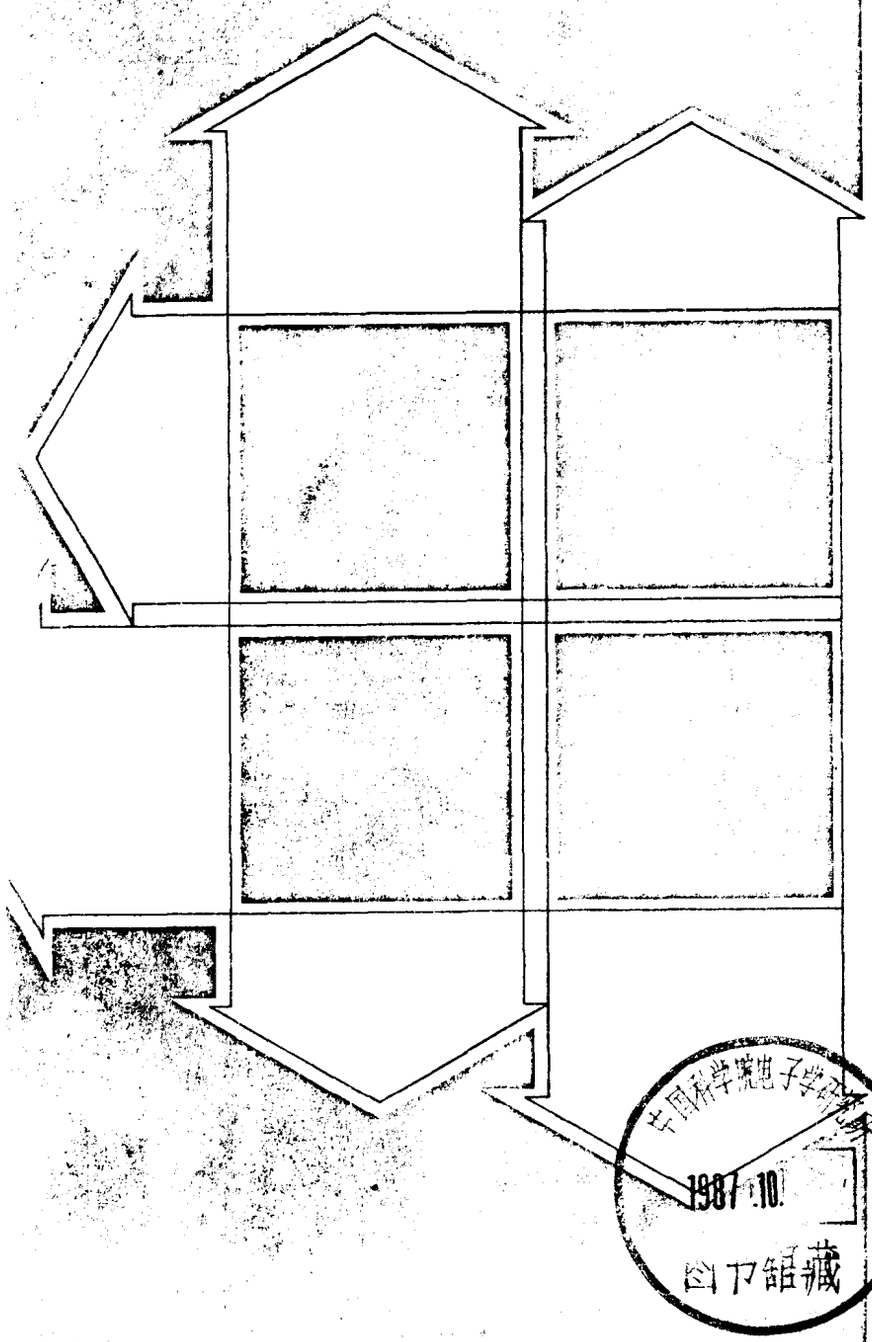
The 5th International Conference on

DISTRIBUTED COMPUTING SYSTEMS

PROCEEDINGS



The 5th International Conference on

DISTRIBUTED COMPUTING SYSTEMS

Denver, Colorado
May 13-17, 1985

SPONSORED BY

IEEE COMPUTER SOCIETY

1884 1984 **THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.**

In cooperation with
Association for Computing Machinery (ACM)
Information Processing Society of Japan (IPSJ)
Institut National de Recherche en Informatique et en Automatique (INRIA)

