DATA FLOW COMPUTING

Theory and Practice

Edited by John A. Sharp

DATA FLOW COMPUTING:

THEORY AND PRACTICE

edited by

John A. Sharp

Department of Mathematics and
Computer Science
University of Wales
Swansea
United Kingdom

ABLEX PUBLISHING CORPORATION
NORWOOD, NEW JERSEY

Printed in the United States of America

Library of Congress Cataloging-in-Publication Data

Data flow computing : theory and practice / edited by John A. Sharp.
p. cm.
Includes bibliographical references and index.
ISBN 0-89391-654-4
1. Computer architecture. 2. Data flow computing. I. Sharp, J. A. (John A.). 1955-
QA76.9.A73D38 1991
004.2'2—dc20 91-30622
CIP

Ablex Publishing Corporation
355 Chestnut St.
Norwood, NJ 07648

List of Authors

Makoto Amamiya, Department of Information Systems, Kyushu University, Japan
Mark Anderson, Palomar Software Inc, USA
Katsuhiko Asada, Department of Electronic Engineering, Osaka University, Japan
Ed A. Ashcroft, Arizona State University, Tempe, USA
Eric Barszcz, NASA Ames Research Center, California, USA
Francine Berman, Department of Computer Science and Engineering, University of California at San Diego, USA
Laxmi N. Bhuyan, The Center for Advanced Computer Studies, University of Southwestern Louisiana, Lafayette, USA
Lubomir Bic, Department of Information and Computer Science, University of California at Irvine, USA
William W. Carlson, Department of Electrical and Computer Engineering, University of Wisconsin, Madison, USA
Thomas J. W. Clarke, Computer Laboratory, University of Cambridge, UK
Tony Faustini, Department of Computer Science, Arizona State University, Tempe, USA
Jose A. B. Fortes, School of Electrical Engineering, Purdue University, Indiana, USA
Jean-Luc Gaudiot, Department of Electrical Engineering Systems, University of Southern California, Los Angeles, USA
Dipak Ghosal, Institute of Advanced Computer Studies, University of Maryland, USA
Janice Glasgow, Department of Computing and Information Science, Queen's University, Kingston, Canada
Jayantha Herath, George Mason University, Fairfax, Virginia, USA
Susantha Herath, Keio University, Yokohama, Japan
Jagan Jagganathan, SRI International, Menlo Park, USA
Shinji Komori, LSI R&D Laboratory, Mitshubishi Electric Corp., Itami, Hyogo, Japan
G. H. MacEwen, Queen's University, Kingston, Canada
Satoshi Matsumoto, VLSI Development Laboratory, Sharp Corporation, Tenri, Nara, Japan

Souichi Miyata, VLSI Development Laboratory, Sharp Corporation, Tenri, Nara, Japan

Walid Najjar, Department of Electrical Engineering Systems, University of Southern California, Los Angeles, USA

Hiroaki Nishikawa, Department of Electronic Engineering, Osaka University, Japan

Carlos Ruggerio, Instituto de Fisco and Quimica de S. Carlos, Brazil

Nobuo Saito, Keio University, Yokohama, Japan

John Sargeant, Department of Computer Science, University of Manchester, UK

John A. Sharp, Department of Mathematics and Computer Science, University of Wales, Swansea, UK

Kenji Shima, Product Development Laboratory, Mitshubishi Electric Corp., Amagasaki, Hyogo, Japan

D. B. Skillicorn, Oxford University, England; on leave from Queen's University, Kingston, Canada

Vason P. Srini, Electronics Research Laboratory, University of California, Berkeley, USA

Hiroaki Terada, Department of Electronic Engineering, Osaka University, Japan

Bill Wadge, Department of Computer Science, University of Victoria, Canada

Yuk Kui Wong, AT&T Network Systems, UK, Ltd.

Yoshinori Yamaguchi, Electrotechnical Laboratory, Niiharigun, Ibraki, Japan

Toshitsugu Yuba, Electrotechnical Laboratory, Niiharigun, Ibraki, Japan

Preface

This book contains a selection of chapters on the general subject of data flow computing. A quick introduction to the subject, including a rough guide to the terminology, is provided in Chapter 1. In Chapter 2 the introductory theme is continued with a comparison of the various data flow computing models that have been proposed.

