Gottlieb, Israel

STRUCTURED DATA FLOW: A QUASI-SYNCHRONOUS INTERPRETATION OF DATA DRIVEN COMPUTATIONS

City University of New York PH.D. 1986

PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark ✓.

1. Glossy photographs or pages _____
2. Colored illustrations, paper or print _____
3. Photographs with dark background _____
4. Illustrations are poor copy _____
5. Pages with black marks, not original copy _____
6. Print shows through as there is text on both sides of page _____
7. Indistinct, broken or small print on several pages ✓
8. Print exceeds margin requirements _____
9. Tightly bound copy with print lost in spine _____
10. Computer printout pages with indistinct print _____
11. Page(s) _____ lacking when material received, and not available from school or author.
12. Page(s) _____ seem to be missing in numbering only as text follows.
13. Two pages numbered _____. Text follows.
14. Curling and wrinkled pages _____
15. Dissertation contains pages with print at a slant, filmed as received ✓
16. Other _____

University
Microfilms
International

Structured Data Flow

A Quasi-synchronous Interpretation of Data Driven Computations

By

Israel Gottlieb

A Dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy. The City University of New York

1986

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy

Jan 27, 1991
Date

Chair of Examining Committee

Jan 29, 1991
Date

Executive Officer

Professor Michael Anshel

Professor Ali Ghozati

Professor John Moyne

Professor Howard Wasserman

Professor Jerry Waxman

Supervisory Committee

The City University of New York

Abstract

STRUCTURED DATA FLOW
A Quasi-Synchronous Interpretation of Data Driven Computations

by
Israel Gottlieb

Adviser Professor Jacob Rootenberg

This thesis develops an interpretation of data flow graphs which imposes an additional level of execution discipline on the usual data flow mechanism A data flow graph is first partitioned into *computation paths* in such a way that every node belongs to exactly one path As in data flow, a node becomes enabled when all of its data have arrived However, the scheduled activity which results is not restricted to the individual node The assigned processor may continue to execute the successors of that node along the computation path to which it belongs This execution proceeds in a sequential fashion much as it does in a conventional von Neumann processor In order for the nodes encountered in the course of such sequential execution to be executable, any operands they require must be available If this is found not to be the case, the execution sequence is suspended via a form of exception handling

A set of basic programming constructs based on this approach is developed and an operational semantics is defined for them Determinacy and liveness in a modified form are demonstrated for computations expressed in this system

The question of what constitutes an optimal partition of data flow graphs, under the SDF execution discipline, is addressed An optimality criteria for static programs is defined and an algorithm for achieving it is given

A preliminary design of an architecture for implementing the SDF execution discipline is presented The performance improvement over conventional data flow which can be expected from SDF is discussed

Acknowledgements

I would like to express my gratitude and appreciation to Professor Jacob Rootenberg for his unflagging support and sage advice throughout my graduate work.

Thanks are also due to the members of my doctoral committee for their involvement in this thesis. I would like especially to thank Dr. Howard Wasserman for the time and effort that he put in with me on the graph theoretical portions of this work.

I would also like to thank Dr. Chris Vickery for the time we spent discussing the SDF concept when it was still just an idea.

On a personal note, I would like to thank Dr. Jerry Waxman for having lured me into computer science in the first place and remaining a steadfast friend ever since.

As with most doctoral theses, the most trying burden was borne by the author's spouse. No amount of written thanks would adequately express the significance of her support.

Table of Contents

List of Figures

List of Tables

Introduction

1. Foreword

Computer structures for parallel processing have been the focus of considerable research in the past two decades The progress which has been made, however, might well be seen as disappointing when compared to the results achieved in other areas of computer science during the same period System overhead continues to plague virtually all multiprocessor implementations with more than a few processors, frustrating their designers' hopes for attaining anything close to the theoretical potential of such systems Further, even this theoretical potential has eluded much effort at generalization, i e how to best partition the work of an algorithm into parallel processes Such questions as the best level of granularity for a given problem, the type of controlling mechanism -- whether centralized or distributed, synchronous vs asynchronous activities and even whether or not there exists a parallel machine which is efficient for a large class of problems, remain the subject of considerable debate The work in this thesis is a contribution in this arena Our focus is on parallel machines, ones which can be expected to handle a significant class of problems efficiently This is in contrast to work which focuses on algorithms designed to utilize parallelism more effectively or on the development of dedicated architectures for particular algorithms such as e g the systolic arrays of Kung et al [27]

Very roughly, one can divide the directions which have been taken by parallel machine researchers into two classes The first assumes that the basic building block of a system is a conventional von Neumann computer In this thesis a von

Neumann computer is characterized by a) a list of instructions stored sequentially which facilitates a very simple automatic mechanism for progressing through the execution of a program: fetch, increment a program counter and execute the instruction, repeated until the instruction list is terminated, and b) a memory in which data and intermediate results are stored, accessed in the course of the execution of the instructions by a naming of locations in the memory. To construct a parallel computing system, several (or large numbers) of these conventional machines are connected together in some way so that they can cooperate on a single problem. The focus of this research has invariably been on the method of interconnection. To appreciate the significance of this aspect of multiprocessor systems, one need only note that physical installations with several processors have been successfully used for many years in *multiprogramming* environments. Each job on the system is assigned a single processor, the multiple processors allowing jobs to be running concurrently. This is easily facilitated because the communication between separate jobs is minimal or zero. It is an entirely different matter when a single job or algorithm is contemplated for parallel execution. Partitioning the job into separately executable processes invariably implies significant communication between the processes. For example, to execute a matrix multiplication in parallel, the same row or column must be used as a multiplier by every processor, hence they must all compete for access to that data. For high performance multiprocessor systems the emphasis has been on *tightly coupled* systems, i.e. where communication is not implemented by actual transmission of data along some medium, e.g. a bus, but rather via a common memory which can be accessed by most or all processors in the system. Here the cost of communication is translated into the problem of contention by different processors for the same

memory location (the matrix multiply is an example). State of the art efforts along these lines are exemplified by the Ultracomputer project at New York University [21].

Our work in this thesis belongs to the other basic approach to parallel machines, that which attempts a radical departure from the von Neumann architecture. The idea that such a departure is called for was eloquently promoted by John Backus in 1978 [22]. He argued largely from a language developer's point of view, that the increasingly large software systems being developed would become impossible to implement efficiently and debug properly if the basic von Neumann architecture were to continue in use and dictate the design of programming languages. The difficulty stems from the *memory state*, which is a basic part of programming a von Neumann computer. Variables correspond to memory locations and, in order to account for all possible side effects of changing the value of a variable, a programmer must be able to keep track of its effect on all parts of the program; in effect, he or she must be aware of the entire memory state. This becomes an impossibly complex task for large systems. Backus argued for functional programming systems in which there is no such thing as a memory state. Other authors [7, 31, 13] pointed out the increasing need for systems with ultra-high speed computing capability and the fact that parallel execution is the only way to speed up a system based on logic components of a given switching speed. The von Neumann computer, with its sequential execution discipline, would seem to be an ill suited candidate for the basic building block of such systems. Thus, the quest for alternate architectures was motivated by at least two quite different considerations. Salient among the approaches being pursued in the last decade is the so called *data flow architecture*. The work presented here

relates to the data flow approach to supercomputing

2. Background

By 1970 a number of graph based formal models of parallel computation had been developed A directed graph can be thought of as an intuitively appealing representation of a partial order An algorithm can be expressed as a set of activities and a partial order, where the ordering specifies dependencies between the activities That is, if activity a must precede activity b, then $a < b$ in the partial ordering We can represent each activity as a node or vertex in the graph and insert a directed arc from node a to node b if, for the corresponding activities it is true that $a < b$ Thus a directed graph is a natural way of expressing parallel computation Of course, such a graph model does not help us decide what the activities are, i.e what operations to include in a node This *level of granularity* can be taken to be anything from the most primitive logical operations at the bit level to activities which are sophisticated algorithms in their own right The earliest models were developed in the context of relatively few processors and attempted to answer questions such as how best to assign activities to the available pool of processors with the object of achieving the shortest completion time (see for example [11]) With the advent of large scale integration the issue of limited processor resources became less central and theoretical work focused on how best to make use of an unlimited supply of processors [2, 24, 30, 28, 26] . Most of these models made no assumptions as to the delay associated with the execution of each activity, only the sequencing implied by the explicit dependences was assumed Thus, for any given computation represented within

the model, considerable variation was possible as to the exact sequence of operations in an execution of that computation.

In order to derive various properties of these models, it was found necessary to add a certain amount of 'interpretation' to the graph itself, over and above the simple association of nodes with activities and arcs with dependencies. The method of communication between activities had to be made explicit, and the arcs between nodes presented a natural and workable way of defining the locus of such communication. Thus, when the activity represented by a node terminates, it was understood to forward the result of its computation to other activities, exactly as specified by the arcs incident to it which are directed to other nodes. Since the delay incurred by different activities is unspecified in the model, the outputs of one activity can accumulate in the communication channel (i.e. on the 'arc') faster than the activity to which they are directed can consume them. Some provision for multiple operands on arcs must therefore be made. A common device employed in these models is the FIFO queue; each arc is defined as an unbounded first-in-first-out buffer.

Much of the theoretical work on these models focused on developing control mechanisms that would support decision and iteration constructs in such a way that the determinacy of the computations represented within the model could be demonstrated. By determinacy is meant that every execution of a computation represented within the model would yield the same set of outputs for a given set of inputs. Other properties which were investigated include conditions for the termination of computations, deadlock, conditions for boundedness of the queues and the equivalency of different computation schemas within a given model.

Once the movement of data on arcs is defined as the sole basis for sequencing of activities, it follows that an activity should be allowed to execute as soon as all input operands have arrived No central control mechanism is required, each locus of activity can be locally controlled and may proceed asynchronously to other activities the only control mechanism being the detection of the arrival of operands and the enabling of an activity when all such operands have been made available Thus the notion of *data flow* or *data driven execution* The data travelling on the arcs in a data flow computation schema are usually referred to in the literature as *tokens*, while the activities associated with each node are called operators The level of granularity which has been assumed by most data flow researchers is a very low level one, operators are often addition, incrementation etc

The next wave of research attempted to bring these models to some degree of implementation, if only an abstract one This usually took the form of a base language to be executed by a hypothetical machine, and perhaps a higher level language which reflected the unusual nature of the sequencing in dataflow as well as the lack of any memory state (see discussion of the functionality of data flow below) Two better known such languages are ID (for Irvine Dataflow) and VAL (for Value oriented Language) [7, 2] Many problems had to be resolved even at this level of implementation First of all, the unbounded FIFO queue had to be avoided The simplest approach, taken by Dennis, is to permit only one token on any arc at a time A machine implementing such a discipline is know as a 'static' machine because only one instantiation of the computation represented by a node in a data flow graph can be active at a time, hence the execution follows

the static structure of the graph in some sense (the term is more meaningful in contrast to the 'dynamic' machine, discussed later). To enforce such a discipline, a given node must know that a successor to which it supplies a token has in fact consumed that token before it attempts to produce another. This is implemented by requiring nodes to receive an ACKnowledgement token from their successors, indicating that the latter have consumed the previously proferred token, before attempting to execute. Thus, for a node to be *fireable* (equivalent to 'enabled' -- used in the literature to denote that all of a node's operands have arrived and it may execute), a new condition is added that all of its ACK s have arrived. In effect the ACKs are additional operands, the criteria for fireability remain the same. Of course this approximately doubles the number of tokens which must be exchanged in the course of a computation -- a severe overhead.

