The Design and Analysis of Asynchronous Up-Down Counters [0]

by

J.P.L. Segers
Department of Mathematics and Computing Science
Eindhoven University of Technology
The Netherlands

[0]This report was presented to the Eindhoven University of Technology in fulfillment of the thesis requirement for the degree of Ingenieur in de Technische Informatica. The work was done while visiting the University of Waterloo from September 1992 until May 1993.

Acknowledgements

I am very thankful to my supervisor Jo Ebergen from the University of Waterloo in Canada for listening to me, answering my questions, and for carefully reading this manuscript. He made my stay in Waterloo very educational.

I thank all members of the MAVERIC research group for some interesting discussions on designing up-down counters. In one of those discussions Peter Mayo gave me the idea for an up-down counter with constant power consumption.

There are more people at the Computer Science Department of the University of Waterloo that deserve to be acknowledged. I will not try to mention all of them, because I would undoubtedly forget someone. Hereby I thank all of them.

The International Council for Canadian Studies is acknowledged for their financial support. The Government of Canada Award that they awarded to me made my stay in Waterloo financially possible.

I thank Rudolf Mak for getting me in touch with Jo Ebergen, and Franka van Neerven for helping with all kinds of organizational details that had to be taken care of before I could go to Waterloo.

Finally, I thank Twan Basten for being patient with me and listening to me during the stay in Waterloo.

Abstract

The goal of this report is to investigate *up-down counter* implementations in the framework of delay-insensitive circuits. An up-down counter is a counter on which two operations can be performed: an increment by one and a decrement by one. For N larger than zero, an up-down N-counter counts in the range from zero through N. In the counters we design, the value of the counter, or its count, cannot be read, but it is possible to detect whether the counter's value is zero, N, or somewhere in between. Up-down counters have many applications. For example, they can be useful in implementing queues or stacks.

Various implementations for up-down N-counters are presented for any N larger than zero. All counter designs are analyzed with respect to three performance criteria, namely area complexity, response time, and power consumption. One of the designs is optimal with respect to all three performance criteria. Its area complexity grows logarithmically with N, and its response time and power consumption are independent of N.

Contents

0 Introduction **1**

0.0 Synchronous Up-Down Counter Implementations 2

0.1 Designing Asynchronous Circuits 3

0.2 Results of the Thesis 5

0.3 Thesis Overview 6

1 Trace Theory and Delay-Insensitive Circuits **7**

1.0 Introduction 7

1.1 Trace Theory and Commands 7

1.2 Extending Commands 9

1.3 Basic Components 13

1.4 Decomposition 13

1.5 Delay-Insensitivity and DI decomposition 17

1.6 Sequence Functions 18

2 Formal Specification of Up-Down Counters **20**

2.0 Introduction 20

2.1 An Up-Down Counter with an *ack-nak* Protocol 20

2.2 An Up-Down Counter with an *empty-ack-full* Protocol 21

3 Some Simple Designs **24**

3.0 Introduction 24

3.1 Unary Implementations 25

3.1.0 Specification of the Cells 26

3.1.1 Correctness of the Implementation 27

3.1.2 Performance Analysis 29

3.2 A Binary Implementation 33

3.2.0 Specification of the Cells 33

3.2.1 Correctness of the Implementation 35

3.2.2 Implementations for General N 37

3.2.3 Performance Analysis 38

4 An Implementation with Parallelism **40**

4.0 Introduction 40

4.1 Specification of the Cells 41

4.2 Correctness of the Implementation 43

4.3 Implementations for General N 47

4.4 Performance Analysis 47

4.4.0 Area Complexity 47

4.4.1 Response Time Analysis 47

4.4.2 Power Consumption 55

4.5 The *ack-nak* Protocol 55

5 An Implementation with Constant Power Consumption **58**

5.0 Introduction 58

5.1 Specification of the Cells 60

5.2 Correctness of the Implementation 61

5.3 Performance Analysis 64

5.3.0 Response Time 65
5.3.1 Power Consumption 69

6 Conclusions and Further Research **71**
6.0 Introduction 71
6.1 Conclusions 71
6.2 Further Research 72

Bibliography **76**

Chapter 0

Introduction

Counters of all kinds are used in a variety of digital circuits. The kinds of counters used include binary counters, Gray code counters, ring counters, and up-down counters. Most of these counters cycle through a number of states, each state representing a natural number. Because of this cyclic behavior, the next state can be determined from the present state. An up-down counter behaves differently. It counts up or down, depending on the input received.

For many counters, the value of the counter, or its count, can be read by the environment. Sometimes, however, there is no need to be able to read the value of the counter. In the case of a modulo-N counter, for example, it can be sufficient to detect when the count modulo N is equal to zero. In the case of an up-down counter with a counting range from zero through N, it can be sufficient to detect when the counter's value is one of the boundary values. In [DNS92], for example, an up-down counter is used to find out whether a FIFO-queue is empty, full, or neither empty nor full. In [EG93a] the use of an up-down counter is proposed for a similar purpose.

In this report we specify and design up-down counters that count in the range from zero through N, for N larger than zero. We call such counters up-down N-counters, or just N-counters. The counters we specify and implement are of the kind where the environment of the counter can only detect whether the counter's value is zero, N, or neither zero nor N. It cannot read the counter's value. Being able to detect whether a counter's value is zero is called empty detection. Detecting whether the counter's value is N is called full detection.

We want to investigate whether or not it is possible to design N-counters with empty and full detection of which emptiness and fullness can be detected within an amount of time independent of N after an input has been sent to the counter. If only empty and full detection are required in a specific application, such a counter could have a faster response time than a readable counter

for which the detection of boundary values is implemented a posteriori. A readable up-down N-counter requires a number of "memory cells" that is at least logarithmic in N. It is hard to imagine that in such a counter the new value can be read after an amount of time independent of N, if no broadcasting of signals is allowed.

In the design of counters we can distinguish between synchronous and asynchronous counters. In this report we devote our attention to asynchronous counters, i.e., counters that do not use a global clock signal. Before giving some advantages of asynchronous circuits, we briefly discuss synchronous counter implementations. We have not found any designs for asynchronous up-down counters in the existing literature.

0.0 Synchronous Up-Down Counter Implementations

A synchronous counter is a counter that uses a global clock for its operation. Designing synchronous counters is usually considered a standard exercise. Synchronous designs can be found in many textbooks on logic design, such as [Man91]. There are not many articles on synchronous counters. In some of them the counters are used to illustrate particular circuit design methods, as in [LT82, CSS89]. In those articles the maximum clock frequency usually depends on the size of the counter and the counter size is limited. In the design proposed in [LT82] this is most apparent in the circuitry that implements the empty detection: it uses gates with a number of inputs that depends on the size of the counter. The authors of this article say

> Traditional counters, both asynchronous and synchronous, suffer either from slow speed (in the asynchronous case) since there is a carry or borrow propagation during counting, or from irregularity (in the synchronous case) due to the control gate of each stage being different.

The counter they design is used as a bracket counter. The object is matching opening and closing brackets in a string of brackets. Hence the authors are interested in testing whether the counter's value is zero. We show that asynchronous counters with empty detection are not necessarily slow due to carry or borrow propagation, and that their structure can be very regular.

Guibas and Liang [GL81] describe an implementation of a binary up-down counter by a transition table. They do not formally prove that this implementations is correct. The idea for the implementation is the same as for the counter design presented in Chapter 4 of this report. Their counter does not have full detection and requires a global clock signal. Guibas and Liang conjecture

that there is a correspondence between their *binary* up-down counter design and a stack design presented in the same paper. This conjecture is not explained. A correspondence between stack implementations and *unary* counters is much more obvious, as we show in Chapter 3. Finally, Guibas and Liang claim that their counter design can be made totally asynchronous. They do not give any justification for this claim.

In [JB88] an up-down $2N$-counter is implemented by N identical modules. The inputs for incrementing and decrementing the counter are broadcast to all modules. This results in an implementation where the new value of the counter can be read after a constant number of clock cycles after the counter has received an input, under the assumption that the input signal can be broadcast to all modules in a constant amount of time. In this report we look at counter designs in which the inputs are sent to one module only.

Oberman wrote a textbook on counting and counters [Obe81]. The book contains a large number of counter implementations of all kinds, among which up-down counters in Chapter 2. Some commercially available counters are discussed, and some simpler up-down counters are presented for educational purposes. Oberman does not discuss the performance of the implementations he presents.

Parhami proposes some up-down counter designs in [Par87]. His counters behave like modulo-N counters (see e.g. [EP92]) when the counter is incremented in its full state, and the value of the counter cannot be read. He also considers counters that can represent negative values. His binary counter design has the drawback that its specification assumes every cell to have two neighbors. The result is that in the implementation internal signals have to be generated to simulate the behavior of these (physically nonexistent) cells. Parhami's work is based on [GL81].

In the above articles no binary counter implementations are presented for general N. Usually the maximum count is a power of two minus one.

0.1 Designing Asynchronous Circuits

At present, most digital circuits use a global clock. They perform their computations in lockstep with the pulses of the clock. The correctness of the operation of such synchronous circuits depends on the delays of its elementary building blocks: they should be no longer than a fixed number of clock periods.

As circuits become larger and larger, distribution of the clock signal over the circuit becomes increasingly more difficult. This becomes apparent when looking at DEC's Alpha microprocessor

for example [BBB+92]. In asynchronous circuits there is no global clock signal. The goal of this report is to design asynchronous up-down counters of a special kind; we aim for delay-insensitive implementations, see e.g. [Ebe89, Udd86]. The correct operation of delay-insensitive circuits does not depend on bounds for the delays of its parts, i.e. delays of basic components and connection wires.

The absence of the need to distribute a global clock signal is not the only advantage of delay-insensitive circuits. The advantages include the following:

- Delay-insensitive circuits have better scalability than synchronous circuits. The reason for better scalability is that in scaling down the dimensions of a circuit (size and in synchronous circuits the clock period), the delays in wires do not scale down proportionally. For synchronous circuits this means that the timing constraints have to be verified again and that the clock frequency may have to be adjusted. For delay-insensitive circuits it is not a problem since the correctness of their operation does not depend on delays in connection wires.

- Delay-insensitive circuits have better modifiability than their synchronous counterparts. In a delay-insensitive circuit parts of the circuit can be redesigned, e.g. to obtain better performance or a smaller circuit, without affecting the correctness of the whole as long as the functionality of the redesigned part does not change. In synchronous designs this is not possible without verifying all the timing constraints as well.

- Asynchronous circuits possibly have a better performance than synchronous circuits. Synchronous circuits exhibit worst-case behavior, since the clock frequency has to be adjusted to the worst-case delay in computations. In asynchronous circuits, a computation step is made as soon as possible. So asynchronous circuits tend to exhibit average-case behavior.

- Asynchronous circuits possibly have a lower power consumption. A reason for the lower power consumption of asynchronous circuits is the absence of the clock, which ticks continuously, even when no computation is being executed. In [BBB+92] it is stated that in DEC's Alpha microprocessor chip, 17.3% of the power dissipation comes from the clock generation (clock distribution is not included in this number). Thus, circuits without a clock may have a lower power consumption. Moreover, absence of a clock may reduce cooling problems in large circuits.

- In delay-insensitive circuits metastable behavior does not cause errors. In synchronous circuits metastable behavior may cause errors when the behavior lasts longer than the clock

period. In delay-insensitive circuits the time it takes to reach a stable state only influences the performance, not the correctness of the circuit.

Currently a lot of research is devoted to designing and analyzing asynchronous circuits [vBKR+91, Bru91, Dil89, Ebe89, Gar93, Mar90, JU90, RMCF88, Sut89, Udd86]. There are still many problems to be solved, like analyzing performance measures of the designed circuits, liveness properties, and testing.

Some interesting performance criteria for asynchronous circuits are their area complexity, response time, and power consumption. The area complexity of a circuit can be analyzed by counting the number of basic elements in the circuit, provided that the circuit does not have long connection wires between its basic elements. We use this basic element count as a measure for the area complexity of our designs.

The response time of an asynchronous circuit can be defined as the delay between an output event of the circuit and the last input event that has to take place before that output can occur. A possible measure for the response time is the number of internal events that have to occur sequentially between an input to the circuit and the next output. For a class of asynchronous circuits this can be formalized by sequence functions [Zwa89, Rem87]. Sometimes counting events is not sufficient. We show this in a later chapter and propose a different way to estimate the response time.

Van Berkel was one of the first to identify low power consumption as an attractive property of asynchronous circuits [vB92]. He analyzes power consumption of his implementations by counting the number of communications of the handshake components in his circuits, see e.g. [vB93]. We estimate the power consumption of our counter designs by counting the average number of events per external event in our specification language.

0.2 Results of the Thesis

In this report we concentrate on the design of delay-insensitive up-down counters. The up-down counters are operated by a handshake protocol between the counter and its environment. The counters have inputs *up* and *down*. If an *up* is received from the environment, then the counter is incremented and an acknowledgement is sent to the environment — provided that the counter was not at its maximum value before the *up* was received. In the same way, the counter is decremented if a *down* is received — provided that the counter's value was greater than zero before the *down* was received. Each input is acknowledged in such a way that the counter's environment knows whether the counter is empty, full, or neither. The counter's value cannot be read by its environment.

The proposed implementations are analyzed with respect to the three performance criteria mentioned in the previous section. Since we do not present transistor implementations, we do not have exact numbers for the measures. We analyze the order of growth of the three performance criteria in terms of N. To indicate the order of growth we use Ω to indicate a lower bound, $\mathcal{O}$ to indicate an upper bound, and Θ to indicate a tight bound. A tight bound is both a lower bound and an upper bound.

In the following we design a number of up-down counter implementations. Some of them are similar to synchronous counter implementations found in the literature. These implementations show that a global clock is not required.

In the response time analysis of one of the counters we show that under certain assumptions sequence functions may not be adequate to determine the response time of asynchronous circuits. We analyze the response time under the weaker assumption that basic elements have variable, but bounded, delays. Ai definition for 'bounded response time' is proposed, to be used instead of 'constant response time' as defined in [Zwa89] when the weaker assumptions apply.

Furthermore, an up-down N-counter design is presented, for any N greater than zero, with optimal growth rates for area complexity, response time, and power consumption. The area complexity of this type of counter is logarithmic in its size, and the response time and power consumption are independent of its size. We can even prove that the response time is bounded according to our definition, which is a stronger result than proving constant response time.

0.3 Thesis Overview

The goal of this report is to examine possible delay-insensitive implementations for up-down N-counters. Before we can give any implementation, we need a specification. In Chapter 1 we give an overview of the formalism we use for describing the specifications and implementations, and we introduce the correctness concerns for implementations. In Chapter 2 we present two specifications for up-down counters and Chapters 3, 4, and 5 are devoted to designing and analyzing implementations. The implementation presented in Chapter 5 is a new one. It presents a method for designing up-down counters with constant power consumption, bounded response time, and logarithmic area complexity for any N. Chapter 6 contains some concluding remarks and suggestions for further research.

Chapter 1

Trace Theory and Delay-Insensitive Circuits

1.0 Introduction

The formalism we use to specify delay-insensitive circuits and to verify the correctness of implementations is introduced in [Ebe89]. Behaviors of circuits are described by strings of events, called *traces*, over a certain alphabet. This is formalized in trace theory. Our specifications are so-called commands. They are similar to regular expressions. Commands are a way to specify regular trace structures, a subclass of trace structures.

Implementations consist of sets of connected components. Each of the components in an implementation can be specified by a command. A set of components that implements a specification is called a decomposition of that specification.

This chapter contains an introduction to trace theory, the command language, and decomposition. A more extensive introduction can be found in [Ebe91].

1.1 Trace Theory and Commands

Components are specified by commands. Commands prescribe the possible sequences of communications between components and their environment.

The underlying semantics for commands is trace theory [vdS85, Kal86]. Every command is associated with a *(directed) trace structure*. A directed trace structure is a triple $\langle I, O, T \rangle$. I and O are

alphabets; I is the input alphabet and O the output alphabet. The input alphabet I represents the input terminals of the specified component and the output alphabet O represents its output terminals. The set of possible communication behaviors of the component is given by T; T is called the trace set of the trace structure. It is a set of sequences over $I \cup O$. A trace structure that describes the communication between a component and its environment has disjoint input and output alphabets. There are no bidirectional communication channels.

For a trace structure S, the input alphabet, output alphabet, and trace set are denoted by $\mathbf{i}S$, $\mathbf{o}S$, and $\mathbf{t}S$ respectively. Furthermore we define the alphabet of S as $\mathbf{i}S \cup \mathbf{o}S$. It is denoted by $\mathbf{a}S$. For command C, we use $\mathbf{i}C$, $\mathbf{o}C$, $\mathbf{a}C$, and $\mathbf{t}C$ to denote the input alphabet, output alphabet, alphabet, and trace set of the corresponding trace structure.

The command language consists of atomic commands and operators. Since commands are used to described the behavior of components we want the corresponding trace structures to have a non-empty trace set. Moreover, we want the trace set to be prefix-closed. This means that for any trace in the trace set, all its prefixes are in the trace set as well. A non-empty, prefix-closed trace structure is also called a process.

The atomic commands are $\emptyset$ ε, $b?$, $b!$, and $!b?$, where b is an element of a sufficiently large set of names. The atomic commands correspond to trace structures in the following way:

$$\begin{array}{rllll}
\emptyset & \langle\emptyset, & \emptyset, & \emptyset\rangle \\
\varepsilon & \langle\emptyset, & \emptyset, & \{\,\varepsilon\,\}\rangle \\
b? & \langle\{\,b\,\}, & \emptyset, & \{\,b\,\}\rangle \\
b! & \langle\emptyset, & \{\,b\,\}, & \{\,b\,\}\rangle \\
!b? & \langle\{\,b\,\}, & \{\,b\,\}, & \{\,b\,\}\rangle.
\end{array}$$

In this report we simply write b for the command $!b?$. This does not cause any confusion; if b occurs in a command, it is an atomic command, and not a symbol. There are seven operators defined on commands. For commands C and D and alphabet A we have:

$$\begin{array}{rcllll}
C; D & = & \langle \mathbf{i}C \cup \mathbf{i}D, & \mathbf{o}C \cup \mathbf{o}D, & (\mathbf{t}C)(\mathbf{t}D)\rangle \\
C \mid D & = & \langle \mathbf{i}C \cup \mathbf{i}D, & \mathbf{o}C \cup \mathbf{o}D, & \mathbf{t}C \cup \mathbf{t}D\rangle \\
*[C] & = & \langle \mathbf{i}C, & \mathbf{o}C, & (\mathbf{t}C)^*\rangle \\
\mathbf{pref}\, C & = & \langle \mathbf{i}C, & \mathbf{o}C, & \{\, t : (\exists\, u :: tu \in \mathbf{t}C) : t\,\}\rangle \\
C \lceil A & = & \langle \mathbf{i}C \cap A, & \mathbf{o}C \cap A, & \{\, t : t \in \mathbf{t}C : t \lceil A\,\}\rangle \\
C \parallel D & = & \langle \mathbf{i}C \cup \mathbf{i}D, & \mathbf{o}C \cup \mathbf{o}D, & \{\, t : t \in (\mathbf{a}C \cup \mathbf{a}D)^* \wedge t\lceil \mathbf{a}C \in \mathbf{t}C \wedge t \lceil \mathbf{a}D \in \mathbf{t}D : t\,\}\rangle.
\end{array}$$

The seventh operator is treated separately in the next section. The first three operations are well known from formal language theory. Juxtaposition and * denote concatenation and Kleene closure

of sets of strings. The **pref** operator constructs prefix-closed trace structures from its arguments. The projection of a trace t on an alphabet A, denoted by $t \upharpoonright A$, is trace t with all occurrences of symbols not in A deleted. We also use another notation to describe projection. For command C, we write

$$|[\, A :: C \,]|$$

as an alternative for

$$C \upharpoonright (\mathbf{a}C \setminus A).$$

This alternative notation has the advantage that the set of symbols to be hidden, A, appears before the command C. The symbols occurring in A can be interpreted as internal symbols of C. The trace set of the *weave* of two commands C and D, consists of the interleavings of the traces described by C and D. We stipulate that unary operators have higher binding power than binary operators. Of the binary operators weaving has the highest priority, followed by concatenation, and finally union.

With these operations every command corresponds to a regular trace structure. This means that components specified by commands have a finite number of states.

A result from trace theory that we use later is the following.

PROPERTY 1.1.0. For trace structures R and S, and alphabet A

$$(R \parallel S) \upharpoonright A = (R \upharpoonright A) \parallel (S \upharpoonright A) \quad \Leftarrow \quad \mathbf{a}R \cap \mathbf{a}S \subseteq A.$$

□

A proof can be found in [Kal86].

