bin,\ sr! : un,\ sa? : bin)$
$=$    { by definition }
  $|[$ **var** $k : bin$ ::
    **initially** $sa = 0,\ k = 0$ ::
    **pref** ( **if** $B.i$ **then** $r?$ $\|$ $(sr!;\ sa?)$ | $\neg\ B.i$ **then** $r?$ **fi** ; $S.i$;
        $*[$ **if** $B.i$ **then** $(a!;\ r?)$ $\|$ $(sr!;\ sa?)$ | $\neg\ B.i$ **then** $(a!;\ r?)$ **fi** ; $S.i$ $]$)
  $]|$
```

where $S.i$ represents the statement that calculates the next value of a (possibly using some local variables) and $B.i$ is a guard. Notice that in each repetition step there is one pair of communications with the environment through a and r, and possibly one pair of communications with the subcomponent through sr and sa. The occurrence of a communication with the subcomponent depends on the value of $B.i$. We try to find appropriate values for $B.i$ and $S.i$ by changing the specification of *CELLi*. The specification for *PCELLi* should still satisfy the decomposition for the modulo-N counter in which *CELLi* was used.

As an example we consider *CELL*1 once more.

```
   $CELL1(r? : un,\ a! : bin,\ sr! : un,\ sa? : bin)$
=      { by definition }
   |[ var $k : bin$ ::
    initially $sa = 0,\ k = 0$ ::
    pref * [ $r?;\ k := (k+1) \bmod 2$
             ; if $k = 0$ then $sr!;\ sa?$ | $k \neq 0$ then $skip$ fi
             ; if $sa \neq 0 \lor k \neq 0$  then $a := 1$
              | $sa = 0 \land k = 0$  then $a := 0$
              fi ; $a!$ ]
   ]|
```

We first switch $k := (k+1) \bmod 2$ with the succeeding **if** ...**fi** , thereby also changing the guards of the selection.

```
   pref * [ $r?$; if $k \neq 0$ then $sr!;\ sa?$ | $k = 0$ then $skip$ fi
            ; $k := (k+1) \bmod 2$
            ; if $sa \neq 0 \lor k \neq 0$  then $a := 1$
             | $sa = 0 \land k = 0$  then $a := 0$
             fi ; $a!$ ]
```

After some unfolding and reordering we get a specification of the desired form, where

```
   $B.1 = (k \neq 0)$
   $S.1 =$ ($k := (k+1) \bmod 2$;
         if $sa \neq 0 \lor k \neq 0$  then $a := 1$
         | $sa = 0 \land k = 0$  then $a := 0$
         fi )
```

The other cells only require some unfolding and reordering of $a!$; $r?$ to obtain the desired form. We just give the outcome of this exercise here. For *PCELL*0 we get

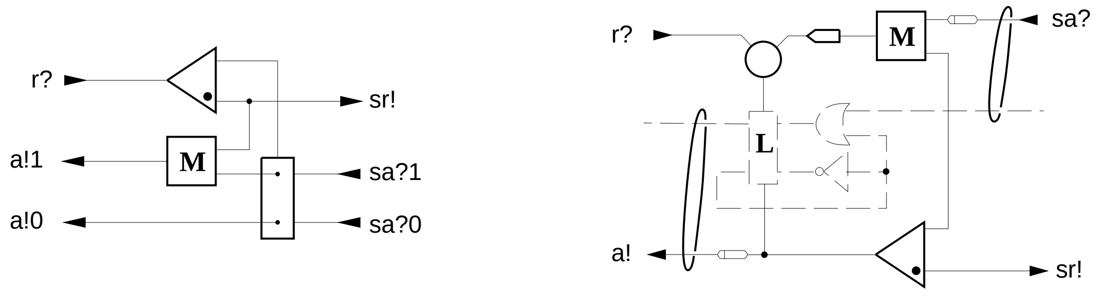
```
   $B.0 = true$
   $S.0 =$ if $sa = 0$ then $k := (k+1) \bmod 2$ | $sa \neq 0$ then $skip$ fi
         ; if $sa \neq 0 \lor k \neq 0$  then $a := 1$
          | $sa = 0 \land k = 0$  then $a := 0$
          fi ;
```

For *PCELL*2 we get

$$B.2 = (k = 1)$$

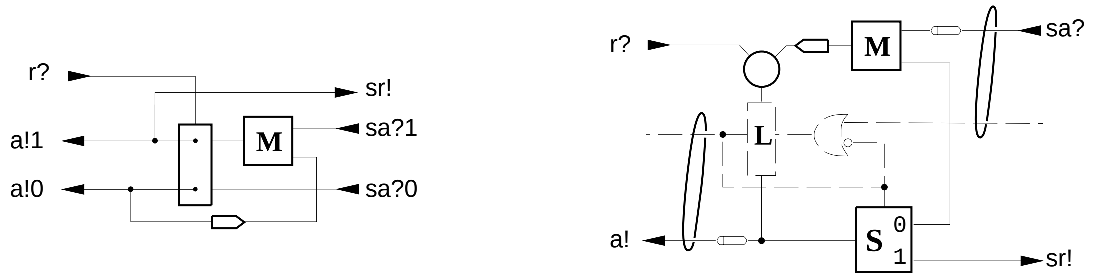
$$\begin{aligned} S.2 = (\ & \textbf{if } k = 0 \vee sa \neq 0 \textbf{ then } k := 1;\ a := 1 \\ |\ & k = 1 \wedge sa = 0 \textbf{ then } k := 0;\ a := 0 \\ & \textbf{fi}\ ) \end{aligned}$$

For each of the cells *PCELLi*, $0 \leq i \leq 2$ we can try to find an implementation using a dual-rail transition encoding for the binary channels. It turns out that *PCELLi* can be decomposed into P and *CELLi*, for $0 \leq i \leq 2$. In other words, we can put the implementation of cell P in front of the implementation for *CELLi*, and we get an implementation for *PCELLi*. These are not the only implementations for *PCELLi*, however. For example, for *PCELL*1 and *PCELL*2 some smaller implementations can be obtained. See Figures 12 and 13. The verification of these implementations is left to the reader. Notice that the combinational logic in Figures 12(b)



(a) Using dual-rail transition encoding (b) Using single-rail data bundling

Figure 12: Implementations for *PCELL*1



(a) Using dual-rail transition encoding (b) Using single-rail data bundling

Figure 13: Implementations for *PCELL*2

and 13(b) are straightforward implementations of the statements $S.1$ and $S.2$. Furthermore, the implementations for the control part correspond to the network given in Figure 5(b). The condition $B.i$ is the input to the SELECT that controls the communications with the subcounter. Notice also that if the data input to the SELECT alternates in value, as is the case with $S.1$, then the SELECT boils down to a TOGGLE.

6 Performance Analysis

By now we have many ways to decompose a modulo-N counter. How do we compare these decompositions? Do there exist some measures for our parallel programs which are similar to the running time or space complexity of an algorithm? Are 'time' and 'space' the only interesting performance aspects? How do we measure each performance aspect?

We consider three performance criteria: area complexity, power consumption, and response time. For each performance criterion we give a performance measure. Our estimates for each of these performance measures satisfy four properties. First, for all measures we only look to an order-of-growth estimate; no estimates are given in terms of square millimeters, microwatts, or nanoseconds. Second, our estimates are based on the program texts: they do not rely on a specific implementation of the basic cells. Third, certain conditions apply in order for our estimates to be accurate first-order approximations. Some of these conditions are discussed later. Fourth, calculating these estimates can be done on the back of an envelope. We emphasize once more that our performance estimates are just first-order approximations. We should always be aware that the hidden constants and second order terms can vary heavily among implementations. A more detailed performance estimate will require further knowledge of the particular implementation.

As usual in the analysis of algorithms, we use the notations $\mathcal{O}$, Ω, and Θ to denote an upper bound, lower bound, and tight bound for the order of growth of a function.

6.1 Area Complexity

Our first performance criterion is area complexity. The area complexity is a rough estimate for the area occupied by a physical implementation such as an integrated circuit. As a measure for area complexity we take the number of 'basic' components in the decomposition. Here, a basic component can be any component as long as the number of states is bounded by a predetermined constant. (Notice that for a given constant there can only be a bounded number of basic components.) For regular implementations consisting of a linear array of basic components, like our modulo-N counter implementations, our measure gives a good first-order approximation. The accuracy of the estimate may change, however, when the connections among the basic components become more complex.