ISBN 0-8186-0617-7
IEEE Catalog Number 85CH2149-3
Library of Congress Number 85-60194
IEEE Computer Society Order Number 617

COMPUTER SOCIETY PRESS

Published by IEEE Computer Society Press
1109 Spring Street
Suite 300
Silver Spring, MD 20910

ISBN 0-8186-0617-7 (paper)
ISBN 0-8186-4617-9 (microfiche)
ISBN 0-8186-8617-0 (case)
IEEE Catalog Number 85CH2149-3
Library of Congress Number 85-60194
IEEE Computer Society Order Number 617

Order from: IEEE Computer Society
Post Office Box 80452
Worldway Postal Center
Los Angeles, CA 90080

IEEE Service Center
445 Hoes Lane
Piscataway, NJ 08854

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

General Chairman's Message

Welcome to the 1985 International Conference on Distributed Computing Systems. The design, development, and deployment of effective distributed computing systems requires the integration of skills from many disciplines including architecture, database management, fault tolerance, networks, operating systems, and software, to name a few. The breadth of the field is so great that this conference was created in 1979 to provide a forum bringing together interested workers from industry and academia for a week of intense interaction, synergism, and cross-fertilization.

This is the first conference after "annualization." The change to an annual cycle was made to provide more timely interaction between workers in this fast-paced field. Now that the conference is on an annual basis, planning should be simpler: each year the International Conference on Distributed Computing Systems will be held in mid-May. Perhaps you will want to mark your calendars now for the 1986 conference in Boston. Mike Liu is the general chairman. Based on my experience in working with Mike over the last several years, I can assure you that the 1986 conference will be highly professional in all respects.

It is my pleasure to acknowledge the prodigious work of our many "volunteers." The committee chairs, the program committee, and the referees have all given freely of that most precious of all commodities, their time. (It is interesting to note that people who are busy always seem willing and able to take on additional tasks.) Please accept my sincere thanks for jobs well done: we couldn't have done this without your help. Finally, I extend my thanks to all participants in the International Conference on Distributed Computing Systems: authors, panelists, session chairs, and most of all, attendees. You are the greatest!

Earl Swartzlander
General Chairman

Preface

This Proceedings of the 1985 International Conference on Distributed Computing Systems is the fifth in the series of meetings sponsored by the IEEE Computer Society Technical Committee on Distributed Processing since October 1979.

The conference received more than 130 submissions from 14 countries, each of which was evaluated by three referees. From these evaluations, 63 papers were selected by the 52-member Program Committee for presentation at the conference and for inclusion in the Proceedings. It is a great pleasure for me to serve as Program Chairman for the conference. I am pleased that the conference is upholding the tradition of being a high-quality, truly international technical meeting.

The technical program is classified into seven subject areas, each of which was handled by a Program Vice-Chairman responsible for that area: Architectures, Networks, Operating Systems and Languages, Distributed Databases, Fault Tolerance, Performance Evaluation, and Applications. I would like to take this opportunity to thank the respective Vice-Chairmen, H.C. Torng, Chuan-lin Wu, John A. Stankovic, David Cohen, K.H. (Kane) Kim, Bill P. Buckles, and Charles J. Graff, for handling the review of papers in their areas. I would also like to express my sincere appreciation to other members of the Program Committee for assisting the review process, to Bernard P. Zeigler and Horst F. Wedde for organizing sessions on Models of Distributed Processes, and to Andre van Tilborg and Edwin C. Foudriat for organizing the two panel sessions.

I would like to acknowledge the tremendous amount of support received from Earl Swartzlander, General Chairman, and Charles Vick, Steering Committee Chairman. Many others, including 168 referees who are not members of the Program Committee, have given freely of their time and expertise. Their names are listed in this Proceedings, but their real satisfaction is that their efforts have helped in making this conference a success.

I should like to call to your attention a special issue of the IEEE Transactions on Computers on distributed computing that will be published in December 1985. All authors of the 63 papers appearing in this Proceedings have been invited to submit their revised/modified/expanded versions for consideration. It is an honor for me to serve as Guest Editor for this special issue.

Finally, I would like to thank Jack B. Dennis, the winner of the 1984 ACM-IEEE/CS Eckert-Mauchly award for his contributions to data-flow computer architecture, for accepting our invitation to be the Keynote Speaker. I have known Jack for more than 10 years and have high respect for his technical contributions in many aspects of computer science and engineering. I trust that you will enjoy his keynote address, which is timely and fresh.