We then begin to examine the whole process of program design and implementation, starting at the highest level. There are many design methodologies in use, such as those attributed to Michael Jackson and Ed Yourdon, among others, which claim to use a data flow approach. In Chapter 3 we examine the relationship between these approaches and the data flow model introduced in Chapter 1. The theme of how the data flow approach affects the programmer is continued in Chapter 4, which discusses the relationship between data flow and both functional programming and logic programming.

Chapters 5 and 6 turn to the implementation of data flow programming ideas on real hardware with comparisons of various data flow machines that have been proposed. Assessing the performance of the alternatives is clearly an important topic, and methods for doing so are discussed in Chapters 7 to 9.

How to improve the performance is the topic of the next series of contributions. We kick off with a discussion of multilevel execution in data flow architectures in Chapter 10. The problem of removing unwanted tokens in data flow computations is covered in Chapter 11. Chapter 12 sees a presentation of a model for executing data flow programs which reduces some of the overload involved in allowing tokens to drive the execution of individual operations. This is done by identifying sequences of operations which cannot be done out of order. In Chapter 13 ways of finding the best such sequences for allocation to processors are discussed.

Many people regard finding as much parallelism as possible in programs as the best way of achieving maximum speed-up. This is not always the case, and in Chapter 14 the problems that occur when we have too much potential parallelism are studied.

In Chapter 15 we take a look at various aspects of the traditional static data flow model with reference in particular to handling loop structures.

Next we have the first of a series of contributions describing specific computer architectures. Reduced instruction set computers are very much in vogue at the moment, and Chapter 16 shows how these ideas may be combined with the data

flow approach to computing. The Japanese approach is reflected in Chapters 17 and 18 with a discussion of a one-chip, data-driven processor, and a graph reduction model.

The aim of parallelism is often considered to be merely to get increased throughput. However, parallel processing is a useful way of increasing reliability by computing the same result a number of times in parallel. The possibility of using the data flow approach to achieve fault-tolerant parallel processing is the subject of Chapter 19.

Chapters 20 and 21 take a slightly wider definition of data flow than is commonly accepted with an introduction to a variation on the data flow theme termed *intensional programming* and a discussion of the syntax, semantics, and applications of *operator nets*.

The final chapter consists of a very personal view of where research into data flow might be heading, together with a brief survey of some of the work around the world that is not directly represented by papers in this volume.

I would like to take this opportunity to thank all the authors who agreed to contribute to this book. Thanks are also due to Ablex Publishing Corp. for suggesting that I might like to put together a collection of research papers on data flow. I apologize profusely to all concerned for the length of time it has taken for this volume to actually be published. I could try to lay the blame at the door of the various contributors, but the unfortunate truth is that I have been rather lax in chasing papers and actually getting round to the task of putting it all together. Despite the delays I hope that this book will prove a valuable source of ideas for all working in and around the data flow area.

Thanks are due to the Institute of Electrical and Electronic Engineers for permission to reprint the papers by Vason Srini as Chapters 5 and 19 and Chapter 17 by Hiroaki Terada et al. I would also to thank Springer-Verlag for permission to reprint the paper by Thomas Clarke as Chapter 16.

As editor I must take final responsibility for any errors or omissions in the chapters that follow. My earnest hope is that I have not introduced too many errors in trying to correct the few minor typographical ones in the contributors' original manuscripts.

John A. Sharp

Table of Contents

1

A Brief Introduction to Data Flow

John A Sharp

The purpose of this chapter is to provide a brief introduction to the field of data flow computing as discussed in this book. In doing so I will attempt to clarify some of the terminology used which inevitably varies between different researchers. The most obvious difference is in fact the very name of the field. Some workers (myself included) use the two word phrase data flow, *while many others (including the majority of contributors to this volume) concatenate the phrase into one word* dataflow. *In editing the volume I considered adopting one convention throughout, but in the end I decided to leave well enough alone, and hope that a few footnotes would suffice to avoid too much confusion.*

It must be remembered that the chapters have been prepared in isolation, so that there is inevitably a fair amount of duplication of introductory material. I felt that it would be unwise to attempt to remove this, not only as different workers use slightly different terminology, but also since I expect that many people will be tempted to dip in to the book and read only selected contributions. Thus it is not unreasonable to expect each chapter to stand alone.