Before we study the area complexity of some decompositions, let us make a list of the decompositions we have so far. Let $ModC(N)$ and $sModC(N)$ be abbreviations for $ModC(N,\ r?,\ a!)$ and $ModC(N,\ sr?,\ sa!)$ respectively. The cells have the usual input and output channels. We take $N > 1$.

$$\begin{aligned} ModC(2N) &\rightarrow (\ CELL0\ ,\ ModC(N)\ ) && (0)\\ ModC(2N) &\rightarrow (\ CELL1\ ,\ sModC(N)\ ) && (1)\\ ModC(N+1) &\rightarrow (\ CELL2\ ,\ sModC(N)\ ) && (2) \end{aligned}$$

$$\begin{aligned} ModC(2N) &\to (\ PCELL0\ ,\ sModC(N)\ ) \quad (3)\\ ModC(2N) &\to (\ PCELL1\ ,\ sModC(N)\ ) \quad (4)\\ ModC(N+1) &\to (\ PCELL2\ ,\ sModC(N)\ ) \quad (5)\\ ModC(N) &\to (\ P\ ,\ sModC(N)\ ) \quad (6)\end{aligned}$$

First, we remark that the number of states of each cell is independent of N and therefore bounded. Accordingly, we can consider these cells as 'basic' components in our area complexity analysis. Second, we remark that the decompositions (3)-(5) are similar to decompositions (0)-(2), as far as area complexity is concerned. Finally, we observe that it does not make sense to use decomposition step (6) to obtain an efficient area complexity. For these reasons, we only concentrate on the first three decompositions.

With decomposition step (2) we can decompose any modulo-N counter for $N > 1$ into basic components $CELL2$ and a modulo-2 counter. The area complexity of such a decomposition is linear in N, that is $\Theta(N)$. If we use decomposition step (0) or (1) when N is even and decomposition step (2) when N is odd, we obtain a decomposition into basic components $CELL0$, $CELL1$, $CELL2$, and a modulo-2 counter with a logarithmic area complexity, that is, the area complexity is $\Theta(\log N)$. (Notice that at least every other decomposition step is of the form (0) or (1), which gives the logarithmic complexity.)

Is there a decomposition into basic components that has an area complexity that grows less than logarithmic in N? In fact, what is the lower bound of the area complexity of the modulo-N counter taken over all decompositions into basic components? It turns out that we cannot do any better than logarithmic in N. Here is an argument why. Consider a network with k basic components, and each component has at most q states for a given constant q. This network can implement a specification with at most $O(q^k)$ states. The modulo-N counter has $\Theta(N)$ states. Consequently, any decomposition into basic components of the modulo-N counter has at least $\Omega(\log N)$ basic components. A similar reasoning can be applied to any specification to obtain a lower bound for the area complexity.

6.2 Power Consumption

Our second performance measure is power consumption. In physical terms, the power consumption of an integrated circuit is the energy dissipated per time unit. Since in our abstract approach there is no time metric, we consider the energy dissipated per external action. As a measure for the energy we take the total number of communication actions in a behavior. Furthermore, in this note we are not interested in incidental peaks in the power consumption: we only consider the power consumption over the long term. That is, we amortize all communication actions over the external actions. Finally, since the power consumption may depend on what external actions are performed by the environment, we assume a worst-case environment for our power consumption analysis. A worst-case environment is an environment that communicates with the implementation in such a way that the total number of communication actions is maximized over the long term. For these reasons, we take as a measure for the power consumption the total number of communication actions amortized over the external communication actions for a worst-case environment. In order for this measure to be a good first-order approximation a number of conditions must be satisfied. Some of these conditions are discussed below.

The power consumption of a circuit is determined by the dynamic and static power consumption. The dynamic power consumption is dominated by the charging and discharging of capacitances. The static power consumption in CMOS circuits is due to leakage current. Our measure, which is based on counting communication actions, is intended to be an estimate of the dynamic power consumption. In order for this 'communication count' to be a good first-order approximation for the power consumption, the static power consumption should be negligible to the dynamic power consumption. This condition can be met in a CMOS implementation, if the frequency at which the communication actions occur is high enough. If the frequency becomes too low, static power consumption is no longer negligible [35].

The second condition is that all voltage transitions in the implementation require about the same amount of energy. The amount of energy needed for a transition depends on the load capacitance. Load capacitances can vary orders of magnitudes in an integrated circuit. For example, the load capacitance of a long wire with a large fan-out is much higher than the load capacitance of a short wire with a fan-out of only one. In particular, in an implementation with an irregular structure and large differences in capacitances, our estimate becomes inaccurate.

The third condition is that the power consumption of (CMOS implementations of) our basic components is proportional to the number of external communication actions performed on it. This assumption requires, for example, that the implementations of our basic components do not exhibit any livelock or metastable behavior.

If one of the conditions above is not satisfied, then the power consumption can only be higher. In this respect, our measure will always give a lower bound for the order of growth of the power consumption.

Let us return to defining the power consumption of a decomposition in terms of the 'communication count.' For a decomposition into basic components, the *power consumption of a behavior* t is defined as the number of communication actions in t amortized over the number of external communication actions in t, where t is 'long enough'. We take the amortized power consumption of a behavior in order to spread out evenly the cost of all communication actions over the external communication actions. For example, initialization effects can be spread out in this way over many external communication actions. If we defined the power consumption of a behavior simply as the number of all communication actions divided by the external communication actions, we could get extremely high power consumptions during initialization or could even get division by zero. The *power consumption of the decomposition* is defined as the maximum of the power consumptions over all behaviors of the decomposition. We say that the decomposition has bounded power consumption if its power consumption is bounded from above by a constant. These definitions have been inspired by [2].

With these definitions we can calculate the power consumption for various decompositions. As a first example, we take the decomposition where only decomposition step (2) (i.e., *CELL*2) is used. Here are some behaviors of that decomposition

$$r_0 a_0,\ r_0 r_1 a_1 a_0,\ r_0 r_1 r_2 a_2 a_1 a_0, \ldots$$

where r_0 represents a communication on channel r between the environment and cell 0, r_1 represents a communication on channel r between cell 0 and cell 1, etc. A similar meaning applies to a_i. See Figure 14. For the decomposition of a modulo-N counter, there are $N-2$ components *CELL*2 and one modulo-2 counter as end cell. The first external communication on

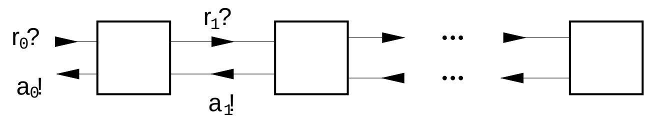


Figure 14: A linear array of cells

channel r propagates to cell 0 and then returns via channel a, the second external communication propagates to cell 1 and then returns via the a channels, and so on. The $N-1$st and N-th external communication propagate to the end cell and then return via the a channels. These behaviors then repeat. So for every part of a behavior where $2N$ external communication actions (N on channel r and N on channel a) occur, there will be a total of

$$(\sum_{i=1}^{N-2} 2i) + 4(N-1) = (N-1)(N+2)$$

external and internal communication actions. So the power consumption amortized over the external communication actions for this part of the behavior is $\Theta(N)$. Since every behavior consists of a repetition of these part behaviors, every behavior has a power consumption of $\Theta(N)$. Consequently, the complete decomposition also has a power consumption of $\Theta(N)$.

Let us calculate the power consumption of a decomposition of the modulo-N counter where only decomposition step (0) is used. In such a decomposition each external action on the r channel propagates through the entire array of $\Theta(\log N)$ cells to the end cell and then back on the a channels. Consequently, every behavior has a power consumption of $\Theta(\log N)$, and therefore this decomposition also has a power consumption of $\Theta(\log N)$.