Ming T. (Mike) Liu
Program Chairman

Fifth International Conference on Distributed Computing Systems

General Chairman
Earl Swartzlander, *TRW*

Standing Committee Chairman
Charles H. Vick, *Auburn University*

International Associate Chairman
H.J. Siegel, *Purdue University*
Helmut Kerner, *Austria*
R.C.T. Lee, *Taiwan*
Gerard LeLann, *France*
Soichi Noguchi, *Japan*
Maria Giovanna Sami, *Italy*
Sigran Schindler, *Germany*
G.J. Smit, *Netherlands*
C.M. Woodside, *Canada*

Professional Societies Liaison
Larry D. Wittie, *SUNY Stony Brook*

Publicity
S. Diane Smith, *Lawrence Livermore Labs*

Local Arrangements
John Pollicmus, *Computer Technology Associates*

Treasurer
Leah Jamieson, *Purdue University*

Awards Chairman
C.V. Ramamoorthy, *University of California, Berkeley*

Tutorial Chairman
Troy Nagle, *University of North Carolina*

Program Chairman
Ming T. Liu, *Ohio State University*

Special Publications Chairman
Stephen F. Lundstrom, *Stanford University*

Vice Chairman
Architectures
H.C. Torng, *Cornell University*
Metin Feridun, *IBM Laboratories*
Bryan Law[illegible] *Livermore Laboratory*
Hungwen Li[illegible] *Watson Research Center*
Warren A. Mo[illegible], *AT&T Bell Labs*
Kang G. Shin, *University of Michigan*
David Tjon, *IBM Kingston*

Vice Chairman
Operating Systems and Languages
John A. Stankovic, *University of Massachusetts*
Ray Campbell, *University of Illinois*
Domenico Ferrari, *University of California, Berkeley*
John K. Gallant, *AT&T Bell Labs*
Martin McKendry, *Georgia Institute of Technology*
James McGraw, *Lawrence Livermore National Laboratory*
Al Spector, *Carnegie-Mellon University*

Vice Chairman
Distributed Databases
David Cohen, *Teknekron Infoswitch*
Daniel H. Fishman, *HP Labs*
Hector Garcia-Molina, *Princeton University*
Donald F. Hayden, Jr., *AT&T Information Systems*
Gideon Lidor, *AT&T Bell Labs*
Jane W.S. Liu, *University of Illinois*
Manda B. Sury, Lockheed

Vice Chairman
Networks
Chuan-lin Wu, *University of Texas, Austin*
Dharma P. Agrawal, *North Carolina State University*
Paul D. Amer, *University of Delaware*
M.G. Gouda, *University of Texas, Austin*
Robert J. McMillen, *Hughes Aircraft*
Richard Y. Oh, *AT&T Bell Labs*
Zvonko G. Vranesic, *University of Toronto*

Vice Chairman
Fault Tolerance
K.H. (Kane) Kim, *University of South Florida*
Tom Anderson, *University of Newcastle Upon Tyne*
Ed C. Foudriat, *NASA Langley Research Center*
Herbert Hecht, *SoHaR, Inc.*
H. Kopetz, *Technische Universitaet Wien*
Zary Segall, *Carnegie-Mellon University*
Raif M. Yanney, *TRW*

Vice Chairman
Performance Evaluation
Bill P. Buckles, *University of Texas, Arlington*
Walter H. Kohler, *University of Massachusetts*
Michael K. Molloy, *University of Texas, Austin*
Charles H. Sauer, *IBM Austin*
Paul Spirakis, *New York University*
Kishor Trivedi, *Duke University*
Bernard P. Zeigler, *Wayne State University*

Vice Chairman
Applications
Charles J. Graff, *U.S. Army CECOM*
Angus Andrews, *Rockwell International*
Henry Chuang, *University of Pittsburgh*
Raymond Liuzzi, *Rome Air Development Center*
Fred Petry, *Tulane University*
Derdek Morris, *Stevens Institute of Technology*
Peter A. Ng, *University of Missouri*
Paul Schneck, *Office of Naval Research*
Norman F. Schneidewind, *Naval Postgraduate School*

