The Design Problem SCPP-A

Charles E. Molnar
Huub Schols

SMLI TR-95-49 December 1995

Abstract:

Much design effort toward a Sproull Counterflow Pipeline Processor has been focused on management of movements of Instructions and Results in the pipelines so that every Instruction and Result that pass one another meet and interact in exactly one stage of the pipeline. The full SCPP design problem poses other requirements as well, such as creation and deletion of items flowing in the pipelines, scheduling of execution of instructions only in stages with the required hardware, and high speed.

Nevertheless, even a simplified version of the design problem that ignores the latter requirements has resisted synthesis using existing formal methods. At a workshop on Asynchronous VLSI Design held in Israel on March 20-22, 1995, Alain Martin of Caltech discussed his synthesis methodology and tools, which he claimed can translate almost any Communicating Sequential Process (CSP) program to a circuit by systematic procedure. Since our essential requirements for movement of Instructions and Results had been expressed by us as a 5-state FSM graph that is easily interpreted as a CSP program, we asked Martin to demonstrate how his method would be applied to this problem.

At the suggestion of the workshop organizer, Dr. Ran Ginosar of the Technion, Dr. Huub Schols presented the challenge to all of the workshop attendees, and produced the careful documentation contained here. Several thoughtful responses to our challenge are cited in the list of references. They lead us to conclude that the problem that we have posed is indeed difficult and worthy of further study and analysis.

Martin has declined to provide us with any information about a solution that he claimed to have found after the workshop.

M/S 29-01
2550 Garcia Avenue
Mountain View, CA 94043

email address:
charles.molnar@eng.sun.com

© Copyright 1996 Sun Microsystems, Inc. The SML Technical Report Series is published by Sun Microsystems Laboratories, a division of Sun Microsystems, Inc. Printed in U.S.A.

Unlimited copying without fee is permitted provided that the copies are not made nor distributed for direct commercial advantage, and credit to the source is given. Otherwise, no part of this work covered by copyright hereon may be reproduced in any form or by any means graphic, electronic, or mechanical, including photocopying, recording, taping, or storage in an information retrieval system, without the prior written permission of the copyright owner.

TRADEMARKS
Sun, Sun Microsystems, and the Sun logo are trademarks or registered trademarks of Sun Microsystems, Inc. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd. All SPARC trademarks, including the SCD Compliant Logo, are trademarks or registered trademarks of SPARC International, Inc. SPARCstation, SPARCserver, SPARCengine, SPARCworks, and SPARCompiler are licensed exclusively to Sun Microsystems, Inc. All other product names mentioned herein are the trademarks of their respective owners.

For information regarding the SML Technical Report Series, contact Jeanie Treichel, Editor-in-Chief <jeanie.treichel@eng.sun.com>.
For distribution issues, contact Amy Tashbook Hall, Assistant Editor <amy.hall@eng.sun.com>.

The Design Problem SCPP-A

Charles E. Molnar
Sun Microsystems Laboratories
2550 Garcia Avenue
Mountain View, CA 94043

Huub Schols
Department of Mathematics and Computer Science
Eindhoven University of Technology
PO Box 513
5600 MB Eindhoven
The Netherlands

1 Introduction

In this monograph, we present a specification of the design problem that we have called SCPP-A. This design problem is related to the Sproull Counterflow Pipeline Processor, [SpSuMo94a, SpSuMo94b]. On one hand, this problem is small: there exists a five-state diagram that specifies the communication behavior of the process to be designed. On the other hand, the problem seems to have an intrinsic property that is hard to deal with. We hope that this problem is an interesting and small example that encourages many people to apply their favorite design method.

2 Specification of SCPP-A

The SCPP-A has N stages, $N \geq 5$, and N processors; both stages and processors are numbered from 0 up to $N - 1$.

2.1 Instructions and results

Instructions flow up the pipeline of stages from $stage(i)$ to $stage(i + 1)$, for $0 \leq i < (N-1)$; to this purpose, output channel $stage(i).L$ is connected to (is the same channel as) input channel $stage(i + 1).K$, see Figure 1. The type of this channel is *INSTR*.

Results flow down the pipeline of stages from $stage(j)$ to $stage(j - 1)$, for $0 < j \leq (N - 1)$; to this purpose, output channel $stage(j).T$ is connected to (is the same channel as) input channel $stage(j - 1).S$. The type of this channel is *RES*.