1.2 Extending Commands

We extend the command language to make it easier to specify finite state machines. Finite state machines can be expressed with the operators introduced so far, but it is not always easy. Introducing tail recursion will remedy this. Ebergen introduced tail recursion to specify finite state machines in [Ebe89]. The proofs of the claims made in this section can be found there. Defining the meaning of a tail-recursive specification requires some lattice theory. A general introduction to lattice theory can be found in [Bir67].

A function f is a *tail function* if it is defined by

$$f.R.i = \mathbf{pref}\ (\ |\ j : 0 \leq j < n : (S.i.j)(R.j))$$

for vector of trace structures R of length n, matrix of trace structures S of size $n \times n$, and for $0 \leq i < n$. Matrix S determines f uniquely. We assume that every row of S contains at least one non-empty trace structure.

Let I be the union of the input alphabets of the elements of S and let O be the union of the output alphabets of the elements of S. The set of all vectors of length n of non-empty, prefix-closed trace structures with input alphabet I and output alphabet O is denoted by $\mathcal{T}^n.I.O$. On $\mathcal{T}^n.I.O$ a partial order can be defined:

$$R \preceq R' \equiv (\forall\, i : 0 \leq i < n : \mathbf{t}(R.i) \subseteq \mathbf{t}(R'.i))$$

for trace structures R and R' in $\mathcal{T}^n.I.O$. With this partial order, $\mathcal{T}^n.I.O$ is a complete lattice with least element $\perp^n.I.O$, where $\perp.I.O = \langle I, O, \{\, \varepsilon\, \} \rangle$. The least upper bound operation on this lattice is pointwise union of vectors of trace structures and the greatest lower bound operation is pointwise intersection.

Moreover, for a matrix S that has a non-empty trace structure in each of its rows, the tail function f induced by S is continuous. This means that this tail function has a least fixpoint (Knaster-Tarski theorem). As usual, we denote the function that maps tail functions to their least fixpoints by μ.

The relation between finite state machines and fixpoints of tail functions can be described as follows. Consider a finite state machine with states $q.i$ for $0 \leq i < n$ and initial state $q.0$. If trace structure $S.i.j$ is non-empty, then there is a transition from $q.i$ to $q.j$ labeled with $S.i.j$. For $0 \leq k$ and $R.k \in \mathcal{T}^n.I.O$ we define:

$$\begin{aligned} R.0 &= \perp^n.I.O \\ R.k &= f.(R.(k-1)) \qquad \text{for } 1 \leq k. \end{aligned}$$

In words this means that $\mathbf{t}R.k.i$, for $0 \leq i < n$, is the prefix-closure of the union of the trace sets obtained by concatenating the k trace structures on a path of length k starting in state $q.i$. The trace structure corresponding to the finite automaton is $(\ |\ k : 0 \leq k : R.k.0)$ It can be proved that $\mu.f.i = (\ |\ k : 0 \leq k : R.k.i)$, so $\mu.f.0$ is the trace structure corresponding to the finite automaton.

For a trace structure defined by a tail function we can use fixpoint induction to prove properties of the trace structure. We use this in Chapter 5. Here we formulate the fixpoint induction theorem for the lattice of vectors of trace structures and tail functions only.

THEOREM 1.2.0. (Fixpoint induction theorem)
Let f be a (continuous) tail function and let P be an inductive predicate such that $P.(\perp^n.I.O)$ holds and f maintains P, i.e.,

$$P.R \quad \Rightarrow \quad P.(f.R),$$

for $R \in \mathcal{T}^n.I.O$. Then $P.(\mu.f)$ holds. □

A predicate is inductive if

$$(\forall\, R : R \in V : P.R) \quad \Rightarrow \quad P.(\bigsqcup R : R \in V : R),$$

for any non-empty, directed subset V of $\mathcal{T}^n.I.O$ (a directed set or chain in a partial order is a set of which the elements are totally ordered).

A tail function can also be specified by a matrix of commands instead of trace structures. This is essentially the way in which we use tail recursion.

As a small example we give a specification using tail recursion and use fixpoint induction to show that this specification can be simplified.

Tail function $f \in \mathcal{T}^4.\{\,a\,\}.\{\,b\,\}$ is defined by the matrix

$$\begin{bmatrix} \emptyset & a? & \emptyset & \emptyset \\ \emptyset & \emptyset & b! & \emptyset \\ \emptyset & \emptyset & \emptyset & a? \\ b! & \emptyset & \emptyset & \emptyset \end{bmatrix}$$

To make tail-recursive specifications more readable, we will use the following format in the rest of this report:

$$\begin{aligned} S.0 &= \mathbf{pref}\ (a?; S.1) \\ S.1 &= \mathbf{pref}\ (b!; S.2) \\ S.2 &= \mathbf{pref}\ (a?; S.3) \\ S.3 &= \mathbf{pref}\ (b!; S.0). \end{aligned}$$

We can use fixpoint induction to prove a property of this specification. Predicate P is defined by

$$P.R \equiv (R.0 = R.2),$$

for $R \in \mathcal{T}^4.\{\,a\,\}.\{\,b\,\}$. P is an inductive predicate: for any subset V of $\mathcal{T}^n.\{\,a\,\}.\{\,b\,\}$ we have

$$
\begin{array}{ll}
 & P.(\bigsqcup R : R \in V : R) \\
\equiv & \quad \{ \text{ Definition of } P \ \} \\
 & (\bigsqcup R : R \in V : R).0 = (\bigsqcup R : R \in V : R).2 \\
\equiv & \quad \{ \text{ Definition of } \bigsqcup \ \} \\
 & (\mid R : R \in V : R.0) = (\mid R : R \in V : R.2) \\
\Leftarrow & \quad \{ \text{ Leibniz } \} \\
 & (\forall R : R \in V : (R.0 = R.2)) \\
\equiv & \quad \{ \text{ Definition of } P \ \} \\
 & (\forall R : R \in V : P.R).
\end{array}
$$

All predicates that express that two components of a vector of trace structures are the same are inductive predicates. For example, predicate Q, defined by:

$$Q.R \equiv (R.1 = R.3)$$

is inductive too.

It is obvious that both $P.(\bot^4.\{a\}.\{b\})$ holds and that $P.(\bot^4.\{a\}.\{b\})$ holds: all components are equal to $\bot.\{a\}.\{b\}$.

Next we show that f maintains $P \wedge Q$. For any R such that $Q.R$ holds we derive:

$$
\begin{array}{ll}
 & f.R.0 \\
= & \quad \{ \text{ Definition of } f \ \} \\
 & \textbf{pref}\ (a?; R.1) \\
= & \quad \{ \ Q.R \ \} \\
 & \textbf{pref}\ (a?; R.3) \\
= & \quad \{ \text{ Definition of } f \ \} \\
 & f.R.2.
\end{array}
$$

Similarly we can derive that $f.R.1 = f.R.3$ if $P.R$ holds. Thus, by the Fixpoint Induction Theorem, we conclude that $\mu.f.0$ is equal to $\mu.f.2$ and that $\mu.f.1$ is equal to $\mu.f.3$. This means that

$$
\begin{array}{l}
\mu.f.0 = \textbf{pref}\ (a?; \mu.f.1) \\
\mu.f.1 = \textbf{pref}\ (b!; \mu.f.0).
\end{array}
$$

So $(\mu.f.0, \mu.f.1)$ is equal to the least fixpoint of the tail function defined by

$$
\begin{array}{l}
S.0 = \textbf{pref}\ (a?; S.1) \\
S.1 = \textbf{pref}\ (b!; S.0).
\end{array}
$$

The fixpoint operator μ has some nice properties. A property we use is that $\mu.f \upharpoonright A$ can be obtained by removing all symbols not in A from the defining equations for f.

1.3 Basic Components

In this section we discuss a number of basic components that can be used to implement larger components. We only introduce the components that are used in this report.

The first component is the WIRE. A WIRE component has one input and one output. Its communication behaviors are all sequences in which input and output alternate, starting with an input, if any.

An IWIRE component has one input and one output as well. Its communication behaviors are alternations of inputs and outputs as well, but starting with an output, if any. IWIREs can be used for starting computations.

The MERGE component is a component with two inputs and one output. Again, inputs and outputs alternate in the communication behaviors. After one of the two inputs is received, an output is produced.

A TOGGLE has one input and two outputs. Communication behaviors of the TOGGLE start with an input, if any, and inputs and outputs alternate. The outputs are produced alternatingly at each of the two output terminals.

The next element is the 1-by-1 JOIN, also called JOIN. It has two inputs and one output and is used for synchronization. After both inputs have been received, an output is produced, and this behavior repeats. We also use an initialized JOIN. An initialized JOIN behaves as a join for which one input event has already taken place.

Table 1.0 contains the commands and schematics for the introduced basic components.

1.4 Decomposition

A decomposition of a specification is a set of components that implements the behavior of the specification according to four correctness concerns. Before we introduce the correctness concerns of decomposition, we describe how specifications should be interpreted.

Earlier we saw that every specification corresponds to a regular trace structure with disjoint input and output alphabets. The alphabet of the trace structure represents the connections between the component and its environment, the terminals of the component. The input alphabet consists of the terminals at which the environment may produce events. The output alphabet consists of the terminals at which the component may produce events.

Basic component	**Command**	**Schematic**
WIRE$(a;b)$	**pref** $*[a?;b!]$	$a?$ → $b!$
IWIRE$(a;b)$	**pref** $*[b!;a?]$	$a?$ → $b!$
MERGE$(a,b;c)$	**pref** $*[a?;c! \mid b?;c!]$	$a?$, $b?$ → M → $c!$
TOGGLE$(a;b,c)$	**pref** $*[a?;b!;a?;c!]$	$a?$ → $b!$, $c!$
JOIN$(a,b;c)$	**pref** $*[(a?\|b?);c!]$	$a?$, $b?$ → $c!$

Table 1.0: Basic components.

A behavior of the component describes the order in which events occur at the terminals. The trace set of the trace structure describing the component specifies which behaviors can occur. Initially the sequence of events that have occurred is empty. Consider a communication behavior t such that ta is also a valid communication behavior. If a is an input symbol, this means that the environment may produce an a event after behavior t. If a is an output symbol, this means that the component may produce an a event after behavior t has occurred.

We note two things. First, an input or output is not guaranteed to occur, even though it might be the only event that can occur after a certain behavior. Second, our specifications prescribe the behavior of the environment as well as of the component. This means that correct operation of the component is guaranteed only in the case that the environment behaves as specified.

In the above we mentioned the environment of a component a number of times. The outputs of the component are the inputs of its environment and the inputs of the component are the outputs of the environment. We can turn the environment of component S into a component $\bar{S}$ by interchanging the input and output alphabets.

DEFINITION 1.4.0. Let S be a trace structure. Its reflection $\bar{S}$ is defined by $\bar{S} = \langle \mathbf{o}S, \mathbf{i}S, \mathbf{t}S \rangle$. □

In a decomposition of a component S into components $T.i$ for $1 \le i < n$, the network produces the outputs as specified by S. Its environment produces the inputs as specified by S. Equivalently we can say that the outputs of $\bar{S}$ are the inputs to the network. Therefore we consider the network

consisting of $\bar{S}, T.1, \ldots, T.(n-1)$ in defining decomposition formally.

DEFINITION 1.4.1. Let $1 < n$. Component S can be decomposed into the components $T.1, \ldots, T.n$, denoted by

$$S \rightarrow (i : 1 \leq i < n : T.i),$$

if Conditions (0), (1), (2), and (3) below are satisfied.

Let $T.0$ be the reflection of S and define $W = (\,\|\, i : 0 \leq i < n : T.i)$.

(0) The network is closed:

$$(\bigcup i : 0 \leq i < n : \mathbf{o}(T.i)) = (\bigcup i : 0 \leq i < n : \mathbf{i}(T.i)).$$

(1) There is no output interference:

$$(\forall\, i, j : 0 \leq i < j < n : \mathbf{o}(T.i) \cap \mathbf{o}(T.j) = \emptyset).$$

(2) There is no computation interference:

$$t \in \mathbf{t}W \wedge x \in \mathbf{o}(T.i) \wedge tx \restriction \mathbf{a}(T.i) \in \mathbf{t}(T.i) \quad \Rightarrow \quad tx \in \mathbf{t}W,$$

for any trace t, symbol x, and index i with $0 \leq i < n$.

(3) The set of network behaviors is complete:

$$\mathbf{t}W \restriction \mathbf{a}S = \mathbf{t}S.$$

□

Condition (0) states that there are no dangling inputs and outputs. Condition (1) states that no two outputs are connected to each other. These two conditions are structural conditions on the network. Conditions (2) and (3) are behavioral conditions. The former states that, after any behavior of the network, all outputs that can be produced by any of the components, can be accepted by the components for which that symbol is an input. The last condition ensures that all behaviors of the specification S may occur in the implementation. This means that an implementation that accepts all inputs, but never produces any output, is not acceptable. The last condition does not guarantee,

however, that after a certain behavior a specified output or input actually occurs. It merely rules out implementations where this specified output or input is guaranteed never to occur.

Verifying absence of computation interference formally is often very laborious and the proofs are not very readable. In the correctness proofs of the counter implementations we verify absence of computation interference informally.

An automatic verifier for decompositions has been developed and is described in [EG93b]. This verification tool, called VERDECT has a slightly more restrictive syntax than the command language described here. But since there is a standard way to specify finite state machines in this restrictive syntax, all regular trace structures can be described.

A theorem that is useful in verifying decompositions, is the *Substitution Theorem.* It allows hierarchical decomposition of components.

THEOREM 1.4.2. (Substitution Theorem)
Let $S.0, S.1, S.2, S.3$, and T be components such that $S.0 \rightarrow (S.1, T)$ and $T \rightarrow (S.2, S.3)$. Then $S.0 \rightarrow (S.1, S.2, S.3)$ if $(\mathbf{a}(S.0) \cup \mathbf{a}(S.1)) \cap (\mathbf{a}(S.2) \cup \mathbf{a}(S.3)) = \mathbf{a}T$. □

The condition on the alphabets of the components states that the decompositions of $S.0$ and T only have symbols from $\mathbf{a}T$ in common. By renaming the internal symbols in the decomposition of T this condition can always be satisfied.

A proof of the Substitution Theorem can be found in [Ebe89]. The theorem can be generalized to decompositions with larger numbers of components.

The following lemma is useful for showing that the network behaviors in the decomposition

$$S \rightarrow (T.1, T.2)$$

are complete.

LEMMA 1.4.3. Let S, $T.1$ and $T.2$ be non-empty, prefix-closed trace structures. Then

$$\mathbf{t}(\bar{S} \parallel T.1 \parallel T.2) \upharpoonright \mathbf{a}S = \mathbf{t}S \quad \Leftarrow \quad \mathbf{t}S \subseteq \mathbf{t}(T.1 \parallel T.2) \upharpoonright \mathbf{a}S$$

□

PROOF. We derive:

$$\mathbf{t}(\bar{S} \parallel T.1 \parallel T.2) \upharpoonright \mathbf{a}S = \mathbf{t}S$$

$\equiv$ { Antisymmetry of $\subseteq$; definition of reflection }

$\mathbf{t}(S \parallel T.1 \parallel T.2)\lceil\mathbf{a}S \subseteq \mathbf{t}S \ \wedge\ \mathbf{t}S \subseteq \mathbf{t}(S \parallel T.1 \parallel T.2)\lceil\mathbf{a}S$

$\equiv$ { Definition of $\parallel$ }

$\mathbf{t}S \subseteq \mathbf{t}(S \parallel T.1 \parallel T.2)\lceil\mathbf{a}S$

$\Leftarrow$ { Property 1.1.0 }

$\mathbf{t}S \subseteq \mathbf{t}((S\lceil\mathbf{a}S) \parallel ((T.1 \parallel T.2)\lceil\mathbf{a}S)) \ \wedge\ (\mathbf{a}(T.1) \cup \mathbf{a}(T.2)) \cap \mathbf{a}S \subseteq \mathbf{a}S$

$\equiv$ { Set calculus }

$\mathbf{t}S \subseteq \mathbf{t}((S\lceil\mathbf{a}S) \parallel ((T.1 \parallel T.2)\lceil\mathbf{a}S))$

$\equiv$ { $\mathbf{t}S\lceil\mathbf{a}S = \mathbf{t}S$ }

$\mathbf{t}S \subseteq \mathbf{t}(S \parallel ((T.1 \parallel T.2)\lceil\mathbf{a}S))$

$\Leftarrow$ { Definition of $\parallel$ }

$\mathbf{t}S \subseteq \mathbf{t}(T.1 \parallel T.2)\lceil\mathbf{a}S.$

$\square$

1.5 Delay-Insensitivity and DI decomposition

Decomposition as introduced in the previous section corresponds to designing *speed-independent* implementations. The correct operation of speed-independent circuits does not depend on delays in components, but it may depend on delays in connection wires. To check whether a network of components is a delay-insensitive implementation of a specification, we define DI decomposition. In DI decomposition the delays of connection wires are taken into account.

Taking delays of connection wires into account is done by replacing the components in a decomposition by a version with renamed terminals and by introducing WIRE components that have one of the new terminals as input and the corresponding old terminal as output or vice versa.

Formally this is done as follows. For a network of components $T.i$, with $1 \leq i < n$, we define $enc.(T.i)$ by renaming every symbol a to a_i. Furthermore we define $wires.(T.i)$ as the set of WIRE components WIRE$(a_i; a)$ for a an output of $T.i$ and WIRE$(a; a_i)$ for a an input of $T.i$. Now we say that the network consisting of components $T.i$, with $1 \leq i < n$ is a DI decomposition of S, denoted by $S \overset{DI}{\to} (i : 0 \leq i < n : T.i)$, if and only if

$$S \to (i : 0 \leq i < n : enc.(T.i), wires.(T.i)).$$

Verifying DI decomposition is more laborious than verifying (speed-independent) decomposition. However, if the components in a network are all DI components, then the two forms of decomposition are equivalent for that network. A component C is a DI component if it can be decomposed into its enclosure and the corresponding WIREs, that is,

$$C \rightarrow (enc.C, wires.C).$$

1.6 Sequence Functions

As mentioned before, sequence functions can be used to obtain a measure for the response time of implementations. For a network of components, the response time is defined as the worst-case delay between an output from the network and the last input on which that output depends. We say that for a given trace structure an output event *depends* on an input event, if the input event has to precede that output according to the trace structure.

Sequence functions map events of *cubic* processes onto natural numbers. They are used to describe the ordering of events of such processes and to estimate response times. We present a short introduction to sequence functions. Details can be found in [Zwa89].

Let X be a process. If

$$\begin{aligned} &(\forall\, t, a, b, c : tac \in \mathbf{t}X \wedge tbc \in \mathbf{t}X \wedge a \neq b : tc \in \mathbf{t}X) \\ \wedge\; &(\forall\, t, a, b : ta \in \mathbf{t}X \wedge tb \in \mathbf{t}X \wedge a \neq b : tab \in \mathbf{t}X \wedge tba \in \mathbf{t}X \wedge [tab] = [tba]), \end{aligned}$$

then X is said to be a *cubic* process.

The set of occurrences of a process X is the set $\{\, (a, \ell.(t \restriction a)) \mid ta \in \mathbf{t}X \,\}$. It is denoted by $\mathbf{occ}.X$. A cubic process X can be characterized by a partial order $\ll$ on the set of occurrences $\mathbf{occ}.X$. This partial order is given by

$(a, i) \ll (b, j)$ if and only if in each trace of X that contains occurrence (b, j), (a, i) precedes (b, j),

for $(a, i), (b, j) \in \mathbf{occ}.X$.

Let X be a cubic process determined by partial order $\ll$ and let σ be a function from $\mathbf{a}X \times \mathbb{N}$ to $\mathbb{N}$ satisfying

$$(\forall\, a, b, i, j : (a, i) \ll (b, j) : \sigma.a.i < \sigma.b.j).$$

Then σ restricted to $\mathbf{occ}.X$ is a sequence function for X.

For process X and sequence function σ for X, the occurrence (a, i) may be interpreted as the moment in time at which (a, i) takes place. With this interpretation σ can be seen as a possible (synchronous) behavior of the component described by X.

For a given network we can define a sequence function for the weave of its components, assuming that the resulting trace structure is cubic. Doing this, we make the implicit assumption that all delays in connection wires are zero. Wire delays can be included by introducing explicit WIRE components in the network.

A network of components is said to have *constant response time* if there is a sequence function σ for the weave of the components W such that $\sigma.b.j - \sigma.a.i$ is bounded from above by a constant, for (b, j) an output occurrence and (a, i) the last input occurrence (the last one according to σ) such that $(a, i) \ll (b, j)$.

Chapter 2

Formal Specification of Up-Down Counters

2.0 Introduction

In this chapter we present two formal specifications of up-down counters. Both specifications specify an up-down counter that counts in the range from 0 to N for some N larger than zero, with both bounds included. We refer to such a counter as an *up-down N-counter* or *N-counter*.

The second of these specifications will be the starting point for the implementations in the rest of this thesis. As was mentioned before, we use commands as our specification language.