1.1 MOTIVATION

In the past few years there has been an increasing interest in data flow programming techniques. This interest has been motivated partly by the rapid advances in technology (and the needs for distributed processing techniques), partly by a desire for faster throughput by applying parallel processing techniques, and partly by a search for a programming tool that is closer to the problem-solving methods which people naturally adopt than are current programming languages. Current languages are often referred to as problem oriented, but their design has been strongly influenced by the design of the computers that they are used with and thus are still to a certain extent machine oriented.

With the advent of the microprocessor and the ever-advancing technology of integrated circuits it is becoming increasingly obvious that we need to break

away from the straight-jacket of the conventional approach to computing. The principles upon which it has been based are becoming less and less realistic in the light of present day knowledge. One obvious illustration of the inadequacy of the conventional approach is the way in which millions of memory cells are associated with only one processor. The same technology is used to develop both processors and memory circuits, yet the processor is being efficiently utilized and is in constant use, whereas memory cells are sitting idle for most of the time, and being grossly under utilized. We are not proposing that we should use memory cells merely to increase some theoretical utilization measure, but rather indicating that there is clearly a possibility that, by using the memory available more efficiently, we may either be able to run programs faster or execute larger programs with the same resources.

Although there have been earlier attempts to break away from the traditional approach, none have been totally successful in achieving popular support and also they have nearly always assumed that any language implementation would be based on a single sequential processor.

The so-called software crisis has led many eminent computer scientists to call for a more structured approach to the design of programs. This is another reason for us to rethink our whole approach to programming in the light of current technology. The data flow approach discussed in this book is an approach which provides an attractive alternative for the solution of the above problems.

1.2 THE BASIC IDEAS

In order to do anything useful with a computer system, we must be able to specify what operations we want to be carried out. This usually implies that a program needs to be written. Exceptions to this are dedicated systems that are designed to implement a limited range of functions. Even then the designer of the system has had to specify a form of program for the system to execute, either permanently stored in memory or hard-wired into the control circuitry of the machine itself.

In specifying the operations to be done, we make assumptions about what primitive operations are available and also about how they are carried out. In other words we have in mind a picture of how the computations we are specifying will be executed. This picture of how things are done is referred to as a *model of computing*. Programmers are not always aware that this is what they are doing, since any programming language automatically provides us with a model of computing. This model may be an abstraction of the physical processes involved in the operation of computer hardware. Conventional programming languages nearly all assume the existence of a primitive set of arithmetic operations (add, multiply, etc.). They also assume that these operations are carried out sequentially on data stored in some form of memory device. Alternatively, the

model may be a more formal one, developed using mathematical theory, which is totally independent of hardware operation. Lisp uses a formal model of computing that is less hardware dependent, but which still assumes that it will be implemented on a traditional digital computer.

Most existing programming languages were designed for use on computers based on John von Neumann's original concept. This is why they implicitly take as their model of computing the traditional execution cycle (fetch; execute; store) proposed by von Neumann and adopted in virtually all digital computers since. This model of computing is referred to both as the *von Neumann model* and the *control flow model.*

1.3 SOME TERMINOLOGY

Our aim in this chapter is to briefly introduce the model of computing known as the data flow model. In order to do so, let us introduce some terminology. Some of the definitions given will be rather imprecise, but they will be adequate to give the reader a flavor and more precise definitions may be found in later chapters.

A *program* may be defined as the specification of a set of operations that are required in order to carry out some task.

Examples of the sort of operations we are talking about are conventional machine operations (such as add, subtract, etc.).

Two aspects of this definition of a program should be emphasized.

1. No ordering of the operations is implied.
2. Not all operations will be executed for all sets of input data.

The subset of operations that are executed given a particular set of input data defines a *computation.*

Two ordering conventions can be defined. The *control flow* ordering is based on the idea of a temporal sequence of operations. The *data flow* ordering, on the other hand, is based on the need for data.

A *data flow program* is one in which the ordering of operations is not specified by the programmer, but is that implied by the data interdependencies.

A *control flow program* is one in which there is a total ordering of operations specified by the programmer.