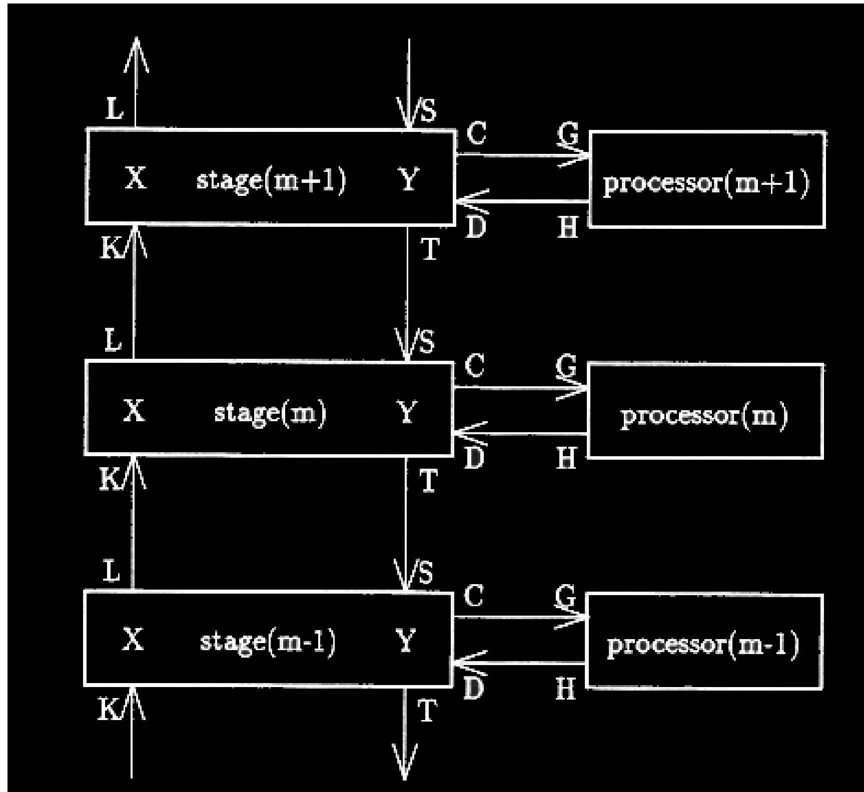

Figure 1. Part of a SCPP-A

2.2 Buffers

There is no buffering on channels. In *stage*(k), $0 \leq k \leq (N-1)$, there is a one-place buffer X of type *INSTR* and there is a one-place buffer Y of type *RES*. Via channel *stage*(k).K, instructions are input into buffer *stage*(k).X; via channel *stage*(k).L, instructions are output from buffer *stage*(k).X. Via channel *stage*(k).S, results are input into buffer *stage*(k).Y; via channel *stage*(k).T, instructions are output from buffer *stage*(k).Y.

2.3 Interaction of instructions and results

The full version of the Sproull Counterflow Pipeline is presented in [SpSuMo94b]. It is not as restricted as the simplified version of the pipeline that we present here. In this simplified version, no instructions should disappear, spontaneously be created, nor overtake one another; the same holds for results. Instructions flow upward through the SCPP-A as fast as possible and results flow

downward through the SCPP-A as fast as possible; however, there is one restriction: each instruction, which flows upward, has to meet and interact with each result, which flows downward. They have to meet in some stage; in which stage they meet is irrelevant.

If an instruction and a result meet in *stage*(*k*), neither of them is output before they have interacted. They interact in *processor*(*k*). To this purpose *stage*(*k*) has an output channel *stage*(*k*).*C*, that is connected to input channel *processor*(*k*).*G* of *processor*(*k*); furthermore, *stage*(*k*) has an input channel *stage*(*k*).*D*, that is connected to output channel *processor*(*k*).*H*. These channels have type *VIR*; the type *VIR* is a vector with two fields: one of type INSTR and one of type *RES*.

Every *processor*(*k*), $0 \le k \le N - 1$, repeatedly receives a vector <*X*,*Y*> via channel *processor*(*k*).*G*, where *X* is the instruction and *Y* is the result. It applies a vector function to this vector; this results in the vector <*X'*,*Y'*>, which has type *VIR*. The vector <*X'*,*Y'*> is sent via channel *processor*(*k*).*H*. Now one cycle of the repetition has been completed.