2.1 An Up-Down Counter with an *ack*-*nak* Protocol

In the first up-down counter specification we use four terminals, two of which are inputs to the counter, and two of which are outputs. A behavior of the counter is a sequence of alternating inputs and outputs.

The two input terminals are *up* and *down*. A transition at terminal *up* indicates that the counter should be incremented by one. A transition at terminal *down* indicates that the counter should be decremented by one.

The output terminals are called *ack* and *nak*. A transition at terminal *ack* indicates that the most recent input has been processed by the counter and that it is ready to receive the next input.

A transition at the other output terminal, *nak*, indicates that the most recent input has *not* been processed. Hence it does not influence the current count.

We define the *current count* of the counter as the number of *up* transitions that have been processed minus the number of *down* transitions that have been processed. Sometimes we refer to the current count simply by *count*.

A transition at *nak* occurs in two cases only. The first case in which it occurs is when the current count is N and the last input received is an *up*. The second case is when the current count is 0 and the last input received is a *down*.

DEFINITION 2.1.0. For $0 < N$ the up-down N-counter with *ack*-*nak* protocol is specified as element 0 of the least fixpoint of the following equations in $S = (S.0, \ldots, S.N)$:

$$\begin{aligned} S.0 &= \mathbf{pref}\ (up?;\, ack!;\, S.1 \mid down?;\, nak!;\, S.0) \\ S.i &= \mathbf{pref}\ (up?;\, ack!;\, S.(i+1) \mid down?;\, ack!;\, S.(i-1)) \qquad \text{for } 0 < i < N \\ S.N &= \mathbf{pref}\ (up?;\, nak!;\, S.N \mid down?;\, ack!;\, S.(N-1)). \end{aligned}$$

□

With this specification, the last output produced does in general not give any information about the state of the counter. If the counter sends an *ack* output to its environment, the counter may be empty, full, or neither. In the specification presented in the next section, the last output produced by the counter contains information about its state.

2.2 An Up-Down Counter with an *empty*-*ack*-*full* Protocol

The up-down counter presented in this section has five terminals, two input and three output terminals. The input terminals are the same as in the previous specification.

The output terminals are *empty*, *ack*, and *full*. The N-counter sends an *empty* signal after receiving an input that makes the current count zero. A *full* signal is sent after the receipt of an input that makes the current count N. All other inputs that would not cause the current count to go beyond the counting range $[0..N]$ are acknowledged by an *ack* event.

DEFINITION 2.2.0. For $0 < N$ we define $UDC.N$, the up-down N-counter with *empty*-*ack*-*full* protocol, as follows. First we specify $UDC.1$:

$$UDC.1 = \mathbf{pref}\ *[up?;\, full!;\, down?;\, empty!] \parallel (ack!)^0,$$

where $(ack!)^0$ specifies trace structure $\langle \emptyset, \{ ack \}, \{ \varepsilon \} \rangle$. Having weavand $(ack!)^0$ ensures that the trace structure corresponding to $UDC.1$ has ack in its output alphabet.

For $1 < N$ we define $UDC.N$ as element zero of the least fixpoint of

$$
\begin{array}{lcl}
S.0 & = & \mathbf{pref}\ (up?; ack!; S.1) \\
S.i & = & \begin{cases}
\mathbf{pref}\ (up?; ack!; S.(i+1) \mid down?; empty!; S.(i-1)) & \text{for } i = 1 \text{ and } 2 < N \\
\mathbf{pref}\ (up?; ack!; S.(i+1) \mid down?; ack!; S.(i-1)) & \text{for } 1 < i < N-1 \\
\mathbf{pref}\ (up?; full!; S.(i+1) \mid down?; ack!; S.(i-1)) & \text{for } 1 < i \text{ and } i = N-1 \\
\mathbf{pref}\ (up?; full!; S.(i+1) \mid down?; empty!; S.(i-1)) & \text{for } i = 1 \text{ and } N = 2
\end{cases} \\
S.N & = & \mathbf{pref}\ (down?; ack!; S.(N-1)).
\end{array}
$$

□

From the definition of $UDC.N$ it is obvious that our specifications put constraints on the behavior of the environment. For the counter to function correctly, its environment should adhere to the specified protocol. For example, if the counter is in its initial state, the environment is not allowed to send an *up* signal followed by two *down* signals. This means that for this specification the current count after a certain behavior is simply the number of *up*'s minus the number of *down*'s.

The counter specified in Definition 2.1.0 allows any sequence of inputs, but even there some constraints are put on the environment's behavior. The environment is not allowed to send two inputs to the counter without an *ack* or *nak* happening between the two inputs.

Note that it is not hard to build an up-down counter as specified in the previous section, using an implementation for the specification given in this section. A cell implementing the following behavior could be added:

$$
\begin{aligned}
S.0 &= \mathbf{pref}\ (up?; sup!; S.3 \mid down?; nak!; S.0) \\
S.1 &= \mathbf{pref}\ (up?; sup!; S.3 \mid down?; sdown!; S.3) \\
S.2 &= \mathbf{pref}\ (up?; nak!; S.2 \mid down?; sdown!; S.3) \\
S.3 &= \mathbf{pref}\ (sempty?; ack!; S.0 \mid sack?; ack!; S.1 \mid sfull?; ack!; S.2).
\end{aligned}
$$

The terminals of the *UDC* implementation should be renamed to *sup*, *sdown*, *sempty*, *sack*, and *sfull*. If the current count of the *UDC* implementation is zero and the environment sends a *down* input, then a *nak* is sent to the environment. If the current count of the *UDC* implementation is at its maximum and an *up* input is received from the environment, a *nak* is sent as well. In all other cases, the input is propagated to the *UDC* implementation. The type of acknowledgement received from this counter determines the behavior of the cell upon receiving the next input.

Building a *UDC.N* from an counter with *ack-nak* protocol is less straightforward.

From now on, we use the words *(up-down) N-counter* to refer to the counter specified in this section.

Chapter 3

Some Simple Designs

3.0 Introduction

We design two implementations for *UDC.N*. The term *implementation* is used here to denote a network of components that is a decomposition of the specification (as defined in Chapter 1) and in which each of the components has a number of states that is independent of the number of states of the specification. We do not design an implementation at the gate level.

The step from a specification with a variable number of states to a network of a variable number of components with a constant number of states each, is the most important step in the design on the way to a low-level implementation (e.g. gate-level implementation). The decomposition of specifications with a fixed, finite number of states into basic components or gates has been studied extensively [Chu87, RMCF88, LKSV91, MSB91, ND91, DCS93].

In this chapter two implementations for *UDC.N* are presented and proved correct, using the four correctness criteria described in Chapter 1. Furthermore a performance analysis of the two implementations is given.

All our implementations, in this and in following chapters, consist of linear arrays of cells. There are two types of cells in such an array: the end cell and the other cells. The end cell has only five terminals for communication with its environment. The other cells have ten terminals: five for communication with their left environment, and five for communication with their right environment. Figure 3.0 depicts block diagrams for the two types of cells.

Based on Figure 3.0 we refer to communications at terminals *up*, *down*, *empty*, *ack*, and *full* as communications with the left environment. Communications at the other terminals are referred to

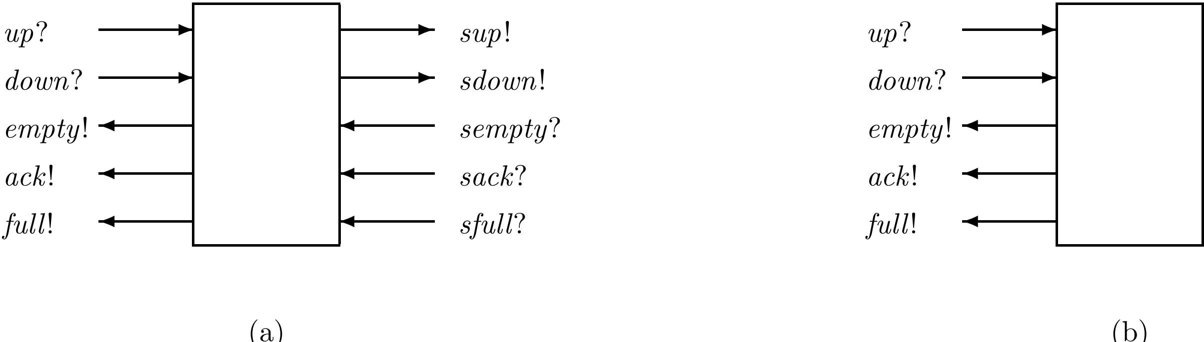

Figure 3.0: (a) block diagram for the general cell; (b) block diagram for the end cell.

as communications with the right environment or subcomponent. The terminals for communication with the subcomponent start with an *s* as a mnemonic reminder.

In an implementation of an up-down counter we consider the cells to be numbered starting at zero. The leftmost cell is numbered zero.

3.1 Unary Implementations

In unary counter implementations the current count of the counter is the sum of the internal counts of the cells in the arrays. Denoting the internal count of cell i by $c.i$, the current count is

$$(\Sigma\, i : 0 \leq i < N : c.i).$$

Unary counter implementations may be useful when the maximum count is small. For large counting ranges they are not particularly useful, since the number of cells needed to implement an N-counter is $\Omega.N$.

There is a close relation between unary implementations of up-down counters and the control structures for stack implementations. A unary counter implementation can be seen as a stack in which only the number of elements on the stack is relevant, not the actual data values. A number of (control structures for) delay-insensitive stack implementations have been proposed [Mar90, JU91]. Here, a very simple implementation is presented. The response time is not very good, but the specified cells have only a few states.

3.1.0 Specification of the Cells

Our unary implementation of the N-counter consists of an array of N cells. Each cell can be either empty or full. We only allow a prefix of the array of cells to be full, i.e., if a cell is empty, all its successors are empty. Marking all full cells with a 1, the current count is represented by the string of 1's interpreted as a unary number.

DEFINITION 3.1.0. The end cell of the unary N-counter described above is simply a $UDC.1$. For the other cells, let $C^0 = (C^0.0, C^0.1, C^0.1', C^0.2, C^0.2', C^0.3)$ be the least fixpoint of the following equations in S:

$$
\begin{aligned}
S.0 &= \textbf{pref}\ (up?; ack!; S.1) \\
S.1 &= \textbf{pref}\ (up?; sup!; S.1' \mid down?; empty!; S.0) \\
S.1' &= \textbf{pref}\ (sack?; ack!; S.2 \mid sfull?; full!; S.3) \\
S.2 &= \textbf{pref}\ (up?; sup!; S.1' \mid down?; sdown!; S.2') \\
S.2' &= \textbf{pref}\ (sack?; ack!; S.2 \mid sempty?; ack!; S.1) \\
S.3 &= \textbf{pref}\ (down?; sdown!; S.2').
\end{aligned}
$$

Now $C^0.0$ specifies the behavior of the cell. □

VERDECT shows that the state graph corresponding to command $C^0.0$ has twelve states. The components of C^0 in Definition 3.1.0 can be considered a subset of those states. We often refer to the elements of this subset as the *named states*. A specification of an up-down counter cell based on the stack design in [JU91] has 38 states; our specification of a counter cell based on Martin's lazy stack in [Mar90] has sixteen states. The counter cell based on [JU91] may seem to be unnecessarily complicated, but it has a better response time and the number of cells needed to implement an N-counter is half the number of $C^0.0$ cells or cells based on the lazy stack needed for an N-counter.

To clarify the behavior of the $C^0.0$ cell we give some assertions that hold in $C^0.0$, $C^0.1$, $C^0.2$, and $C^0.3$ (in these four states the counter is waiting for input from its environment).

- $C^0.0$: the current count is zero,
- $C^0.1$: the current count is one,
- $C^0.2$: the current count is larger than one and smaller than the maximum count of the cell and its subcounter,
- $C^0.3$: the current count is equal to the maximum count of the cell and its subcounter.

3.1.1 Correctness of the Implementation

Proving that the implementation presented in the previous section satisfies the specification requires proving that for all N larger than zero

$$UDC.N \rightarrow ((i : 0 \leq i < N-1 : s^i C^0.0), s^{N-1}\,UDC.1)$$

where $s^i C^0.0$ is $C^0.0$ with all terminals prefixed by i s's.

We give a proof by induction on N. The basic step is easy: the proof obligation is

$$UDC.1 \rightarrow (UDC.1),$$

which is a property of decomposition.

For the inductive step we reduce the proof obligation by applying the Substitution Theorem. The remaining proof obligation is:

$$UDC.(N+1) \rightarrow (C^0.0, s\,UDC.N).$$

Proving that this simplification is justified requires the careful verification of the alphabet conditions of the Substitution Theorem, but the proof is not very hard.

Verifying the two structural conditions for the decomposition of $UDC.(N+1)$ into a $C^0.0$ cell and a $UDC.N$ is easy. The network consisting of $\overline{UDC.(N+1)}$, $C^0.0$, and $UDC.N$ is closed and there is no output interference. We concentrate on the behavioral conditions. First we verify that the network behaviors are complete, and then we look at absence of computation interference.

The cases $N = 1$ and $N > 1$ are treated separately. The reason is that these two cases were also distinguished in the specification of $UDC.N$. We present the proof for the case $N > 1$ only.

First we construct the set of network behaviors $\mathbf{t}(\overline{UDC.(N+1)} \parallel C^0.0 \parallel s\,UDC.N)$. We do this by constructing a set of defining equations for $C^0.0 \parallel s\,UDC.N$, and then looking at the weave of the result and $\overline{UDC.(N+1)}$.

The set of named states for $C^0.0 \parallel s\,UDC.N$ will correspond to a subset of the Cartesian product of the named states of the weavands. The starting state corresponds to the product of the two starting states of the weavands. Then the other states are obtained by looking at the possible events in the corresponding states of the weavands. An event at a terminal that occurs in both weavands, is possible in the weave only if it is possible in both weavands. An event at a terminal that occurs in only one of the weavands, is possible in the weave if it is possible in that weavand. The named

states of the weave are numbered according to the numbers of the states of the weavands to which they correspond.

The defining equations for $C^0.0 \parallel s\,UDC.N$ are

$$
\begin{array}{lcl}
S.0.0 & = & \mathbf{pref}\ (up?;\, ack!;\, S.1.0) \\
S.1.0 & = & \mathbf{pref}\ (up?;\, sup;\, sack;\, ack!;\, S.2.1 \\
 & & \qquad |\, down?;\, empty!;\, S.0.0) \\
S.2.i & = & \left\{ \begin{array}{ll}
\mathbf{pref}\ (up?;\, sup;\, sack;\, ack!;\, S.2.(i+1) & \\
\qquad |\, down?;\, sdown;\, sempty;\, ack!;\, S.1.(i-1)) & \text{for } i = 1 \text{ and } 2 < N \\
\mathbf{pref}\ (up?;\, sup;\, sack;\, ack!;\, S.2.(i+1) & \\
\qquad |\, down?;\, sdown;\, sack;\, ack!;\, S.2.(i-1)) & \text{for } 1 < i < N-1 \\
\mathbf{pref}\ (up?;\, sup;\, sfull;\, full!;\, S.3.(i+1) & \\
\qquad |\, down?;\, sdown;\, sack;\, ack!;\, S.2.(i-1)) & \text{for } 1 < i \text{ and } i = N-1 \\
\mathbf{pref}\ (up?;\, sup;\, sfull;\, full!;\, S.3.(i+1) & \\
\qquad |\, down?;\, sdown;\, sempty;\, ack!;\, S.1.(i-1)) & \text{for } i = 1 \text{ and } N = 2
\end{array} \right. \\
S.3.N & = & \mathbf{pref}\ (down?;\, sdown;\, sack;\, ack!;\, S.2.(N-1)).
\end{array}
$$

As before, $C^0.0 \parallel s\,UDC.N$ is the first component of the least fixpoint of this set of equations. Now we could apply the same construction to these equations and the reflection of $UDC.(N+1)$. But if we hide the internal symbols in the equations for $C^0.0 \parallel s\,UDC.N$, thus obtaining the equations for $|[\, \mathbf{a}(s\,UDC.N) :: C^0.0 \parallel s\,UDC.N\,]|$, we get:

$$
\begin{array}{lcl}
S.0.0 & = & \mathbf{pref}\ (up?;\, ack!;\, S.1.0) \\
S.1.0 & = & \mathbf{pref}\ (up?;\, ack!;\, S.2.1 \\
 & & \qquad |\, down?;\, empty!;\, S.0.0) \\
S.2.i & = & \left\{ \begin{array}{ll}
\mathbf{pref}\ (up?;\, ack!;\, S.2.(i+1) & \\
\qquad |\, down?;\, ack!;\, S.1.(i-1)) & \text{for } i = 1 \text{ and } 2 < N \\
\mathbf{pref}\ (up?;\, ack!;\, S.2.(i+1) & \\
\qquad |\, down?;\, ack!;\, S.2.(i-1)) & \text{for } 1 < i < N-1 \\
\mathbf{pref}\ (up?;\, full!;\, S.3.(i+1) & \\
\qquad |\, down?;\, ack!;\, S.2.(i-1)) & \text{for } 1 < i \text{ and } i = N-1 \\
\mathbf{pref}\ (up?;\, full!;\, S.3.(i+1) & \\
\qquad |\, down?;\, ack!;\, S.1.(i-1)) & \text{for } i = 1 \text{ and } N = 2
\end{array} \right. \\
S.3.N & = & \mathbf{pref}\ (down?;\, ack!;\, S.2.(N-1)).
\end{array}
$$

It is easily verified that this is another way to write down the defining equations for $UDC.(N+1)$. Thus we have established

$$\mathbf{t}(C^0.0 \parallel s\,UDC.N) \upharpoonright \mathbf{a}(UDC.(N+1)) = \mathbf{t}(UDC.(N+1)).$$

We can now apply Lemma 1.4.3 to obtain

$$\mathbf{t}(\overline{UDC.(N+1)} \parallel C^0.0 \parallel s\,UDC.N) \upharpoonright \mathbf{a}(UDC.(N+1)) \;=\; \mathbf{t}(UDC.(N+1)).$$

To prove that there is no computation interference, we consider the communication between $\overline{UDC.(N+1)}$ and $C^0.0$ and between $s\,UDC.N$ and $C^0.0$ separately.

First we check whether $C^0.0$ causes computation interference in $\overline{UDC.(N+1)}$. In Definition 2.2.0 we see that after each *up* event, both *full* and *ack* are enabled. In Definition 3.1.0 we see that after each *up* event, the next communication with the environment is either a *full* or an *ack*. Similarly we have that after a *down* event the next output to the environment is one of *empty* or *ack*, and both these events are enabled in $\overline{UDC.(N+1)}$ after a *down* has been sent.

Next we look at the possibility of computation interference in $C^0.0$ caused by $\overline{UDC.(N+1)}$. First we observe that after each occurrence of *ack* in the definition of $C^0.0$, both inputs are enabled. Second, we see that after a *full* event, a *down* input is allowed. This is also the only enabled event in $\overline{UDC.(N+1)}$ after a *full* has been received. Third, in both $C^0.0$ and $\overline{UDC.(N+1)}$ the only event that can occur after a transition at *empty* is *up*. Hence we may conclude that that there is no computation interference between $C^0.0$ and $\overline{UDC.(N+1)}$.

By similar reasoning it can be verified that there is no computation interference between $s\,UDC.N$ and $C^0.0$. Since there are no terminals connecting $\overline{UDC.(N+1)}$ to $s\,UDC.N$, we can now conclude that there is no computation interference.

3.1.2 Performance Analysis

In the previous section we proved that an N-counter can be implemented by $N-1$ cells of type $C^0.0$ and a 1-counter. In this section we analyze the area complexity, response time and power consumption of such an implementation.

Area Complexity

Suppose that we have hardware implementations of $C^0.0$ and the 1-counter. Then a hardware implementation of an N-counter can be made of a linear array of implementations of cells, with

connections only between neighboring cells. This means that the amount of area used for the connection wires is relatively small. Since we are only interested in the order of the area used for possible hardware implementations of counters, the number of cells into which an N-counter is decomposed gives an accurate enough measure for this.

The number of $C^0.0$ cells into which an N-counter is decomposed grows linearly with N. We may ask ourselves whether we can do better than that. Consider a network of k components, and assume that each component has a fixed number of states, which is at least two. Then the number of states of the network can grow at most exponentially with k. Thus the number of cells for any N-counter implementation grows at least logarithmically with N. In the next section we show that a number of cells that grows logarithmically with N indeed enough to implement an N-counter.

Response Time

We judge our designs not only by their area complexity, but also by their response time, i.e., the time that elapses between an input to the counter and the succeeding output.

Since we use high-level descriptions of our cells, we can only give an estimate for the response time. In our analysis of the response time, we assume that the cells are implemented such that there is an upper bound for the delay between two consecutive external events of a cell. This means that the analysis is not valid for implementations of our cells in which some kind of lock (deadlock or livelock) may occur. If livelock can occur in the implementation of a cell, then there is no upper bound for the time elapsing between an input to that cell and the next output.