Iteration is problematic in data flow because tokens must be circulated through the same operators many times. This means that no further instantiations of a loop can be permitted while a first one is still in progress because otherwise operators would not be able to distinguish between tokens of different instantiations. This problem of token confusion was found to plague any data flow computation whose graph contained cycles, which of course, are exactly what loops are. One approach to this problem (used in e.g. [34]) is to use only recursion to express iteration. The body of the loop becomes a procedure which is called recursively. A special *apply* operator is defined which takes two inputs the operands required by the procedure being invoked, and the name of the procedure. This name consists of a pointer to the text of a data flow program which describes the procedure. The apply operator allocates an activation record for an instantiation of the procedure. Initially it contains only the input

operands for the procedure However, it also contains fields which are in one-to-one correspondence with the intermediate results which will be generated by the nodes in the procedure data flow graph The procedure executes, using the activation record as its intermediate storage for the results of its internal operators A pointer to the successor of the apply operator is part of the activation record. when the procedure completes. its results are sent as input operands from the activation record to the node which is successor to the apply Thus many instances of the same data flow code can be executing concurrently

An important advantage of data flow which must be appreciated is its functional nature In addition to being a vehicle for parallel computing data flow also offers a solution to the problem posed by Backus that of a complex memory state which makes the correct development of large systems extremely difficult All data flow operators are assumed to be *functional* in nature, that is, their outputs are a function only of their inputs They contain no state information, i e have no history of previous computations which could affect the result of the current one Hence the notion of a variable, in the sense of a locus whose value can be changed and referred to in its different versions, does not exist in data flow Only values themselves are created and destroyed as tokens pass from one operator to the next One important benefit of functional systems is the absence of *side effects* Since the values used by an operator are anihilated by that operator and have no relation to any other operator, it follows that operators cannot inadvertently interfere with each other This makes debugging large systems much simpler By contrast, in conventional systems, changing the value of a variable can adversely effect the correct execution of any number of modules whose scope includes that variable Of course the programming advantages of

data flow may be outweighed by their disadvantages. In particular, expressing programs in the form of directed graphs is extremely unwieldy and the resulting programs quickly become a mass of spaghetti. Whether or not high-level versions of these languages such as VAL and ID, can overcome enough of this problem has yet to be proven in practice.

3. The Dennis Machine

In addition to the problems described above, dataflow presents some thorny problems when one attempts to bring the implementation down to the level of a workable machine architecture. In order to appreciate these difficulties, we will look at the basic 'static' machine designed by Dennis and his associates at MIT [15, 16]. The later development in this thesis makes use of the Dennis architecture as a starting point. The other major type of data flow architecture, known as the 'dynamic' or 'tagged' machine will be briefly described in a subsequent section.

The basic Dennis architecture is shown in figure 1.1. Each *instruction cell* corresponds to an operator in the base machine language. As we have seen earlier the base language is essentially the directed graph representation of the program. The instruction cells are held in an *instruction memory* and are individually addressable. An instruction cell contains storage for the following fields.

1) operation code
2) receiver fields
3) receiver count field
4) ACKnowledgement count field
5) destination address fields

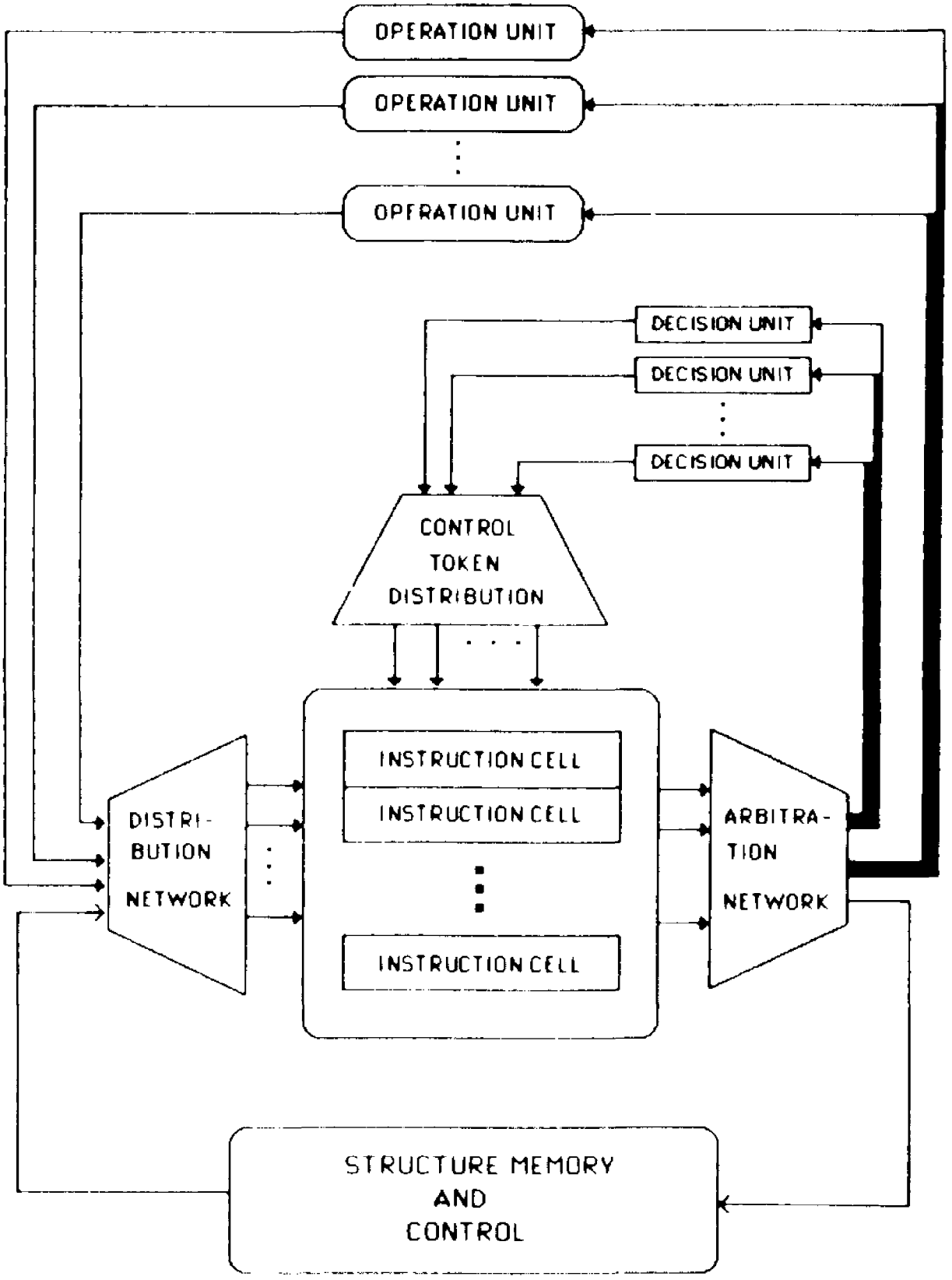


Fig 1.1 The Dennis Machine

Items (1) and (5) remain fixed throughout the execution of the program The receiver fields are the locations to which the input arguments for this operation are stored The receiver count field is initially set to the number of inputs required for this instruction As each input arrives, the receiver count field is decremented As explained earlier, each recipient of a token from this instruction must return an ACKnowledgement indicating that it has consumed the previous token and has room for another in its receiver Accordingly, the ACK count field is initialized to the number of successors indicated in the destination address fields As each ACK arrives, the count is decremented When both counts are zero, the instruction cell becomes *fireable* The instruction cell includes logic to assemble an *operation packet* consisting of the op-code, operands and destination addresses and forwards these to the *arbitration network* The latter sends the packet to the correct execution unit based on the op-code The execution units perform the required operation and form one or more (depending on the number of destination addresses) *result packets* consisting of the result value and a destination address and forwards these to the *distribution network* A destination address consists of an instruction cell address and a receiver number The distribution network routes the result packet to the appropriate instruction cell where it is written to the correct receiver and the count decremented Certain data flow operators will generate only a boolean value, e g a comparator Since only a bit is required to represent these values, separate access lines to the instruction memory are shown for them Such decision operations work in essentially the same way as any other data flow operator, however, having a separate *control network* for them takes advantage of the simpler read/write requirements of single bit values

In general, each activity in the Dennis machine goes through a life cycle consisting of operation packet formation, arbitration, execution, result packet formation and distribution. This sequence of events constitutes the basic beat of the machine. The expectation is that many many activities will be in progress at the same time, each a different station in the cycle. Accordingly, the basic mechanism of this other and similar data flow architectures is often referred to as a *circular pipeline*.

Note that an instruction cell is understood to be a separate physical unit with its own logic to directly accept operands, test a count, assemble a packet and forward it to the Arbitration network. In any real system the number of such units would be impractical, hence a number of them would share the controlling logic. That is, each would essentially be a memory module with each addressable location comprising a 'cell'. One controller would receive result packets, each containing enough information to address the correct cell and port number within the module. In addition to the usual addressing/read/write capabilities, each such memory controller (Mc) would contain processing logic to test control bits and assemble a packet as well as reinitializing a cell. This is essentially the 'cell-block' idea (described in [15]), it amounts to the usual interleaved memory with additional processing logic in each controller.

Instead of sharing a multiplexed bus however, each of these memory modules is separately connected to its own port from the Distribution network (input) and to the Arbitration network (output). This allows greater concurrency in forwarding packets because the forced sequentiality of a shared bus is eliminated. The expectation is that this would allow a sufficient rate of packet flow to fully exploit

algorithm parallelism

Note that such networking is applicable in any system for increasing the bandwidth from processor to memory, and is in fact employed in various forms in most multi-processor schemes currently proposed [21, 23] The main difference between an N x N net (the usual kind in conventional multi-processor organizations) and an N x M net with M < N (such as the arbitrator in the Dennis machine) is that the latter must suffer a higher level of blocking and hence delay due to two or more input ports requesting the same output port This delay is not recovered by the subsequent routing through an M x N (such as the distnet) The fine granularity of dataflow and its intention to concurrently enable large numbers of computational activities necessitates simple, non-homogeneous processors -- if a reasonable level of processor utilization is to be achieved This means that competition for a particular type of execution unit between activities is inevitable and hence that the equivalent of an N x M net with M < N is enforced regardless of the number of processors actually provided (up to some prohibitively expensive overkill) The arbnet configuration reflects these realities More conventional multi-processor systems are targeted to a much higher level of granularity and a proportionately lower degree of parallelism They can therefore afford to provide homogeneous, full fledged processors for each memory module

4. Problems with Data Flow

Despite the fundamental appeal of the dataflow model of computation to the

supercomputer researcher, its real-life deficiencies -- as reflected in currently proposed architectures -- are considerable. A review of the important problems has been presented by Kuck et al [19]. Two of the most salient of these difficulties, and ones which will be of interest to us in this thesis are

1) The fact that *instruction pipelining*, a basic performance enhancement employed in conventional supercomputers, is not possible in current data flow architectures.
2) The long processing pipeline which must be traversed between every pair of interdependent instructions, and
3) The lack of an adequate data structuring mechanism

We shall briefly describe each of these. Firstly, consider any pair of instructions *a* and *b*, represented by adjacent nodes in a data flow graph such that an arc is directed from *a* to *b*. It is clear that no part of the execution of both instructions can be in progress concurrently. This follows directly from the implementation of each instruction as an independent activity whose sole criteria for activation is the arrival of the requisite input tokens. Thus since the tokens which will enable *b* must be generated by the execution of *a*, it is clear that *b* cannot 'know' to begin until *a* has completely finished. Conventional supercomputers employ a technique called *instruction pipelining* to enhance performance. The fetch and decode stages of the execution pipeline can each work on one instruction, dispatch it to the actual instruction execution hardware and then begin the same processing on a next instruction well before the first has completed. This type of

pipelining of instruction processing stages is entirely proscribed by the Dennis architecture

Further, since the entire processing pipeline of the Dennis machine must be traversed by every instruction, it follows that any pair of interdependent instructions (such as *a* and *b* above) will be separated by the full pipeline delay Virtually all of the proposed data flow architectures rely on a relatively long processing pipeline similar to that of the Dennis machine (see for example [33]) Good utilization of such a long pipeline can only be effected by algorithms with ultra high levels of inherent parallelism It has been estimated that some 600 concurrent operations would be required to fill the typical data flow pipeline [19] Note that this would have to be the average level of parallelism, not a burst level, if the pipeline were to be used efficiently

Finally, data structures represent a serious problem in data flow For scalar tokens the actual transmission of operands from one activity to another (by e g writing directly to a field that is part of the instruction) is feasable For large structures such as arrays of data, the copying and storage requirements implied by the actual movement of data along arcs would be prohibitive, both in time and space Various proposals have been put forward to deal with this problem Among the better known are the structure memory of Dennis and Ackerman [14 1] and the I structures of Arvind [5] In the former a special memory is incorporated into the system whose sole function is to store data structures All such structures are in the form of trees, where each non-terminal node represents a substructure and the leaf nodes contain the actual data Arbitrarily complex structures can be represented in this scheme A simple one dimensional

array is implemented much like a list in LISP, essentially a binary tree in which one side of every subtree (the same one) is a leaf node except the last one. Pointers to these structures are passed between operators rather than the structures themselves. To avoid massive copying, a reference count is associated with every node. Whenever two copies of a structure token (actually, its pointer) are made and input to different operators, the reference count of root node of that structure is incremented. As long as the reference count is 1, updates to the structures can be made by operators without any problem; the write operation is made to the structure memory. If the reference count is > 1, however, and an update is requested to a particular node within the referenced structure, the entire structure, up to and including the node which was changed, must be copied. The rest of the structure can continue to be shared by different activities. The reference count of the new structure is set to 1 and that of the old one decremented by 1. In this way, both activities have their own 'view' of the structure up to and including any part of it in which they differ. The scheme is similar to the usual LISP implementations of CONS cells except that no cyclic structures are permitted and there can be many structures being concurrently accessed.