What would the power consumption be of a decomposition where only step (1) is used? Notice that in this decomposition each cell propagates every other input r further in the linear array of cells. So if in a communication behavior $2k$ external communication actions take place, at most $2k/2$ communication actions take place between the first and second cell, at most $2k/4$ communication actions take place between the second and third cell, and so on. Consequently, each communication behavior with $2k$ external communication actions has a power consumption of at most

$$(\sum_{i=0}^{\infty} 2k/2^i)/2k \leq 2$$

From this observation we may conclude that every behavior has a bounded power consumption, and therefore the decomposition also has a bounded power consumption.

What would be the power consumption of a decomposition using step (1) for even N and step (2) for odd N? It turns out that such a decomposition also has bounded power consumption. The argument why this is so is only a slight elaboration of the argument used in the previous analysis. First we observe that for *CELL*2 after any behavior the number of communication actions on channels sr and sa is at most the number of communication actions on channels r and a. Second, we observe that at most every other cell in the decomposition is of type *CELL*2.

This means that the total number of communication actions in the complete decomposition is at most double that of a decomposition where only $CELL1$ is used. Since a decomposition where only $CELL1$ is used has a bounded power consumption, a decomposition where $CELL1$ is used for even N and $CELL2$ for odd N also has a bounded power consumption.

Can we improve any of these bounds if we use any of the steps (3)-(5) instead of (0)-(2) respectively? It turns out that none of these bounds improves. Notice that by using any of the components $PCELLi$ instead of $CELLi$, $0 \leq i \leq 3$, many actions can take place in parallel, but the communication behavior between any two neighboring cells remains almost invariant for a given external behavior. The only difference is that some extra internal communications take place for computing some responses ahead of time. The number of extra internal actions is at most $O(L)$, where L is the length of the array. Since we only consider long-term behaviors where the number of external actions $k \gg L$, these extra communication actions are negligible in calculating the power consumption. For these reasons the results of the analyses of the 'sequential' cells are still valid when using the 'parallel' cells.

6.3 Response Time

Our third performance measure is the response time of a decomposition. The response time is the delay between the receipt of an input and the production of the succeeding output. The response time is always measured from the time the *last* input arrives that enables the production of that output to the actual production of that output. We are particularly interested in the worst-case response time.

In calculating the response time of a decomposition, we assume that the response times of our basic components are bounded from above and below by fixed constants. Consequently, all implementations of the basic components may not exhibit livelock, deadlock, or metastable behavior, since then there is no guaranteed upper bound. We do not require that the response times of basic components are constant. Response times may vary arbitrarily between lower and upper bound. For example, response times may depend on the actual values of the data received, like addition may depend on the actual values added. Response times may also vary over different instances of the same basic component or may vary over time. These delay assumptions are more general than the assumptions made in [27], for example, and may sometimes lead to results that are too pessimistic. A more detailed performance analysis and optimization technique, which is based on Martin's design approach, is given by Burns in [5]. Techniques for analyzing the throughput and latency of micropipelines are proposed in [30, 36].

Let us see if we can calculate the response times of the various decompositions for the modulo-N counter. If we only consider the steps (0)-(2), then the decompositions do not exhibit any parallel behavior. Let us first consider a decomposition that uses only step (2). In the worst case an input r propagates through all $N-2$ cells to the end cell and then back. Consequently, this decomposition has a response time that grows linearly with N. What is the response time of a decomposition using only step (0)? In such a decomposition every input propagates through all $\Theta(\log N)$ cells and then back. Consequently, the decomposition has a response time of $\Theta(\log N)$. A decomposition using only step (1) also has a response time of $\Theta(\log N)$, since in the worst case an input r propagates through all $\Theta(\log N)$ cells and then back.

Response time analysis is perhaps the most difficult analysis to perform, in particular when there is a high degree of parallelism. In order to facilitate the analyses of decompositions using steps (3)-(6), we present some theorems on response times for linear arrays of cells that exhibit parallel behavior.

We consider a linear array of cells as given in Figure 15. We assume that each cell in the

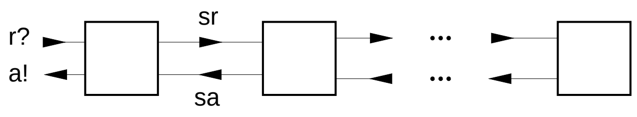


Figure 15: A linear array of cells

linear array has a communication behavior given by

$$\textbf{pref}(\mathit{INIT};\ *[\ \textbf{if}\ B\ \ \textbf{then}\ (a!;\ r?)\ \|\ (sr!;\ sa?)\ \mid\ \mathrm{not}(B)\ \ \textbf{then}\ (a!;\ r?)\ \textbf{fi}\ ;\ S\ ]) \qquad (7)$$

where S represents the computation of the values to be communicated on a and sr, and INIT is some initial behavior that can be represented by a proper 'postfix' of the behavior in the repetition. For example, $\mathit{INIT} = r?;\ S$ or $\mathit{INIT} = (r?\ \|\ (sr!;\ sa?));\ S$. The guard B depends on the values that are communicated and possibly the values of the local variables. Since we are only interested in when which communication action can occur, we have abstracted as much as possible from the computation of the values that are communicated. There are two special cases for the guards in which we are interested. One case is where $B = \mathit{true}$ in each repetition step; so in every repetition step there is a pair of communications on a and r and on sr and sa. The other case is where at most every other repetition step $B = \mathit{true}$. More specifically, if B_i represents the value of guard B in repetition step $i, i \geq 0$, then we have

$$B_i \Rightarrow \mathrm{not}(B_{i+1}) \quad \text{for all } i \geq 0$$

In other words, if in a certain repetition step there is a pair of communications on sr and sa, then in the next repetition step there are no communications on sr and sa.

Each cell has two outputs: a and sr. For each of these outputs, each of the inputs can be the last input to arrive that enables the production of that output. Accordingly, the response times for a cell are denoted by

$$\tau(r?;a!),\ \tau(sa?;a!),\ \tau(sa?;sr!),\ \tau(r?;sr!)$$

The response time $\tau(r?;a!)$ is the time it takes to produce output $a!$ after input $r?$ has been received, where we assume that input $r?$ is the last input that enables output $a!$. The other response times are described similarly. The response times of the cells may vary. For example, they may vary over different instances of a cell, over time, or they may depend on the state of a cell. We assume, however, that all response times τ of each cell have upper bound Δ and lower bound δ, that is

$$\delta \leq \tau \leq \Delta$$

The end cell has a behavior given by

$$\textbf{pref} * [\ r?;\ a!\ ]$$

The response time of this end cell is given by $\tau_L(r?; a!)$. We also assume that the response time of the end cell has lower bound d and upper bound D, that is

$$d \leq \tau_L \leq D$$

Finally we let R stand for the response time of the decomposition. Since we have chosen to calculate the worst-case response time, R is the maximum of all $R.i$, $i \geq 0$, where $R.i$ is the maximum delay between occurrence i of input r and output a in the decomposition. We have the following theorems.

Theorem 2 *If for all cells in the decomposition $B = true$ in* every *repetition step, then*

$$R \leq D + 2L(\Delta - \delta)$$

where L is the number of non-end cells in the linear array.

With the proper distribution of delays, this upper bound can indeed be attained. The delays then should be independent of each other. Recall furthermore that delays may vary in 'time and space,' that is, in occurrence of an output and in instance of a component. The freedom of varying delays independent of each other and in time and space may give a too pessimistic bound for some applications. If more information is available about the possible delay distributions, then tighter upper bounds may be obtained.

Theorem 3 *If for all cells in the decomposition $B = true$ in* at most every other *repetition step, and D satisfies*

$$D \leq \Delta + \Delta^2/\delta + \delta L^2 - 2\Delta L$$

then

$$R \leq \Delta + \Delta^2/\delta$$

where L is the number of non-end cells in the linear array.

Notice that here the upper bound for the response time R is independent of L, the number of cells in the array. The response time R only depends on the values of δ and Δ. In other words, the response time is bounded under any delay distribution (in time and space) and for any length of the array. The requirement for D, the maximum response time for the end cell, is easy to satisfy. For example, $D \leq \Delta$ will do for any L. For large L, however, D may be chosen much larger such that still a bounded response time is guaranteed. (In fact D may increase quadraticly with L.) This property can be exploited by taking, for large L, an end cell that is very slow, but has a low power consumption, for example.