We cannot give a sequence function for the unary counter implementation presented earlier, since the behavior of the cells is not cubic. However, for the proposed unary implementation we do not have to use sequence functions, a more informal analysis will suffice. The reason is that the behaviors of the unary implementation are purely sequential. As a result, the number of internal events that occur in the implementation between receiving an input from the environment and sending the corresponding output to the environment is a good measure for the response time.

In the best case there are no internal events between an input and the next output of the counter. This occurs when the first cell of the array (the one that communicates with the environment) is in state $S.0$ or in state $S.1$ and the environment sends an *up* or *down* respectively. If the first cell of the counter is in state $S.3$, then the response time is larger. In this case all cells will go to state $S.3$. Upon receiving a *down* input, the first cell forwards this to the second cell, which in turn forwards it to the third cell, and so on. After all cells have received an input, all cells send an acknowledgement to their environment, one after another, starting with the last cell. So the

following trace might occur:

$$s = t\ down\ sdown \ldots s^{N-1}down\ \ s^{N-1}empty \ldots sack\ ack,$$

where t is a trace with N more *up* than *down* events. The number of internal transitions between the last *down* and *ack* is $2(N-1)$. In general, the response time is determined by the current count of the counter. If the current count is i, for some i larger than one, then the number of internal transitions between the next input and corresponding output is $2(i-1)$.

Power Consumption

The third performance criterion for the implementation is its power consumption. Power consumption of a circuit consists of static and dynamic power consumption. Charging and discharging of capacitances and short-circuit current during switching add to the dynamic power consumption. In CMOS circuits the static power consumption is due to leakage currents. More information on the power consumption of CMOS circuits can be found in [WE85].

The power consumption is analyzed here by counting the number of internal transitions per external transition. Some conditions have to be met for this to be an accurate estimate for the power consumption.

One condition is that the static power consumption is negligible compared to the power consumed during switching. This assumption is justified if the cells are implemented in CMOS technology and the frequency at which transitions occur is high enough. If the frequency of transitions becomes too low, leakage current becomes the main factor in the power consumption.

The other condition is that all transitions require about the same amount of power. The amount of power needed for a transition depends on the load capacitance and the voltage change required for that transition. Load capacitances, in turn, depend on the layout. For example, the load capacitance of a long wire is larger than that of a shorter one. Due to the regular, linear structure of our implementations, we may assume that the load capacitances are similar. We also assume that the voltage change for transitions is uniform throughout the implementation.

Furthermore we assume that the number of internal events in a hardware implementation of our cells is proportional to the number of external events of those cells. This assumption requires the absence of livelock in the cells and the absence of metastable behavior.

The result of the above assumptions is that we can analyze the power consumption by comparing the number of external communications in a behavior (communications in which the environment

partakes) to the number of internal communications. For implementations that do not satisfy the assumptions, counting the numbers of internal and external events gives a lower bound for the power consumption.

For a given network of components, the power consumption of a behavior t of that network is defined as the length of t divided by the number of external communications (communications with the environment) on t. For the power consumption of the network, we take the maximum of the power consumptions of its behaviors. We say that a network has constant power consumption if its power consumption is bounded from above by a constant.

The measure that we use for the power consumption is the same as that used by van Berkel in [vB93].

For the N-counter implementation using C^0 cells the power consumption grows linearly with N: if the count is N and *down* and *up* inputs arrive alternatingly, then each of those inputs causes $2(N-1)$ internal communications.

The power consumption of any unary up-down counter implementation using cells that have a bounded number of neighbors is determined by the way in which the cells are connected. For example, for an implementation consisting of a linear array of cells, the power consumption is at least linear in N. We show that for unary counter implementations constant power consumption cannot be attained. This can be seen by looking at a trace consisting of $\Theta.N$ consecutive *up*'s (with their corresponding acknowledgements). According to our specification in the previous chapter, such a trace is a possible behavior of *UDC.N*. So, it is sufficient to prove that for any implementation of *UDC.N* and any behavior u of that implementation such that $u \upharpoonright \mathbf{a}(UDC.N)$ consists of $\Theta.N$ consecutive *up*'s, the power consumption of u increases with N.

To see that unary counters with constant power consumption do not exist, we reason as follows. We assume that in the implementation $\Theta.N$ cells can be distinguished, each having a maximum internal count independent of N. If an *up* is sent to the counter, at least one cell has to change state so that its internal count goes up. If $\Theta.N$ consecutive *up*'s are sent to the counter, there are $\Omega.N$ changes in internal counts. Since the maximum internal counts of the cells are independent of N, we even know that the internal counts of $\Omega.N$ different cells change.

Let S be the set of cells whose internal counts change, and for cell $s \in S$, define $d.s$ as the number of cells on the path from the external inputs of the counter to cell s (cell s itself excluded). From the above it follows that the number of internal transitions for an external behavior consisting of $\Theta.N$ consecutive *up*'s is bounded from below by

$$(\Sigma\, s : s \in S : d.s).$$

Due to our assumption that each cell has $\mathcal{O}.1$ neighbors, this number is in $\Omega.(N \log N)$. Hence the power consumption for $\Theta.N$ consecutive *up*'s is $\Omega.(\log N)$. By the definition of power consumption, we see that the power consumption of the counter grows at least logarithmically with N as well.

For implementations consisting of a linear array of cells, the number of internal transitions for $\Theta.N$ consecutive *up*'s grows at least quadratically with N: the average distance of a cell to the environment is $N/2$. So the power consumption for such an implementation grows at least linearly with N.

Since the control structure of a stack implementation can be translated into a unary up-down counter implementation, these results for the power consumption of up-down counters also hold for stack implementations. So the power consumption of the control part of any N-place stack implementation grows at least logarithmically with N.

3.2 A Binary Implementation

3.2.0 Specification of the Cells

In the counter implementation as proposed in the previous section, a unary representation was used. In this section we use a binary representation. Again cells can be full or empty, but now the current count of an array of cells is not the number of cells that is full. The current count is obtained by assigning weights to the cells. Cell i is assigned weight 2^i. The current count is the sum of the powers of two of the weights of the full cells. In formula, the current count is

$$(\Sigma\, i : 0 \leq i < k : c.i * 2^i),$$

where k is the number of cells and $c.i$ is the internal count of cell i.

Using only two types of cells, an end cell and a non-end cell, we can make $(2^k - 1)$-counters for any k larger than zero. In Section 3.2.2 we explain how to make N-counters for any N greater than zero.

Before we define the behavior of the cells, we give invariants for the states in which an input from the environment is expected. With binary counting, a cell can be either full or empty and at the same time the subcounter can be full, empty, or neither. Thus we get the following named states:

$S.0$: the cell is empty and the subcounter is empty (current count is zero),
$S.1$: the cell is full and the subcounter is empty (current count is one),
$S.2$: the cell is empty and the subcounter is neither full nor empty,
$S.3$: the cell is full and the subcounter is neither full nor empty,
$S.4$: the cell is empty and the subcounter is full,
$S.5$: the cell is full and the subcounter is full.

Suppose that the counter is in one of the above states. If an *up* input is received and the cell is empty, an output is sent and the cell goes to the next state. Information about the current count of the subcounter (which is encoded in the state of the cell) determines which output is sent. If the cell is full and an *up* input is received, then an *sup* is sent to the subcounter, provided that the subcounter is not full. Then the cell waits for an input from the subcomponent, sends and output to its left environment, and goes to the next state. The operation upon receiving a *down* input is analogous.

Definition 3.2.0. The end cell of the binary counter is a $UDC.1$ cell. Let

$$C^1 = (C^1.0, C^1.1, C^1.1', C^1.2, C^1.2', C^1.3, C^1.4, C^1.5)$$

be the least fixpoint of

$$\begin{aligned}
S.0 &= \textbf{pref}\ (up?; ack!; S.1) \\
S.1 &= \textbf{pref}\ (up?; sup!; S.1' \mid down?; empty!; S.0) \\
S.1' &= \textbf{pref}\ (sack?; ack!; S.2 \mid sfull?; ack!; S.4) \\
S.2 &= \textbf{pref}\ (up?; ack!; S.3 \mid down?; sdown!; S.2') \\
S.2' &= \textbf{pref}\ (sack?; ack!; S.3 \mid sempty?; ack!; S.1) \\
S.3 &= \textbf{pref}\ (up?; sup!; S.1' \mid down?; ack!; S.2) \\
S.4 &= \textbf{pref}\ (up?; full!; S.5 \mid down?; sdown!; S.2') \\
S.5 &= \textbf{pref}\ (down?; ack!; S.4).
\end{aligned}$$

The behavior of the counter cell is specified by $C^1.0$. □

The two extra named states were introduced to avoid having to write down transitions more than once. In state $C^1.1'$ an *sup* has been sent to the subcomponent and the cell is waiting for an output from the subcomponent. In state $C^1.2'$ the cell is waiting for an output from the subcomponent after having sent an *sdown*.

If an up-down counter implemented with $C^1.0$ cells is full, then all the cells in the implementation are full. This means that we can only implement N-counters for numbers N whose binary representation does not have any 0's. How to make counters for general N is discussed in a separate section.

VERDECT shows that the state graph for this specification has sixteen states. Although the number of states per cell is only slightly larger than for the unary counter implementation discussed in the previous section, the area complexity of N-counter implementations using $C^1.0$ cells grows logarithmically with N.

3.2.1 Correctness of the Implementation

An implementation of a $(2^k - 1)$-counter using C^1 cells and a $UDC.1$ cell is correct if

$$UDC.(2^k - 1) \rightarrow ((i : 0 \leq i < k-1 : s^i C^1.0), s^{k-1}\, UDC.1),$$

for any k larger than zero. The structure of the proof is the same as that for the unary counter. We prove the decomposition by induction on k. The basic step of the induction is exactly the same. For the inductive step we have:

$$\begin{array}{ll} & UDC.(2^{k+1} - 1) \rightarrow ((i : 0 \leq i < k : s^i C^1.0), s^k\, UDC.1) \\ \Leftarrow & \quad \{ \text{ Substitution Theorem and induction hypothesis } \} \\ & UDC.(2^{k+1} - 1) \rightarrow (C^1.0, s\, UDC.(2^k - 1)). \end{array}$$

We prove something stronger than this, viz.

$$UDC.(2N + 1) \rightarrow (C^1.0, s\, UDC.N),$$

for $1 \leq N$. For this last decomposition we verify the two behavioral conditions. In particular, we consider only the case $1 < N$.

We start by looking at the weave $C^1.0 \| s\, UDC.N$. We use the construction explained in Section 3.1.1 to obtain a set of equations that define this weave. The equations are

$$\begin{array}{rcl} S.0.0 & = & \textbf{pref}\ (up?; ack!; S.1.0) \\ S.1.0 & = & \textbf{pref}\ (up?; sup; sack; ack!; S.2.1 \\ & & \quad |\, down?; empty!; S.0.0) \\ S.2.1 & = & \textbf{pref}\ (up?; ack!; S.3.1 \\ & & \quad |\, down?; sdown; sempty; ack!; S.1.0) \end{array}$$

$$
\begin{aligned}
S.2.i &= \textbf{pref}\ (up?;ack!;S.3.i \\
&\qquad |\,down?;sdown;sack;ack!;S.3.(i-1)) \qquad \text{for } 2 \le i < N \\
S.3.i &= \textbf{pref}\ (up?;sup;sack;ack!;S.2.(i+1) \\
&\qquad |\,down?;ack!;S.2.i) \qquad \text{for } 1 \le i < N-1
\end{aligned}
$$

$$
\begin{aligned}
S.3.(N-1) &= \textbf{pref}\ (up?;sup;sfull;ack!;S.4.N \\
&\qquad |\,down?;ack!;S.2.(N-1)) \\
S.4.N &= \textbf{pref}\ (up?;full!;S.5.N \\
&\qquad |\,down?;sdown;sack;ack!;S.3.(N-1)) \\
S.5.N &= \textbf{pref}\ (down?;ack!;S.4.N).
\end{aligned}
$$

Defining equations for $|[\,\mathbf{a}(s\,UDC.N) :: C^1.0 \parallel s\,UDC.N\,]|$ are now obtained by hiding the internal symbols:

$$
\begin{aligned}
S.0.0 &= \textbf{pref}\ (up?;ack!;S.1.0) \\
S.1.0 &= \textbf{pref}\ (up?;ack!;S.2.1 \mid down?;empty!;S.0.0) \\
S.2.1 &= \textbf{pref}\ (up?;ack!;S.3.1 \mid down?;ack!;S.1.0)
\end{aligned}
$$

$$
\begin{aligned}
S.2.i &= \textbf{pref}\ (up?;ack!;S.3.i \mid down?;ack!;S.3.(i-1)) \qquad \text{for } 2 \le i < N \\
S.3.i &= \textbf{pref}\ (up?;ack!;S.2.(i+1) \mid down?;ack!;S.2.i) \qquad \text{for } 1 \le i < N-1
\end{aligned}
$$

$$
\begin{aligned}
S.3.(N-1) &= \textbf{pref}\ (up?;ack!;S.4.N \mid down?;ack!;S.2.(N-1)) \\
S.4.N &= \textbf{pref}\ (up?;full!;S.5.N \mid down?;ack!;S.3.(N-1)) \\
S.5.N &= \textbf{pref}\ (down?;ack!;S.4.N).
\end{aligned}
$$

Denote the least fixpoint of these equations by B and denote the least fixpoint of the defining equations for $UDC.(2N+1)$ by A (so $A.0 = UDC.(2N+1)$). Then $|[\,\mathbf{a}(s\,UDC.N) :: C^1.0 \parallel s\,UDC.N\,]|$ is equal to $B.0.0$. In particular

$$
\begin{aligned}
A.0 &= B.0.0 \\
A.1 &= B.1.0 \\
A.i &= B.(2+(i\,\mathbf{mod}\,2)).(i\,\mathbf{div}\,2) \qquad \text{for } 1 < i < 2N \\
A.(2N) &= B.4.N \\
A.(2N+1) &= B.5.N.
\end{aligned}
$$

This shows that

$$
|[\,\mathbf{a}(s\,UDC.N) :: C^1.0 \parallel s\,UDC.N\,]| = UDC.(2N+1),
$$

or equivalently,

$$(C^1.0 \parallel s\,UDC.N) \restriction \mathbf{a}(UDC.(2N+1)) = UDC.(2N+1).$$

Again we can apply Lemma 1.4.3 to conclude that the completeness condition of the decomposition is fulfilled.

In order to verify that there is no computation interference in the network consisting of $\overline{UDC.(2N+1)}$, $C^1.0$, and $s\,UDC.N$, we look at the communication between $C^1.0$ and $s\,UDC.N$.

From Definition 2.2.0 it follows that after each *up*, the next communication with the environment is one of *full* and *ack*. So in $s\,UDC.N$ an *sup* may be followed by either an *sfull* or an *sack*. These two are exactly the events that become enabled in $C^1.0$ after the occurrence of an *sup*. Also, if an *sdown* input occurs in $s\,UDC.N$, then an *sempty* or an *sack* will be sent, and both are enabled in $C^1.0$ after an *sdown*.

Similar arguments show that $C^1.0$ does not cause computation interference in $s\,UDC.N$ and that there is no computation interference between $\overline{UDC.(2N+1)}$ and $C^1.0$. Hence the network is free of computation interference.

3.2.2 Implementations for General N

In summary, we have proved the following results:

$$\begin{aligned} UDC.(N+1) &\to (C^0.0, s\,UDC.N) \\ UDC.(2N+1) &\to (C^1.0, s\,UDC.N). \end{aligned}$$

This means that with cells $C^0.0$ and $C^1.0$, and a $UDC.1$ cell, we can implement N-counters for any N larger than zero. If N is odd, then we use a C^1 cell as head cell and find an implementation for an (N **div** 2)-counter. If N is even then we use a C^0 as head cell and find an implementation for an $(N-1)$-counter. Note that in any counter implementation obtained using this strategy, the number of C^0 cells is at most the number of C^1 cells. Moreover, in such a counter implementation, only one of any two neighboring cells is of type $C^0.0$.

There is another way to implement an N-counter for general N, using a binary representation. It requires a cell that sends a *full* to its environment upon receiving an *sfull* from its subcomponent when its internal count is zero. A specification for such a cell is easily obtained from the specification

for the $C^1.0$ cell. It can be specified using seven defining equations:

$$\begin{aligned}
S.0 &= \textbf{pref}\ (up?; ack!; S.1) \\
S.1 &= \textbf{pref}\ (up?; sup!; S.1' \mid down?; empty!; S.0) \\
S.1' &= \textbf{pref}\ (sack?; ack!; S.2 \mid sfull?; full!; S.4) \\
S.2 &= \textbf{pref}\ (up?; ack!; S.3 \mid down?; sdown!; S.2') \\
S.2' &= \textbf{pref}\ (sack?; ack!; S.3 \mid sempty?; ack!; S.1) \\
S.3 &= \textbf{pref}\ (up?; sup!; S.1' \mid down?; ack!; S.2) \\
S.4 &= \textbf{pref}\ (down?; sdown!; S.2').
\end{aligned}$$

Denoting this cell by C, we have

$$UDC.(2N) \rightarrow (C, s\,UDC.N).$$

The growth rates for the area complexity, response time, and power consumption of an implementation using cells of this type and $C^1.0$ cells are the same as for an implementation using $C^0.0$ and $C^1.0$ cells.

3.2.3 Performance Analysis

A $(2^k - 1)$-counter implemented with C^1 cells and a $UDC.1$ consists of k cells. The implementation of a general N-counter using as many $C^1.0$ cells as possible (and as few $C^0.0$ cells as possible) also has a number of cells that grows logarithmically with N. Therefore we have achieved the optimal growth rate for the area complexity of up-down counter implementations.

For the response time, we notice that the implementations do not have any parallel behavior. As was the case in the unary implementation described in Section 3.1, an input may be propagated from the first cell to the last cell and an output of the last cell is propagated back to the first cell before an output to the environment occurs. Since the number of cells grows logarithmically with N, the response time does so as well.

If the implementation consists of C^1 cells and a $UDC.1$ cell only, the response time depends on the current count as follows. It is determined by the length of the suffix of ones in the binary representation of the current count (in case the next input is an *up*), or by the length of the suffix of zeroes in the binary representation of the count (in case the next input is a *down*).

The power consumption of this implementation grows logarithmically with N too. If all C^1 cells in the implementation are in state $C^1.3$, all C^0 cells are in state $C^0.2$, and the last cell has internal

count zero, then an *up* input is propagated all the way to the last cell in the array. This corresponds to incrementing the count when all cells except the last have internal count one. If the *up* input is followed by a *down*, this *down* input is also propagated all the way to the end of the array. Thus, there are behaviors where, after a bounded prefix, all inputs cause logarithmically many internal communications. Since the number of these inputs is unbounded, the power consumption grows logarithmically with N, assuming that the assumptions made in the section on power consumption of the unary implementation hold for this binary implementation as well.

Chapter 4

An Implementation with Parallelism

4.0 Introduction

The implementations presented in Chapter 3 have a response time that grows linearly with the number of cells of the implementation. In this chapter we present an implementation that has a better response time. Under certain assumptions one can conclude that the response time of this implementation does not depend on the number of cells. If the assumptions are weakened, however, the response time still depends on the number of cells.

Better response times can be obtained by designing implementations with parallelism. The unary and binary counters from Chapter 3 do not have any parallelism; their behaviors are strictly sequential.

Designing implementations with parallelism is more difficult than designing sequential implementations. A good way to specify parallel behaviors for linear arrays of cells is specifying the behaviors of a cell with respect to its left and right environments separately. The two partial behaviors of the cell are then weaved together. The proper synchronization between the partial behaviors is obtained by introducing internal symbols.

In specifications with parallelism, the commands language results in smaller specifications than, for example, state graphs. The reason is that, due to the weave operator, we do not have to represent parallelism by giving all interleavings of the events that may occur in parallel. Another advantage of the commands language will become evident in the correctness proof of the proposed implementation.

In this chapter we analyze the response time of the designed implementations by first abstracting

away from the different inputs and from the different outputs. This idea is based on the response time analysis of the stack design in [JU91]. The abstract implementation is analyzed using sequence functions and so-called *timing functions*. The underlying assumption for sequence functions is that delays are constant. The assumption for our timing functions is weaker. We assume that delays may vary between fixed lower and upper bounds. This seems to correspond more naturally to asynchronous implementations.

4.1 Specification of the Cells

As before, the end cell of an array of counter cells is a *UDC*.1 cell. The other cells are specified by a weave of two sequential behaviors, the behavior with respect to the environment and the behavior with respect to the subcomponent. We start with an explanation of the former.