The implementation of this scheme requires a separate *structure memory* with a fairly involved control logic (shown at the bottom of figure 1.1). The tokens which are exchanged between instruction cells which operate on data structures are pointers to structure memory locations. Special data flow operators called (among others) *select* and *append* are used to read and write an element of a data structure, respectively. Referring to figure 1.1, the select operator takes two arguments: the address of the structure in the structure memory and an identifier

of the particular element being read An operation packet of these arguments and the select op-code is sent to the operation logic of the structure controller which reads the selected structure item The structure controller forms a result packet consisting of a copy of the item and a destination address of an instruction which needs to make use of the item -- this will be a successor of the select instruction The packet is sent to the distribution network as usual The append instruction takes the arguments that the select does plus a value which is to replace the selected structure element It is processed in essentially the same way The structure memory and its controlling logic can be thought of as a specialized execution unit which has access to a data base

There are two serious drawbacks to this scheme Firstly it is clear from this description that approximately double the number of memory accesses will be required for structure operations as opposed to scalar ones, apart from the overhead associated with reference counts and other housekeeping One access must be directed to the instruction memory which contains the operators and the structure pointer tokens, and then, after decoding the instruction, another access must be made to the structure memory to get at the 'real' data The second problem is that concurrent operations on the same structure must be severely curtailed If two activities are to update the same structure, they will each have to be provided with pointer tokens to that structure as input By the logic described above, the reference count of the structure will be 2, and when an update is attempted, a second copy of the structure will be created The result will be two logical structures, each with one of the changes, instead of one structure reflecting both changes Thus the updates must be done sequentially

The solution of 'I' structures is to permit only a very restricted type of access to data structures. the sequence of operations to a structure must have a property known as *monotonicity*. We shall not elaborate on this here, except to note that this type of access is quite limiting when compared to the simple freedom of read/write access enjoyed by the sequentially stored array of a conventional machine

These difficulties are perhaps less surprising when one considers that dataflow is a high-performance *model* of computation. i.e it is an advanced *theoretical* tool -- but that as such its premises fly in the face of basic high performance *architectural principles* High performance architectures invariably draw their power from careful synchronization and dedicated structures Dataflow, with its insistence on complete asynchrony at the lowest level, would seem to preclude such techniques from the start

Early dataflow research based its hopes for high performance on massive parallelism One might well characterize much of current data flow hardware work as an attempt to implement a high performance theoretical model of computation on low performance architectures Massive parallelism cannot correct this basic deficiency

An implicit concession to this apparently very basic difficulty is contained in some of the more recent proposals for quasi dataflow architectures e g Kuck's dependence driven and Hwang & Su's event driven models [18, 22] These proposals represent a renouncement of dataflow at the lowest level and an appeal to higher abstraction The key issue then becomes one of process scheduling

more akin to the kind dealt with in conventional multiprocessors. The approach to array accesses presented in [20] is in a similar vein; whole array operations are subsumed within an operator. The Piecewise Dataflow concept [29] is an interesting proposal for dataflow-like scheduling of processes on two levels of granularity. The distinction between the two levels however is based on a construct -- the basic block -- which seems to have been chosen more as a matter of convenience and software compatibility than because of any inherent relevance to parallelism. There is no theoretical groundwork for the choice of granularity level.

5. Objectives of the Thesis

Of the three problems described above, we shall, in this thesis, propose a solution to the first two, and indirectly to the last one. Our proposal shall consist of a different *interpretation* of a graph modelled computation than that assumed by the usual data flow models. We shall retain the meaning ascribed by data flow to the arcs in a graph, i.e. that they represent the movement of data between operators and are the sole criteria for dependency between operations. For this reason we shall refer to our scheme as an interpretation of data flow graphs, that is, one which includes the elements of the data flow interpretation but go beyond it. However, this is to be understood in the above sense, i.e. that we retain the meaning of arcs as used in data flow. In other respects, our interpretation will differ significantly.

The interpretation which we propose involves two parts. Firstly, we begin with a data flow graph and apply a transformation to it which consists of *partitioning* the graph into a set of disjoint *sequential computation paths* (see figure 1.2). Every node in the graph must be assigned to exactly one such path. Each such computation path corresponds to a directed path (in the graph theoretic sense) in the data flow graph of length ≥ 0, i.e. it consists of one or more nodes. This transformation also effects a partitioning of dataflow arcs into two classes. Those connecting successive nodes on a computation path will be referred to as SEQ arcs. The other type of arc connects nodes that have been assigned to different computation paths. They will be referred to as COM arcs. It is clear that this partition can be effected in many different ways. The particular partition or partitions which would be of interest depends on the second part of our scheme, an interpretation of *partitioned* data flow graphs, i.e. what is meant by an execution of such a graph.

The interpretation is as follows. Each sequential computation path is assigned a processor. The significance of the term *sequential computation path* is that any set of successive operators that lie on a directed path in a data flow graph, could theoretically be executed in a sequential and uninterrupted fashion, *provided that any operands required by those operators which are supplied by operators not on the same path, will have arrived by the time they are needed*. Thus we assign a processor to each computation path with the intention of having it fetch and execute the successive instructions on the path in the manner of a von Neumann processor -- hence the term SEQ for the arcs connecting successive operators on a path. The token which traverses the SEQ arcs on a particular path, will be referred to as the HOME datum for that path. This datum can be kept in a

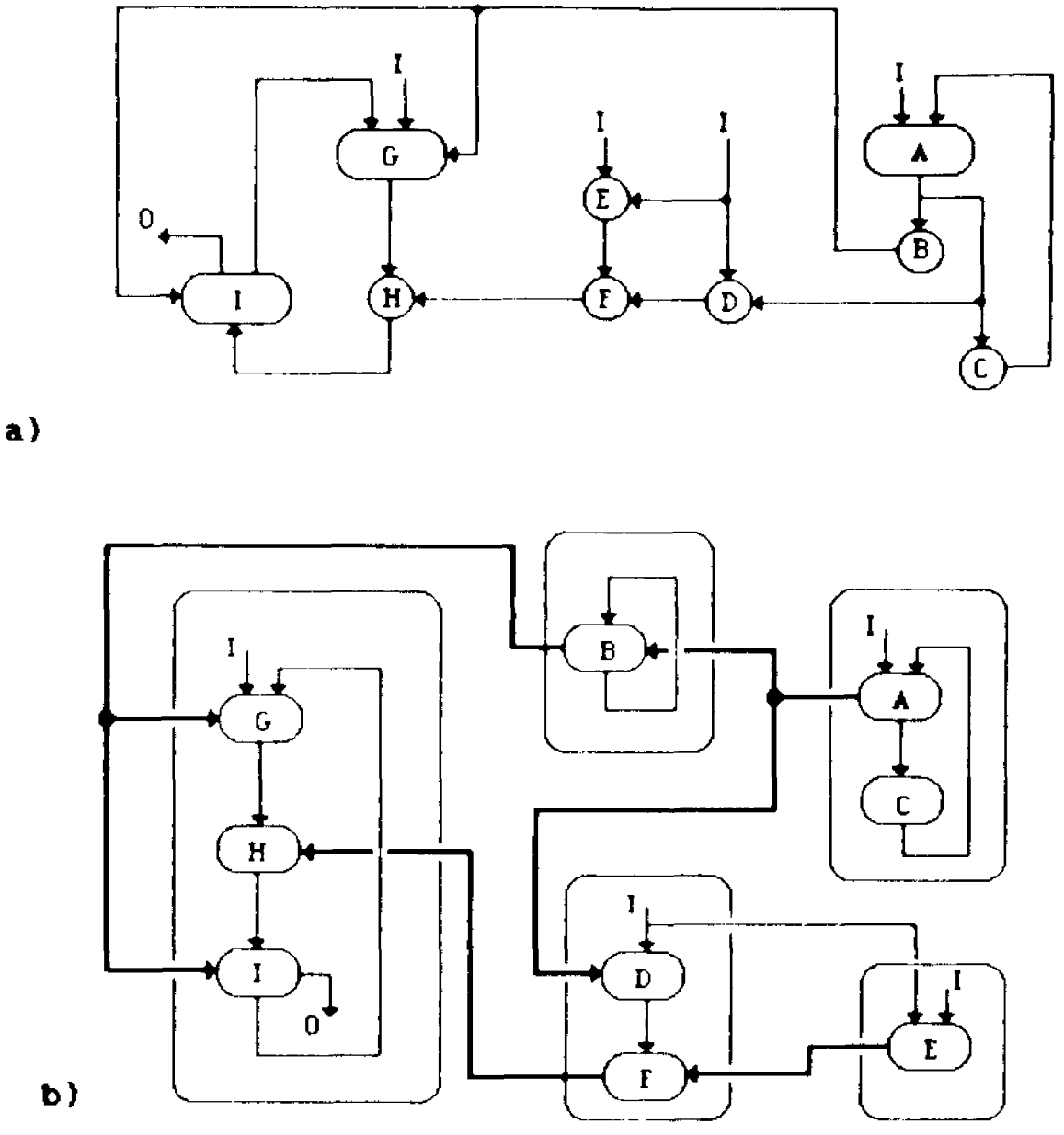


Fig 1.2 Illustration of SDF Partitioning

(a) shows an example dataflow graph represented in the usual way In (b), the nodes and arcs have been partitioned into communicating critical paths, each of which may be executed sequentially 'I' denotes inputs, 'O' shows where the output is taken Each rounded square enclosure represents one element of the partition, each consists of a single sequential path The narrow lines represent sequential flow of control while the thicker ones show the communication between paths

high speed local register and may be thought of as a one word 'context' of the process corresponding to the entire computation path. It need not actually 'move' anywhere; the instructions that comprise the path effect successive transformations on this context register's contents. Ordinary von Neumann control flow branches are permitted, but *only within a given path*.

COM arcs are so called because they represent an interprocessor communication cost. A node which requires more than one input operand must receive such operand(s) from another processor executing another computation path. A COM arc delivers such operand(s).

We will refer to the scheme presented here as Structured Dataflow (SDF) because of the partitioning imposed on the basic dataflow model and because of the quasi-synchrous execution discipline employed. Processors executing a computation path will adhere to the classical fetch/execute cycle for successive instructions. We would expect a machine implementing such a scheme to store the instructions on a computation path in physically sequential memory locations, so as to facilitate efficient implementation of the fetch/exec. SDF represents a reintroduction to some extent of the von Neumann computing element into a data flow type of environment. We should be precise as to what extent we are in fact making use of the classic von Neumann architecture. Referring to the two features with which we characterized a von Neumann machine earlier, we note that SDF introduces the first but not the second of the von Neumann machine elements. That is, we employ sequential instruction fetch and execution, albeit in a novel way. However, there is no memory state, the instructions in SDF remain entirely functional and data are created only to the

extent that they are to be forwarded as input to a particular instruction. In other words, only values are created, not variables. Thus the functionality of data flow is retained. Our objective is to introduce the advantages of the conventional processor, particularly its amenability to instruction pipelining and vector processing without its drawbacks, i.e. its complex memory state.