When *stage*(*k*) receives a vector <*X'*,*Y'*> via channel *stage*(*k*).*D*, *X'* overwrites *stage*(*k*).*X* and *Y'* overwrites *stage*(*k*).*Y*. Notice that the old values are lost, but that the number of values remains the same. Hereafter, outputting is allowed to be resumed. However, it is possible that one instruction meets and interacts with two or more subsequent results in the same stage; analogously, it is possible that one result meets and interacts with two or more subsequent instructions in the same stage. Initially, the pipelines are empty, no communication is pending on a channel, and no buffer in a stage is filled: all are empty.

3 The Goal of the Design Problem SCPP-A

The goal of this design problem is to design the SCPP-A. A suggestion is to design all stages. We strongly prefer all stages to be identical; furthermore, since they communicate with each other, they have to “fit”: activity/passivity of channels, deadlock, etc. As a consequence, a design decision taken in a stage with respect to the communication with the top part of the pipeline, influences the communication of this same stage with the bottom part of the pipeline. The designer has to decide in which stage he or she deals with choosing along which channel to communicate.

4 Communication Behavior of a Stage

In this section, we capture the communication behavior of a single stage in two formalisms, viz. a state graph and a CSP program. The particular way in which the original specification is translated into a formalism may be crucial with respect to the possibilities for manipulating the formal specification in the formalism, as Alain Martin pointed out. One has to be aware that different translations may be possible.

4.1 State graph

The specification above results in the state graph of the communication behavior of a stage shown in Figure 2.

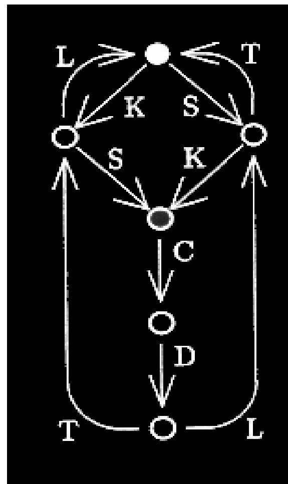

Figure 2. State graph of the communication behavior of a stage

The top state, marked by the full circle, is the initial state.

4.2 Reduced communication behavior

For simplicity, we are willing to combine the subsequent communications via channels *C* and *D* into one communication via channel *Z*. If you think this helps, use the state graph shown in Figure 3; otherwise, stick to the one in Figure 2.

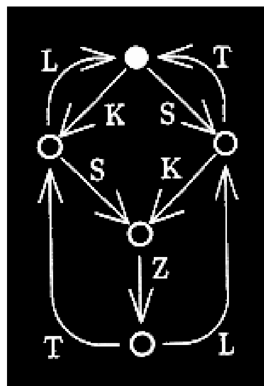

Figure 3. State graph of the simplified communication behavior of a stage

A possible CSP-program (but we know of several other ones) for the simplified communication behavior of a stage is:

```
*[   *[       [ K;L | S;T ] ] ;     [ K;S | S;K ]
   ; *[ Z ; [ L;K | T;S ] ] ; Z ; [ L;T | T;L ]
 ]
```

4.3 Communication between stage and processor

The interesting part of the communication behavior of the stages is the interaction of the communication between the stages with the communication between a stage and its processor. Abstracting from the latter communication by projecting away this communication, leads to the uninteresting state graph shown in Figure 4a; one may want to argue that here the instruction flow and the register flow are independent of each other, as shown in the two state graphs in Figure 4b.

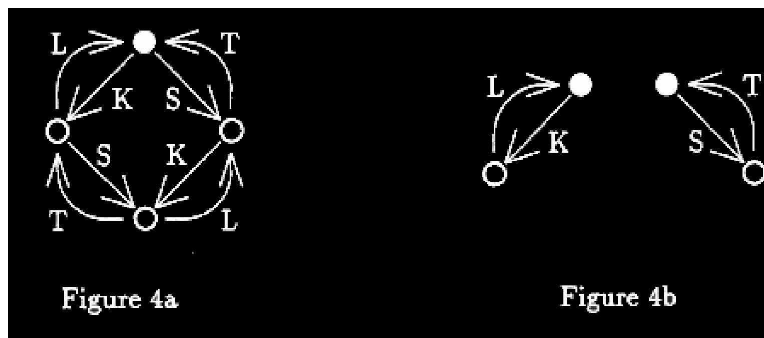

Figure 4. State graph of the communication behavior of a stage after projection

5 Conclusion

We hope that you will find many solutions for designing the SCPP-A specified in this monograph. Huub Schols (email: schols@win.tue.nil) will collect the solutions sent. He will post an overview of the solutions received to the asynchronous mailing list.