For the behavior with respect to the environment only the emptiness or fullness of the cell is encoded in the named states. This is the only information needed to determine whether communication with the subcomponent must be initiated. If the cell is full and an *up* input is received, then there is a carry propagation to the next cell. If the cell is empty, an *up* input does not cause a carry propagation. Upon receiving a *down* input, there is a borrow propagation if and only if the cell is empty.

Initiation of communication with the subcomponent is done by introducing two internal symbols, *su* and *sd*. They should be interpreted as 'send an *sup* to the subcomponent' and 'send an *sdown* to the subcomponent'.

For determining which output must be sent to the environment after an input has been received, we introduce three additional internal symbols, viz., *se*, *sn*, and *sf*. The occurrence of an *se* event is to be interpreted as 'the subcomponent is empty'. Similarly, *sf* can be interpreted as 'the subcomponent is full' and *sn* as 'the subcomponent is neither full nor empty'.

We must make sure that our definition for the communication with the subcomponent justifies the interpretation of the internal symbols.

Definition 4.1.0. We use two named states for the description of the external behavior. A state 0 which indicates that the cell is empty, and state 1 which indicates that the cell is full. We get

the following equations in $S = (S.0, S.1)$:

$$
\begin{aligned}
S.0 = \mathbf{pref}\ (\ & (up?; ((se \mid sn); ack! \mid sf; full!) \\
& \mid down?; sd; ack! \\
&); S.1) \\
S.1 = \mathbf{pref}\ (\ & (up?; su; ack! \\
& \mid down?; ((sf \mid sn); ack! \mid se; empty!) \\
&); S.0).
\end{aligned}
$$

We denote the least fixpoint of these equations by D^2.

The behavior with respect to the subcomponent is specified using three states, encoding whether the subcomponent is empty, neither full nor empty, or full. The internal behavior is described by $E^2.0$, where E^2 is the least fixpoint of

$$
\begin{aligned}
S.0 = \mathbf{pref}\ (& se; S.0 \\
& \mid su; sup!; (sack?; S.1 \mid sfull?; S.2) \\
&) \\
S.1 = \mathbf{pref}\ (& sd; sdown!; (sack?; S.1 \mid sempty?; S.0) \\
& \mid sn; S.1 \\
& \mid su; sup!; (sack?; S.1 \mid sfull?; S.2) \\
&) \\
S.2 = \mathbf{pref}\ (& sf; S.2 \\
& \mid sd; sdown!; (sack?; S.1 \mid sempty?; S.0) \\
&).
\end{aligned}
$$

The behavior of the counter cell is specified by the command

$$C^2 = |[\, se, sn, sf, sd, su :: D^2.0 \parallel E^2.0 \,]|\,.$$

□

The state graph for this cell has 29 states. Conceptually the specification is related to the binary counter of [GL81].

Before we turn to the correctness proof for the counter implementation consisting of an array of C^2 cells and a $UDC.1$, we try to specify a cell without using internal symbols.

Suppose that both the cell and its subcounter are empty, so the current count is zero. When an *up* input is received, the cell immediately sends an *ack* to its environment. This is captured by the following equation:

$$S.0 = \textbf{pref}\ (up?; ack!; S.1).$$

Next the cell waits for another input, which may be an *up* or a *down*. If a *down* arrives, an *empty* output is produced and the cell returns to its initial state. If the input is an *up*, then an *sup* is sent to the subcomponent. Since the cell itself becomes empty again, an *ack* can be sent to the environment at the same time. This behavior is formalized as follows:

$$\begin{aligned} S.1 = \textbf{pref}\ (&down?; empty!; S.1 \\ &|up?; sup! \| ack!; S.2). \end{aligned}$$

In state 2 inputs from the environment and the subcomponent may arrive in either order. There are several possibilities:

$$\begin{aligned} S.2 = \textbf{pref}\ (&up? \| sack?; ack!; S.3 \\ &|up? \| sfull?; full!; S.4 \\ &|down? \| sack?; sdown! \| ack!; S.5 \\ &|down? \| sfull?; sdown! \| ack!; S.5) \end{aligned}$$

Even though the above three equations describe only a part of the behavior of the proposed cell, we already have six named states.

Moreover, the (partial) specification of this cell is incorrect. Note that after trace *up ack up ack* the environment has no way of knowing that it has to wait for an internal action (*sup*) to occur before sending the next input to the counter. A counter implemented by cells like this suffers from computation interference. This shows that one has to be careful in specifying behaviors in which things can happen in parallel.

4.2 Correctness of the Implementation

We prove that for $1 \leq k$ a network of $k-1$ components of type C^2 and one $UDC.1$ implements a 2^k-1-counter. For $k = 1$ we only have to prove

$$UDC.1 \rightarrow (UDC.1).$$

As mentioned before, this is a property of decomposition, so there is nothing left to prove. We use this case as the basic step for an inductive proof.

As before, the proof obligation for the inductive step can be reduced to

$$UDC.(2N+1) \rightarrow (C^2, s\,UDC.N)$$

by applying the Substitution Theorem. Before we prove that this last decomposition is valid, we introduce abbreviations for some alphabets:

$$A_0 = \mathbf{a}(s\,UDC.N)$$
$$A_1 = \{\, sd, su, se, sn, sf \}.$$

The structural conditions for this decomposition can be verified easily. For the behavioral conditions we consider the case $N > 1$ only; the case $N = 1$ is similar, but easier. For the completeness of the network behaviors we derive:

$$
\begin{array}{ll}
 & |[\, A_0 :: C^2 \parallel s\,UDC.N \,]| \\
= & \quad \{ \text{ Definition of } C^2 \ \} \\
 & |[\, A_0 :: |[\, A_1 :: D^2.0 \parallel E^2.0 \,]| \parallel s\,UDC.N \,]| \\
= & \quad \{ \text{ Property 1.1.0 with } |[\,.\,]| \text{ instead of } \restriction (\mathbf{a}(s\,UDC.N) \cap A_1 = \emptyset) \ \} \\
 & |[\, A_0 \cup A_1 :: D^2.0 \parallel E^2.0 \parallel s\,UDC.N \,]| \\
= & \quad \{ \text{ Define } F = E^2.0 \parallel s\,UDC.N, \text{ see Property 4.2.0 } \} \\
 & |[\, A_0 \cup A_1 :: D^2.0 \parallel F \,]| \\
= & \quad \{ \text{ Property 1.1.0 with } |[\,.\,]| \text{ instead of } \restriction \mathbf{a}(D^2.0) \cap A_0 = \emptyset \ \} \\
 & |[\, A_1 :: D^2.0 \parallel |[\, A_0 :: F \,]| \,]| \\
= & \quad \{ \text{ Define } G = |[\, A_0 :: F \,]|, \text{ see Property 4.2.1 } \} \\
 & |[\, A_1 :: D^2.0 \parallel G \,]| \\
= & \quad \{ \text{ Define } H = D^2.0 \parallel G, \text{ see Property 4.2.2 } \} \\
 & |[\, A_1 :: H \,]| \,.
\end{array}
$$

This derivation is made possible by the structure of the specification for C^2 as a weave of two sequential behaviors, one with respect to its left environment and one with respect to its right environment.

The second step of the derivation allows us to circumvent the construction of $D^2.0 \parallel E^2.0$, the state graph of which has 29 states. The fourth step allows us to hide the symbols of the alphabet of $s\,UDC.N$, which in turn allows for an easy specification of G.

PROPERTY 4.2.0. For $N > 1$ the weave of $E^2.0$ and $sUDC.N$ can be specified as the least fixpoint of

$$
\begin{array}{lll}
S.0.0 = & \textbf{pref}\ (se; S.0.0 & \\
 & \quad | su; sup; sack; S.1.1) & \text{for } 1 < N \\
S.1.i = & \left\{ \begin{array}{l}
\textbf{pref}\ (su; sup; sack; S.1.(i+1) \\
\quad | sn; S.1.i \qquad\qquad\qquad\qquad \text{for } i = 1 \text{ and } 2 < N \\
\quad | sd; sdown; sempty; S.0.(i-1)) \\
\textbf{pref}\ (su; sup; sack; S.1.(i+1) \\
\quad | sn; S.1.i \qquad\qquad\qquad\qquad \text{for } 1 < i < N-1 \\
\quad | sd; sdown; sack; S.1.(i-1)) \\
\textbf{pref}\ (su; sup; sfull; S.2.(i+1) \\
\quad | sn; S.1.i \qquad\qquad\qquad\qquad \text{for } i = N-1 \text{ and } 2 < N \\
\quad | sd; sdown; sack; S.1.(i-1)) \\
\textbf{pref}\ (su; sup; sfull; S.2.(i+1) \\
\quad | sn; S.1.i \qquad\qquad\qquad\qquad \text{for } i = 1 \text{ and } N = 2 \\
\quad | sd; sdown; sempty; S.0.(i-1))
\end{array} \right. & \\
S.2.N = & \textbf{pref}\ (sf; S.2.N & \\
 & \quad | sd; sdown; sack; S.1.(N-1)) & \text{for } 1 < N.
\end{array}
$$

So the weave of $E^2.0$ and $sUDC.N$ is the first component of the least fixpoint of these equations. Thus, F is the first component of the least fixpoint of these equations. □

With G defined as

$$|[\, A_0 :: F \,]|$$

we can obtain defining equations for G by omitting symbols from A_0 from the defining equations for F. We omit the first index in the named states of F in writing down the defining equations for G.

PROPERTY 4.2.1. G is the first component of the least fixpoint of

$$
\begin{array}{lll}
S.0 & = \textbf{pref}\ (se; S.0 \mid su; S.1) & \\
S.i & = \textbf{pref}\ (su; S.(i+1) \mid sn; S.i \mid sd; S.(i-1)) & \text{for } 0 < i < N \\
S.N & = \textbf{pref}\ (sf; S.N \mid sd; S.(N-1)). &
\end{array}
$$

□

PROPERTY 4.2.2. The first component of the least fixpoint of

$$
\begin{aligned}
S.0.0 &= \mathbf{pref}\ (up?;\, se;\, ack!;\, S.1.0 \mid down?) \\
S.1.0 &= \mathbf{pref}\ (up?;\, su;\, ack!;\, S.0.1 \mid down?;\, se;\, empty!;\, S.0.0)
\end{aligned}
$$

$$
\begin{aligned}
S.0.i &= \mathbf{pref}\ (up?;\, sn;\, ack!;\, S.1.i \mid down?;\, sd;\, ack!;\, S.1.(i-1)) && \text{for } 0 < i < N \\
S.1.i &= \mathbf{pref}\ (up?;\, su;\, ack!;\, S.0.(i+1) \mid down?;\, sn;\, ack!;\, S.0.i) && \text{for } 0 < i < N
\end{aligned}
$$

$$
\begin{aligned}
S.0.N &= \mathbf{pref}\ (up?;\, sf;\, full!;\, S.1.N \mid down?;\, sd;\, ack!;\, S.1.(N-1)) \\
S.1.N &= \mathbf{pref}\ (up? \mid down?;\, sf;\, ack!;\, S.0.N)
\end{aligned}
$$

is H. □

PROPERTY 4.2.3. The defining equations for $|[\, A_1 :: H \,]|$ are

$$
\begin{aligned}
S.0.0 &= \mathbf{pref}\ (up?;\, ack!;\, S.1.0 \mid down?) \\
S.1.0 &= \mathbf{pref}\ (up?;\, ack!;\, S.0.1 \mid down?;\, empty!;\, S.0.0)
\end{aligned}
$$

$$
\begin{aligned}
S.0.i &= \mathbf{pref}\ (up?;\, ack!;\, S.1.i \mid down?;\, ack!;\, S.1.(i-1)) && \text{for } 0 < i < N \\
S.1.i &= \mathbf{pref}\ (up?;\, ack!;\, S.0.(i+1) \mid down?;\, ack!;\, S.0.i) && \text{for } 0 < i < N
\end{aligned}
$$

$$
\begin{aligned}
S.0.N &= \mathbf{pref}\ (up?;\, full!;\, S.1.N \mid down?;\, ack!;\, S.1.(N-1)) \\
S.1.N &= \mathbf{pref}\ (up? \mid down?;\, ack!;\, S.0.N).
\end{aligned}
$$

□

The defining equations for $|[\, A_1 :: H \,]|$ are almost the same as those for $UDC.(2N+1)$: $S.i.j$ corresponds to defining equation no. $i + 2j$ of $UDC.(2N+1)$. The only difference is that more inputs are accepted: in the initial state a *down* input is accepted, and when the counter is full an *up* input is accepted. This means that

$$\mathbf{t}(UDC.(2N+1)) \subseteq \mathbf{t}(C^2 \parallel s\,UDC.N) \restriction \mathbf{a}(UDC.(2N+1)).$$

Lemma 1.4.3 gives the required result: the network behaviors are complete.

For absence of computation interference it is sufficient to look at computation interference between $\overline{UDC.(2N+1)}$ and $D^2.0$, and between $E^2.0$ and $s\,UDC.N$. The reason is that in $D^2.0$ after each

output action, both inputs are enabled; there is no synchronization with $E^2.0$ after output actions. For $E^2.0$ this holds as well: after an output has been sent, $E^2.0$ is ready to receive input without having to synchronize with $D^2.0$ first.

Verifying that there is no computation interference between $D^2.0$ and the environment, and between $E^2.0$ and the subcounter is analogous to the verification of absence of computation interference for the designs in Chapter 3. Therefore we omit it here.

4.3 Implementations for General N

Using C^2 cells and a *UDC*.1 cell we can only implement N-counters for N equal to $2^k - 1$, for certain k larger than zero. As in the previous chapter, we can use $C^0.0$ cells to overcome this deficiency.

Another possibility would be to specify a cell that has internal count zero when the counter is full. The problem in specifying such a cell is that it does not exhibit parallel behavior. If the cell has internal count one and receives an *up* input, it can initiate communication with its subcomponent, but it cannot send an acknowledgement back to its left environment before receiving an input from its subcomponent. If the subcomponent sends an *sack*, then the cell should send an *ack*, and if the subcomponent sends an *sfull*, then the cell should send a *full*. For a number N that has only one 1 in its binary representation this would result in an implementation without any parallel behavior. This has consequences for the response time.

4.4 Performance Analysis

4.4.0 Area Complexity

In the correctness proof for the proposed implementation we saw that k cells implement a $(2^k - 1)$-counter. Thus the implementation's area complexity grows logarithmically with the maximum count. In general, any N-counter can be implemented using C^2 cells, C^0 cells and a *UDC*.1 cell using $\Theta.(\log N)$ cells.

4.4.1 Response Time Analysis

For our response time analysis we use sequence functions, see [Zwa89] and Chapter 1. For simplicity's sake we only consider implementations using C^2 cells and a *UDC*.1 cell.

The specification of a C^2 cell is not cubic. To obtain a cubic trace structure we transform the specification in a number of steps. We make sure that the transformations do not influence the outcome of the response time analysis. There are several transformations we can apply to the specification without influencing (the growth rate of) the response time.

The first step is abstracting away from different channels for which there is input non-determinism. In our counter, for example, the environment controls whether an *up* or *down* input will arrive at the counter. In a cubic specification this non-determinism is not allowed. Renaming terminals does not change (the growth rate of) the response time.

So the first transformation comprises replacing the two inputs from the left environment by one input (r) and replace the three outputs to the left environment by one output (a). We do the same for the terminals for communication with a subcomponent. Furthermore we replace the internal symbols *se*, *sn*, and *sf* by a new internal symbol p and replace *sd* and *su* by an internal symbol q.

We get the following defining equations:

$$\begin{aligned} S.0 &= \mathbf{pref}\ ((r?;p;a! \mid r?;q;a!);S.1) \\ S.1 &= \mathbf{pref}\ ((r?;p;a! \mid r?;q;a!);S.0) \end{aligned}$$

for the behavior with respect to the left environment, and

$$\begin{aligned} S.0 = \mathbf{pref}\ &(p;S.0 \\ &|q;sr!;(sa?;S.1 \mid sa?;S.2)) \\ S.1 = \mathbf{pref}\ &(q;sr!;(sa?;S.0 \mid sa?;S.1) \\ &|p;S.1 \\ &|q;sr!;(sa?;S.1 \mid sa?;S.2)) \\ S.2 = \mathbf{pref}\ &(q;sr!;(sa?;S.0 \mid sa?;S.1) \\ &|p;S.2) \end{aligned}$$

for the behavior with respect to the right environment.

In the next step we rewrite the equations for the behavior with the left environment and we identify p and q. This does not influence the response time. For the communication with the left environment we now have

$$S.0 = \mathbf{pref}\ *[r?;q;a!]$$

and for the communication with the right environment

$$\begin{aligned}
S.0 &= \textbf{pref } (q; S.0 \\
&\qquad\qquad |q; sr!; sa?; (S.1 \mid S.2)) \\
S.1 &= \textbf{pref } (q; S.1 \\
&\qquad\qquad |q; sr!; sa?; (S.0 \mid S.1 \mid S.2)) \\
S.2 &= \textbf{pref } (q; S.2 \\
&\qquad\qquad |q; sr!; sa?; (S.0 \mid S.1)).
\end{aligned}$$

We are interested in the worst-case delay between an input r and the succeeding output a. The worst-case delay occurs when there is much communication between a cell and its subcomponent. Therefore we remove the non-determinism by omitting the alternatives in which no communication with the subcomponent occurs. For the defining equations we obtain

$$S.0 = \textbf{pref } *[r?; q; a!]$$

and

$$\begin{aligned}
S.0 &= \textbf{pref } (q; sr!; sa?; (S.1 \mid S.2)) \\
S.1 &= \textbf{pref } (q; sr!; sa?; (S.0 \mid S.1 \mid S.2)) \\
S.2 &= \textbf{pref } (q; sr!; sa?; (S.0 \mid S.1)).
\end{aligned}$$

The result of this step is that after every input from the left environment, there is communication with the subcomponent. Such behavior is indeed possible in the up-down counter as can be seen in the following example.i Representing the current count by a binary number with the most significant bit on the left, the receipt of an *up* when the current count is 01^n, for some n larger than zero, causes all cells (except the end cell) to communicate with their subcomponents. The new count is 10^n, and now the receipt of a *down* input causes all cells to communicate with their subcomponent. Alternating *up*'s and *down*'s results in a behavior where every cell communicates with its subcomponent upon receiving an input.

With some trace calculus the equations for the behavior with respect to the subcomponent can be simplified further. The abstract behavior of the counter cell C^2 is described by

$$\begin{aligned}
C^2_{abs} = |[\; & q :: \\
& \quad \textbf{pref } *[r?; q; a!] \\
& \| \; \textbf{pref } *[q; sr!; sa?] \\
&]| \; .
\end{aligned}$$

This is the specification for the control structure of a micropipeline.

The abstract behavior of the end cell is described by

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

i.e., a simple WIRE.

Now consider a network consisting of $k-1$ components of type C^2_{abs} and a wire. Define

$$W.k = (\,\|\, j : 0 \leq j < k-1 : s^j C^2_{abs}) \,\|\, \textbf{pref } *[s^{k-1}r?; s^{k-1}a!].$$

Then $\sigma \in \textbf{occ}.(W.k) \rightarrow \mathbb{N}$, defined by

$$\sigma.(s^j r).i = j + 2i$$
$$\sigma.(s^j a).i = j + 2i + 1,$$

for $0 \leq j < k$ and $0 \leq i$, is a sequence function for $W.k$ showing that $W.k$ has constant response time:

$$\begin{array}{ll} & \sigma.(s^j a).i - \sigma.(s^j r).i \\ = & \{ \text{ Definition of } \sigma \} \\ & j + 2i + 1 - j - 2i \\ = & \{ \qquad \} \\ & 1. \end{array}$$

From the above it follows that, using the formalism of sequence functions, the response time of $W.k$ is independent of k.

In a decomposition of a C^2 cell there may be some internal events that have to occur between an input to the cell and the next output. The cell has a finite number of states. So, assuming that there is no livelock, the number of internal events that occur between any input and the next output, is bounded. Thus an up-down N-counter implementation made of C^2 cells has a response time that is independent of N as well, provided the implementations of the cells do not suffer from livelock. By the response time of an up-down counter implementation we mean the delay between an *up* or *down* and the succeeding *empty*, *ack*, or *full*.

For implementations of general N-counters using C^2 and $C^0.0$ cells we can also prove constant response time. The reason is that in an N-counter implementation at most half of the cells are of

type $C^0.0$ and that there are no neighboring cells of type $C^0.0$. An implementation of an N counter using a strictly binary representation as suggested in Section 4.3 may not have constant response time. This is a result of the lack of parallelism for certain values of N.

The existence of a sequence function showing constant response time corresponds to a synchronous implementation with constant response time. We want to know whether asynchronous implementations have bounded response time. Here the term 'bounded response time' is used instead of 'constant response time.' One of the reasons is that in asynchronous implementations the response time may vary due to variations in the delays of basic components, although there is an upper bound for the response time. Another reason is that we want to avoid confusion: in this report the term 'constant response time' is used according to the definition in [Zwa89] and the term 'bounded response time' is reserved for response time analysis on networks in which the response time of basic components may vary.