In the course of the thesis we shall

1) Describe a preliminary architecture for implementing the SDF interpretation.
2) Argue, on the basis of that architecture, that the performance potential of SDF should be higher than that of data flow.
3) Establish criteria for a partition which is optimal in the sense that it achieves the best performance under the interpretation, and give an algorithm for obtaining such a partition.
4) Develop the rudiments of an SDF programming system, i.e. devise constructs to facilitate decision and iteration within the SDF interpretation, and
5) Show that computations represented in the programming system are determinate and live, i.e. that it is a viable system.

The main thrust of our work will be (4) and (5), that is, to show that an SDF system can theoretically be made to work. The notions of determinacy and liveness have been discussed at length in the literature and have been shown to hold for various graph based models of parallel computation. Determinacy is generally

taken as a prerequisite for any viable system and is assumed in most data flow implementations The notion of liveness is usually defined in the context of some set of states of a formal model of machine which are defined to be final or terminal states Liveness is then defined to hold for a machine or schema with a given initial configuration of a terminal state will always be reached Intuitively, the idea of liveness is meant to reflect whether or not a machine or computation model can be expected to make progress A system which is not live is subject to deadlock or is in some sense inconsistently defined Some data flow systems have assumed liveness as a prerequisite for their programs and have restricted the premissible constructs to insure it [31] Others leave it to the programmer to avoid deadlock and combine operators in a consistent manner We shall use the term liveness in a manner appropriate to the somewhat more implementaion oriented context of this thesis Informally, an SDF programming construct will be considered live if it consumes all of its input data and generates the expected output for each A more careful definition will be given in the appropriate section It turns out that these properties do not automatically carry over from data flow to SDF, and this is the focus of (5)

Nevertheless, the thrust of the SDF scheme is to improve performance over the basic dataflow approach Because of this, and because an implementation oriented description of a computing scheme provides a concretization which can help the reader gain an intuitive sense of the ideas presented, we have included a description of an architecture which implements the SDF scheme The case for SDF as a system with better performance than data flow will be argued at the end of this section We emphasize that this design is entirely preliminary and that the issues dealt with in (4) and (5) above must precede a detailed architectural

implementation

The optimal partition developed in (3) relates to a static computation schema, i.e one which contains no decision elements that vary with the dynamic progress of a computation Issues surrounding the dynamic optimization of SDF schemas will be treated briefly in the last section of the thesis where we indicate the directions of future possible research

A Preliminary SDF Architecture

1. The Architecture

We conceive of an architecture which includes the capabilities of the Dennis machine, i e a memory controller (Mc) which can accept a result token, test the destination cell, forward an enabled packet if necessary and re-initialize the cell However, this action is to account for only part of the instruction processing A significant part of it will be done via a different mechanism, i e sequential instruction fetch & execute at the point following the enabled packet's address Thus, *enabled packets also contain a program counter* (PC) *value*

We therefore retain the interleaved memory with enhanced controller for instruction cells Instead of individual cells however, the basic program unit stored in a module is a *computation path* and is logically associated with the token which flows along a directed path in the data flow graph -- the Home datum We would like to use memory interleaving to increase the throughput of tokens from instruction cells to processors Since enabling of nodes along a computation path cannot occur in parallel, it follows that interleaving should be done *across* critical paths

We now have two types of instruction enabling function associated with a memory module (figure 2 1a) The Dennis machine type Mc with its test, packet assembly and related capability and a von Neumann type fetch/decode/execute function The latter has a PC which is incremented for each fetched instruction and can also decode unconditional branches Results of operations executed along

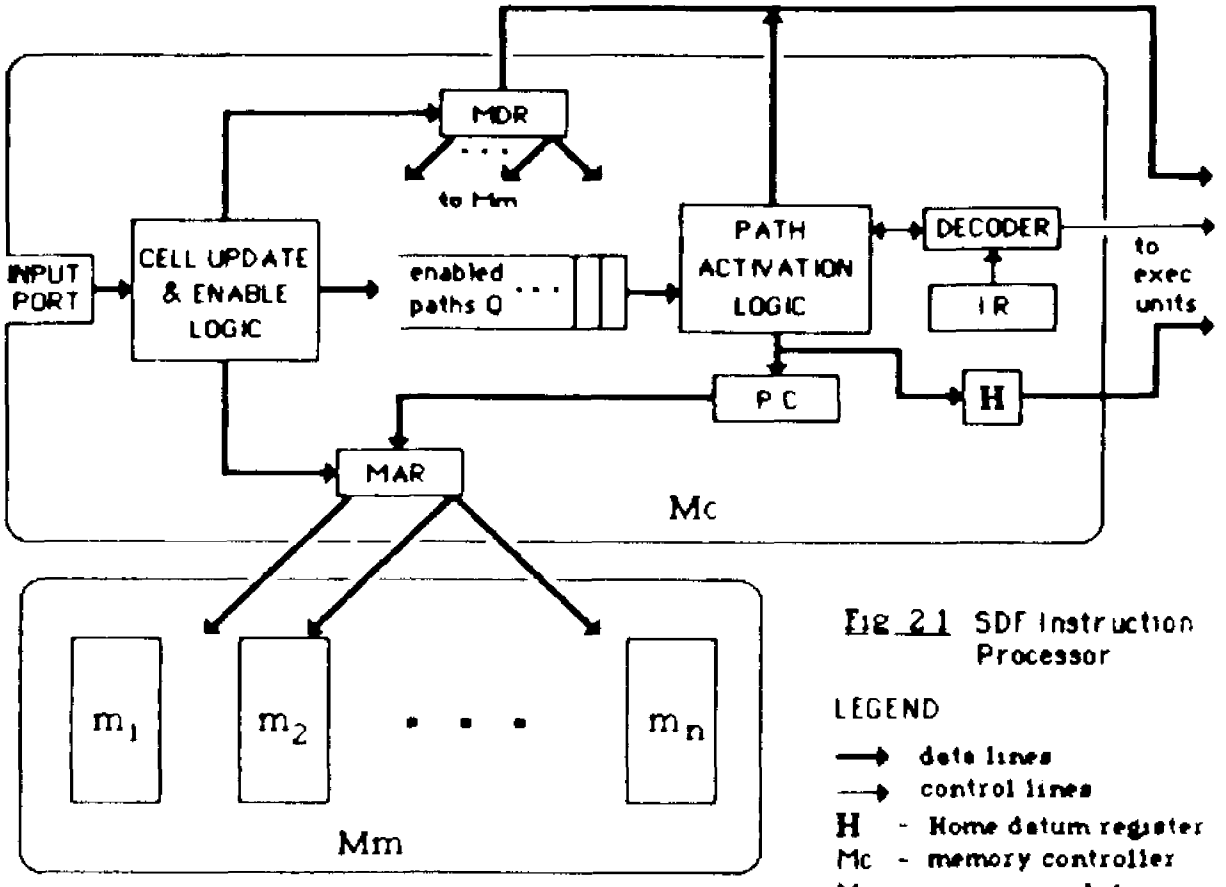


Fig 2.1 SDF Instruction Processor

LEGEND

→ data lines

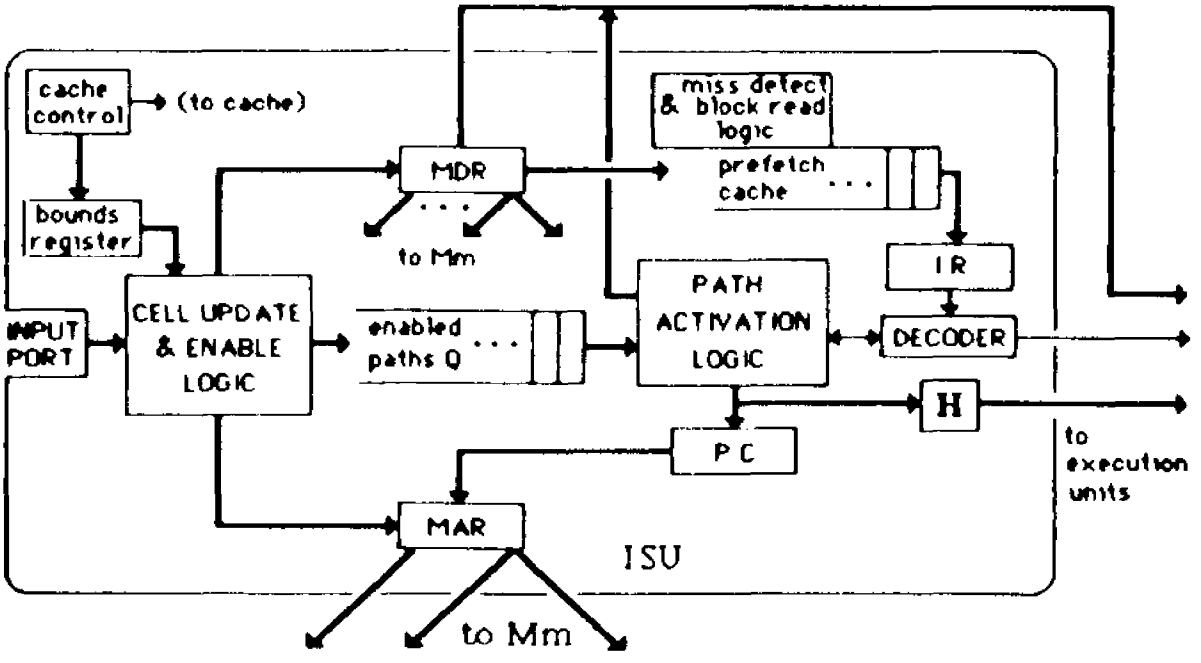
→ control lines

H - Home datum register

Mc - memory controller

Mm - memory module

ISU - instruction stream unit



a computation path are treated in one of two ways: 1) They may update the H datum and/or 2) they may forward results to an instruction in another path. In the latter case the address of the destination is specified as a triple ‹ p, i, t › where p is a path number, i is the instruction number within that path and t is a bit which specifies the type of token being store, i.e. whether it is an operand or a control ACK. Assuming only binary or unary operations are permitted, only one operand is ever required because the H datum is always available. The latter will be maintained in a hardware register during the time that the path is active. Therefore, only one of two operands for a binary operation need be transmitted, while control tokens need only increment an ACK count. When, during the course of executing a computation path, an instruction is encountered with input operand missing, the cell is *marked* with a bit and the path is cancelled. Conversely, when a result is forwarded to an instruction cell, its operand field and ACK count are checked. If all inputs have arrived, *and* the cell is marked, an operation packet corresponding to the instruction is formed and placed in a buffer. The packet includes the PC value of the instruction as well as the H datum, i.e. it contains the information necessary to resume sequential execution of a path at the point where it was last suspended. The packets thus represent re-enabled paths. The buffer in which they are deposited is organized as a queue. When the currently executing path (if any) becomes disabled, the next enabled path in the buffer is activated as follows. The H datum is loaded into its hardware register, the packet operand is forwarded to an appropriate execution unit and the address of the next instruction is latched into the PC. Fetch/Exec continues.

A problem arises as to where the H datum should be stored during the time a path is inactive. One approach is to have a field in each instruction containing a path

ID. This ID would be used to index into a table containing H data. An extra memory reference would be required each time a path is enabled or disabled. To avoid this we could provide room for the H datum itself in each instruction cell. Either way, considerable extra memory is being used.

Note however, that it is because of the existence of control tokens that any problem arises at all. Otherwise we could simply store the H datum in the field where the missing operand is to arrive. There is only one such operand and -- if there were no control tokens -- the path would be enabled as an immediate consequence of the operand's arrival. At that point the H datum goes into a hardware register and no longer needs the storage field. This suggests that the SDF approach presented here might benefit from the incorporation of a dynamic dataflow interpretation -- one which employs label matching instead of ACKs [33, 6]. It is the author's belief that a dynamic version of SDF would be the implementation of choice and would realize much more of its potential. We shall treat this possibility briefly in the last section. For the purposes of the rather more theoretical treatment presented here however, we have assumed the static interpretation.