What is the response time for a decomposition using only *PCELL0*? Such a decomposition has $L = O(\log N)$ cells, and each cell, except the end cell, is of the form (7). Furthermore, in each repetition step there is a pair of communications with both neighbors. Consequently, Theorem 2 applies. From Theorem 2 we then derive that the decomposition has a response time of at most $O(\log N)$, if $\Delta \neq \delta$. If $\Delta = \delta$, then the response time is bounded by the response time of the end cell. (It can be proven that the upper bound of $O(\log N)$ can be attained, if the distribution of the delays is such that, for example, 1's propagate twice as fast as 0's through the cells.)

In order to calculate the response time of a decomposition using only $PCELL1$ we can apply Theorem 3. Notice that $PCELL1$ communicates on channels sr and sa in at most every other repetition step. We assume that the end cell has a response time of at most Δ, that is, $D \leq \Delta$. By Theorem 3 it now follows that this decomposition has a bounded response time.

For a decomposition using $PCELL1$ for even N and $CELL2$ for odd N, we can also use Theorem 3 to conclude a bounded response time. This conclusion follows from looking at the composite behavior of $PCELL1$ followed by $CELL2$. Without proof we mention that this behavior is also of the form (7) and has the property that at most every other repetition step there is pair of communications on channels sr and sa.

There are many other combinations of decomposition steps we haven't analyzed. For example, what is the response time of a decomposition using $PCELL2$ only? What is the response time of a decomposition using $PCELL0$ and $PCELL2$? Or, of a decomposition using $PCELL1$ and $PCELL2$? None of the two theorems directly applies to these decompositions. For most of these combinations the analysis is non-trivial. Having more general theorems than the two we presented would be helpful to calculate response times of such decompositions.

7 The Up-Down N Counter

7.1 Specification

An up-down counter is a component on which two operations can be performed: an 'up', which is an increment by one, and a 'down', which is a decrement by one. For each of those operations, one of three replies will be sent back to the environment. The reply depends on the count and the range of the counter. The count of the counter is the number of ups minus the number of downs. The range of the up-down N counter is $[0..N]$, where $N > 0$. Initially, the count of the counter is 0. If after a down, the count of the up-down N counter reaches 0, then the reply will be 'empty'. If after an up, the count of the up-down N counter reaches N, then the reply will be 'full'. Otherwise, the reply will be 'ack'. In order for the count of the up-down N counter to stay in the range $[0..N]$, we assume that the environment will not attempt a down when the count is 0 and will not attempt an up when the count is N. Notice that the environment is informed after each operation whether the counter is empty, full, or neither. We stipulate that initially the counter is empty.

Here is a specification of the up-down N counter using guarded commands. We take the following type definitions throughout these notes.

type $ud = \{up, down\}$
type $efa = \{empty, full, ack\}$

The specification reads

```
  $UDC(N : int,\ r? : ud,\ a! : efa)$
=    { by definition }
  $|[$ var $n : [0..N]$ ::
    initially $n = 0$ ::
    pref $*[\ r?;$
            if $r = up \ \wedge\ n = N - 1$  then $a := full;\ n := n + 1$
            | $r = up \ \wedge\ n < N - 1$  then $a := ack;\ n := n + 1$
            | $r = down \ \wedge\ n = 1$    then $a := empty;\ n := n - 1$
            | $r = down \ \wedge\ n > 1$    then $a := ack;\ n := n - 1$
            fi ; $a!$
            ]
  $]|$
```

For the repetition we have the invariant

$$n = (\#r?up - \#r?down)$$

Notice that the two alternatives

$r = down \ \wedge\ n = 0$
$r = up \ \wedge\ n = N$

do not occur in the guarded selection, since we assume that the environment does not attempt a down when the count is 0 and not attempt an up when the count is N. (By definition of the guarded selection, the **if** ..**fi** statement amounts to an *abort* in states where these alternatives would apply.)

The important property we should remember is the precondition for every output a:

$$(a = \mathit{full}) \equiv (n = N) \quad \wedge \quad (a = \mathit{empty}) \equiv (n = 0)$$

Here is one special case of the up-down N counter. For $N = 1$ we have

$$\mathbf{pref} * [\ r?up;\ a!\mathit{full};\ r?down;\ a!\mathit{empty}\ ]$$

Given the specification of the up-down N counter we are asked to find an efficient asynchronous implementation for this component for any $N > 0$. Try it, before reading any further.

7.2 Related Work and Results

There are many implementations of up-down counters commercially available. (See, for example, the data books of several manufacturers.) There are also many applications for an up-down counter. For example, they can be used in any application that keeps track of a count within a range $[0..N]$, and where the only operations on the count are increments and decrements by one and testing whether the value of the count is 0 or N. Applications that come to mind immediately are semaphores, bounded stacks, and bounded queues.

There is also quite a rich literature on all sorts of counters. Designing synchronous implementations (as opposed to asynchronous implementations) of an up-down counter is usually considered a standard exercise in almost every textbook on digital design. In [25] one can find designs for many types of counters, including up-down counters. In [14] a synchronous implementation of an up-down counter is discussed, where the output is available after a constant number of clock cycles. This design, however, does not detect whether the counter is full. The idea of this implementation is the same as the idea underlying our first design. In [16] an up-down $2N$ counter is implemented by N identical modules. The output is also available after a constant number of clock cycles under the assumption that the inputs can be broadcast to all modules in a constant amount of time. In our designs no broadcast is needed. In [26], which is based on [14], several up-down counter designs are presented. The counters are slightly different in the sense that they behave like modulo-N counters when an increment occurs in the full state.

Most counters described in the literature are counters that report the value of the count in some radix representation after each operation. For the up-down counter, however, there is no need to know the value of the count after each operation. The only information that is needed is whether the count is equal to one of the two boundary values, and, if so, which one. It is, however, possible to implement an up-down counter that is based on a counter that reports the value of the count after each operation. From this value the response *full, empty*, or *ack* can be calculated. This calculation must consider all digits in the radix representation. If you want to obtain a bounded response time, it is hard to imagine that this could be achieved with such a design.

Most published implementations we have found are synchronous implementations and are based on some sort of binary representation of the count. As a consequence the clock frequency depends on the counter size N. For most designs this is due to the carry and borrow propagation during increments and decrements [20]. These carry and borrow propagations give a response time that is at least logarithmic in N. Furthermore, every synchronous implementation based on a binary representation of the count has to clock in each period about $\log N$ storage devices, making the power consumption at least proportional to $\log N$.

In conclusion, we have found that all published designs are clocked designs, have a logarithmic area complexity, usually do not give a response time analysis, do not give a power consumption analysis, and only apply to very few values of N (usually only for values $2^k - 1, k > 0$). We present several designs for the up-down N counter that have an area complexity of $\log N$, but unlike all previously known designs, are unclocked, have a bounded response time, have a bounded power consumption, and apply to any $N > 0$. These bounds are asymptotically optimal.

7.3 A First Decomposition

The derivation of our first decomposition is based on the usual binary representation of numbers. Suppose the count is represented by k bits, $k > 0$. An increment can be implemented by adding 1 to the least significant bit modulo 2, where a carry may propagate possibly all the way to the most significant bit. Similarly, a decrement is implemented by subtracting 1 from the least significant bit modulo 2, where a borrow may propagate possibly all the way to the most significant bit. Keeping track of whether the counter is full or empty can be done while returning to the least significant bit from a carry or borrow propagation. Assuming that the counter is full when all bits are 1 and empty when all bits are 0, then we can have the following scenario. Each cell records whether the rest of the (more significant) bits are all 1 (subcounter full), all 0 (subcounter empty), or neither all 1 nor all 0. After an increment to a cell, the response is 'full' if the subcounter is full and the new value of the bit is 1. Similarly, after a decrement to a cell, the response is 'empty' if the subcounter is empty and the new value of bit is 0. In all other cases the response is 'ack.'

In order to give a formal derivation, we show that a $2N + 1$ counter can be decomposed into an N counter and a cell. Let $N > 0$. We first give the specification of the $2N + 1$ counter. The specification for the cell will arise in the derivation.