Under certain assumptions about the delays between events of C^2_{abs} we can prove that the response time of an array of k of these cells increases linearly with k. We assume that the delay between certain events may vary between a lower and an upper bound. This seems to correspond to an asynchronous implementation in a more natural way than the assumptions made for response time analysis with sequence functions. We show that under our variable-delay assumptions the response time of an array of k cells of type C^2_{abs} may grow linearly with k.

For C^2_{abs} we cannot assume upper bounds for the delays between outputs and succeeding inputs. The only delays we can assume to be bounded from above are the delays between $(r, 0)$ and $(a, 0)$, between $(r, 0)$ and $(sr, 0)$, between $(r, i+1)$ or (sa, i) (whichever occurs last) and $(a, i+1)$, and between $(r, i+1)$ or (sa, i) (whichever occurs last) and $(sr, i+1)$. For simplicity's sake, we assume one lower bound δ and one upper bound Δ for all of these delays.

With the assumed lower and upper bounds for the delays between events, we can write down *timing functions* for the cells of an array of k components of type C^2_{abs} and one WIRE component. The functions are just sequence functions if δ and Δ are integer-valued. Timing functions can be viewed as functions that generate timed traces. They correspond to possible asynchronous behaviors of the components they describe, rather than synchronous behaviors.

A pictorial representation of the array of C^2_{abs} components is helpful in writing down timing functions. A C^2_{abs} component can be implemented by an initialized 1-by-1 JOIN. This is a 1-by-1 JOIN that behaves as a normal JOIN in which a transition has already occurred at one of the two input terminals. This implementation is due to Sutherland [Sut89]. Sutherland calls the 1-by-1 JOIN a C-element or rendez-vous element. An initialized 1-by-1 JOIN can be decomposed into a normal JOIN and an IWIRE. For the initialized JOIN element we assume that a delay of at least δ and at

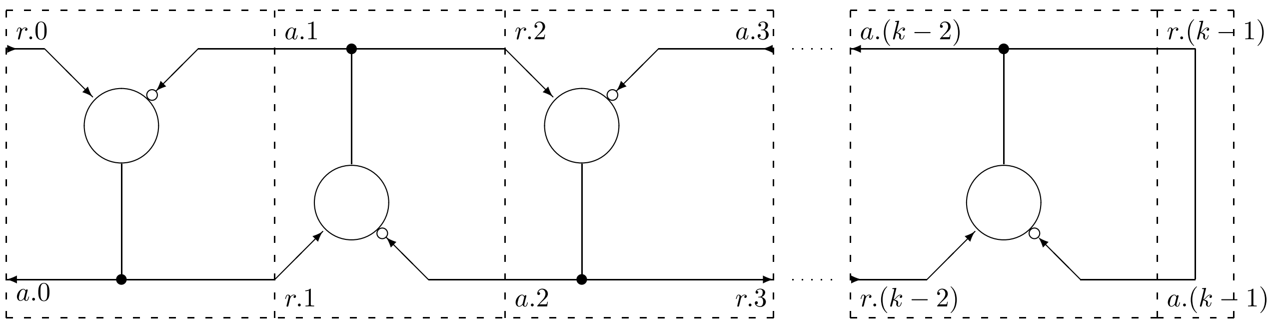

Figure 4.0: Implementation of abstract counter cells: a micropipeline.

most Δ time units occurs between the time that the output becomes enabled and the time that the output has taken place. Furthermore we assume that the delays in connection wires are zero; this poses no restriction on the validity of the results: with non-zero wire delays the same result can be obtained. With these assumptions, the implementation of a C^2_{abs} component with a JOIN satisfies exactly the delay assumptions made earlier for the C^2_{abs} cell. In Figure 4.0 the implementation is depicted. The initialized input to the JOINs is indicated with a bubble.

The idea for obtaining a timing function for which all initialized JOINs have bounded response time, but the network does not, is the following. We let the first input to the array propagate to cell $k-2$ as fast as possible, we let the second input propagate to cell $k-3$ as fast as possible, and so on. Then we let the acknowledgements propagate back as slowly as possible. The result is a timing function for which the delay between $(r.0, k-2)$ and $(a.0, k-2)$ depends on k. Although it is not very likely that this distribution of delays will occur in practice, it is possible in theory.

For cell j, with j at least zero and smaller than $k-2$, we have the following timing function:

$$\begin{array}{rcll}
\mathcal{T}_j.(r.j).i &=& (j+2i)*\delta & \text{for } 0 \le i < k-1-j \\
\mathcal{T}_j.(r.j).i &=& (k-2)*\delta + (k-j+2*(i-k+1+j))*\Delta & \text{for } k-1-j \le i \\
\mathcal{T}_j.(a.j).i &=& (j+2i+1)*\delta & \text{for } 0 \le i < k-2-j \\
\mathcal{T}_j.(a.j).i &=& (k-2)*\delta + (k-j-1+2*(i-k+2+j))*\Delta & \text{for } k-2-j \le i
\end{array}$$

$$\begin{array}{rcll}
\mathcal{T}_j.(r.(j+1)).i &=& (j+2i+1)*\delta & \text{for } 0 \le i < k-2-j \\
\mathcal{T}_j.(r.(j+1)).i &=& (k-2)*\delta + (k-j-1+2*(i-k+2+j))*\Delta & \text{for } k-2-j \le i \\
\mathcal{T}_j.(a.(j+1)).i &=& (j+2i+2)*\delta & \text{for } 0 \le i < k-3-j \\
\mathcal{T}_j.(a.(j+1)).i &=& (k-2)*\delta + (k-j-2+2*(i-k+3+j))*\Delta & \text{for } k-3-j \le i.
\end{array}$$

For the last micropipeline cell, i.e. cell $k-2$, the timing function $\mathcal{T}_{k-2}$ can be used:

$$\begin{aligned}
\mathcal{T}_{k-2}.(r.(k-2)).i &= (k-2)*\delta + 2i*\Delta && \text{for } 0 \leq i \\
\mathcal{T}_{k-2}.(a.(k-2)).i &= (k-2)*\delta + (2i+1)*\Delta && \text{for } 0 \leq i \\
\mathcal{T}_{k-2}.(r.(k-1)).i &= (k-2)*\delta + (2i+1)*\Delta && \text{for } 0 \leq i \\
\mathcal{T}_{k-2}.(a.(k-1)).i &= (k-2)*\delta + (2i+1)*\Delta && \text{for } 0 \leq i.
\end{aligned}$$

Finally, for cell $k-1$, which is just a WIRE, we have:

$$\begin{aligned}
\mathcal{T}_{k-1}.(r.(k-1)).i &= (k-2)*\delta + (2i+1)*\Delta && \text{for } 0 \leq i \\
\mathcal{T}_{k-1}.(a.(k-1)).i &= (k-2)*\delta + (2i+1)*\Delta && \text{for } 0 \leq i.
\end{aligned}$$

Now we have to show that these timing functions for the separate cells form a timing function for the whole array of connected cells. According to [Zwa89, Theorem 2.5.13] it is sufficient to show that for all a such that a occurs in the alphabet of two cells, say j and j', the values for $\mathcal{T}_j.a.i$ and $\mathcal{T}_{j'}.a.i$ are the same for all i for which (a,i) occurs in a network behavior. It is easily verified that this property holds for our network of cells.

The values $\mathcal{T}_0.(r.0).i$ and $\mathcal{T}_0.(a.0).i$ can be seen as the moments in time at which external communications may occur in an asynchronous behavior of the network. Due to our particular choice of the timing functions, an output of a JOIN occurs at least δ and at most Δ time units after it becomes enabled. So the basic components in the network have a bounded response time. However, if we consider the delay between $r.0$ and $a.0$ events, we see that this depends on k, the number of cells:

$$\begin{aligned}
& \mathcal{T}_0.(a.0).(k-2) - \mathcal{T}_0.(r.0).(k-2) \\
= & \quad \{ \text{ Definition of } \mathcal{T}_0 \text{ } \} \\
& (k-2)*\delta + (k-2)*\Delta - 2*(k-2)*\delta \\
= & \quad \{ \text{ Algebra } \} \\
& (k-2)*(\Delta - \delta)
\end{aligned}$$

If $\delta < \Delta$ and the response time of basic components may vary arbitrarily between these two bounds, then the delay depends linearly on the number of cells in the array. Therefore we do not consider this implementation to have bounded response time. We consider an implementation of the micropipeline to have bounded response time if and only if for all timing functions $\mathcal{T}$ for that implementation where the delays of the basic components are bounded by a constant, the delay between inputs from the environment and corresponding outputs to the environment is bounded by a constant independent of the length of the pipeline.

The choice of the bounds for the delays of the basic components may influence the results of the analysis. There are several possibilities. They range from assuming the same bounds for all components, or introducing parameters for all components, so even delays of components of the same type may have different bounds. Introducing more parameters introduces more freedom for choosing delays, and therefore it will be harder to prove bounded response time. It is unclear what assumptions should be made, especially if the circuit in question is an abstraction of some other circuit. Here the micropipeline was obtained as an abstraction of an up-down counter. A questionable assumption we made is that wires have no delays. Introducing wire delays between the output of a JOIN element and the input to the next JOIN gives us the same result, but the analysis becomes more tedious. The result stays the same because adding delays in each cell amounts to adding the same delays in all the directed cycles that can be seen in Figure 4.0.

Something else that is not clear is how to define bounded response time for delay-insensitive circuits in general. Our counters and the micropipeline have the nice property that inputs and outputs alternate. Our definition of bounded response time is valid for this type of circuit only. In general inputs and outputs do not alternate and then a different formulation of bounded response time is required.

Proving bounded response time as defined here is harder than proving constant response time using sequence functions. In the definition of bounded response time there is a *universal* quantification over timing functions, while the definition of constant response time contains an *existential* quantification over sequence functions. Methods for proving bounded response time still have to be investigated.

We still have to show that the result we proved for the response time of a micropipeline is also valid for up-down counter implementations based on C^2 cells.

In a decomposition of a C^2 cell the delays of the basic components on distinct paths from input terminals to output terminals may have different bounds. In C^2_{abs} cells there is only one terminal for receiving input from the left environment and one for receiving input from the right environment. Similarly, there is only one output terminal to each of the environments. In our analysis the same bounds for delays were assumed for the four paths in the C^2_{abs} cell implementation.

Given an implementation of a C^2 cell, we can set δ to the maximum of the lower bounds for the delays of the different paths from inputs to outputs. We can set Δ to the minimum of the upper bounds for the delays of the different paths from inputs to outputs. With these values for δ and Δ, and a timing function $\mathcal{T}$ for the micropipeline, we can make a timing function for the counter implementation made of C^2 cells. If $\mathcal{T}$ is a timing function showing that the micropipeline does not have bounded response time, then $\mathcal{T}$ can be used to show that the counter implementation built

from C^2 cells does not have bounded response time either, provided that the value found for δ is smaller than that for Δ. The timed trace generated by $\mathcal{T}$ corresponds to a trace of the counter starting with internal count $2^{k-1} - 1$ and alternating *up*'s and *down*'s instead of r's.

4.4.2 Power Consumption

Let $1 \leq k$ and assume that we have an array of k cells of type C^2 and one *UDC*.1 cell. The current count of the counter is the sequence of internal counts of the cells, interpreted as a binary number with the most significant bit on the right.

For an implementation of an $(2^k - 1)$-counter using $C^1.0$ cells, the representation of the current count is exactly the same. This means that if the current counts of these two implementations, one with C^2 cells and one with $C^1.0$ cells, are the same, they have the same number of carry propagations if they receive the same input. So, if both counters are fed the same sequence of inputs, the numbers of internal transitions as a result of those inputs are the same.

In Chapter 3 we showed that the binary N-counter implementation using $C^1.0$ cells has a power consumption that grows logarithmically with N. By the argument above and the definition of power consumption, we conclude that the power consumption of the N-counter implementation using C^2 cells grows logarithmically with N as well.

4.5 The *ack-nak* Protocol

In Chapter 2 we gave two specifications for up-down counters. In the one we used so far, up-down counters have three outputs. In the other one, there are two outputs, viz., *ack* and *nak*.

Designing a counter that consists of a linear array of cells that have outputs *ack* and *nak* instead of *empty*, *ack*, and *full* is harder, because input from the subcomponent of a cell does not give information about the subcomponent's state. We illustrate this by attempting to specify the cells of a binary implementation with internal parallelism based on the *ack-nak* protocol.

As before, we try to specify a cell's behaviors with respect to environment and subcomponent separately, starting with the behavior w.r.t the environment. This behavior is very similar to that of the C^2 cells. The only difference is in the synchronization after a *down* has been received in state

zero or an *up* has been received in state one: F is defined as the least fixpoint of

$$\begin{aligned} S.0 &= \textbf{pref } (up?;\, ack!;\, S.1 \\ &\qquad\quad |down?;\, (se;\, nak!;\, S.0 \mid sd;\, ack!;\, S.1)) \\ S.1 &= \textbf{pref } (up?;\, (sf;\, nak!;\, S.1 \mid su;\, ack!;\, S.0) \\ &\qquad\quad |down?;\, ack!;\, S.0). \end{aligned}$$

If a *down* is received when in state $F.0$, decrementing the subcounter regardless of its state does not work. The output to the environment depends on the state of the subcounter. So the state of the subcounter has to be known in order to achieve the parallelism.

The specification of the behavior w.r.t. the subcounter becomes different. Consider the following equations.

$$\begin{aligned} S.0 &= \textbf{pref } (se;\, S.0 \\ &\qquad\quad |su;\, sup!;\, sack?;\, S.3) \\ S.1 &= \textbf{pref } (su;\, sup!;\, sack?;\, S.3 \\ &\qquad\quad |sd;\, sdown!;\, sack?;\, S.4) \\ S.2 &= \textbf{pref } (sf;\, S.2 \\ &\qquad\quad |sd;\, sdown!;\, sack?;\, S.4). \end{aligned}$$

States zero, one, and two encode the state of the subcounter, as before. But now the new state of the subcounter cannot be determined from the last input received from the subcounter. In state three the subcounter is not empty, but it may be either full or non-full. Similarly, in state four the subcounter is not full, but it may be either empty or non-empty. We need extra equations to determine the new state. The easiest way of doing this is sending an extra *sup* or *sdown*. This gives

$$\begin{aligned} S.3 &= \textbf{pref } (sup!;\, (sack?;\, sdown!;\, sack?;\, S.1 | snak?;\, S.0)) \\ S.4 &= \textbf{pref } (sdown!;\, (sack?;\, sup!;\, sack?;\, S.1 | snak?;\, S.0)) \end{aligned}$$

for states three and four. If the cell is in state three, then the subcounter is not empty. We have to determine whether it is full or not. If an *sack* is received from the subcounter after an *sup* has been sent to the subcounter, the subcounter was not full. An *sdown* is sent to return the subcounter to its original state. If an *snak* is received, then the subcounter is full. Since the counter's value does not change when it is full and receives an *up*, no *sdown* has to be sent to cancel the effect of the extra *sup*.

We see that by using the *ack-nak* protocol, we get a more difficult specification for the cells. In our solution there is extra communication between cells. We have not investigated whether there are

other implementations for counters with the *ack-nak* protocol, without the extra communication overhead.

From this exercise we learn that it might be wise to keep some information of the state of neighboring cells when specifying the behavior of a cell. Of course those neighboring cells should provide enough information to make this possible. Without this information it is harder to obtain parallel behavior.

Chapter 5

An Implementation with Constant Power Consumption

5.0 Introduction

In this chapter we design an N-counter with constant response time and constant power consumption for any N larger than zero. To achieve this we use redundant number systems with radix 2. In the implementation some cells use digit set $\{0, 1, 2\}$ and other cells use digit set $\{0, 1, 2, 3\}$. Which digit set is used for a cell in the implementation of an N-counter depends on the value of N.

In a number system with radix 2 and digit set $\{0, 1, \ldots, m\}$, the string $d_{l-1} \ldots d_0$ represents the number

$$(\Sigma\, i : 0 \leq i < l : d_i 2^i),$$

as in the normal binary system. For m larger than one most numbers have more than one representation. The possible representations of four for m equal to two are 12, 20, and 100 (most significant digit on the left).

The advantage of using redundant number systems is that there will be fewer carry and borrow propagations per input in our implementation. It is the reduction in the number of carry and borrow propagations per input that allows an implementation with constant power consumption.

The number system with radix 2 and digit set $\{0, 1, 2\}$ is known as the binary stored-carry (or BSC) number system [Par90]. There does not seem to be an accepted name for the binary number

system with digits 0, 1, 2, and 3. In the conclusions of [Par90] the name binary stored-double-carry (or BSDC) number system is proposed. This name is used in the rest of this report.

We want to be able to make an implementation for an N-counter, for N larger than zero, with the property that the counter is full if and only if all cells in the implementation are full. As pointed out in Section 4.3, this is necessary to obtain parallel behavior in the implementation for any N. Thus, using cells with maximum internal counts of two and three, and possibly one for the end cell of an implementation, we have to be able to represent all positive numbers with digits 2, 3, and possibly 1 for the most significant digit.

We need cells with maximum internal counts of two and three to reduce the average number of carry and borrow propagations per input of a behavior. If we use cells with maximum internal counts of one and two, then there is no redundancy in the representation for some numbers. For example, for $N = 2^k - 1$, the N's only representation using digits 1 and 2, and radix 2, is a sequence of 1's, a normal binary number. An implementation of such a counter uses only cells with maximum internal count 1. So the current count of the implementation is represented in the same way as for the implementation of Chapter 4. The number of carry and borrow propagations per input is the same as well, and so is the power consumption.

The next lemma shows that all positive numbers can be represented as a radix-2 number using digits 2 and 3, and possibly 1 for the most significant digit.

LEMMA 5.0.0. Any natural number larger than 0 can be represented (uniquely) by radix-2 number $d_{l-1} \ldots d_0$ for some l larger than zero, where $d_i \in \{2, 3\}$ for $i < l - 1$ and $d_{l-1} \in \{1, 2, 3\}$. □

PROOF. The proof is an easy induction. For the base case of the induction we observe that the lemma is true for 1, 2, and 3.

Now let $3 < N$. If N is even, then $N = 2(N-2)\,\mathbf{div}\,2 + 2$. Since $(N-2)\,\mathbf{div}\,2$ is smaller than N, we have by induction hypothesis that there is an l larger than zero and there are $d_0, \ldots, d_{l-1}$ such that the radix-2 number $d_{l-1} \ldots d_0$ represents $(N-2)\,\mathbf{div}\,2$, all but the most significant digit are elements of $\{2, 3\}$, and the most significant digit d_{l-1} is in $\{1, 2, 3\}$. Then N can be represented by the radix-2 number $d_{l-1} \ldots d_0 2$.

If N is odd, we can just add a 3 to the radix-2 representation of $(N-3)\,\mathbf{div}\,2$, which exists by the induction hypothesis. □

Thus, we need three types of end cells and two types of other (non-end) cells to be able to implement any N-counter. The three end cells are *UDC*.1, *UDC*.2, and *UDC*.3. The two other types of cells are a BSC cell and a BSDC cell.

5.1 Specification of the Cells

The specifications of the BSC and BSDC cells are very similar, and both are similar to the specification of cell C^2.

DEFINITION 5.1.0. Let D^3 be the least fixpoint of the following equations in S:

$$\begin{aligned} S.0 &= \textbf{pref}\ (up?; ack!; S.1 \\ &\qquad\quad |\, down?; sd; ack!; S.1) \\ S.1 &= \textbf{pref}\ (up?; ((se\,|sn); ack! \mid sf; full!); S.2 \\ &\qquad\quad |\, down?; ((sf|sn); ack! \mid se; empty!); S.0) \\ S.2 &= \textbf{pref}\ (up?; su; ack!; S.1 \\ &\qquad\quad |\, down?; ack!; S.1). \end{aligned}$$

Then the BSC counter cell is specified by

$$C^3 = |[\, se, sn, sf, sd, su :: D^3.0 \parallel E^2.0 \,]|\, .$$

□

The state graph of the specification of C^3 has 35 states.

DEFINITION 5.1.1. Let D^4 be the least fixpoint of

$$\begin{aligned} S.0 &= \textbf{pref}\ (up?; ack!; S.1 \\ &\qquad\quad |\, down?; sd; ack!; S.1) \\ S.1 &= \textbf{pref}\ (up?; ack!; S.2 \\ &\qquad\quad |\, down?; ((sf|sn); ack! \mid se; empty!); S.0) \\ S.2 &= \textbf{pref}\ (up?; ((se\,|sn); ack! \mid sf; full!); S.3 \\ &\qquad\quad |\, down?; ack!; S.1) \\ S.3 &= \textbf{pref}\ (up?; su; ack!; S.2 \\ &\qquad\quad |\, down?; ack!; S.2). \end{aligned}$$

The component

$$C^4 = |[\, se, sn, sf, sd, su :: D^4.0 \parallel E^2.0 \,]|$$

specifies the BSDC counter cell. □

The state graph of the BSDC cell has 49 states.