Hence:

A *data flow computation* is one in which the operations are executed in an order determined by the data interdependencies and the availability of resources.

A *control flow computation* is one in which the operations are executed in an order predetermined by a control convention, used by the programmer.

Two varieties of data flow computation can be distinguished.

1. *data-driven computations* in which operations are executed in an order determined by the availability of input data.
   and
2. *demand-driven computations* in which operations are executed in an order determined by the requirements for data.

The data flow approach is often associated solely with the use of data-driven computations. In this book we have taken a rather more liberal line and some of the contributors include both data-driven and demand-driven execution strategies.

The notion of a computation is such that the same set of operations are carried out whenever the computation is executed. With a program we expect a different set of operations to be carried out, depending upon the input data. A program thus represents a set of computations. In order to decide which operations are to be carried out, we need to introduce some sort of *run-time data-dependent decision mechanism*. It is in the set of such mechanisms introduced that the different notations used by various researchers tend to differ most (provided they accept the basic data-driven model).

1.4 AN OVERVIEW OF THE DATA FLOW MODEL

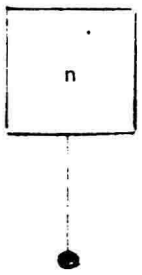
Data flow research has been proceeding now since the late 1960s, with the first paper being, as far as I am aware, by Adams [Ada70]. This chapter is in fact based on Adams's doctoral thesis, which was published in 1968 [Ada68]. Most people have been introduced to data flow through the work at MIT, and in particular the work of Jack Dennis [Den74]. This approach may be regarded as an example of the classic, data-driven data flow model.

The classic way of representing computations in the data flow model is based on a graphical notation for programs with nodes representing operations, and arcs representing data dependencies.

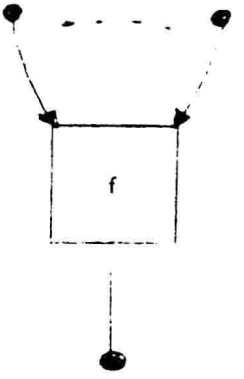
Four early notations are presented in [Ada70], [Den74], [Rum77], and [Kos73]. Most subsequent work has been based on Dennis's notation [Den74], but in passing we will mention some of the features included in the other models.

1.4.1 Basic Nodes for Computations

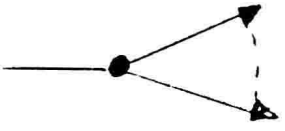
In order to represent simple computations we require three basic nodes: a *source node* or *constant generator* to introduce constant values, some sort of *copy* or *duplicate node* in order to replicate values, and a set of *operator* or *function nodes*. The set of primitive nodes introduced is usually roughly equivalent to the



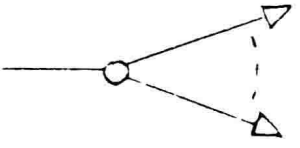
Source actor or node
(generates data value n)



Data actor or function node
(applies function f to inputs)



Data link
(allows for multiple copies
of data values)



Control link
(allows for multiple copies
of boolean values)

Figure 1.1. Primitive nodes in Dennis's notation.

instruction set of a conventional machine. Dennis's notation for representing these primitives is summarized in Figure 1.1.

1.4.2 Run-time Data Dependent Decision Mechanisms

In order to represent programs additional nodes capable of directing values along selected paths are required. Dennis [Den74] introduces two sorts of nodes (see Figure 1.2): the *merge* node, where the output token placed on the output arc is selected by the value (true or false) of the token on the control input, and *T & F Gates*, where an output token (of value equal to the input token) is placed on the output arc if the value of the control token is true (for a T-gate) or false (for an