This problem was originally posed (be it in a slightly different form) at a Workshop on Asynchronous VLSI Design that was held at Kibbutz Ginosar in Israel on March 20-22, 1995 under the sponsorship of the Ministry of Science of Israel. It was presented there in response to Alain Martin's claim to have a methodology for deriving circuits from CSP programs. Jo Ebergen had

already presented at the Workshop his approach to the problem, using semaphores.

6 Acknowledgements

We acknowledge Alain Martin for suggesting to specify this design problem in English, rather than in a formalism (like a CSP program or a state graph). Alain has made a very fine point by arguing that the translation of the specification to the formalism may be one of the most difficult aspects of this design problem.

We also acknowledge Ian Jones and Jo Ebergen for suggesting important improvements in a draft of this monograph.

7 Epilogue—September 1996

We received several responses to the challenge, each reflecting considerable thought and effort, and each presenting a somewhat different approach to the problem, and some alternative solutions. [LuUd96] and [KoNe95] are relatively complete and self-contained reports. Ebergen discussed his approach at the Israel Workshop, and in slides presented at SunLabs in Mountain View. [EbBe95] touches on the problem in pages 6–8. An approach to this problem using Petri-Net models has been presented by Cortadella, Kishinevsky, Lavagno. and Yakovlev, see [CoKiLaYa95].

Loewenstein's paper [Lo95] and technical report [Lo96] were not a response to our specific challenge, but deal with broader verification issues in the counterflow pipeline processor design.

Martin claimed that a student of his, Robert Southworth, had produced an elegant solution to the problem shortly following the Workshop, but Martin declined to communicate the solution or give us a reference to it.

8 References

[CoKiLaYa95] Cortadella, J., M. Kishinevsky, L. Lavagno, and A. Yakovlev. “Synthesizing Petri nets from State-Based Models.” *Universitat Politèchnica de Catalunya Technical Report UPC-DAC-95-09* (1995). (ftp.ac.upc.es in directory /pub/archives/cad/petrify, file UPC-DAC-95-09.ps.gz.)

[EbBe95] Ebergen, Jo, and Robert Berks. “VERDECT: A Verifier for Asynchronous Circuits.” 1995. (http://maveric0.uwaterloo.ca/publication.html.)

[KoNe95] Korver, Wilbert, and Ivailo Nedelchev. “An Asynchronous Implementation of SCPP-A.” *Technical Report CSRG95-07*, Department of Electrical Engineering, University of Surrey, Guilford GU2 5XH, England. 1995.

[Lo95] Loewenstein, Paul N. “Formal Verification of Counterflow Pipeline Architecture.” Higher Order Logic Theorem Proving and its Applications in *Lecture Notes in Computer Science* 971 (1995): 261–276.

[Lo96] Loewenstein, Paul N. “Formal Verification of Counterflow Pipeline Architecture.” *Sun Microsystems Laboratories Technical Report SMLI TR-96-53* (April 1996).

[LuUd96] Lucassen, Paul G., and Jan Tijmen Udding. “On the Correctness of the Sproull Counterflow Pipeline Processor.” *Proceedings of the Second International Symposium on Advanced Research in Asynchronous Circuits and Systems* (1996): 112–120.

[SpSuMo94a] Sproull, Robert F., Ivan E. Sutherland, and Charles E. Molnar. “Counterflow Pipeline Processor Architecture.” *Sun Microsystems Laboratories Technical Report SMLI TR-94-25* (April 1994).

[SpSuMo94b] Sproull, Robert F., Ivan E. Sutherland, and Charles E. Molnar. “Counterflow Pipeline Processor Architecture.” *IEEE Design & Test of Computers* 11, no. 3 (Fall 1994): 48–59.