An implementation of a (2^k-1)-counter using these cells behaves different from an implementation using C^2 cells when given the same input. However, for both types of cells the communication with respect to the subcomponent is described by the same command that specifies the communication with respect to the subcomponent for the C^2 cell. This is yet another advantage of specifying behaviors of cells as a weave of sequential commands that describe the behavior at different boundaries of those cells.

5.2 Correctness of the Implementation

Since we have several types of cells, we have several proof obligations. Again, the proof is inductive. For the basic step we have three proof obligations, viz.

$$UDC.N \to (UDC.N)$$

for $1 \le N \le 3$. For the inductive step we apply the Substitution Theorem. It is sufficient to prove that for N larger than one

0. $UDC.(2N+2) \to (C^3, s\,UDC.N)$, and
1. $UDC.(2N+3) \to (C^4, s\,UDC.N)$.

The first parts of the proofs of 0 and 1 are identical to the first part of the correctness proof of the implementation in Chapter 4. For $i = 3, 4$ and any $1 \le N$ we have:

$$\begin{array}{ll} & |[\, A_0 :: C^i \parallel s\,UDC.N \,]| \\ = & \quad \{ \text{ See page 44 } \} \\ & |[\, A_1 :: D^i \parallel G \,]| \,. \end{array}$$

From this point on we treat the two cases separately.

0. The defining equations for $|[\, A_1 :: D^3 \parallel G \,]|$ are

$$\begin{aligned} S.0.0 &= \textbf{pref}\ (up?; ack!; S.1.0 \mid down?) \\ S.1.0 &= \textbf{pref}\ (up?; ack!; S.2.0 \mid down?; empty!; S.0.0) \\ S.2.0 &= \textbf{pref}\ (up?; ack!; S.1.1 \mid down?; ack!; S.1.0) \end{aligned}$$

$$\left.\begin{aligned} S.0.i &= \mathbf{pref}\ (up?; ack!; S.1.i \mid down?; ack!; S.1.(i-1)) \\ S.1.i &= \mathbf{pref}\ (up?; ack!; S.2.i \mid down?; ack!; S.0.i) \\ S.2.i &= \mathbf{pref}\ (up?; ack!; S.1.(i+1) \mid down?; ack!; S.1.i) \end{aligned}\right\}\ \text{for } 0 < i < N$$

$$\begin{aligned} S.0.N &= \mathbf{pref}\ (up?; ack!; S.1.N \mid down?; ack!; S.1.(N-1)) \\ S.1.N &= \mathbf{pref}\ (up?; full!; S.2.N \mid down?; ack!; S.0.N) \\ S.2.N &= \mathbf{pref}\ (up? \mid down?; ack!; S.1.N). \end{aligned}$$

We see that $S.2.i = S.0.(i+1)$ for $0 \le i < N$. So $|[\, A_1 :: D^3 \parallel G\,]|$ is also the first component of the least fixpoint of

$$\begin{aligned} S.0.0 &= \mathbf{pref}\ (up?; ack!; S.1.0 \mid down?) \\ S.1.0 &= \mathbf{pref}\ (up?; ack!; S.0.1 \mid down?; empty!; S.0.0) \end{aligned}$$

$$\left.\begin{aligned} S.0.i &= \mathbf{pref}\ (up?; ack!; S.1.i \mid down?; ack!; S.1.(i-1)) \\ S.1.i &= \mathbf{pref}\ (up?; ack!; S.0.(i+1) \mid down?; ack!; S.0.i) \end{aligned}\right\}\ \text{for } 0 < i < N$$

$$\begin{aligned} S.0.N &= \mathbf{pref}\ (up?; ack!; S.1.N \mid down?; ack!; S.1.(N-1)) \\ S.1.N &= \mathbf{pref}\ (up?; full!; S.2.N \mid down?; ack!; S.0.N) \\ S.2.N &= \mathbf{pref}\ (up? \mid down?; ack!; S.1.N). \end{aligned}$$

In these equations $S.i.j$ corresponds to $S.(i+2j)$ in the defining equations for $UDC.(2N+2)$. The only difference is that in $S.0.0$ a *down* input is allowed and in $S.2.N$ an *up* input is allowed. From this we conclude

$$\mathbf{t}(UDC.(2N+2)) \subseteq \mathbf{t}(C^3 \parallel sUDC.N) \restriction \mathbf{a}(UDC.(2N+2)),$$

and hence Lemma 1.4.3 gives us that the network behaviors are complete.

1. The defining equations for $|[\, A_1 :: D^4 \parallel G\,]|$ are

$$\begin{aligned} S.0.0 &= \mathbf{pref}\ (up?; ack!; S.1.0 \mid down?) \\ S.1.0 &= \mathbf{pref}\ (up?; ack!; S.2.0 \mid down?; empty!; S.0.0) \\ S.2.0 &= \mathbf{pref}\ (up?; ack!; S.3.0 \mid down?; ack!; S.1.0) \\ S.3.0 &= \mathbf{pref}\ (up?; ack!; S.2.1 \mid down?; ack!; S.2.0) \end{aligned}$$

$$\left.\begin{aligned} S.0.i &= \textbf{pref}\ (up?;ack!;S.1.i \mid down?;ack!;S.1.(i-1)) \\ S.1.i &= \textbf{pref}\ (up?;ack!;S.2.i \mid down?;ack!;S.0.i) \\ S.2.i &= \textbf{pref}\ (up?;ack!;S.3.i \mid down?;ack!;S.1.i) \\ S.3.i &= \textbf{pref}\ (up?;ack!;S.2.(i+1) \mid down?;ack!;S.2.i) \end{aligned}\right\}\ \text{for } 0 < i < N$$

$$\begin{aligned} S.0.N &= \textbf{pref}\ (up?;ack!;S.1.N \mid down?;ack!;S.1.(N-1)) \\ S.1.N &= \textbf{pref}\ (up?;ack!;S.2.N \mid down?;ack!;S.0.N) \\ S.2.N &= \textbf{pref}\ (up?;full!;S.3.N \mid down?;ack!;S.1.N) \\ S.3.N &= \textbf{pref}\ (up? \mid down?;ack!;S.2.N). \end{aligned}$$

There are $4N + 4$ defining equations, while a $UDC.(2N + 3)$ is specified with only $2N + 4$ equations. As in the proof under 0, we want to show that some of the equations specify the same trace structure. In particular, states $S.2.i = S.0.(i + 1)$ and $S.3.i = S.1.(i + 1)$ for $0 \leq i < N - 1$. But here it is harder to see which states are redundant than it was in the previous proof. We use fixpoint induction to prove that states $S.2.i$ and $S.3.i$ are redundant for $0 \leq i < N$.

As explained in Chapter 1, the defining equations for $|[\, A_1 :: D^4 \,\|\, G \,]|$ define a tail function f with least fixpoint $\mu.f$. In terms of the fixpoint we have to show that $\mu.f.2.i = \mu.f.0.(i + 1)$ and $\mu.f.3.i = \mu.f.1.(i+1)$ for $0 \leq i < N$. Using fixpoint induction this requires showing that predicate P, defined by

$$P.S \;\equiv\; (\forall i : 0 \leq i < N : S.2.i = S.0.(i+1) \,\wedge\, S.3.i = S.1.(i+1)),$$

for vector of trace structures $S \in \mathcal{T}^{4(N+1)}.(\mathbf{a}(UDC.N))$, is inductive, $P.(\bot^{4(N+1)}.(\mathbf{a}(UDC.N)))$ holds, and that f maintains P. As pointed out in Chapter 1, any assertion of this form is inductive. Therefore we do not give the proof.

Since $\bot^{4(N+1)}.(\mathbf{a}(UDC.N)).i = \bot.(\mathbf{a}(UDC.N))$ for all $i \in [0..4N + 5)$, it is obvious that $P.(\bot^{4(n+1)}.(\mathbf{a}(UDC.N)))$ holds.

For proving the last condition, let $S \in \mathcal{T}^{4(N+1)}.(\mathbf{a}(UDC.N))$ and assume that $P.S$ holds. We have to prove that $P.(f.S)$ holds as well. Let $0 \leq i < N$. Then

$$\begin{aligned} f.S.2.i &= \textbf{pref}\ (up?;ack!;S.3.i \mid down?;ack!;S.1.i) \\ f.S.3.i &= \textbf{pref}\ (up?;ack!;S.2.(i+1) \mid down?;ack!;S.2.i) \\ f.S.0.(i+1) &= \textbf{pref}\ (up?;ack!;S.1.(i+1) \mid down?;ack!;S.1.i) \\ f.S.1.(i+1) &= \textbf{pref}\ (up?;ack!;S.2.(i+1) \mid down?;ack!;S.0.(i+1)). \end{aligned}$$

Using the fact that $P.S$ holds we see that $f.S.2.i = f.S.0.(i+1)$ and $f.S.3.i = f.S.1.(i+1)$ for $0 \le i < N$. This shows that $P.(f.R)$ holds. By fixpoint induction and the definition of f we now have

$$\begin{aligned}
\mu.f.0.0 &= \textbf{pref}\ (up?;\, ack!;\, \mu.f.1.0 \mid down?) \\
\mu.f.1.0 &= \textbf{pref}\ (up?;\, ack!;\, \mu.f.0.1 \\
&\qquad\qquad \mid down?;\, empty!;\, \mu.f.0.0)
\end{aligned}$$

$$\left.\begin{aligned}
\mu.f.0.i &= \textbf{pref}\ (up?;\, ack!;\, \mu.f.1.i \\
&\qquad\qquad \mid down?;\, ack!;\, \mu.f.1.(i-1)) \\
\mu.f.1.i &= \textbf{pref}\ (up?;\, ack!;\, \mu.f.0.(i+1) \\
&\qquad\qquad \mid down?;\, ack!;\, \mu.f.0.i)
\end{aligned}\right\}\ \text{for } 0 < i < N$$

$$\begin{aligned}
\mu.f.0.N &= \textbf{pref}\ (up?;\, ack!;\, \mu.f.1.N \\
&\qquad\qquad \mid down?;\, ack!;\, \mu.f.1.(N-1)) \\
\mu.f.1.N &= \textbf{pref}\ (up?;\, ack!;\, \mu.f.2.N \\
&\qquad\qquad \mid down?;\, ack!;\, \mu.f.0.N) \\
\mu.f.2.N &= \textbf{pref}\ (up?;\, full!;\, \mu.f.3.N \\
&\qquad\qquad \mid down?;\, ack!;\, \mu.f.1.N) \\
\mu.f.3.N &= \textbf{pref}\ (up? \mid down?;\, ack!;\, \mu.f.2.N).
\end{aligned}$$

From this and from the definition of $UDC.(2N+3)$ we conclude that $\mathbf{t}(\mu.f.0)$ contains $\mathbf{t}(UDC.(2N+3))$. Denoting the least fixpoint of the defining equations for $UDC.(2N+3)$ by A, we see that $\mu.f.0.i$ corresponds to $A.(2i)$, $\mu.f.1.i$ corresponds to $A.(2i+1)$, and $\mu.f.j.N$ corresponds to $A.(2N+j)$ for $0 \le i < N$ and $0 \le j < 4$. Applying Lemma 1.4.3 once more, we find that the network behaviors are complete.

Proving that there is no computation interference in the two decompositions is analogous to proofs given before; we omit the proofs here.

5.3 Performance Analysis

There are three criteria by which we judge the efficiency of the implementation: its area complexity, its response time, and its power consumption. In this section we analyze all three.

Calculating the area complexity is easy. Implementing an N-counter requires $\Theta.(\log N)$ of the specified cells.

5.3.0 Response Time

For the response time analysis we transform the behaviors of the cells into cubic ones in a similar way as in Chapter 4. Again, we must make sure that the transformations do not change the order of growth of the response time.

The behavior of the BSC cell is transformed in three steps. In the first step inputs *up* and *down* are replaced by r, outputs *empty*, *ack*, and *full* are replaced by a, internal symbols *se*, *sn*, and *sf* are replaced by p, and *sd* and *su* are replaced by q. We get the following defining equations:

$$\begin{aligned} S.0 &= \textbf{pref } ((r?;a! \mid r?;q;a!);S.1) \\ S.1 &= \textbf{pref } (r?;p;a!;(S.0 \mid S.2)) \\ S.2 &= \textbf{pref } ((r?;a! \mid r?;q;a!);S.1) \end{aligned}$$

and

$$\begin{aligned} S.0 = \textbf{pref } &(p;S.0 \\ &|q;sr!;sa?(S.1 \mid S.2)) \\ S.1 = \textbf{pref } &(q;sr!;sa?(S.0 \mid S.1) \\ &|p;S.1 \\ &|q;sr!;sa?(S.1 \mid S.2)) \\ S.2 = \textbf{pref } &(q;sr!;sa?(S.0 \mid S.1) \\ &|p;S.2) \end{aligned}$$

for the behaviors with respect to the left and right environment respectively. These equations can be simplified to

$$\begin{aligned} S.0 &= \textbf{pref } ((r?;a! \mid r?;q;a!);S.1) \\ S.1 &= \textbf{pref } (r?;p;a!;S.0) \end{aligned}$$

and

$$S.0 = \textbf{pref } *[p \mid q;sr!;sa?].$$

The next transformation consists of omitting the first alternative of the choice in the behavior with respect to the left environment. This does not influence the worst-case response time: the other

alternative gives rise to the worst-case behavior because of the synchronization on q. After doing this we can reduce the number of internal symbols to one to obtain

$$
\begin{aligned}
C^3_{abs} = |[\ & q :: \\
& \quad \mathbf{pref}\ *[r?; q; a!; r?; q; a!] \\
& \| \ \mathbf{pref}\ *[q; sr!; sa?; q] \\
&]|
\end{aligned}
$$

as the abstract behavior for the BSC cell. Note that this is a possible specification for a 4-phase to 2-phase protocol.

For the BSDC cell we can do something similar. Some more steps are involved (all of the same kind as for the BSC cell), but the result is the same. The abstract behavior for the BSDC cell can be defined by C^4_{abs} where

$$
\begin{aligned}
C^4_{abs} = |[\ & q :: \\
& \quad \mathbf{pref}\ *[r?; q; a!; r?; q; a!] \\
& \| \ \mathbf{pref}\ *[q; sr!; sa?; q] \\
&]|\ .
\end{aligned}
$$

The possible end cells, $UDC.1$, $UDC.2$, and $UDC.3$ are all transformed into $\textsc{wire}(r; a)$ components.

To estimate the response time, we first define sequence functions for

$$W.k = (\ \|\ j : 0 \le j < k-1 : s^j C^3_{abs}) \ \| \ \mathbf{pref}\ *[s^{k-1}r?; s^{k-1}a!],$$

for $1 \le k$.

Define σ by

$$
\begin{aligned}
\sigma.(s^j r).i &= j + 2^{j+1} * i \\
\sigma.(s^j a).i &= j + 2^{j+1} * i + 1
\end{aligned}
$$

for $0 \le i$ and $0 \le j$. Then σ restricted to $\mathbf{occ}.(W.k)$ is a sequence function for $W.k$. According to σ all N-counters implemented by BSC and BSDC cells have the same, constant, response time.

The next question we may ask ourselves is whether we have bounded response time if we allow delays of basic components to vary between a lower bound δ and an upper bound Δ. We are interested in finding out whether an implementation with bounded response time exists. So, contrary to our approach in Chapter 4, we do not try to prove anything about the specifications of the abstract cells,

but we look at an implementation. In Figure 5.0 a network of $k-1$ abstract cell implementations is depicted. We prove that the response time of this network is independent of k under the assumption that the response times of the basic components are bounded from below by δ and from above by Δ. For the same reasons as in Chapter 4 we can then conclude that the C^3 and C^4 cells can be used to implement up-down counters with bounded response time. Informally the argument is as follows.

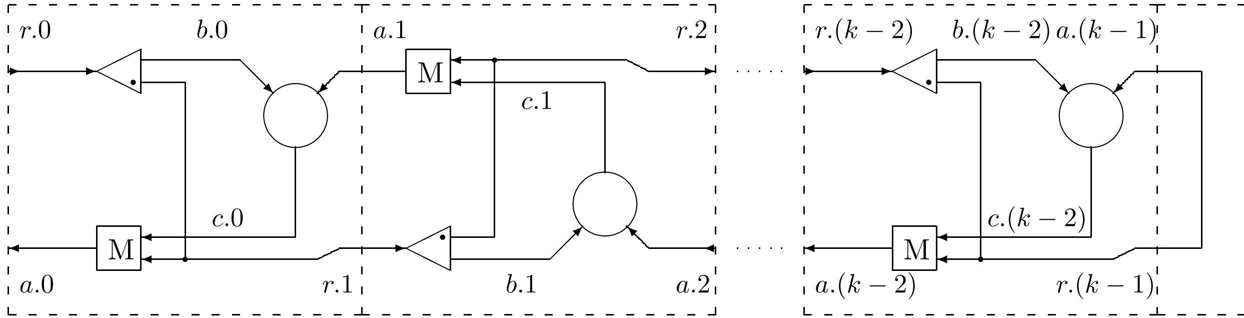

Figure 5.0: An array of C^3_{abs}/C^4_{abs} cell implementations.

If we look only at the structure of the network (see Figure 5.0), the delay between corresponding $r.j$'s and $a.j$'s increases at most linearly in $k-j$. Consider cell $j-1$, for some j greater than zero. Under certain assumptions for the minimum delay between an $a.(j-1)$ event and the next $r.(j-1)$ event, we can even prove that the maximum delay between an $r.(j-1)$ event and the corresponding $a.(j-1)$ event is no larger than the maximum delay between an $r.j$ event and the corresponding $a.j$ event.

The delay between an $a.j$ event and the next $r.j$ event increases with j. As a result, for large enough k there is a cell for which the assumptions for the minimum delay between its outputs to its left environment and the following inputs from its left environment are satisfied. The cell number of this cell determines the response time of the counter, and that cell number does not depend on the number of cells. Hence the counter has bounded response time.

We now give a formal proof. Let $\mathcal{T}$ be a timing function for the array depicted in Figure 5.0 such that according to $\mathcal{T}$ the response time of the JOIN, MERGE, and TOGGLE elements is bounded from below by δ and from above by Δ, and the response time of WIREs is zero. This assumption about the response time of WIREs is not crucial to the argument given below. We prove two properties of $\mathcal{T}$:

0. $$(\forall i,j : 0 \le i \wedge 0 \le j < k-1 \wedge \mathcal{T}.(r.j).(i+1) - \mathcal{T}.(a.j).i \ge 3*\Delta - 2*\delta$$
$$: \mathcal{T}.(a.j).i - \mathcal{T}.(r.j).i \le 3*\Delta),$$

and

1. $(\forall\, i, j : 0 \leq i \wedge 0 \leq j < k : \mathcal{T}.(r.j).(i+1) - \mathcal{T}.(a.j).i \geq 3j * \delta)$.

Proof of 0. The proof is by induction.

- Basic step ($j = k-2$). By the assumptions that the delays of the basic components used in Figure 5.0 are at most Δ and that wire delays are zero, we see that

  $$\mathcal{T}.(a.(k-2)).i - \mathcal{T}.(r.(k-2)).i \leq 3 * \Delta$$

  for any i.

- Inductive step. Let $0 < j < k-1$ and let $0 \leq i$. Assume that the delay between $a.(j-1)$ and consecutive $r.(j-1)$ is at least $3*\Delta - 2*\delta$. The delay between occurrence no. $2i$ of $r.(j-1)$ and occurrence no. $2i$ of $a.(j-1)$ is the delay of a toggle plus the delay of a merge component. Thus

  $$\mathcal{T}.(a.(j-1)).(2i) - \mathcal{T}.(r.(j-1)).(2i) \leq 2 * \Delta.$$

  For the other occurrences of $r.(j-1)$ and $a.(j-1)$ we derive:

  $$\begin{array}{ll}
  & \mathcal{T}.(a.(j-1)).(2i+1) - \mathcal{T}.(r.(j-1)).(2i+1) \\
  \leq & \{\ \mathcal{T}.(a.(j-1)).(2i+1) \leq \mathcal{T}.(c.(j-1)).i + \Delta \text{ and} \\
  & \quad \mathcal{T}.(r.(j-1)).(2i+1) \geq \mathcal{T}.(b.(j-1)).i - \Delta\ \} \\
  & \mathcal{T}.(c.(j-1)).i - \mathcal{T}.(b.(j-1)).i + 2*\Delta \\
  \leq & \{\ \text{Delay of JOIN is at most } \Delta\ \} \\
  & (\mathcal{T}.(b.(j-1)).i \ \mathbf{max}\ \mathcal{T}.(a.j).i) - \mathcal{T}.(b.(j-1)).i + 3*\Delta \\
  = & \{\ \text{Distribution of } + \text{ over } \mathbf{max}\ \} \\
  & 0\ \mathbf{max}\ (\mathcal{T}.(a.j).i - \mathcal{T}.(b.(j-1)).i) + 3*\Delta \\
  \leq & \{\ \text{Induction hypothesis}\ \} \\
  & 0\ \mathbf{max}\ (\mathcal{T}.(r.j).i + 3*\Delta - \mathcal{T}.(b.(j-1)).i) + 3*\Delta \\
  \leq & \{\ \mathcal{T}.(r.j).i \leq \mathcal{T}.(a.(j-1)).(2i) - \delta \text{ and } \mathcal{T}.(b.(j-1)).i \geq \mathcal{T}.(r.(j-1)).(2i+1) + \delta\ \} \\
  & 0\ \mathbf{max}\ (\mathcal{T}.(a.(j-1)).(2i) - \mathcal{T}.(r.(j-1)).(2i+1) + 3*\Delta - 2*\delta) + 3*\Delta \\
  = & \{\ \mathcal{T}.(r.(j-1)).(2i+1) - \mathcal{T}.(a.(j-1)).(2i) \geq 3*\Delta - 2*\delta\ \} \\
  & 3*\Delta.
  \end{array}$$

Proof of 1. This proof is also by induction.