Since the storing of operands to instruction cells will be occuring concurrently with the fetch/execute function, it is possible that such a store will be directed to the same module from which instructions are being fetched, indeed such a store could be directed to the same computation path as the one currently being executed. We will therefore take each module to be itself interleaved. This interleaving will be low-order, i.e. it will place successive nodes along a critical path in different modules. Thus concurrent cell update and fetch accesses will be

directed to different parts (interleaved modules) of the memory module with a high probability

Processing of an arrived packet is independent of, and occurs in parallel with sequential fetch/execute processing The only intercourse between these functions occurs when a PC fetch and packet store/update access the same module in the low-order interleaving -- in which case one will have to wait (logically, the fetch should take precedence unless the store provides an operand for that very instruction)

Actually, the arrangement shown in figure 2 1a, with its PC, IR and other logic, goes well beyond the memory control function It is responsible for the transmission of operation packets as well as the distribution of results -- functions which demand considerable resources and are a primary design focus in any multi-processor system Accordingly, we have labeled it an Instruction Issue and Update (ISU) unit in figure 2 1b

The fact that each Mm is interleaved suggests the block reads of instruction paths are possible and this is a next enhancement step (fig 2 1b) A prefetch buffer for high speed instruction access is added to the Mc It parallels the interleaving of the Mm itself except that it is only one location deep for each module, in effect, a register file In addition to the usual expidition of instruction issue provided by a cache, the prefetch buffer also serves to further separate memory accesses and reduce collisions since all fetches generated by a given path address the cache while stores do not This is much more significant in dataflow than in conventional architectures because all 'load' type memory accesses are included

in fetches due to the immediate operand character of dataflow instructions. Stores generated by other ISU's however, may in general be targeted to instruction cells on an active path, i.e those contained in some prefetch buffer. A bounds register(s) will maintain information as to what part of memory is in the buffer, the destination address of an incoming packet is compared with this and if the destination is in the buffer the update proceeds there rather than to memory. The registers comprising the prefetch buffer must be provided with split multiplexers to access them so that a store of an operand may proceed into one of the instructions in the buffer while the fetch function is in progress. If the path is cancelled a block write from the point of cancellation and on (in the buffer) must be made to memory to retain the newly arrived operands which have not yet been used. Alternatively, a bit map register could be used to limit write-back to those modules corresponding to prefetch registers which actually received an operand. Note that this represents a significant easing of the cache coherence problem. Because the control function for each memory module is uniquely associated with an instruction stream processor (ISU), writes to a memory module are readily checked against the contents of the corresponding prefetch buffer. No other 'processor' can access instructions from this module -- hence the write-back is effective.

The decoding function determines which type of execution (E) unit a packet should be sent to and may optionally change the PC for unconditional jumps. We are assuming for the moment that non-homogeneous processors are employed. Such a decoding function may be provided with each instruction issue unit and each decoder will directly demultiplex all the different types of E units. This is not an unconscionable number of cross points, the number of types of E units is

limited and we need demultiplex only the *types*. A cluster of identical E units may be managed by logic which forwards an operation packet to any available unit and returns an ACK when latched in. It may have some buffering. Alternatively, a single decoder may strobe a number of IR's in turn, ACKing each with the same signal. Only one demultiplexer would be required. The former provides a potentially higher density packet stream. Two packets requiring different units may be forwarded in parallel in the first scheme but will be separated by at least one clock and likely more in the second. Even packets requiring the same unit will experience unecessary delay in the second scheme if they happen to be separated some distance in the polling order.

Similar considerations apply to the result forwarding function. In figure 2.2, each operand packet forwarded to an execution unit type also has its destination addresses deposited in a holding register asociated with each individual E unit. This maintains a FIFO correspondence between results emerging from the unit and destination information in the register. This register must be buffered to a depth equal to the pipe delay of the E unit. Each emerging result will pop a destination info field from the queue and forward a complete result packet to a completed result register (CRR) -- also buffered. An addressing (A) unit strobes the latter register in each E type cluster. The A unit is directly demultiplexed to the input ports of all ISU's -- and the results are forwarded. As in the decode function, separate, full demultiplexers may be associated with each CRR for better parallelism.

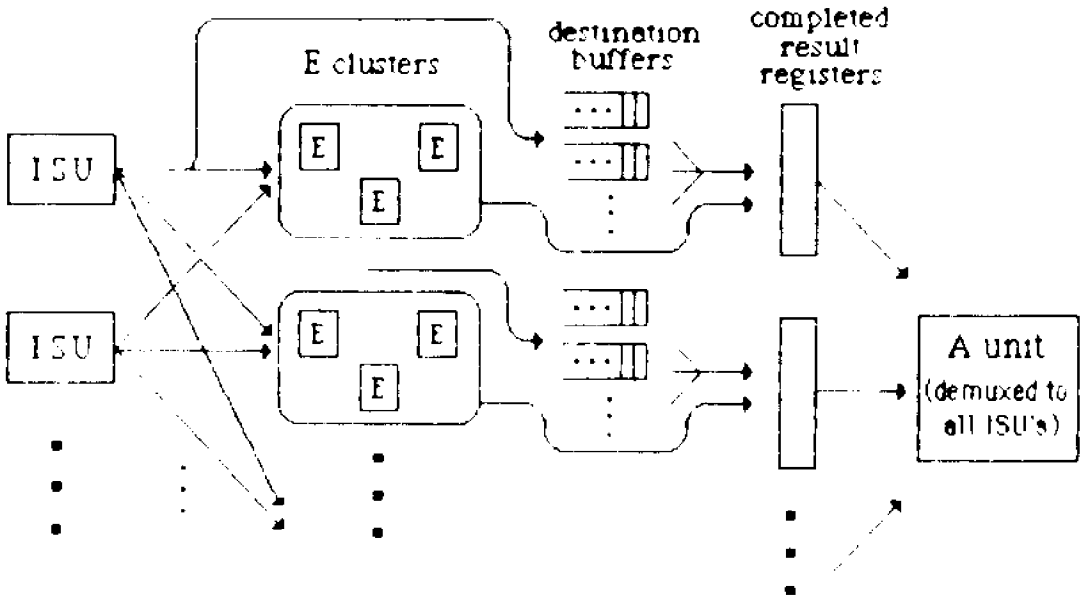


Fig. 2.2 Result Forwarding Between ISU's

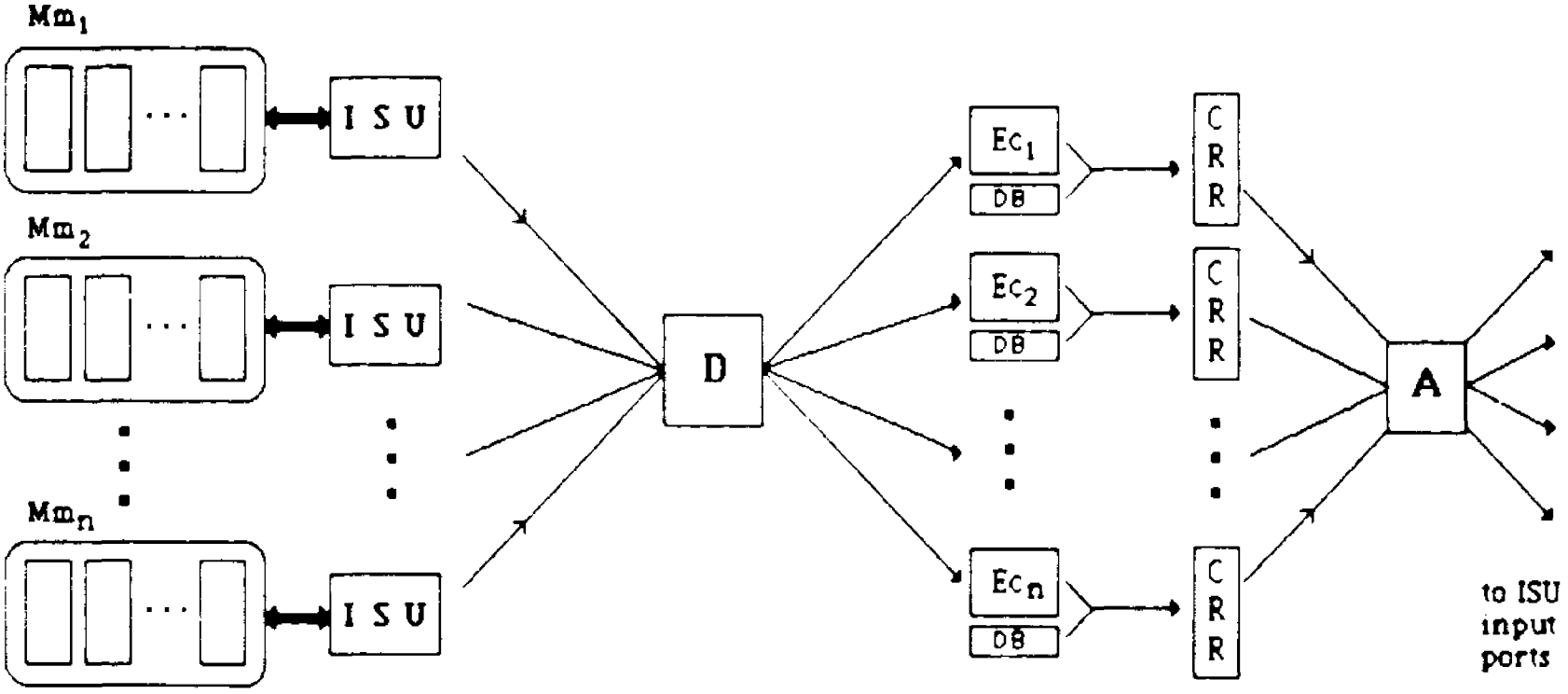
To reduce the buffering requirement in the above arrangement one might use only a single random access buffer for each cluster Each operation packet would be appended a tag designating the buffer location in which its destination information has been temporarily stored Only the tags would have to be FIFO'd -- a more economical proposition

We have incorporated full crossbar connectivity for the connections between the ISU's and the E units as well as from the A units back to the ISU's. Our initial design goal is for a smallish, practical machine and hence the crossbar is relevant In principle however, a log delay type of network could be employed

for either or both of these connections

A somewhat different architectural approach than the one described thus far might be considered in light of the declining cost of basic processor power. Allow a full complement of arithmetic and logical processing capability for each ISU and avoid all network or even crossbar switching delay for the fetch/exec function. Particularly since the ISU is a fairly intricate piece of hardware, the addition of an ALU may not be overly significant. The emphasis would be on high speed networking to distribute results. Only one crossbar or network would be required.

Figure 2.3 shows an overview of the system. Note that if we wish to achieve tight coupling between ISU's and memory modules, it will be necessary to have relatively small memory modules and dedicate a processor to each one. The alternative is to allow arbitrary assignment from a pool of processors to any memory module. whenever a computation path becomes enabled, the path activation logic would remain part of the memory control function while the fetch/execute facility would be a separately assigned processor. However, for a sizable number of processors and memories, we will have to resort to packet switching and the associated delay to implement the many processor to memory connections. Even if a processor is dedicated to every memory module, we are still unable to have concurrent instantiations of the same path -- unless multiple fetch/exec units were dedicated to each memory module. This is an expensive and likely a wasteful proposition. Thus the length of the pipeline traversed by the instruction processing function may still have to be longer than in a conventional serial processor. For as long as the fetch/exec sequential mode is



ISU -- Instruction Issue/Update Unit
Mm -- Memory module
D -- Decoder
A -- Adress unit (result forwarding)
Ec -- Execution cluster type
DB -- Destinations buffer
CRR -- Completed Result Register

Both D and A functions may be replicated for each ISU and CRR respectively The connection between D and E units as well as between A and ISU units may be provided by a full crossbar or a log delay type of network

Fig 2.3 Overview of the SDF Architecture

active however, the overhead of assembling an instruction packet as well checking and updating operand counts is avoided. The key point however, is that successive instructions along a computation path, may follow each other into the pipeline and need not be separated by its full delay

2. Performance Enhancement in the SDF Architecture

The design thus far is similar in important respects to the Dennis machine. Its discernible advantages over the latter include the following. The Dennis machine is limited in parallelism by the number of Instruction 'cells'. Since a module may contain non-interdependent instructions and conversely, two modules may contain dependent instructions, overkill in the number of modules and associated arbnet ports is required to attain good parallelism. Indeed, the original conceptual model of the Dennis machine prescribed a separate unit for every 'cell' -- for exactly this reason. Our scheme however guarantees that a module contains only dependent nodes and hence -- by implication -- two modules are more likely to contain independent nodes. It is however, still possible for nodes in different module to be interdependent since one may transmit operands to the other. Nevertheless, it follows that the number of modules required to achieve a given level of throughput, as well as the traffic intensity through the decoder network will be lower than for the Dennis Arbnet. From a practical point of view this would allow entertaining a machine with a smallish number (say 32) of ISU's with full connectivity to E units -- since we would expect more of them to be providing parallel computation more of the time.