List of Referees

Amir Abduelnaga
George B. Adam III
Adarsh K. Arora
Dusham Badel
Daniel Barbara
Andrew G. Barto
Peter Bates
Marc Beacken
Geneva Belford
Mark Benard
L.N. Bhuyan
Paul Blackwell
Joshua Block
Kevin W. Bowyer
Opal A. Brass
Jeff Brumfield
John Bruner
L.F. Cabrera
David A. Carlson
Chung-Kuo Chang
Dah-Ming Chin
Ching-Hua Chow
Ey-Chih Chow
Arturo I. Concepcion
Eric Cooper
Richardo S. Cordori
Dean Daniels
Nathaneil J. Davis IV
A.D. Dehkordi
Nigel Derett
Laurie Dillon
Perry Emrath
N.M. Endo
James C. Ferraus
Eric Fiene
Victor Franco
James L. Frankel
Bruce Galler
Robert Geist
Jack Goldberg
Houssam A. Halabi
Allan W. Haley, Jr.
Alfred C. Hartmann
Al Hayashi
John P. Hayes
William R. Hawe
Michael T. Heins
John H. Holland
Victor Hom
Pei Hsia
Edward Hunter
Ronald R. Hutchins
S.M. Jacobs
Sushil Jajodia
Pankaj Jalote
M. Jameel
Roberto R. Kampfner
Krishna M. Kavi
Robert Keller
Jack Kent
Aaron Kershenbaum
H. Khalil
Robert E. Kinichi
Hank Korth
C.M. Krishna
M. Krishna
Clyde Kruskal
Manoj Kumar
Ming-Yee Lai
Simon S. Lam
Richard LeBlanc
Insup Lee
Yann-Hang Lee
Ernest W. Leggett, Jr.
Dennis Leinbaugh
Steven Leviton
Richard C. Lian
Woei Lin
An-Chi Liu
John Lohse
Frank Luk
Anthony V. Ma
Katie Macrander
Miro Malek
Robert Marti
Bill McDonald
David McNabb
James McSkimin
Sherri Menees
Jai Menon
Bart Miller
Mike Minnich
Toshimi Minoura
J.H. Mirza
Joe Mohan
Clifton L. Moss, Jr.
Nicolas J. Multari
N. Natarajan
Victor P. Nelson
Vladimir Nepustil
James W. Nippert
Dan O'Donnell
Ron Olsson
S. Ong
M. Ossefort
Mourad Oulid-Aissa
Susan S. Owicki
B.K. Padmanabhan
Janak H. Patel
Randy Pausch
Lynn L. Peterson
G.F. Pfister
Frank M. Pittelli
D.K. Pradhan
Russell A. Putzke
C.S. Raghavendra
K.K. Ramakrishnan
Krith Ramamritham
Brain Randell
Daniel A. Reed
Anthony Reeves
Robert G. Reynolds
Mark Riley
Marshall Rose
Robert Rosenthal
L. Rosier
J. Rozenblit
Robin Sahner
A.R.K. Sastry
Prashant S. Sawkar
T. Saydam
Frank Schaffa
Fred B. Schneider
Edmond Schonberg
Peter Schwarz
Robert Seban
A.Sethi
Amit Sheth
Edward N. Shipley
S.K. Shrivastava
Timothy M. Sigmon
Luca Simoncini
Adit D. Singh
Anoop Singhal
Stephen K. Skedzielewsdi
S. Diane Smith
Todd Smith
Arthur Sorkin
Eugene Spafford
Clyde H. Springen
R.J. Stroud
N.K. Swain
Kamaal Thadani
K.S. The
William C. Thibault
L. David Umbaugh
James H. Vellenga
M.K. Vernon
Stephen T. Vinter
Constantin von Altrock
A. von Mayrhauser
Ben Wah
Horst Wedde
Jack Wileden
John Wilkes
Seung Ming-Yang
Pen-Chung Yew
Songman Zhou

Table of Contents

**This paper appears on page 552.

Session 1A
Data Flow Systems

Chairman

H.C. Torng

THE HUGHES DATA FLOW MULTIPROCESSOR

Rex Vedder, Michael Campbell, George Tucker

Microelectronics Engineering Laboratory,
Hughes Aircraft Company, Box 902, El Segundo, CA

ABSTRACT

The Hughes Data Flow Multiprocessor (HDFM) Project has developed a data flow architecture and software environment for high performance signal and data processing. The programming environment allows applications coding in a functional high level language called the Hughes Data Flow Language (HDFL). The HDFL is compiled to a data flow graph form which is then automatically partitioned and distributed to multiple processing elements. The data flow architecture consists of many processing elements connected by a three dimensional bussed-cube packet routing network. The processing elements have been designed for implementation in VLSI to provide large throughput real-time processing for embedded processor applications. The modular nature of the computer allows adding additional processing elements to meet a range of throughput and reliability requirement. Simulation results have demonstrated high performance operation with high level language programmability.