- Basic step ($j = 0$). After having sent an $r.0$, the environment does not send a next $r.0$ before receiving the $a.0$ corresponding to the former $r.0$ (on valid behaviors). Thus

  $$\mathcal{T}.(r.0).(i+1) - \mathcal{T}.(a.0).i \geq 0$$

  for any $i \geq 0$.

- Inductive step. Let $0 \leq j < k-1$ and let $0 \leq i$. We derive:

$$\begin{array}{ll}
 & \mathcal{T}.(r.(j+1)).(i+1) - \mathcal{T}.(a.(j+1)).i \\
\geq & \quad \{ \ \mathcal{T}.(r.(j+1)).(i+1) \geq \mathcal{T}.(r.j).(2i+2) + \delta \ \} \\
 & \mathcal{T}.(r.j).(2i+2) + \delta - \mathcal{T}.(a.(j+1)).i \\
\geq & \quad \{ \ \mathcal{T}.(a.(j+1)).i \leq \mathcal{T}.(a.j).(2i+1) - 2*\delta \ \} \\
 & \mathcal{T}.(r.j).(2i+2) - \mathcal{T}.(a.j).(2i+1) + 3*\delta \\
\geq & \quad \{ \text{ Induction hypothesis } \} \\
 & 3j*\delta + 3*\delta \\
= & \quad \{ \text{ Algebra } \} \\
 & 3*(j+1)*\delta.
\end{array}$$

Let h be the smallest integer solution for j of the equation

$$3j*\delta \geq 3*\Delta - 2*\delta.$$

Then the response time of cell h is bounded from above by $3*\Delta$. Moreover, the response time of the network is bounded as well. The upper bound for the response time depends on h only. Since h is determined by δ and Δ, the upper bound for the response time does not depend on the number of cells in the network.

Incorporating nonzero wire delays into the model results in a bounded response time as well, although the value found for h might be different.

Given implementations of C^3 and C^4 cells, we can choose the values for the delays of the MERGE, TOGGLE, and JOIN elements such that the timed behaviors of the network of Figure 5.0 correspond to timed behaviors of the counter implementation. Therefore we conclude that an N-counter can be implemented with C^3 and C^4 cells, with a response time that does not depend on N.

5.3.1 Power Consumption

The specifications of the BSC and BSDC cells have one property that is important for proving constant power consumption, viz., that the number of communications with their subcomponent is

at most half the number of communications with their environment. We do not prove this formally. It is already indicated by the commands C^3_{abs} and C^4_{abs}.

Let $N > 0$. Consider the weave W of the components of the implementation of the N-counter:

$$W = (\parallel j : 0 \leq j < f.N - 1 : s^j C^{1+g.N.j}) \parallel UDC.(g.N.(f.N-1))).$$

Here f is a function that maps a number to the number of digits of the representation described in Lemma 5.0.0; function g maps the pair (N, i) to digit number i in N's representation.

With V as an abbreviation for $\{ up, down \}$, we derive:

$$\begin{array}{ll}
 & t \in \mathbf{t}W \\
\Rightarrow & \{ \ t \restriction \mathbf{a}(s^i C^{1+g.N.i}) \in \mathbf{t}(s^i C^{1+g.N.i}) \text{ for } 0 \leq i < f.N \ \} \\
 & (\forall i : 0 \leq i < f.N - 1 : 2 * \ell.(t \restriction s^{i+1} V) \leq \ell.(t \restriction s^i V)) \\
\Rightarrow & \{ \text{ Algebra } \} \\
 & (\forall i : 0 \leq i < f.N : 2^i * \ell.(t \restriction s^i V) \leq \ell.(t \restriction V)) \\
\equiv & \{ \text{ Algebra } \} \\
 & (\forall i : 0 \leq i < f.N : \ell.(t \restriction s^i V) \leq (1/2^i) * \ell.(t \restriction V)) \\
\Rightarrow & \{ \ + \text{ is monotonous w.r.t } \leq \ \} \\
 & (\Sigma i : 0 \leq i < f.N : \ell.(t \restriction s^i V)) \leq (\Sigma i : 0 \leq i < f.N : (1/2^i) * \ell.(t \restriction V)) \\
\equiv & \{ \ \mathbf{a}W = (\bigcup i : 0 \leq i < f.N : s^i V); \text{ Algebra } \} \\
 & \ell.t \leq (\Sigma i : 0 \leq i < f.N : 2^{-i} * \ell.(t \restriction V)) \\
\Rightarrow & \{ \ 2^{-i} * \ell.(t \restriction V) \geq 0 \ \} \\
 & \ell.t \leq (\Sigma i : 0 \leq i : 2^{-i} * \ell.(t \restriction V)) \\
\equiv & \{ \text{ Algebra; definition of } V \ \} \\
 & \ell.t \leq 2 * \ell.(t \restriction \{ up, down \})
\end{array}$$

This proves that this implementation has constant power consumption.

Chapter 6

Conclusions and Further Research

6.0 Introduction

In this chapter we discuss the obtained results and present some conclusions. Furthermore we give some suggestions for further research.

6.1 Conclusions

A number of delay-insensitive implementations for up-down counters have been specified and analyzed with respect to their area complexity, response time, and power consumption.

All counter implementations consist of a linear array of cells. The current count of the counter can be derived from the states of these cells. For the simplest of the counters, the current count is just the sum of the internal counts of the cells. This corresponds to unary or radix-1 counting. For the other implementations we used binary or radix-2 counting.

For specifying the behaviors of the cells we used the commands language described in Chapter 1. The weave operator allowed for relatively short specifications, compared to state graphs, for example. Specifying parallel behavior is made easy by specifying the behaviors at the different boundaries separately and then weaving these partial behaviors. Internal symbols can be used to obtain the necessary synchronization. Having the partial behaviors was advantageous in proving the correctness of the proposed implementations. Specifying cells as a weave of partial behaviors also makes it easier to avoid computation interference.

Unary up-down counter implementations turn out to be closely linked to the control parts of stack

implementations. We proved that the power consumption of unary up-down N-counters grows at least logarithmically with N. Since every stack implementation can be seen as an up-down counter, the power consumption of stack implementations grows at least logarithmically with the size of the stack.

The binary counter implementation presented in Chapter 4 shows that counters described earlier, for example in [GL81], can be implemented in a delay-insensitive way. In [GL81] counters with maximum count $2^k - 1$, for some k greater than zero, are designed. We can implement N-counters for any N greater than zero.

Furthermore, in the analysis of the proposed implementation in Chapter 4 we argued that under certain assumptions using sequence functions and the definition of constant response time for sequence functions may not be suitable to analyze the worst-case response time of asynchronous circuits. These assumptions are that the delays of basic components may vary between a lower and an upper bound. A suggestion was made for a definition of bounded response time for a particular class of specifications, namely cubic specifications with alternating inputs and outputs. Subsequently we showed that using this definition, the response time of the implementation depends on the number of cells. For this proof we used an abstraction of the counter cells. The advantage of analyzing the response time of the abstract cells is that each cell has only one input from its left environment and one output to its left environment. As a result, only the delays between that input and that output have to be considered. If there are more inputs and outputs, case analysis might be required.

The counter implementation of Chapter 5 is a new one and is an improvement on all previous implementations. It shows that up-down counters can be implemented with constant power consumption. Constant power consumption was achieved by introducing redundancy in the representation of the current count. Moreover, the implementation's response time is independent of the number of cells, even with respect to our stronger definition. Its area complexity grows logarithmically with its size. Thus, this counter has optimal growth rates with respect to all three performance criteria.

6.2 Further Research

First of all, in this report only high-level implementations of up-down counters are presented. A next step is the decomposition of the cells into smaller (basic) components or directly into transistors.

Second, there are some possible extensions to the up-down counter specified in Chapter 2. A possibility is having the counter count modulo $N + 1$ if the current count is N and another *up* is

received. This is considered useful by some authors [Par87].

Third, in this report we only considered counter implementations consisting of linear arrays of cells. Unary counters can also be implemented by cells configured in a binary tree. Then a logarithmic response time may be obtained without any parallelism in the implementation.

Fourth, counters based on other number systems than radix-1 and radix-2 number systems can be designed. For example, in an implementation with a linear structure, one plus the cell number can be chosen as the weight for its internal count. With digit set $\{0, 1\}$, six can be represented by 100000, 10001, and 1010 (most significant digit on the left). In this way all natural numbers can be represented. Specifying cells for an implementation using this number system requires the introduction of an extra output channel since in each cell the internal count of the subcell is needed in order to determine the next state upon receiving an input. We give a specification for the general cells of such a counter. We have not verified its correctness for all N, but verified a small number of cases using VERDECT.

DEFINITION 6.2.0. Define D^5 as the least fixpoint of

$$
\begin{aligned}
S.0 = \mathbf{pref}\ & (up?; ((se \mid sn); ack_1! \mid sf; empty!); S.1 \\
& \mid down1; (sd_0; ack_0!; S.0 \mid sd_1; ack_1!; S.1) \\
&) \\
S.1 = \mathbf{pref}\ & (up?; (su_0; ack_0!; S.0 \mid su_1; ack_1!; S.1) \\
& \mid down?; ((sf \mid sn); ack_0! \mid se; ack_0!); S.0 \\
&)
\end{aligned}
$$

and define E^5 as the least fixpoint of

$$
\begin{aligned}
S.0 = \mathbf{pref}\ &(se; S.0\\
&|su_0; sup!; (sack_1?; S.2 \mid sfull?; S.3)\\
&)\\
S.1 = \mathbf{pref}\ &(sd_0; sdown!; (sack_0?; S.1 \mid sack_1?; S.2 \mid sempty?; S.0)\\
&|sn; S.1\\
&|su_0; sup!; (sack_1?; S.2 \mid sfull?; S.3)\\
&)\\
S.2 = \mathbf{pref}\ &(sd_1; sdown!; (sack_0?; S.1 \mid sempty?; S.0)\\
&|sn; S.2\\
&|su_0; sup!; (sack_0?; S.1 \mid sack_1?; S.2 \mid sfull?; S.3)\\
&)\\
S.3 = \mathbf{pref}\ &(sf; S.3\\
&|sd_1; sdown!; (sack_0?; S.1 \mid sempty?; S.0)\\
&).
\end{aligned}
$$

Then the counter cell is defined as

$$C^5 = |[\, se, sn, sf, sd_0, sd_1, su_0, su_1 :: D^5.0 \parallel E^5.0 \,]| \,.$$

□

To obtain an implementation of an up-down counter as specified in Chapter 2, the outputs ack_0 and ack_1 of the head cell of this proposed implementation can be merged into one signal ack. The current count of a counter implementation consisting of $k-1$ of these cells, a 1-counter, and a MERGE$(ack_0, ack_1; ack)$ is

$$(\sigma\, i : 0 \leq i < k : c_i * i),$$

where c_i is the internal count of cell i. The implementation of an N-counter requires $\Theta.(\sqrt{N})$ cells of type C^5.

Counters based on other (redundant) number systems may have better average response time than the binary implementations presented in this report.

Fifth, it is not clear how accurate the proposed response time analysis is. Abstracting away from the identity of inputs and outputs of cells as we did in Chapters 4 and 5 may not influence the

growth rate of the response time, but it does influence the constant factors, and we do not know to which extent.

Sixth, we only analyzed the worst-case response time of the proposed designs. One could also analyze the average-case response time. For counters with the same worst-case response time, the average response times might still be different. This seems to be the case for the unary counter implementation of Section 3.1 and a counter based on Martin's lazy stack protocol. For both implementations the worst-case response time is linear in the maximum count, but we suspect the average response time of the latter to be much better. For synchronous implementations this would not be of much interest, since they reflect the worst-case behavior anyway. For asynchronous implementations the difference influences the performance.

Bibliography

[BBB+92] Roy W. Badeau, R. Iris Bahar, Debra Bernstein, Larry L. Biro, William J. Bowhill, John F. Brown, Michael A. Case, Ruben W. Castelino, Elizabeth M. Cooper, Maureen A. Delaney, David R. Deverell, John H. Edmondson, John J. Ellis, Timothy C. Fischer, Thomas F. Fox, Mary K. Gowan, Paul E. Gronowski, William V. Herrick, Anil K. Jain, Jeanne E. Meyer, Daniel G. Miner, Hamid Partovi, Victor Peng, Ronald P. Preston, Chandrasekhara Somanathan, Rebecca L. Stamm, Stephen C. Thierauf, G. Michael Uhler, Nicholas D. Wade, and William R. Wheeler. A 100 MHz macropipelined VAX microprocessor. *IEEE journal of Solid-State Circuits*, 27(11):1585–1598, November 1992.

[Bir67] G. Birkhoff. *Lattice Theory*, volume 25 of *AMS Colloquium Publications*. American Mathematical Society, 1967.

[Bru91] Erik Brunvand. *Translating Concurrent Communicating Programs into Asynchronous Circuits*. PhD thesis, Carnegie Mellon University, 1991.

[Chu87] Tam-Anh Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications*. PhD thesis, MIT Laboratory for Computer Science, June 1987.

[CSS89] Somsak Choomchuay, Somkiat Supadech, and Manus Sangworasilp. An 8 bit presettable/programmable synchronous counter/divider. In *IEEE Sixth International Electronic Manufacturing Technology Symposium*, pages 230–233. IEEE, 1989.

[DCS93] Al Davis, Bill Coates, and Ken Stevens. Automatic synthesis of fast compact asynchronous control circuits. In *Proceedings of the IFIP WG10.5 Working Conference on Asynchronous Design Methodologies*, March 1993.

[Dil89] David L. Dill. *Trace Theory for Automatic Hierachical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations. MIT Press, 1989.

[DNS92] David L. Dill, Steven M. Nowick, and Robert F. Sproull. Specification and automatic verification of self-timed queues. *Formal Methods in System Design*, 1(1):29–60, July 1992.

[Ebe89] Jo C. Ebergen. *Translating Programs into Delay-Insensitive Circuits*, volume 56 of *CWI Tracts*. Centre for Mathemathics and Computer Science, 1989.

[Ebe91] Jo C. Ebergen. A formal approach to designing delay-insensitive circuits. *Distributed Computing*, 5(3):107–119, 1991.

[EG93a] Jo C. Ebergen and Sylvain Gingras. An asynchronous stack with constant response time. Technical report, University of Waterloo, 1993.

[EG93b] Jo C. Ebergen and Sylvain Gingras. A verifier for network decompositions of command-based specifications. In Trevor N. Mudge, Veljko Milutinovic, and Lawrence Hunter, editors, *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences*, volume I, pages 310–318. IEEE Computer Society Press, 1993.

[EP92] Jo C. Ebergen and Ad M. G. Peeters. Modulo-N counters: Design and analysis of delay-insensitive circuits. In Jørgen Staunstrup and Robin Sharp, editors, *2nd Workshop on Designing Correct Circuits, Lyngby*, pages 27–46. Elsevier Science Publishers, 1992.

[Gar93] J.D. Garside. A CMOS VLSI implementation of an asynchronous ALU. In S. Furber and M. Edwards, editors, *IFIP WG 10.5 Working Conference on Asynchronous Design Methodologies*. Elsevier Science Publishers, 1993.

[GL81] Leo J. Guibas and Frank M. Liang. Systolic stacks, queues, and counters. In P. Penfield, Jr., editor, *1982 Conference on Advanced Research in VLSI*, pages 155–164. Artech House, 1981.

[JB88] Edwin V. Jones and Guoan Bi. Fast up/down counters using identical cascaded modules. *IEEE journal of Solid-State Circuits*, 23(1):283–285, February 1988.

[JU90] Mark B. Josephs and Jan Tijmen Udding. Delay-insensitive circuits: An algebraic approach to their design. In J. C. M. Baeten and J. W. Klop, editors, *CONCUR '90, Theories of Concurrency: Unification and Extension*, volume 458 of *Lecture Notes in Computer Science*, pages 342–366. Springer-Verlag, August 1990.

[JU91] Mark B. Josephs and Jan Tijmen Udding. The design of a delay-insensitive stack. In G. Jones and M. Sheeran, editors, *Designing Correct Circuits*, pages 132–152. Springer-Verlag, 1991.

[Kal86] Anne Kaldewaij. *A Formalism for Concurrent Processes*. PhD thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, 1986.

[LKSV91] Luciano Lavagno, Kurt Keutzer, and Alberto Sangiovanni-Vincentelli. Synthesis of verifiably hazard-free asynchronous control circuits. In Carlo H. Séquin, editor, *Advanced Research in VLSI: Proceedings of the 1991 UC Santa Cruz Conference*, pages 87–102. MIT Press, 1991.

[LT82] X. D. Lu and Philip C. Treleaven. A special-purpose VLSI chip: A dynamic pipeline up-down counter. *Microprocessing and Microprogramming*, 10(1):1–10, 1982.

[Man91] M. Morris Mano. *Digital Design*. Prentice Hall, 2nd edition, 1991.

[Mar90] Alain J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*. Addison-Wesley, 1990. UT Year of Programming Institute on Concurrent Programming.

[MSB91] Cho W. Moon, Paul R. Stephan, and Robert K. Brayton. Synthesis of hazard-free asynchronous circuits from graphical specifications. In *Proceedings of ICCAD-91*, pages 322–325. IEEE Computer Society Press, November 1991.

[ND91] Steven M. Nowick and David L. Dill. Automatic synthesis of locally-clocked asynchronous state machines. In *Proceedings of ICCAD-91*, pages 318–321. IEEE Computer Society Press, November 1991.

[Obe81] Roelof M. M. Oberman. *Counting and Counters*. MacMillan Press, 1981.

[Par87] Behrooz Parhami. Systolic up/down counters with zero and sign detection. In Mary Jane Irwin and Renato Stefanelli, editors, *IEEE Symposium on Computer Arithmetic*, pages 174–178. IEEE Computer Society Press, 1987.

[Par90] Behrooz Parhami. Generalized signed-digit number systems: A unifying framework for redundant number representations. *IEEE Transactions on Computers*, 39(1):89–98, 1990.

[Rem87] Martin Rem. Trace theory and systolic computations. In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE: Parallel Architectures and Languages Europe, Vol. I*, volume 258 of *Lecture Notes in Computer Science*, pages 14–33. Springer-Verlag, 1987.

[RMCF88] Fred U. Rosenberger, Charles E. Molnar, Thomas J. Chaney, and Ting-Pien Fang. Q-modules: Internally clocked delay-insensitive modules. *IEEE Transactions on Computers*, 37(9):1005–1018, September 1988.

[Sut89] Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, January 1989.

[Udd86] Jan Tijmen Udding. A formal model for defining and classifying delay-insensitive circuits. *Distributed Computing*, 1(4):197–204, 1986.

[vB92] C. H. (Kees) van Berkel. *Handshake Circuits: An Intermediary between Communicating Processes and VLSI*. PhD thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, 1992.

[vB93] C. H. (Kees) van Berkel. VLSI programming of a modulo-N counter with constant response time and constant power. In S. Furber and M. Edwards, editors, *IFIP WG 10.5 Working Conference on Asynchronous Design Methodologies*. Elsevier Science Publishers, 1993.

[vBKR+91] C.H. (Kees) van Berkel, Joep Kessels, Marly Roncken, Ronald Saeijs, and Frits Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. In *Proceedings of the European Design Automation Conference*, pages 384–389, 1991.

[vdS85] Jan L. A. van de Snepscheut. *Trace Theory and VLSI Design*, volume 200 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[WE85] Neil H. E. Weste and Kamran Eshraghian. *CMOS VLSI Design*. Addison-Wesley VLSI Systems Series. Addison-Wesley, 1985.

[Zwa89] Gerard Zwaan. *Parallel Computations*. PhD thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, 1989.