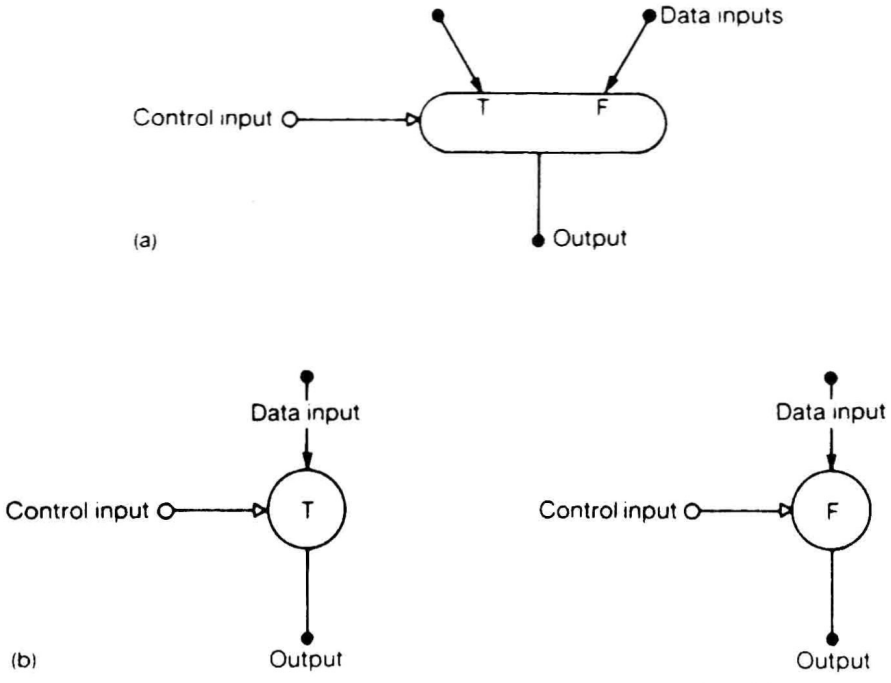


Figure 1.2. Nodes for run-time data-dependent decisions in Dennis's notation. (a) The merge node. Which input arc the output token is taken from depends upon the value (true or false) of the token on the control input. (b) T- and F-gates. An output token (of value equal to the input token) is placed on the output arc if the value of the control token is true (for a T-gate) or false (for an F-gate).

F-gate). Otherwise the input and control tokens are both absorbed. Graphs representing familiar simple loop and conditional constructs of conventional programming can now be constructed (see Figure 1.3).

Rumbaugh [Rum77] assumes that, to produce well-formed data flow graphs, you must always use T & F gates and merge nodes in a controlled manner. This assumption seems to be justified. He introduces two alternative control nodes (Figure 1.4): the *switch*, where the input token is placed on the output arc selected by the control input, and the *merge*, where it is the programmer's responsibility to ensure that only one input arrives at any one time. The input is then placed on the output. The conditional and while-loop constructs given above may also be programmed using Rumbaugh's nodes (Figure 1.5). Note that it is no longer necessary to have an initial token on an arc in the loop graph. It should be noted that Rumbaugh's nodes are more suited to expressing the above constructs, but are not as general, or powerful (in some sense) as Dennis's.

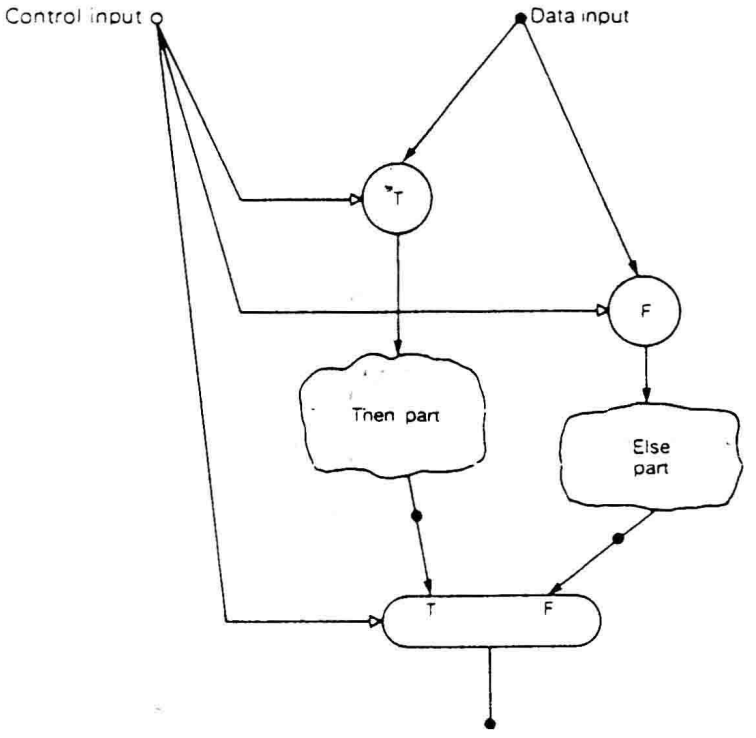


Figure 1.3a. A conditional construct.