1.0 INTRODUCTION

The Hughes Data Flow Multiprocessor (HDFM) Project began in 1981 prompted by the need for high performance, reliable, and easily programmable processors for embedded systems. VLSI now allows a many-fold increase in the number of circuits which can be employed within the small volume and power limitations of the embedded processor. This has led to the increased use of parallel processors to achieve higher performance. Unfortunately, the use of parallel processors increases the programming complexity, requiring an applications programmer to partition and distribute his program to multiple processors, and to explicitly coordinate communication between the processors or shared memory. Applications programming is extremely expensive even using current single processor systems and is often the dominant cost of a system. The goal of the HDFM Project has been to develop a high performance multi-processor system which is programmed in a high level language as a single computer, with the multi-processor configuration being transparent to the user. This not only reduces software cost but allows re-mapping an existing program onto a different numbers of processing elements without re-writing the program; this is fundamental for fast fault recovery and useful for meeting changing real-time performance requirements.

To meet these increasing performance requirements and reduce growing software costs, the HFDM Project has developed a data flow architecture and software environment for high performance signal and data processing. The programming environment allows applications coding in a functional high level language called HDFL (Hughes Data Flow Language). The HDFL is compiled to a data flow graph form which is then automatically partitioned and distributed to multiple processing elements. The data flow architecture consists of many processing elements connected by a three dimensional bussed-cube packet routing network. The processing elements have been designed for implementation in VLSI to provide large throughput real-time processing for embedded processor applications. The modular nature of the computer allows adding additional processing elements to meet a range of throughput and reliability requirement. Simulation results have demonstrated high performance operation with high level language programmability.

2.0 DATA FLOW ORGANIZATION

Multi-processors can be organized by many different methods. Data flow control is particularly attractive because it can express the full parallelism of a problem and reduce explicit programmer concern with inter-processor communication and synchronization. Much work has been published on data flow blossoming from the work started at MIT [Dennis74]. Our work is aimed specifically at performing signal processing problems and the related data processing functions including tracking, control, and display processing on the same processor. We use a micro data flow approach and compile time assignment of tasks to processing elements to get efficient run-time performance.

Data flow, as opposed to the traditional control flow computation model (with program counter), lets the data dependencies of a group of operations determine the sequence in which the operations may be executed. A data flow graph

represents this information using nodes for the operations and directed arcs defining the data dependencies between these nodes, which are called actors. The output result from an actor is passed to other actors via data items called tokens which travel along the arcs.

The actor execution, or firing, occurs when all of the actor's input tokens are present on the actor's input arcs. When the actor fires, it consumes the values off its input arcs, performs its intended operation and puts the result token on its output arc(s). When actors are implemented in an architecture they are called templates. Each template consists of an opcode, slots for operands, and destination pointers which indicate where the results of the operation should be sent. For a more thorough introduction to data flow see [Dennis74] [Dennis80].

3.0 SOFTWARE ENVIRONMENT

The HDFM is programmed in the Hughes Data Flow Language (HDFL), which is a functional high level language based on Val [Ackerman79]. We have developed a compiler which translates from HDFL to a parallel data flow graph form. This graph is then distributed to the processing elements of the multi-processor by a software tool called the Allocator. The Allocator employs static graph analysis to produce a compile-time assignment of program graph to hardware that attempts to maximize the number of operations which can proceed in parallel while minimizing the inter-processing element communication. The software environment for the HDFM is shown in Figure 1. Parallel development of high level language, compiler, and architecture have allowed many hardware/software tradeoffs to be studied resulting in a design which can be efficiently implemented.

3.1 Hughes Data Flow Language (HDFL)

One of the primary goals of our project was to provide a high level language capability for a multi-processor system in order to reduce software cost. This entailed finding a high level language which could easily express the parallelism inherent in many problems. Current sequential languages like Fortran and Pascal were eliminated because of their inherent lack of parallelism. Ada and other multi-tasking languages were rejected because they require explicit programmer concern with creating and synchronizing multiple tasks which adds complexity and cost to software development. Within a specific process, these languages are also subject to the same lack of parallelism as the Fortran-class languages. We determined that an applicative data flow language such as Val [McGraw82] or Id [Arvind78] was needed to allow an effective extraction of parallelism and efficient mapping to multi-processor hardware. This led to the development of the HDFL.