In case no significant sequentiality of execution is realized along any critical path of the dataflow graph of a given algorithm, the architecture described reduces to a machine very much like the Dennis design. Paths would be enabled, execute for one instruction and then, finding a requisite input operand missing for the next instruction -- be immediately disabled. Thus one instruction node at a time is enabled and executed in accordance with the dataflow graph's specification. It is instructive to consider in more detail exactly what additional overhead -- beyond that of the Dennis machine -- would be incurred, if any, for this degenerate case, i.e. where each instruction path executes exactly one instruction and is then disabled. Assuming n input operands for a given instruction node, the processing required to enable that node involves: a) the storing of those n operands into the instruction cell, b) for each such store reading the cell to determine whether the operand count has been exhausted and c) the switching delay implied in assembling an operand packet and forwarding it to the arbnet's decoder. While the last operand supplied to an instruction cell need not actually be written to memory because we can simply forward it as part of the enabled packet, it is still necessary to rewrite the instruction cell to its initial condition -- in preparation for the next time it is fired. Hence we can simply associate n reads and writes with a cell for each of its n input operands -- in addition to the delay in (c). Comparing dataflow and SDF we have

DF: n read/write operations for checking operand counts, updating input operands and re-initializing the cell + packet assembly and forwarding delay (includes any buffering in the arbnet)

SDF n -1 read/write operations for checking operand counts etc as above One operand is always already available as the Home datum One packet assembly delay is incurred as above When an instruction is encountered with missing input operand, it cannot be executed and the cell must be re-written with a 'mark' to indicate that execution is to resume at that point Hence an extra read/write is required

SDF actually pays one read/write less for operand writes than a standard dataflow architecture because of the H datum On the other hand it must always fetch one instruction too many because of the assumption of sequentiality and must also mark the cell which interrupted the fetch/exec sequence Hence the overhead associated with both systems is quite the same for the fully degenerate case

To the extent that sequential execution of instruction paths can be realized, the overhead of assembling an instruction packet (switching delay), as well as the memory accesses for checking and updating the final operand -- are avoided In dataflow, these delays are forced between every successive node on a critical path For every instruction after the first one executed sequentially this gain is pure profit and may be significant

Whether or not the performance enhancements described can actually be realized depends of course on partitioning a dataflow graph into paths in such a way that the probability of sequential execution is high The viability of this is discussed in the next section

The Partitioning of Data Flow Graphs

Two issues present themselves when considering how to organize a dataflow graph for execution in the SDF environment

1) The length of sequential instruction streams The longer the stream the fewer context switches per operation and hence the greater the overall gain in efficiency

2) The waits which a processor may encounter in executing a stream because operands have failed to arrive on time from other processors Such waits represent the need to suspend the current instruction stream temporarily and therefore are equivalent to increasing the number in (1)

Definition: The *level of parallelism* of a graph G, denoted p(G), is equal to the cardinality of the largest set of nodes of the graph such that no precedence relation holds between any pair in the set The nodes in such a set will be referred to as *independent* A *chain* of nodes in a graph G, is a set of nodes on which the precedence relation is a linear order A *partition* of a graph G, will mean a partition of chains Thus a partition will consist of a set of sets of nodes where each such set qualifies as a chain, as well as the precedence relations which the graph specifies between nodes on different chains A partition therefore divides a graphs arcs into two sets those specifying precedence relations between nodes on a chain and those specifying precedence between

nodes on different chains The former will be referred to as SEQ arcs (for sequential), the latter as COM arcs (for communication). A third type of arc, called *forward* arcs in graph theoretical terminology, are those which connect nodes that are more than one node apart on the same chain. As we shall see, this type of arc can be neglected for most analysis The cardinality of a partition of some graph G, i e the number of chains in the partition, will be denoted by $C(P_G)$

Lemma 3.1: $C(P_G) \geq p(G)$

proof: By definition a chain's elements are totally ordered Therefore, for any set of p(G) nodes which participate in the graph's maximum parallelism, each node must be on a different chain Otherwise, at least 2 would be in the same chain, and since no ordering applies between them by the definition of p(G), would violate the defintion of a chain □

We have defined partitions to consist of chains in the usual sense of the term when used in connection with partial orders Other types of partitions are possible To facilitate a discussion of the different types we shall first require the following

Definition: A graph whose nodes all have unit weight is called a *unit weight graph*. We can transform a graph G with arbitrary weight assignment W into a unit weight graph G' as follows For every node in G with weight k in W and

having m input arcs and n output arcs, place in G' a directed path of k unit weight nodes. Connect the m input arcs to the first node on this directed path and the n output arcs to the last one.

Note that the only restriction on W is that weights have rational values. By appropriate multiplication all fractional node weights may be converted to integer values, thus permitting the transformation. For our purposes, the weight associated with a given node will correspond to the time delay of the corresponding instruction. If we convert the nodes in the graph to unit weight, we have that the time delay associated with the execution of a given computation path is proportional to its length. Note that this static assessment of a path's delay is necessarily imperfect since the execution time of certain instructions will vary with the particular operands. Henceforth we shall assume that graphs being used in the context of SDF have been transformed into a unit weight graph, unless otherwise indicated.

Definition: The *level* of a node n in a graph G, denoted $L(n)$, is the number of nodes on the longest directed path to n, not including n itself.

We can now consider the following different approaches to partitioning a Dataflow graph G.

1) A chain is defined as set of nodes $v_1 \ldots v_n$ such that $L(v_i) + 1 = L(v_{i+1})$. If we assume that all nodes at level $i-1$ have executed at time t_i -- this is possible because the graph's nodes are all of unit weight -- then this

approach is similar to the classical scheduling model in which a process is assigned an available processor as soon as soon as all of its precedence requirements have been satisfied In our case a node at level j can have at most j ancestors along any directed path and hence at time t_j all predecessors of nodes at level j have executed

If the partitioning of computations among processors is modeled in this way it is easily shown that one can always find a partition P such that $C(P_G) = p(G)$ and further that the completion time can never be improved upon by using more that p(G) processors [11] . However, such a model ignores the overhead of context switching and communication between processors This simplification is justified if the tasks being scheduled are self contained 'macro' tasks, i e they are large compared to these costs and communication between them is limited Our 'tasks' however are actually primitive operations and hence the cost of allocating and deallocating a processor as well as the level of communication between computations is very significant This model would therefore be a poor approximation of the behavior of an SDF system

2) A chain can be defined as we did above, as a linearly ordered subset of the nodes in G with respect to the precedence relation Note that in this approach, successive elements on a chain do not necessarily correspond to an arc in the Dataflow graph There need only exist a *directed path* in G between such nodes

3) We can use (2) but restrict the definition of a chain by requiring that successive nodes correspond to an arc in G.

4) We can use (3) but restrict chains even further by requiring that successive nodes correspond to successive levels as in (1)

In SDF we shall be interested in (4) which is a combination of (1) and (3). To see why this is so, note first that if successive elements v_i and v_{i+1} on a chain do not correspond to an arc in G, then there is no transfer of data between them in the Dataflow sense and the Read set of v_{i+1} is disjoint with the Write set of v_i. Therefore, a processor assigned to execute the chain will be forced to switch contexts between v_i and v_{i+1}. In terms of the greater efficiency we hope to achieve in SDF by an uninterrupted sequential execution of a serial instruction stream this situation is no better than two chains, one ending at v_i and the other beginning with v_{i+1}. Hence the importance of the requirement in (3).

At the same time, if instruction streams in SDF are to be executed in an uninterrupted fashion we must avoid situations in which an operation cannot be executed because an operand that it requires from a node on a different chain has failed to arrive on time. This is the motivation for the requirement in (1). As a first approximation we will ignore the time delay cost of COM arcs themselves. Therefore, to determine in general if a given operand will arrive at its destination node in time for the latter's execution, we must consider the weight, in terms of

time delay, of the directed path which ends with the COM arc that delivers the operand to that destination. This in turn must be compared with the weight of the chain along which execution is progressing towards the destination node. If the latter is greater than or equal to the former, there will be no wait, otherwise we have a problem. In what follows the term *wait* will be used in this sense.

Definition: A *perfect weight assignment* for some partition P of a graph G is a weight assignment to each node in G such that if G is interpreted as a dataflow graph, no chain in P incurrs any waits for operands. G is not necessarily a unit weight graph.

A *perfect partition* for some graph G and a given weight assignment W for all nodes in G is a partition P such that if G is interpreted as a dataflow graph whose nodes have execution times W, no chain in P incurrs any waits for operands.

We now ask: Under what conditions does there exist a perfect weight assignment for a given partion? True, even if one does exist, we must still show how to find it. More importantly, in practice the weights are given and are relatively inflexible. However, the significance of the above question is that it will allow us to characterize partitions which *cannot* avoid chain waits and hence should be avoided when searching for a perfect partition. The following result achieves this.

Lemma 3.2: A partition P of a graph G has a perfect weight assignment iff, after conversion to a unit weight graph, there does not exist a pair of

successive nodes v_i and v_{i+1} on any chain such that $(L(v_i)+1) < (L(v_{i+1}))$

proof: We prove the 'if' part first -- by contradiction Assume the converse, i e that the condition of the Lemma holds but that in fact there is no perfect weight assignment for P Then for any weight assignment W, there exists at least one forced wait in P The condition stated in the Lemma is equivalent to saying that there does not exist a pair of nodes a and b on any chain such that there exists a directed path in G from a to b which does not include the SEQ arc between them If the wait is experienced by the first node of some chain, we can simply delay the start of that chain by the amount of 'time' of the wait Assume therefore, that there exist a pair of successive nodes a and b on some chain such that node b experiences a forced wait This implies that there exists a directed path p from some first node of some chain to b such that the weight of p is greater than the weight of the path to b along the chain containing a In this case we can modify W by increasing the weight of a arbitrarily and thereby reverse the inequality between the paths This would eliminate the wait and yield a perfect weight assignment Therefore, p must nevertheless include a so that increasing the weight of a increases the weight of both paths equally, thus preserving the forced wait But this implies a subpath from a to b in p Further, p cannot include the SEQ arc joining a and b else b would have only one incident input arc and hence could not experience any waits But this contradicts the condition of the lemma

To prove the 'only if' part we need only note that if a path p exists between two successive nodes on a chain a and b then, since dataflow graphs are not

multigraphs, there must exist at least one node, say c, on p other than a and b. This node must have some finite weight, however small, and therefore the chain containing a and b will be forced to wait after executing a for this amount of time before executing b since the latter is dependent on c. □

The Lemma shows, that for a graph with arbitrary weight assignment, chains whose successive nodes are not at successive levels *must* result in a forced wait. This implies either idling a processor or a context switch, both of which we wish to avoid in SDF. For our particular case of a unit weight graph, the following lemma shows that requiring successive chain elements to have successive level values is also a *sufficient* condition to avoid this type of chain 'break'.

Lemma 3.3: Let P be a partition such that for every pair of successive elements on a chain v_i and v_{i+1}, $L(v_i) + 1 = L(v_{i+1})$. Then P is a perfect partition.

proof: By contradiction. Assume the condition of the lemma is satisfied but that P is not perfect, i.e. there exists a chain C in P that experiences a forced wait. If the wait is experienced by the first node of some chain, then, as before, we can simply delay the start of that chain by the amount of the wait. Assume therefore, that there exist a pair of successive nodes a and b on some chain C, such that node b experiences a forced wait. This implies that there exists a directed path p from some first node of some chain to b, which does not include the SEQ arc from a to b, such that the weight of p is greater than the weight of C up to b. Now since G is a unit weight graph we must have that C up to b consists of n nodes while p consists of $n + k$ nodes for some k, $k \geq 1$. But this implies that $L(b) = n + k$.