```
   $UDC(2N+1 : int,\ r? : ud,\ a! : efa)$
=      { by definition }
   |[ **var** $nn : [0..(2N+1)]$ ::
     **initially** $nn = 0$ ::
     **pref** $*$ [ $r?$;
             **if** $r = up \wedge nn = 2N$   **then** $a := full;\ nn := nn + 1$
             | $r = up \wedge nn < 2N$   **then** $a := ack;\ nn := nn + 1$
             | $r = down \wedge nn = 1$  **then** $a := empty;\ nn := nn - 1$
             | $r = down \wedge nn > 1$  **then** $a := ack;\ nn := nn - 1$
             **fi** ; $a!$
             ]
   ]|
```

In our first step we introduce binary variable k, representing the value of the bit in the cell, and variable n, representing the count of the subcounter. The variables k and n are related to nn by the invariant

$$P: \ nn = 2*n+k \quad \wedge \quad 0 \le n \le N \quad \wedge \quad 0 \le k \le 1$$

From this invariant we derive the following equivalences.

$$\begin{aligned} nn = 2N &\equiv (n = N \ \wedge \ k = 0) \\ nn < 2N &\equiv n \ne N \\ nn = 1 &\equiv (n = 0 \ \wedge \ k = 1) \\ nn > 1 &\equiv n \ne 0 \end{aligned}$$

These observations then lead to the following program for $UDC(2N+1, r, a)$.

```
|[ var $nn : [0..(2N+1)], n : [0..N], \ k : bin$ ::
  initially $n = 0, k = 0$ ::
  pref * [ $r$?
          ; if $r = up \wedge n = N \wedge k = 0$      then $a := full;\ nn := nn + 1$
            | $r = up \wedge n \ne N$                  then $a := ack;\ nn := nn + 1$
            | $r = down \wedge n = 0 \wedge k = 1$    then $a := empty;\ nn := nn - 1$
            | $r = down \wedge n \ne 0$               then $a := ack;\ nn := nn - 1$
            fi
          ; if $r = up \wedge k = 1$      then $n := n + 1$
            | $r = down \wedge k = 0$ then $n := n - 1$
            | $(r = up \wedge k = 0)$ or $(r = down \wedge k = 1)$  then $skip$
            fi
          ; $k := (k + 1) \bmod 2$
          ; $a$!
          ]
]|
```

The alternative $r = up \ \wedge \ k = 1$ corresponds to a carry propagation, here represented by $n := n + 1$, and the alternative $r = down \ \wedge \ k = 0$ corresponds to a borrow propagation, here represented by $n := n - 1$. It is not difficult to see that $nn = 2 * n + k \ \wedge \ 0 \le k \le 1$ are indeed invariants of this program. Let us check that $0 \le n \le N$ is an invariant of the program as well. Because of the semantics of guarded selection, a postcondition for the first guarded selection, which is the precondition for the second guarded selection, is

$$(r = up \ \wedge \ k = 1 \ \Rightarrow \ n < N) \quad \wedge \quad (r = down \ \wedge \ k = 0 \ \Rightarrow \ n > 0)$$

In other words, an increment to n is not done when $n = N$ and a decrement to n is not done when $n = 0$. From this observation we may conclude that $0 \le n \le N$ is an invariant of the repetition.

After the first derivation step, we can make a couple of observations. First, the variable nn is a ghost variable: it is never inspected and is only used for the correctness proof of the derivation step. Consequently, we can remove all statements involving nn from the program. Second, we observe that the only operations on variable n are increments, decrements, and tests whether $n = 0$ or $n = N$. These operations can be performed by an up-down N counter. For this purpose we introduce an up-down N counter with input channel sr and output channel sa, and whenever an increment to n is done we replace this statement by $sr!up; sa?$, whenever a decrement to n is done we replace this statement by $sr!down; sa?$. After every communication on channel sa we can then assert

$$(sa = full) \equiv (n = N) \ \wedge \ (sa = empty) \equiv (n = 0)$$

where $n = (\#sr?up - \#sr?down)$. These observations then lead to the following parallel composition.

```
    $UDC(2N+1, r?, a!)$
=      { def. of $UDC(N,\ sr?,\ sa!)$, see above }
    |[ $\mathbf{chan}\ sr : ud,\ sa : efa\ ::$
       $CELL0(r?,\ a!,\ sr!,\ sa?)\ \|\ UDC(N, sr?, sa!)$
    ]|
```

where $CELL0$ is defined by

```
|[ $\mathbf{var}\ k : bin$ ::
  $\mathbf{initially}\ sa = empty,\ k = 0$ ::
  $\mathbf{pref} * [\ r?$
        ; $\mathbf{if}\ r = up \wedge sa = full \wedge k = 0$          $\mathbf{then}\ a := full$
         | $r = up \wedge sa \neq full$                    $\mathbf{then}\ a := ack$
         | $r = down \wedge sa = empty \wedge k = 1$  $\mathbf{then}\ a := empty$
         | $r = down \wedge sa \neq empty$               $\mathbf{then}\ a := ack$
         $\mathbf{fi}$
        ; $\mathbf{if}\ r = up \wedge k = 1$    $\mathbf{then}\ sr!up;\ sa?$
         | $r = down \wedge k = 0\ \mathbf{then}\ sr!down;\ sa?$
         | $(r = up \wedge k = 0)$ or $(r = down \wedge k = 1)$  $\mathbf{then}\ skip$
         $\mathbf{fi}$
        ; $k := (k+1) \bmod 2$
        ; $a!$
        ]
]|
```

Notice that $CELL0$ will not attempt an $sr!down$ when the subcounter is empty, nor will $CELL0$ attempt an $sr!up$ when the subcounter is full. This property follows immediately from the invariant $0 \leq n \leq N$ of the previous program.

Finally we observe that all conditions for decomposition are satisfied. In particular, no computation interference can occur: in $CELL0$ every output $sr!$ is immediately followed by input $sa?$ and every output $a!$ is immediately followed by input $r?$. Accordingly, we can write

$$
\begin{array}{ll}
 & UDC(2N+1,\ r?,\ a!) \\
\rightarrow & \quad \{ \text{ def. of decomposition } \} \\
 & (\ CELL0(r?,\ a!,\ sr!,\ sa?)\ \ ,\ \ UDC(N, sr?, sa!)\ )
\end{array}
$$

7.4 What about even N?

In the previous section we derived a decomposition that applies to odd N only. If we need to have a decomposition that applies to all N, we also need to find a decomposition that applies to even N. Our next decomposition is a generalization of our first decomposition and is not restricted to even N only.

The previous decomposition was based on the unique binary representation of each number. The invariant we used there was

$$P:\ nn = 2*n+k \quad \wedge \quad 0 \leq n \leq N \quad \wedge \quad 0 \leq k \leq 1$$

where nn represents the count of the $2N+1$ counter and n represents the count of the N counter. What would happen if we enlarged the range of k? For example, what would change in our derivation if we had $0 \leq k \leq K$? It turns out that we need to change our derivation only slightly. Let $K > 0$. Our new invariant is

$$nn = 2*n+k \quad \wedge \quad 0 \leq n \leq N \quad \wedge \quad 0 \leq k \leq K$$

From this invariant we derive the following equivalences.

$$
\begin{array}{rcl}
(nn = 2N+K-1) & \equiv & (n = N\ \wedge\ k = K-1) \\
(nn < 2N+K-1) & \equiv & (n \neq N\ \vee\ k < K-1) \\
nn = 1 & \equiv & (n = 0\ \wedge\ k = 1) \\
nn > 1 & \equiv & (n \neq 0\ \vee\ k > 1)
\end{array}
$$

These observations then lead to the following program.