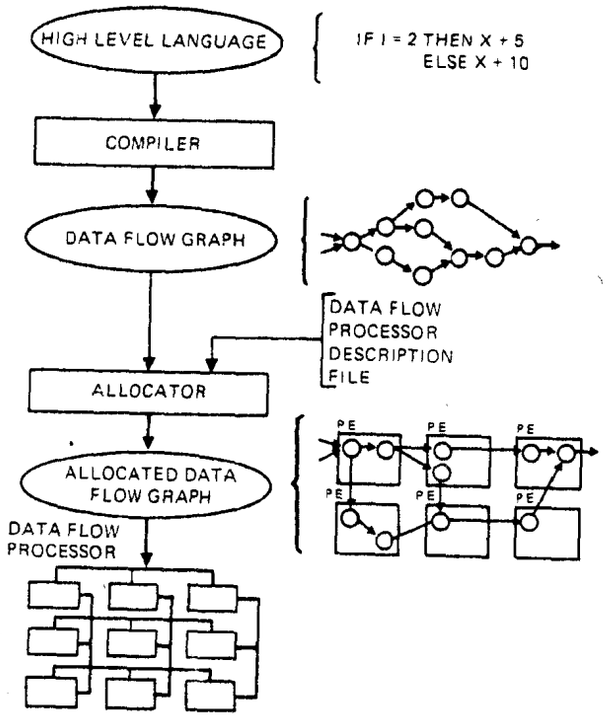


Figure 1. The Data Flow Software Environment Allows Programming the Data Flow Processor in High Level Language

HDFL is a general purpose high level programming language for data flow computers which is designed to allow full expression of parallelism. It is an applicative language [Backus78] but includes the usage of familiar algebraic notation and programming language conventions. HDFL borrows many of its features from Val [McGraw82], a data flow language developed at MIT.

The language is value oriented, allowing only single assignment variables. Its features include strong typing, data structures including records and arrays, conditionals (IF THEN ELSE), iteration (FOR), parallel iteration (FORALL), and streams. The stream capability is similar to that used in the ID language [Arvind78]. A HDFL program consists of a program definition and zero or more function definitions. There are no global variables or side effects; values are passed via parameter passing.

Figure 2 shows a simple example of HDFL which illustrates the flavor of the language. The example consists of a function "foo" which takes four parameters (one record and three integers), and returns one record and one integer. "Result" is a key word beginning the body of a function and "endfun" terminates it. The function body consists of a list of arbitrarily complex expressions separated by commas with one expression per return value. In this example the first expression in the function body is a "record type constructor" which assigns values to the fields of the record result. The conditional below it evaluates to an integer value. Constants and types may be declared before the function header or before the body. Functions may be nested.

```
type xy = record [ x: integer; y: integer ];
constant scalefactor = 2;  %This is a comment.

function foo( xyvar:xy; x0,y1,y2:integer
        returns xy, integer)

constant offset = 1;

result
   xy( xyvar.x + x0, xyvar.y + y1) ,
   if y1 > y2        %Either branch creates a
                      single value.
      then y2 * scalefactor + offset
      else y1 + x0
   endif
endfun
```

Figure 2. HDFL Programs are Composed of Functions

3.2 HDFL Compiler

The compiler translates HDFL to a data flow graph intermediate form composed of primitive data flow actors. Operation proceeds in three phases - syntax-checking and parse tree construction, semantics-checking and augmentation, and finally code generation. Each phase is table-driven, using the Hughes' Translation Table Generator [Tucker82]. Following table-driven code generation is a final post-processing stage to eliminate un-needed code, evaluate constant subgraphs, and perform some optimizations. The graph intermediate form generated by the compiler includes syntactic information and other information which is used by the Allocator.

The primitive actors are those supported directly by the hardware. We currently have 39 primitive actors, some with both 16-bit and 32-bit forms. Many are simple arithmetic and boolean actors such as ADD, others are control actors such as ENABLE and SWITCH, or hybrids like LE5, some are used in function invocation such as FORWARD, and others are used for array and stream handling. The five above named actors and their function are shown in Figure 3.