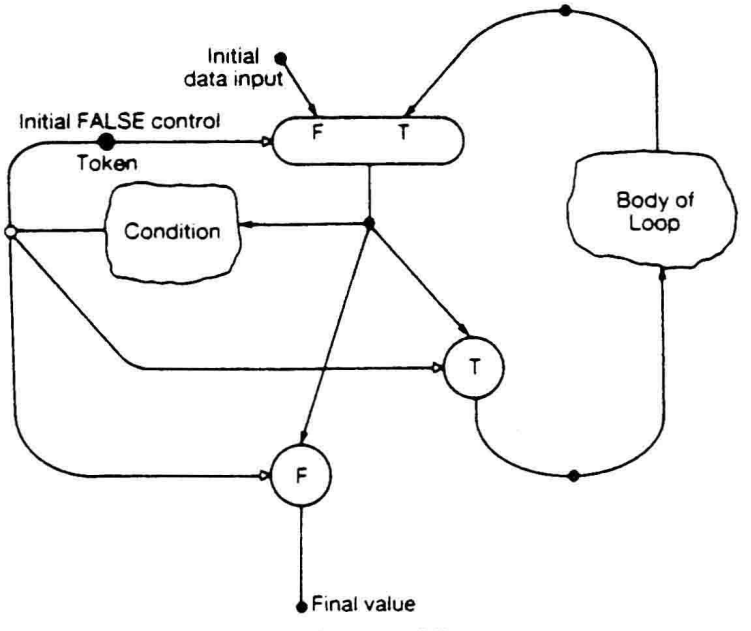


Figure 1.3b. A while loop.

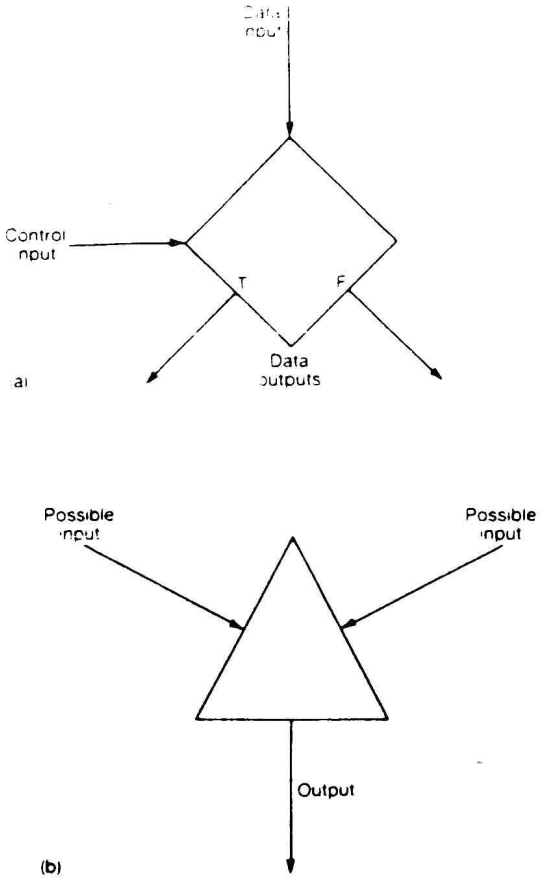


Figure 1.4. Rumbaugh's control nodes. (a) The switch. The input token is placed on the output are selected by the control input. (b) The merge. It is the programmer's responsibility to ensure that only one input arrives at any one time. The input is then placed on the output.

Adams [Ada70] also uses two sorts of nodes (Figure 1.6): *select and route* (roughly equivalent to Dennis's merge node), and *conditional* (and *negative conditional*) *route* (roughly equivalent to Dennis's T and F gates).

1.4.3 Synchronization

Both MIT notations ([Den74], [Rum77]) insist on synchronizing the execution of operations by having all inputs present before execution. On the other hand,