$$
\begin{array}{ll}
 & UDC(2N+K, r?, a!) \\
= & \quad \{ \text{ for def. of } CELL2 \text{ see below } \} \\
 & (\ CELL2(K : int,\ r?,\ a!,\ sr!,\ sa?)\ \ ,\ \ UDC(N,\ sr?,\ sa!)\ )
\end{array}
$$

where $CELL2$ is defined by

```
    $CELL2(K : int,\ r? : ud,\ a! : efa,\ sr! : ud,\ sa? : efa)$
  $=$   { by definition }
    |[ var $k : [0..K]$ ::
      initially $sa = empty,\ k = 0$ ::
      pref $*$ [ $r?$
              ; if $r = up \wedge sa = full \wedge k = K-1$ then $a := full$
               | $r = up \wedge (sa \neq full \vee k < K-1)$ then $a := ack$
               | $r = down \wedge sa = empty \wedge k = 1$ then $a := empty$
               | $r = down \wedge (sa \neq empty \vee k > 1)$ then $a := ack$
              fi
              ; if $r = up \wedge k = K$ then $sr!up;\ sa?$
               | $r = down \wedge k = 0$ then $sr!down;\ sa?$
               | $(r = up \wedge k \neq K)$ or $(r = down \wedge k \neq 0)$ then $skip$
              fi
              ; if $r = up \wedge k \neq K$ then $k := k+1$
               | $r = up \wedge k = K$ then $k := K-1$
               | $r = down \wedge k \neq 0$ then $k := k-1$
               | $r = down \wedge k = 0$ then $k := 1$
              fi
              ; $a!$
              ]
    ]|
```

Now we have many decompositions that apply for even N. For example, we can use *CELL*2 for $K = 2$ or $K = 4$, when N is even. For odd N we also have many decompositions. We can use *CELL*2 for $K = 1$ or $K = 3$, when N is odd. If we are restricted to using *CELL*2 for values of $K = 2$ and $K = 3$, then any N counter for $N \geq 1$ can be decomposed using such cells, if we have at least as end cells a 1 counter, a 2 counter, and a 3 counter. Notice that for $N \geq 1$, we have $2N + 2 \geq 4$ and $2N + 3 \geq 5$.

For $K = 1$, we have the usual binary number system, where each number has a unique representation. For $K > 1$ we use a redundant binary number system, where the digits can range from 0 to K inclusive. For example, for $K = 2$ the number 4 can be represented as 100 and as 012 (least significant digit to the right). This redundancy will pay off later when we examine the power consumption and response time.

7.5 What about Parallelism?

The next step in our derivation of an optimal design is the introduction of some parallelism. We try to do so by allowing as much freedom as possible in the ordering of the communication actions in a cell with the restrictions that the communication behavior on channels r and a remains invariant and the communication behavior on channels sr and sa remains invariant.

We first introduce some parallelism in the specification for *CELL*0. Let us consider the specification for *CELL*0 once more.

```
    $CELL0(r? : ud,\ a! : efa,\ sr! : ud,\ sa? : efa)$
 =      { by definition }
    |[ var $k : bin$ ::
      initially $sa = empty,\ k = 0$ ::
      pref * [ $r?$
              ; if $r = up \wedge sa = full \wedge k = 0$          then $a := full$
                | $r = up \wedge sa \neq full$                    then $a := ack$
                | $r = down \wedge sa = empty \wedge k = 1$      then $a := empty$
                | $r = down \wedge sa \neq empty$                then $a := ack$
                fi
              ; if $r = up \wedge k = 1$       then $sr!up;\ sa?$
                | $r = down \wedge k = 0$   then $sr!down;\ sa?$
                | $(r = up \wedge k = 0)$ or $(r = down \wedge k = 1)$  then $skip$
                fi
              ; $k := (k + 1) \bmod 2$
              ; $a!$
              ]
    ]|
```

Observe that although the value of a is calculated immediately after r has been received, it is sent only after there has been a possible communication on the channels sr and sa. Immediately after sending a the next request on channel r can be received, in order to avoid computation interference. Is it possible to send a (and receive the next request on r) in parallel with sending sr (and receiving sa)? Yes, that can be done. If we unfold the repetition a bit and reorder some statements, we get the following program for *PCELL*0

```
    $PCELL0(r? : ud,\ a! : efa,\ sr! : ud,\ sa? : efa)$
 =      { by definition }
    |[ var $k : bin$ ::
      initially $sa = empty,\ k = 0$ ::
      pref ( $r?;\ S_a$;
            *[ if $r = up \wedge k = 1$       then $(a!;\ r?) \parallel S_k \parallel (sr!up;\ sa?)$
                | $r = down \wedge k = 0$   then $(a!;\ r?) \parallel S_k \parallel (sr!down;\ sa?)$
                | $(r = up \wedge k = 0)$ or
                  $(r = down \wedge k = 1)$ then $(a!; r?) \parallel S_k$
               fi
             ; $S_a$
             ])
    ]|
```

where $S_k = k := (k+1) \bmod 2$ and S_a is the statement that calculates the next value for a given by

```
$S_a$
$=$   { by definition }
   if $r = up \wedge sa = full \wedge k = 0$             then $a := full$
    | $r = up \wedge sa \neq full$                        then $a := ack$
    | $r = down \wedge sa = empty \wedge k = 1$          then $a := empty$
    | $r = down \wedge sa \neq empty$                    then $a := ack$
   fi
```

It is not difficult to verify that the communication behavior of *CELL*0 and *PCELL*0 is the same if we look to the channels r and a only. Similarly, the communication behavior of *CELL*0 and *PCELL*0 is the same if we look to the channels sr and sa only.

The same exercise can be performed on *CELL*2. If we name the resulting specification *PCELL*2, we get

```
$PCELL2(K : int,\ r? : ud,\ a! : efa,\ sr! : ud,\ sa? : efa)$
$=$   { by definition }
   |[ var $k : [0..K]$ ::
      initially $sa = empty,\ k = 0$ ::
      pref ( $r?;\ S_a;$
           *[ if $r = up \wedge k = K$     then $(a!;\ r?) \parallel (k := K - 1) \parallel (sr!up;\ sa?)$
               | $r = down \wedge k = 0$   then $(a!;\ r?) \parallel (k := 1)$       $\parallel (sr!down;\ sa?)$
               | $r = up \wedge k \neq K$   then $(a!;\ r?) \parallel (k := k + 1)$
               | $r = down \wedge k \neq 0$ then $(a!;\ r?) \parallel (k := k - 1)$
              fi
            ; $S_a$
            ])
   ]|
```

where S_a is the appropriate statement calculating the next value of a.

7.6 An Implementation

As an example of an implementation, we briefly present one for *PCELL*0 using data bundling. As an encoding for the data type *ud*, we use a single wire and take by definition $up = 1$ and $down = 0$. As an encoding for the type *efa* we take two wires called f and e. Each value of type *efa* is then encoded by a pair of binary values (f, e) in the following way.

$full = (1, 0)$
$empty = (0, 1)$
$ack = (0, 0)$

The value (1,1) is not used. These encodings then quickly lead to the implementation of Figure 16 for the up-down 1 counter.

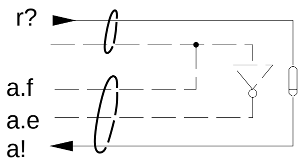


Figure 16: An implementation for *UDC*(1) using data bundling

From the program for *PCELL*0, we derive that the 'new' values for $a = (a.f, a.e)$ and k can be calculated as follows from the inputs r and $sa = (sa.f, sa.e)$.

$$\begin{aligned} a.f &:= r \wedge sa.f \wedge \neg k \\ a.e &:= \neg r \wedge sa.e \wedge k \\ k &:= \neg k \end{aligned}$$

The 'new' value for sr is calculated as follows. If r and k are both zero or both one, then sr becomes zero or one respectively. Otherwise, sr retains its 'old' value. In other words, the 'new' value for sr is the majority of r, k, and the old value of sr. Furthermore, a communication with the subcounter takes place only when r and k are both zero or both one. Therefore, we introduce a binary variable *prop* (of *prop*agate) to direct the communications with the subcounter. The values for sr and *prop* are calculated as follows.

$$\begin{aligned} sr &:= maj(r, k, sr) \\ prop &:= (r \wedge k) \vee (\neg r \wedge \neg k) \end{aligned}$$

where $prop = 1$ if there is a communication on the channel sr, and $prop = 0$ if there is no communication on the channel sr.

The control part of the implementation consists of a JOIN, a MERGE, a SELECT, and some WIREs. The control part is indicated with solid lines in Figure 17. The data part is indicated by dashed lines in Figure 17.