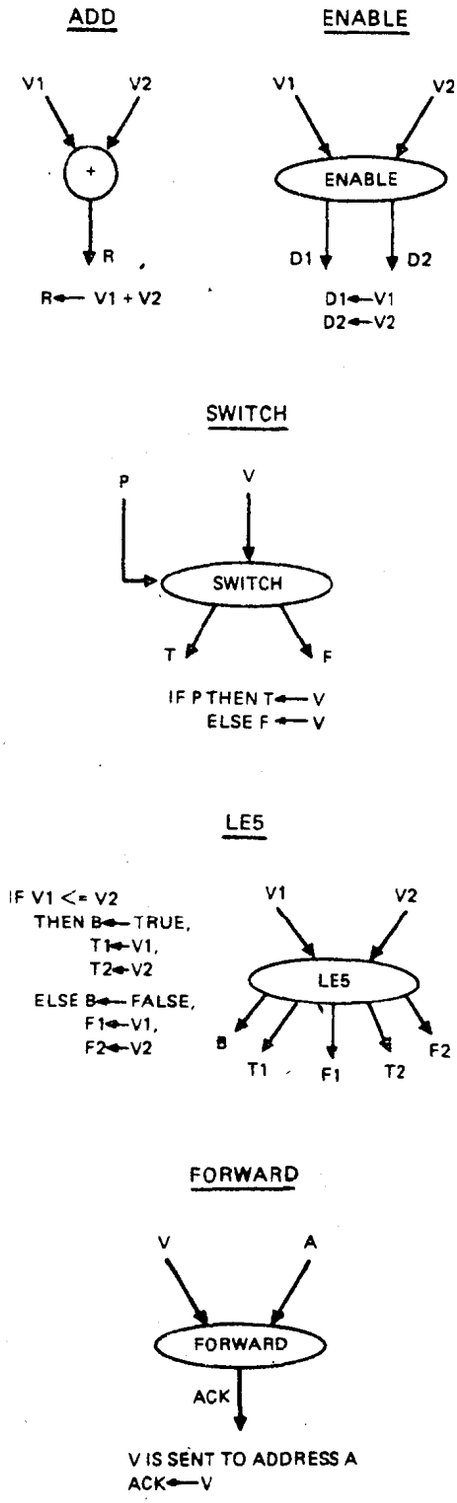


Figure 3. The Primitive Actors are Implemented Directly in Hardware

For each construct in the high level language the compiler has a corresponding data flow graph composed from the primitive actors that implements that function. For example, the data flow graph generated from the HDFL conditional expression, "if y1 <= y2 then y2 * 2 + 1 else y1 + x0 endif" is shown in Figure 4. The then and else branches of the conditional are merged together by sending tokens on these arcs to the same location; this is indicated by merging the output arcs together. Note also that the LE5 actor has some stub output arcs which are not used. The ENABLE actor is present so that when the result of the expression is generated this guarantees that all actors in the graph have fired and the graph is available for further use if desired.

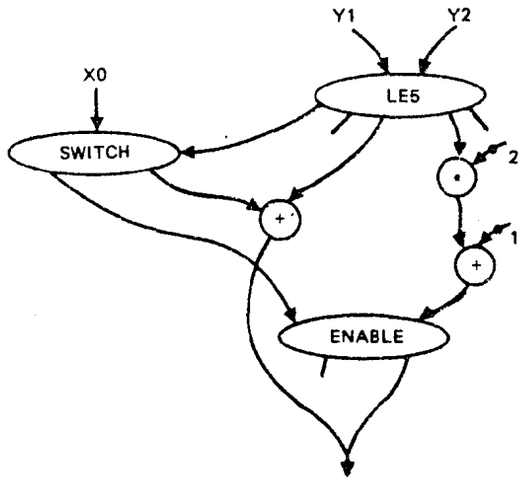


Figure 4. Compiler Generated Data Flow Graph Corresponding to
if y1 <= y2 then y2 * 2 + 1
else y1 + x0 endif

3.3 Allocator

The Allocator is a software tool which takes the data flow graph intermediate form from the compiler and a description of the number and configuration of processing elements and assigns each actor to an actual processing element. Note that each processing element may have many actors assigned to it. The Allocator attempts to satisfy two conflicting goals: (1) maximize the parallelism of execution in the data flow graph by assigning actors that can fire in parallel to different processing elements, and (2) minimize the communication traffic between processing elements by assigning actors that are connected by arcs to the same processing element.

The Allocator algorithm works in a divide and conquer fashion by partitioning the input graph into smaller modules each of which is assigned to a subset of the processing elements. The modules can then be further divided and assigned to smaller sets of processing elements until the trivial assignment of an actor or group of actors to one processing element is made. At each step, the assignment of modules to processing elements is guided by heuristic functions which estimate the communications overhead and parallelism of different distributions of actors so that the best solutions may be selected.