-1 while $L(w) = n-2$. This contradicts the assumption of the lemma. □

In light of the preceding, we are interested in a partition consisting of chains as defined in (4) above, i.e. chains for which both of the following are true for successive nodes v_i and v_{i+1}

1) there is an arc in G connecting the nodes v_i and v_{i+1}

and 2) $L(v_i) + 1 = L(v_{i+1})$

The next step is to obtain a partition which maximizes the average chain length. Since the average chain length is given by $|V|/C(P_G)$, this problem is equivalent to finding a partition with a minimum number of chains. We can apply standard graph theoretic results related to transportation networks to solve this problem [17] -- as follows. First, 'prune' the graph by eliminating all arcs which do not connect nodes at successive levels, call this graph G'. It is easily seen that no vertices are eliminated in this process. We now wish to find a minimum number of vertex disjoint directed paths which cover the vertices in the graph. Since a directed path must follow along arcs in G, condition (1) above is satisfied. Since G' contains only arcs that connect nodes at successive levels, any directed path will also satisfy (2). Hence such a minimum cover will yield the partition we seek. To find the minimum cover we first form a network G'' from G' as follows

1) The vertices in G'' are s and t, the source and sink of the network, as well as two nodes x_i and y_i for each node v_i in G'

2) The edges in G'' are $s \rightarrow x_i$ and $y_i \rightarrow t$ for $1 \leq i \leq |V|$ as well as an edge $x_i \rightarrow y_j$ for every edge $v_i \rightarrow v_j$ in G'

3) The capacity of all edges in G'' is 1

Lemma 3.4: The minimum number of vertex disjoint directed paths which cover the vertices in G' is equal to $|V| - F$, where F is the maximum flow in the network G''

proof: Let M = a minimum size set of vertex disjoint paths which covers V in G' For each path p consisting of vertices $v_1, v_2, \ldots, v_n$ in the set M, let S_p denote the n-1 member set of directed paths in G'' $(sx_1y_2t, sx_2y_3t, \ldots, sx_{n-1}y_nt)$. Now consider $S = \bigcup_{p \in M} S_p$ Since G' is acyclic, each $p \in M$ has no repeated vertices Further, since M consists of vertex disjoint directed paths which cover the vertices in G', it follows that each of those vertices occurs once in exactly one of the paths $p \in M$ Now $|S_p| =$ #(vertices on p) -1 Thus $|S| = \sum_{p \in M} |S_p| = \sum_{p \in M}$ #(vertices on p) $-|M| = |V| - |M|$, from which $|M| = |V| - |S|$

We now show that $|S| = F$, the maximum flow of the network G''. First, S consists of vertex disjoint directed paths from source to sink in G''; hence we can use each such path to flow one unit from source to sink and it follows that $|S| \leq F$ Next let f

be a flow function on G'' which assigns to every arc on each directed path in S a flow of one unit and a flow of zero to all others For any pair of nodes x_i and y_j such that $f(s \to x_i) = f(x_i \to y_j) = f(y_j \to t) = 1$, it follows that no other edge $x_i \to y_k$ or $x_h \to y_j$ can be used to carry flow since only one edge enters x_i and only one leaves y_j, both of capacity 1 Hence there exists no augmenting path for the flow function f and if follows that f is maximal i e $|S| \geq F$ □

Lemma 3 4 tells us that the size of a minimal partition is equal to $|V| - F$, where F is the maximum flow in the network G'' The proof of the lemma can be extended to a construction which yields such a minimal partition However, in addition to finding a minimal set of chains would also like to show that the completion time of a data flow graph can never be improved by using more than p(G) processors That is, we would like to show that this result, which is trivially true for traditional models of process scheduling (definition (1) of chains above), also holds true even when we restrict chains so that successive nodes are one level apart and also correspond to data transmission in the data flow graph (definition (4) of chains) Stated differently, we would like to show that if P is a partition of G into chains as we have defined them, there exists a subset cover Q, of the elements of P, such that all chains in any single subset in Q are levelwise disjoint, i e no pair of nodes from any two chains in the subset have the same level value, and further that $|Q| \leq p(G)$ If this is true, then it follows that a single processor yields the best completion time for any element of Q, since all chains in any one subset must be executed at disjoint times (remember G is a unit weight graph) Hence p(G) processors are sufficient We shall take a different approach to finding an

optimal (minimal) partition, one which will also allow us to prove the existence of the subset cover Q

We shall need the following concepts from graph theory

Definition: A *bipartite graph* G is a graph whose vertices can be partitioned into two sets X and Y such that every edge in G connects an element of X to an element of Y A *maximum matching* in a bipartite graph G is a subset E' of edges in G such that no two edges in E' have any vertex in common and |E'| is maximal

The problem of finding a maximum matching can be thought of as determining the largest number of nodes in Y, each of which is adjacent to a distinct node in X, i e matching the y's to the x s Equivalently, the problem is to match as many of the x's to the y's as possible A number of algorithms are known for finding a maximum matching The reader is referred to [32] for a survey of the literature

The following algorithm takes an arbitrary Dataflow graph which has been transformed into one of unit weight and finds a minimum partition of chains as defined by (4) earlier The algorithm will make use of maximum bipartite matching as well as *splice* arcs which we define below

Definition: A *splice* may connect the last node of a chain A, to the first node of a chain B, iff nodes on the two chains exclude each other in time, i e they are levelwise disjoint

Algorithm 3.1: The algorithm assumes that the level value has been determined

for each node We will denote the node at level j on the ith chain by C_{ij} Each iteration of the algorithm deals with nodes at the next level Thus we begin with nodes at level 0, those without predecessors

Step 1 $L = 0$ Create as many chains as there are nodes at level L For each chain set C_{i0} to one of these nodes

Step 2 If none of the C_{ij} have successors, terminate the algorithm

Step 3 Consider the nodes at level L and L+1 together with the arcs connecting them (there are no arcs connecting nodes at the same level, this is shown below) Let these nodes constitute a bipartite graph with X comprising the nodes at level L and Y the nodes at level L+1 Find a maximum matching for this bipartite partition For each node C_{iL} which has been matched to a node s at level L+1, set C_{iL+1} to s In other words continue the chain C_{ij} with the node s, the arc used in the matching becomes a SEQ arc

Step 4 For any maximum matching there will be a set H of nodes in X that remain unmatched and a set K of nodes in Y that are not used in the matching, $|H|,|K| \geq 0$ For each node $C_{iL} \in H$, terminate the chain C_i at level L and set the arcs connecting its last node to successors at level L+1 to COM arcs Create $|K|$ new chains at level L+1, setting the first node C_{iL+1} of each chain to a different element of K

Step 5 Let J be the set of chains which have been terminated at levels < L. If $|J| \leq |K|$, arbitrarily connect the last node of each chain in J to the first node of one of the new chains created at level L+1 -- with a splice. If $|J| > |K|$, arbitrarily splice the first node of each of the chains created at level L+1 to the last node of a chain in J.

Step 6 L = L+1. Go to step 2.

Note that a chain produced by the algorithm which contains k splices must be considered as $k+1$ chains under the definition we are using since the splice connects nodes that are at least two levels apart. The significance of the splices will become clear shortly. For the moment we ignore them.

We first show that Algorithm 3.1 always yields a minimum partition. We will need the following additional concepts from graph theory.

Definition: Let B be a bipartite graph, X and Y a bipartite partition of B and let W be a subset of X. The *adjacency set* of W, denoted $\mathcal{L}(W)$, is defined as the largest subset of Y such that every node in the subset is adjacent to a node in W. The *deficiency* of W, denoted $\partial(W)$, is defined as $\partial(W) = |W| - |\mathcal{L}(W)|$. The X deficiency of the entire graph B is defined as $\partial_x(B) = MAX_{S \subseteq X} \partial(S)$. Alternatively, if we take Z, a subset of Y, and its adjacency set (with respect to X) as $\mathcal{L}(Z)$, then the deficiency of Z is $\partial(Z) = |Z| - |\mathcal{L}(Z)|$ and we have the Y deficiency of the graph B $\partial_y(B) = MAX_{S \subseteq Y} \partial(S)$. A basic result in graph theory is,

Hall's Theorem: Let B be a bipartite graph, X and Y a bipartite partition of B A maximum matching of B will match exactly $|X| - \partial_x(B)$ nodes in X to a node in Y Similarly, a maximum matching will match $|Y| - \partial_y(B)$ nodes in Y to a node in X

Theorem 3.1: Algorithm 3 1 yields an optimal (minimal) partition of chains

proof: We will require some notation Let G_k denote the graph which is input to the algorithm but restricted to nodes r such that $L(r) \leq k$, V_k will be the set of vertices in G_k Denote by P_{Gk} the partition of G_k corresponding to the partition P generated by the algorithm restricted to the subchains of chains in P comprised of nodes in G_k Let B_k denote the bipartite graph for which the algorithm finds a maximum matching at level K, i e the graph consisting of nodes at levels K and K+1 Denote by X_k and Y_k the bipartite partion of B_k Finally, let G'_k denote the network corresponding to G_k constructed in the same way as was done for Lemma 3 4 The maximum flow of G'_k will be denoted by F_k

The proof shows that for any k

$$C(P_{Gk}) = |V_k| - F_k \tag{3 1}$$

We can then take k = the number of levels in G and by Lemma 3.4 the cardinality of the partition will be minimal. The proof proceeds by induction on k. For the base case we take $k = 0$. By step 1 of the algorithm every node at level 0 is placed on a separate chain and therefore $C(P_{G0}) = |V_0|$. But G_0 has no arcs at all and hence G'_0 has no flow. Thus $F_0 = 0$ and (3.1) holds for the base case.

We now make the induction assumption, i.e. that (3.1) holds for some k and show that this implies that it holds for $k+1$ as well. After finding a maximum matching at each pair of levels k and $k+1$, the algorithm creates as many new chains at level $k+1$ as there are nodes in Y_k which remain unmatched. It follows from Hall's theorem therefore that the number of new chains created at level $k+1$ is $\delta_y(B_k)$. Thus we have that

$$C(P_{G(k+1)}) = C(P_{Gk}) + \delta(B_k) \qquad (3.2)$$

also

$$|V_{k+1}| = |V_k| + |Y_k| \qquad (3.3)$$

Since we want (3.1) to hold for $k+1$, we substitute (3.2) and (3.3) into (3.1) yielding

$$C(P_{Gk}) + \delta_y(B_k) = [\,|V_k| + |Y_k|\,] - F_{k+1}. \qquad (3.4)$$

Using the induction assumption, we can combine (3.1) with (3.4) to yield

$$(3.5) \qquad \partial_y(B_k) - |Y_k| = F_k - F_{k-1}$$

which is the same as

$$(3.6) \qquad F_{k-1} - F_k = [|Y_k| - \partial_y(B_k)]$$

In other words, if (3.1) is to hold for any k, the additional flow in G' due to the inclusion of the nodes at $k-1$ must be equal to $|Y_k| - \partial_y(B_k)$. We now prove this.

We will denote the left side of (3.6) by ΔF. Define $Z \subseteq Y_k$ such that $\partial(Z) = \partial_y(B_k)$. Since there are exactly $\ell(Z)$ nodes x_i connected to nodes y_j in G_{k-1}, where $y_j \in Z$, and since each x_i has only one arc of capacity 1 incident to it from the source, it follows that Z adds at most $\ell(Z)$ units of flow to F_k. Further, the nodes in $Y_k - Z$ can add at most $|Y_k - Z|$ units of flow to F_k since each of them adds exactly one y_j to G'_{k-1} and each such y_j has only one arc of capacity 1 directed out of it to the sink. Thus we have that $\Delta F \leq [|Y_k| - \partial_y(B_k)]$. Now every node x_i in G'_k corresponding to an element of X_k has no arc directed out of it because such an arc would have to be

directed to a node at level $k+1$ and G'_k has nodes x_j corresponding only to nodes in G up to level k. Thus all such nodes in G'_k carry no flow. Further, by Hall's theorem, G_{k+1} adds exactly $|Y_k| - \partial_y(B_k)$ vertex disjoint arcs to G_k and in G'_{k+1} each of these arcs is directed out of a node $x_i \in X_k$. Hence each of these can be used to flow one additional unit from source to sink in G'_{k+1} and we have that $\Delta F \geq |Y_k| - \partial_y(B_k)|$. Thus $\Delta F = |Y_k| - \partial_y(B_k)|$. □

We now wish to show that Algorithm 3.1, in addition to producing an optimal partition, yields a set of chains which can be optimally scheduled on p(G) processors. This is the significance of the 'splices' applied by the algorithm in Step 5. Since a splice connects only chain segments that are at least two levels apart, it follows that the latter of the two segments will have its precedence requirements satisfied only after the first has completed, i.e. their execution is disjoint in time (the graph is a unit weight graph). Thus any number of chains spliced together can be executed by a single processor since no parallelism is restricted by the splicing. For the discussion which follows we will refer to chain segments that have been spliced together by the algorithm as *spliced chains*. Now, if the number of spliced chains generated by the algorithm is equal to p(G), then we have obtained the subset cover Q, discussed earlier, each spliced chain is a subset in the cover and its cardinality is p(G). We can assign a processor to each spliced chain.