7.7 Performance Analysis

Before we are going to analyze the performance of some complete decompositions of the up-down N counter, we give a brief summary of the possible decomposition steps. Below, *UDC*(N) stands for *UDC*(N, $r?$: *ud*, $a!$: *efa*) and *CELL*2(K) stands for *CELL*2(K : *int*, $r?$: *ud*, $a!$: *efa*, $sr!$: *ud*, $sa?$: *efa*). A similar correspondence holds for *PCELL*2(K). (Recall that *CELL*0 is a special case of *CELL*2(K), viz., *CELL*2(1) = *CELL*0)

$$\begin{aligned} UDC(2N+1) &\rightarrow (\ CELL2(1)\ ,\ UDC(N)\ ) && (0) \\ UDC(2N+2) &\rightarrow (\ CELL2(2)\ ,\ UDC(N)\ ) && (1) \\ UDC(2N+3) &\rightarrow (\ CELL2(3)\ ,\ UDC(N)\ ) && (2) \\ UDC(2N+1) &\rightarrow (\ PCELL2(1)\ ,\ UDC(N)\ ) && (3) \end{aligned}$$

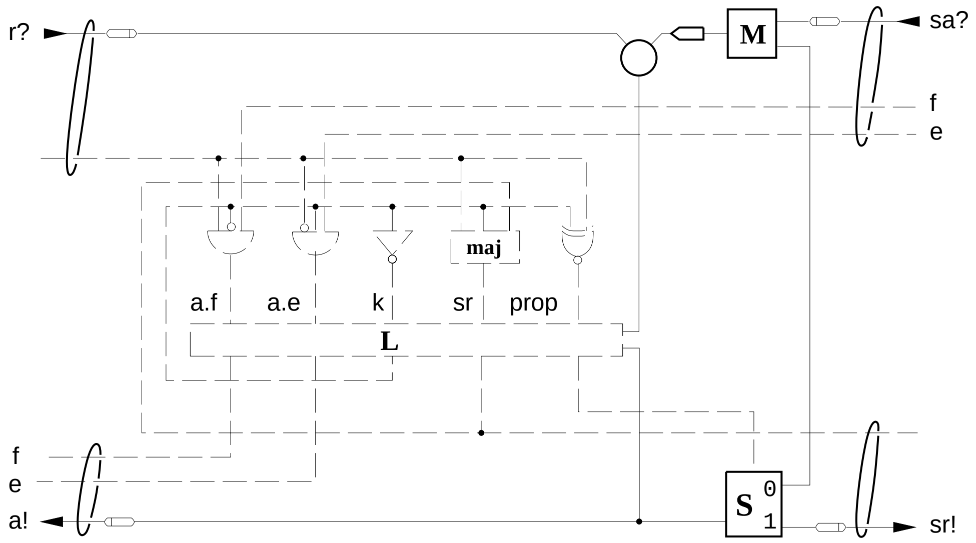


Figure 17: An implementation of *PCELL*0 using data bundling

$$UDC(2N+2) \rightarrow (\ PCELL2(2)\ ,\ UDC(N)\ ) \qquad (4)$$
$$UDC(2N+3) \rightarrow (\ PCELL2(3)\ ,\ UDC(N)\ ) \qquad (5)$$

Let us consider the area complexity of some decompositions first. Our first observation is that any of the cells above can be considered as a basic component, since the number of states of each cell is bounded by some constant, which is independent of N. A similar remark can be made for small counters, like the 1, 2, and 3 counter. Accordingly, a first-order estimate for the area complexity can be obtained by counting the total number of cells and end cells of the decomposition. The area complexity of some decompositions can now be calculated easily. A decomposition of the up-down N counter using any of the cells $CELL2(K)$ or $PCELL2(K)$ for $K = 1, 2, 3$ obviously has area complexity $\Theta(\log N)$. In a similar manner as for the modulo-N counter, we can prove that this bound is optimal.

The analysis for the power consumption is a bit more difficult. Let us first consider the power consumption of a decomposition using only $CELL2(1)$. This decomposition is based on the usual binary number system. In the worst case a request propagates through all $\log N$ cells and comes back, and this can happen repeatedly. For example, when the count of the counter is $(N-1)/2$ (that is, the binary representation consists of all ones, except the most significant bit which is zero), and the sequence *r?up*; *a!ack*; *r?down*; *a!ack* is performed repeatedly. Then, for each request, each bit has to flip once. Consequently, each external communication results in $\Theta(\log N)$ communications, and so the worst-case power consumption over all communication behaviors is $\Theta(\log N)$,

What is the power consumption of a decomposition using $CELL2(2)$ and $CELL2(3)$ with $UDC(1)$, $UDC(2)$, or $UDC(3)$ as end cell? This decomposition uses a redundant binary representation of the count. The important observation is that in both $CELL2(2)$ and $CELL2(3)$ in at most every other repetition step there will be a communication on the channels sr and sa. Accordingly, if in a communication behavior $2k$ external communication actions take place, at most $2k/2$ communication actions take place between the first and second cell, at most $2k/4$ communication actions take place between the second and third cell, and so on. Consequently, each communication behavior with $2k$ external communication actions has a power consumption of at most

$$(\sum_{i=0}^{\infty} 2k/2^i)/2k \leq 2$$

From this observation we may conclude that every behavior has a bounded power consumption, and therefore the decomposition using cells $CELL2(K)$ for $K = 2, 3$ has a bounded power consumption.

Would the power consumption change if we used a $PCELL2$ instead of a $CELL2$? From the programs for $CELL2$ and $PCELL2$ we derive that for each external behavior with $2k$ actions, the maximum number of internal communication actions that can take place in the parallel and the sequential version are the same. The only difference is in the order in which the actions can take place. For this reason, the results of the power consumption analyses for the sequential version are also valid for the parallel version.

Finally we analyze the response time of some decompositions. The 'sequential' decompositions are easy. A decomposition using any of the cells $CELL2$ has logarithmic response time, since in the worst case a request propagates through all $\Theta(\log N)$ cells to the end cell and then returns as an acknowledgement.

For the decomposition using the parallel versions of $CELL2$, we can use Theorems 2 and 3. We assume that the upper bounds and lower bounds for the response times of the cells (including the end cells) are given by Δ and δ respectively. Neither Theorem 2 nor 3 is directly applicable to a decomposition using only $PCELL2(1)$. However, if we assume that the environment gives input requests according to a worst-case scenario, then after an initial behavior each cell will communicate on channels r, a and sr, sa in each repetition step. According to Theorem 2, this decomposition then has a worst-case response time of $O(\log N)$ if $\Delta > \delta$. In order to attain this upper bound, however, we have to assume a very pessimistic delay distribution.

For $PCELL2(2)$ and $PCELL2(3)$ we observe that in at most every other repetition step there is a pair of communications on sr and sa. Accordingly, Theorem 3 is applicable, and we conclude that a decomposition using $PCELL2(2)$ and $PCELL2(3)$ has a bounded response time.

8 Concluding Remarks

We have illustrated some techniques in the design and performance analysis of asynchronous circuits. The techniques were illustrated by means of two examples: the modulo-N counter and the up-down N counter. For both examples we derived several designs that have an area complexity of $\Theta(\log N)$, a bounded power consumption, and a bounded response time. These bounds are optimal. Although the exercises have given us some insights in the alternatives in designing asynchronous circuits, there are still many problems that remain. We mention a few of them.

The final step of our derivations consisted of finding circuit implementations for the cells. Compared to many of the other steps in the design, this last step was rather large and often ad hoc and without a proof. We did so, because the main emphasis of these notes was on the design of the parallel program rather than on the design of circuit implementations for the cells. In order to obtain a complete design method, however, it is important that this last step be investigated more closely, and that a systematic method be found to translate our specifications for the cells into circuit implementations. The choice of implementation for the data types will undoubtedly play a large role in this step.

For all designs, we first started with the design of a sequential program and then transformed this sequential program into a parallel program by reordering the communication actions. Can this be done for every design? That is, for every parallel program, does there exist a sequential program that can be transformed into the parallel program by reordering the statements and communication actions? If so, do there exist some general techniques for reordering communication actions so as to obtain a parallel program?