The partitioning algorithm splits up the data flow graph along boundaries implied by the syntax of the high level language source code. To guide this process, the data flow intermediate graph form includes the syntactic parse tree from the compiler. For example, the first step of the allocator algorithm is to partition the parse tree into subtrees which correspond to high level language functions. Each of these subtrees corresponds to a module which can then be further partitioned into still smaller modules corresponding to smaller language constructs such as loop bodies or expressions. Since each construct in our language is side-effect free, we expect this partitioning to achieve good locality of reference and therefore help minimize inter-processor communication.

4.0 ARCHITECTURE

The HDFM consists of many relatively simple identical processing elements connected by an global packet-switching network. The architecture is designed to be modular and fault tolerant, and has been targeted for VLSI implementation. The interconnection network is integrated with the processing element for ease of expansion and minimization of VLSI chip types.

4.1 Communications Network

The global interconnection network is a three dimensional bussed-cube network as shown in Figure 5 which implements a fault tolerant store-and-forward packet switching network. Each processing element contains queues for storing packets in the communication network, and the appropriate control for monitoring the health of the processing elements and performing the packet routing.

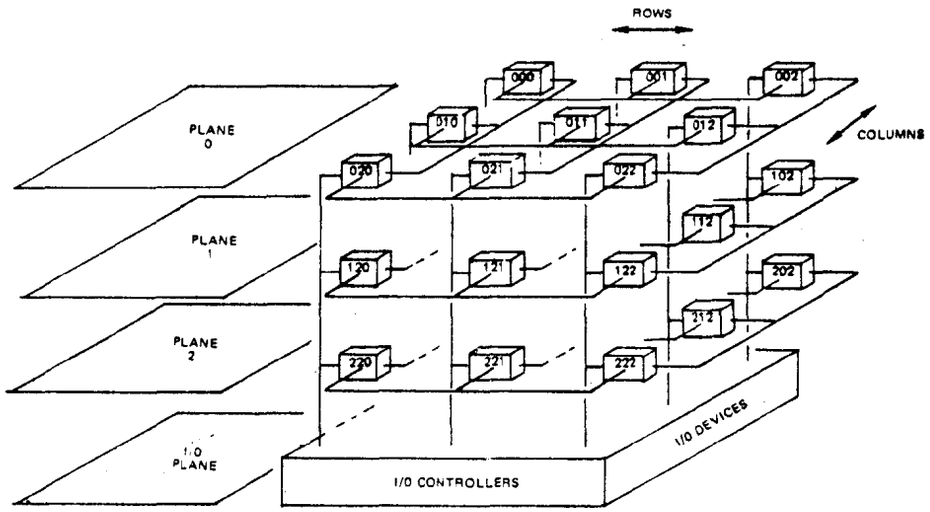


Figure 5. The Communications Network is a Three-Dimensional Bussed Cube Structure - Pictured is a 3 by 3 by 3 Configuration of Processing Elements

The network is optimized for transmission of very short packets consisting of a single token. Each packet consists of a packet type, an address, and a piece of data. Different types of packets include normal token packets, initialization packets, and special control packets for machine re-configuration control. The address of each packet consists of a processing element address and a template address which points to one particular actor instruction within a processing element. The data can be any of the allowed data types of HDFL or control information if the packet is a control packet.

The communications network is designed to be reliable, with automatic retry on garbled messages, distributed bus arbitration, alternate path packet routing, and failed processing element translation tables to allow rapid switch-in and use of spare processing elements.

The communications network can physically accomodate up to an 8x8x8 configuration or 512 processing elements. Many signal processing problems could potentially use this many processing elements without overloading the bus capacity because of the ease of partitioning some of these algorithms. However, for general data processing, the bus bandwidth will begin to saturate above four processirg elements per bus. More processing elements can be added and performance will continue to increase but at lower efficiency per processing element.

4.2 Processing Element

Each processing element consists of two parts - the above mentioned communications functionality and a processing engine which can perform the primitive actor operations and the data flow sequencing control. Each processing element has local memory which is used for program and data storage; there is no global memory. The processing element has been targeted for VLSI implementation and consists of two custom chips - a communications chip and a processor chip - and some commercially available memory parts. This organization is shown in Figure 6.

The communications chip receives packets from the routing network and either forwards them on to other processing elements or sends them to the processor chip. When the processor chip receives a packet, it checks if this token has enabled a template to fire. If so, the operands and opcode for this template are sent to the ALU. The ALU will perform the indicated operation and send the result to be matched with its destination address and sent either back to this same processing element or out into the routing network.