The performance analyses raised some interesting questions as well. For the area complexity we can compute a lower bound for any implementation of a component easily. For the power consumption and response time this is a different matter. How do we calculate the lower bounds for the power consumption and response time for any implementation of a particular component? For example, what is the lower bound of any implementation for an N-place stack or queue? Knowing what the lower bound is of the power consumption of a component, we may able to conclude whether we have found an optimal design. If not, we may want to search for a more efficient design.

Calculating the response time of an implementation was a nontrivial task. We gave some theorems that were very useful, but these theorems applied only to linear arrays of cells and only under certain conditions for the behavior of the cells. How do we calculate the response time for other networks of cells or when the cell's behavior does not satisfy these restricted conditions?

Finally, we mention the problem of formulating appropriate progress conditions. For any design it is important to know whether progress is guaranteed or not. For example, a design should be free from the danger of deadlock or livelock. Our correctness conditions are too weak to guarantee progress in general. (See Section 3.8.) Finding correctness conditions for progress that have a proper justification in the context of asynchronous circuit behaviors and, preferably, are easy to work with is still an open problem.

Acknowledgements

Acknowledgements are due to Ad Peeters for his comments on the modulo-N counter decompositions, Kees van Berkel for directing our attention to the efficient decomposition of modulo-N counters and to analyzing the power consumption of decompositions, Marly Roncken and Joep Kessels who gave the idea for efficient modulo-N counter decompositions [18], Peter Mayo for giving the idea of using a redundant number system to decompose the up-down counter, and the International Council for Canadian Studies for providing financial support for the second author to conduct part of this research [29].

References

[1] C.H. van Berkel, *Handshake Circuits: An Intermediary between Communicating Processes and VLSI.* PhD Thesis, Dept. of Mathematics and Computing Science, Eindhoven University of Technology, 1992.

[2] K. van Berkel, VLSI Programming of a Modulo-N Counter with Constant Response Time and Constant Power, in: in : S. Furber and M. Edwards eds., *Asynchronous Design Methodologies*, IFIP Transactions A-28, North-Holland, 1993, 1–12.

[3] G.M. Brown, Towards Truly Delay-insensitive Circuit Realizations of Process Algebras, in: G. Jones and M. Sheeran eds., *Designing Correct Circuits*, Workshops in Computing, Springer Verlag, Berlin, 1990, 120–131.

[4] J.A. Brzozowski and C-J.H. Seger, A Unified Framework for Race Analysis of Asynchronous Networks, *J. ACM*, **36** (1), 1989, 20–45.

[5] S. Burns and A.J. Martin, Performance Analysis and Optimization of Asynchronous Circuits, in: Carlo H. Sequin ed., *Advanced Research in VLSI*, MIT Press, Cambridge, Mass., 1991, 71–86.

[6] W. Chen, J.T. Udding, and T. Verhoeff, Networks of Communicating Processes and Their (De-)Composition, in: J.L.A. van de Snepscheut ed., *Mathematics of Program Construction*, Lecture Notes in Computer Science 375, Springer-Verlag, Berlin, 1989, 174–196.

[7] W.A. Clark, Macromodular Computer Systems, in: *Proceedings of the Spring Joint Computer Conference,* AFIPS, Academic Press, London, 1967, 335–401.

[8] A. Davis, B. Coates, K. Stevens, Automatic Synthesis of Fast Compact Self-Timed Control Circuits, in : S. Furber and M. Edwards eds., *Asynchronous Design Methodologies*, IFIP Transactions A-28, North-Holland, Amsterdam, 1993, 193–208.

[9] A. Davis, B. Coates, K. Stevens, The Post Office Experience: Designing a Large Asynchronous Chip, in T.N. Mudge et al. eds., *Proceedings of the 26th Annual Hawaii International Conference on System Sciences*, Vol. 1, IEEE Computer Society Press, 1993, 409–418.

[10] E.W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, 1976.

[11] D.L. Dill, *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*, MIT Press, Cambridge, Mass., 1989.

[12] J.C. Ebergen, A Formal Approach to Designing Delay-Insensitive Circuits, *Distributed Computing*, **5** (3), 1991, 107–119.

[13] J.C. Ebergen and A.M.G. Peeters, Modulo-N Counters: Design and Analysis of Delay-Insensitive Circuits, in: J. Staunstrup and R. Sharp eds., *Designing Correct Circuits*, IFIP Transactions A-5, North-Holland, Amsterdam, 1992, 27–46.

[14] L.J. Guibas and F.M. Liang, Systolic Stacks, Queues, and Counters, in: P. Penfield Jr. ed., *Advanced Research in VLSI*, Artech House, 1982, 155–164.

[15] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, London, 1985.

[16] E.V. Jones and G. Bi, Fast Up/Down Counters Using Identical Cascaded Modules, *IEEE Journal of Soli-State Circuits*, 23 (1), 1988, 283–285.

[17] M.B. Josephs and J.T. Udding, Delay-Insensitive Circuits: An Algebraic Approach to Their Design, in: J.C.M. Baeten and J.W. Klop eds., *CONCUR 1990*, Lecture Notes in Computer Science 458, Springer-Verlag, Berlin, 1990, 342–366.

[18] J.L.W. Kessels, Designing Counters with Bounded Response Time, in: W.H.J. Feijen and A.J.M. van Gasteren eds., *C.S. Scholten Dedicata: van oude machines en nieuwe rekenwijzen*, Academic Service, Schoonhoven, 1990, 127–140.

[19] L. Lavagno and A. Sangiovanni-Vincentelli, *Algorithms for Synthesis and Testing of Asynchronous Circuits*, Kluwer Academic Press, 1993.

[20] X.D. Lu and P.C. Treleaven, A Special-Purpose VLSI Chip: A Dynamic Pipeline Up/Down Counter. *Microprocessing and Microprogramming*, 10 (1), 1982, 1–10.

[21] A.J. Martin, Programming in VLSI: From Communicating Processes to Delay-Insensitive Circuits, in: C.A.R. Hoare ed., *Developments in Concurrency and Communication*, Addison-Wesley, Reading, Mass., 1990, 1–64.

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

[23] R.E. Miller, *Switching Theory*, Vol. 2, Chapter 10, Wiley, New York, 1965, 199–244.

[24] S.M. Nowick and D.L. Dill, Synthesis of Asynchronous State Machines Using a Local Clock, *Proceedings of the 1991 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, IEEE Computer Society Press, 1991, 192–197.

[25] R.M.M. Oberman, *Counting and Counters*, Macmillan Press, 1981.

[26] B. Parhami, Systolic Up/Down Counters with Zero and Sign Detection, in: M.J. Irwin and R. Stefanelli eds., *IEEE Symposium on Computer Arithmetic*, IEEE Computer Society Press, 1987, 174–178.

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

[28] F.U. Rosenberger, C.E. Molnar, T.J. Chaney, T.-P. Fang, Q-Modules: Internally Clocked Delay-Insensitive Modules, *IEEE Trans. on Comp.*, C-37, No. 9, 1988, pp. 1005–1018.

[29] J.P.L. Segers, *The Design and Analysis of Asynchronous Up-Down Counters*, Research Report CS-93-29, Computer Science Department, University of Waterloo, 1993.

[30] J. Sparsø and J. Staunstrup, Design and Performance Analysis of Delay-Insensitive Multi-Ring Structures, in: T.N. Mudge et al. eds., *26th Annual Hawaii International Conference on System Sciences*, Vol. 1, IEEE Computer Society Press, 1993, 349–358.

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

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

[33] Alan M. Turing, Lecture to the London Mathematical Society on 20 February 1947. In: Carpentar BE, Doran RW eds., *Charles Babbage Institute Reprint Series for the History of Computing*, vol. 10, MIT Press, Cambridge, Massachusetts, 1986.

[34] J.T. Udding, A Formal Model for Defining and Classifying Delay-Insensitive Circuits and Systems, *Distributed Computing*, **1** (4), 1986, 197–204.

[35] N.H.E. Weste and K. Eshragian, *Principles of CMOS VLSI Design*, Addison-Wesley, 1985.

[36] T.E. Williams, Analyzing and Improving Latency and Throughput in Self-Timed Rings and Pipelines, *Proceedings of 1992 ACM/SIGDA Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, Princeton University, March 1992.