We now prove that the number of spliced chains generated by the algorithm is in

fact p(G) We will require some preliminary lemmas We shall use the notation G_k, P_{Gk}, B_k, X_k, Y_k etc as we did for Theorem 3.1.

Lemma 3.5: The set of nodes at level L-i for any i are independent

proof: Assume to the contrary Then there exist nodes a and b such that L(a) = L(b) = i and a takes precedence over b Then the graph must contain a directed path from a to b By the definition of L, the longest path to a contains i nodes Hence there must exist a path to b consisting of i+k nodes for some k, k ≥1 But then L(b) = i+k □

Lemma 3.6: Let G' be G restricted to nodes at level i and i+1, for arbitrary i There exists an independent set S of nodes in G', such that each subchain corresponding to a chain generated by Algorithm 3 1 but restricted to the levels i and i+1, is represented by a node in the set S That is, no two nodes are on the same chain and for every subchain C there exists a node in S on C

proof: Place all nodes at level i in S By Lemma 3 5, S is independent Find a subset Z of the nodes at level i+1, such that $\partial(Z) = \partial_Y(B_k)$, where $B_k = G'$ and X and Y are the nodes at i and i+1 respectively Replace £(Z) in S by Z By the definition of £, no dependencies have been introduced into S, i e it remains an independent set Further, by Hall's theorem, exactly $\partial_Y(B_k)$ nodes at level i+1

remain unmatched by the algorithm and hence exactly as many new chains are created at level $i+1$. But since $\partial_y(B_k) = \partial(Z) = |Z| - \ell(Z)$, it follows that we have added exactly this many new nodes to S. □

Now consider the following procedure

Procedure MARK (G, P) // takes as input a DAG G and a partition P defined on G//

```
VAR IND Boolean   //indicates whether the markes set of nodes is
                    independent//

Begin
Mark all nodes which begin a chain in P.

WHILE [there exist pairs of marked nodes u and v such
       that there exists a directed path from u to v]   DO

       Begin
       IF [u is the last node on a chain in P]
              THEN Begin IND = false, EXIT End
              ELSE Move the mark from u to its chain successor
       End
End
```

Lemma 3.7: If MARK terminates with IND = True, then $C(P_G) = p(G)$

Otherwise $C(P_G) \geq p(G)$

proof: That $C(P_G) \geq p(G)$ is merely a restatement of Lemma 3.1. The WHILE

condition is actually a statement of the definition of non-independence. Hence if the procedure terminates without IND going false, the WHILE condition must have become false and hence the set of marked nodes must be independent. Since the procedure never marks more nodes then there are chains in P in must be true that $C(P_G) \leq p(G)$. Hence by Lemma 3.1 we have $C(P_G) = p(G)$. □

We are now ready to prove the result we seek.

Theorem 3.2: The number of *spliced* chains generated by Algorithm 3.1 is equal to p(G).

proof: By induction on the level k. The base case is a graph consisting only of nodes at level L-0. By Lemma 3.5 all nodes are independent and hence p(G) = V, and the theorem is trivially true. Now assume that procedure MARK has been applied to k levels of the partition P of G and has terminated with IND = True. By Lemma 3.7 this implies that $C(P_{Gk}) = p(G_k)$ [refer to the earlier notation]. To prove the theorem, we will show that if G_k is augmented to G_{k+1} by adding the nodes at level $k+1$, the procedure MARK can be updated to include those nodes and their arcs without IND going false.

If no new chains are created at level $k+1$ then the induction hypothesis follows from the induction assumption because the marked set can remain unchanged -- no new nodes at level $k+1$ need be included. Assume therefore, that one or more chains are created at level $k+1$.

Let Z be a maximally deficient subset of the nodes at level $t+1$, i. e. $\partial(Z) = \partial_y(B_t)$, B_t a partition X and Y of the nodes at t and $t+1$ respectively. By the method of proof of Lemma 3.6, Z can be added to the set of nodes marked by the procedure MARK without incurring any dependencies on nodes at level t if we first remove any elements of the set $\Gamma(Z)$ of nodes at level t from the marked set. However, not all nodes in $\Gamma(Z)$ are necessarily in the marked set since procedure MARK may have chosen a node at level $<t$ to represent a chain which has a node in $\Gamma(Z)$. However, once we have introduced Z into the marked set, we need no other nodes to represent chains which have nodes in $\Gamma(Z)$ since all such chains are represented by some node in Z. Therefore we will remove all nodes from the marked set which represent chains which have nodes in $\Gamma(Z)$. By Lemma 3.6, the inclusion of Z will introduce no dependencies on nodes at level t which *might* be in the marked set.

Of course there may still be marked nodes at levels $<t$ which have precedence over nodes in Z and therefore the marked set may not be independent. Denote the set of such nodes by Y and the subset of Y consisting of nodes which represent chains that terminate at levels $<t$ by W. Now if

1) $\partial(Z) \leq |W|$ then Algorithm 3.1 will have spliced all new chains created at level $t+1$ to an element of W. Hence no new *spliced* chains are created and we dispense entirely with Z. All chains in P_{Gt} are represented and the marked set remains independent.

2) $\partial(Z) > |W|$, all chains represented in the marked set by an element of W are now represented by an element of Z since the algorithm will have spliced all of the former to one of the latter. Hence we can delete the nodes in W from the marked set

We now claim that the procedure MARK can be updated in case (2) to include level $k+1$ without IND going false. Assume the converse. For IND to go false there must exist a chain C_a with last node u which is marked such that there exists a directed path from u to a marked node v on a chain C_b. Since P_{Gk} has maximum level of $k+1$, the node u on C_a must be at level $\leq k$. However, splicing has eliminated all chain terminations at levels $< k$, therefore the node u on C_a must be at level k. Now the only changes we have made to the marked set produced by MARK are the elimination of nodes in W and the addition of nodes in Z. Hence any dependencies must be introduced by nodes in Z. Since all nodes in Z are at level $k+1$, the node u on C_a cannot be in Z. We must assume therefore, that the node v on C_b is in Z. But C_a cannot have any nodes in $\mathcal{L}(Z)$ since all chains with nodes in $\mathcal{L}(Z)$ are already represented by Z and hence extend to level $k+1$. But since the node u on C_a must be at level k, this implies that C_b has no nodes in Z -- a contradiction. □

An SDF Programming System

1. Introduction

In the previous section it was shown how Dataflow graphs could be partitioned into chains and what the theoretical limits of this partitioning are However, we are not yet in a position to assess the siginificance of organizing computation in this way because we have not yet fully specified an *interpretation* of partitioned graphs which corresponds to an SDF system -- although the rudiments of an interpretation were introduced in an informal way in the first section Indeed the development of the previous section may be thought of as addressing itself only to the theoretical problem of partitioning in any graph, without reference to the meaning of the graph Because of its quasi-synchronous character, SDF will have a different interpretation than the usual ones for graph models of parallel computation and its behavior as a computing system will be governed by correspondingly different parameters

In this section we develop a workable interpretation of SDF that is, how the operations implied in an SDF program are actually carried out Another way of saying this is that we will develop an operational semantics for SDF programs represented as partitioned graphs

As an example of the issues that must be dealt with, consider the case of a simple expression tree We have seen how the tree can be partitioned into chains Intuitively, we might let the matter rest there, the 'interpretation' would amount

to simply assigning a processor to each chain and letting them execute for the length of the chain If an instruction on a chain is fetched and found to be without a requisite operand, we can suspend that chain by storing the Home datum in the instruction -- as described in the first section When the required operand is stored in the instruction's operand field, the chain will be reactivated at that point All processors simply execute to the end of their assigned chains An important question that remains however, is If more than one set of operands are applied to the inputs of the tree, will the set of outputs be a determinate function of those inputs? The question of determinacy has been resolved for a number of data flow models However, each of these incorporates a particular interpretation and the determinacy of the models' computational behavior depends on that interpretation SDF interprets the parallel computation implied in a graphical representation in a significantly different way For example, do we assign a processor to every set of inputs? In other data flow interpretations the answer is usually yes, every complete set of inputs is assumed to enable an independent activity In SDF, a complete set of inputs will certainly imply that a single instruction or 'activity' may be independently enabled However, it may also imply the executability of a succession of instructions, the number and termination point of which is not known at the time of enabling

SDF introduces a unique situation in a parallel computation system in that it retains the Home datum throughout the execution of a chain As a result, if an instruction is unary (requires only one operand), the processor will only need to read the op-code Therefore, if we activate multiple processors for different sets of inputs to the same code, an instantiation of the code which was activated last may get ahead of one which was activated first, thus producing output tokens out of

order and compromising determinacy FIFO processing of tokens on arcs, or the simple ACKing of tokens used in the Dennis machine will not help because, for a unary operation, there are no tokens to FIFO or ACK, the operand is always in the processor

These issues become even more pronounced when considering dynamic constructs such as conditionals and loops -- programming elements which would be required in any real system

In what follows we shall develop the rudiments of a programming system in SDF It will consist of the following basic constituents

1) The *static path* These are program segments which contain no branches or decision elements In a conventional programming language they would correspond to a block of assignment statements However since we are dealing with a functional semantics with data flow there is no 'assignment' facility and hence no statement in the usual sense An expression tree qualifies as a static path but the latter is not restricted to such trees This is because, in functional languages one often encounters functions which are tuple valued, that is they return a structure of values rather than a single scalar value Thus any graph segment which is well-formed (defined later) in the usual data flow sense but contains no 'switches', 'gates' or other decision elements, will qualify, after the application of Algorithm 3 1, as an SDF static path The term *static* reflects the fact that the segments behavior is completely determined before run time

as opposed to e g a loop where the count of iteration may depend on the input data

2) The *conditional* This construct provides a basic decision facility and is semantically similar to the conditional of Rumbaugh [31], Weng [34] and others It chooses one of two execution paths based on a boolean decider token

3) The *loop* This will be our basic iteration construct

These three building blocks will form the basis of a complete, if rudimentary programming facility For each of the constructs we will demonstrate

a) that their outputs are determinate functions of their inputs
   -- over any sequence of such inputs
b) that they are live, and
c) that their expressive power remains significantly general
   despite the regimentation into chains

Our interpretation of the data flow graphical model will be 'static' in the sense that it will employ ACKnowledgements between instruction nodes to regulate the token flow among buffers of limited capacity The increased system traffic due to the ACK's is problematic and has been discussed elsewhere in connection with the Dennis and similar machines [19] SDF may lend itself very well to a dynamic interpretation in the sense used by Arvind and Gostelow [6] For a first system however, the static approach is simpler and better suited to grappling with the

most basic issues

2. Graph Models of Parallel Computation

A number of graph based formal models of parallel computation have been developed and presented in the literature We shall not attempt to survey all or even most of them this has been done elsewhere (see for example [9]) However our development will draw from some of this work and accordingly, we review below two models which will be relevant in the sequel

The Karp & Miller Model

Karp & Miller [24] presented one of the earliest models of parallel computation employing a graph in which nodes represented computational operations and arcs represented the flow of data In this model a computation consists of a directed graph with say h nodes $n_1, n_2, \ldots, n_h$ and l arcs or edges $e_1, e_2, \ldots, e_l$ Each arc must be directed from some node to another With each arc e_p are associated four nonnegative integers A_p, U_p, W_p and T_p

With each arc is associated a FIFO queue Let n_i and n_j be the nodes from which an arc e_p is directed out of and into respectively The four parameters are understood to have the following meaning

1) A_p is the initial number of data items in the FIFO queue associated with e_p

2) U_p is the number of data items added to the queue upon completion of the operation associated with n_i

3) W_p is the number of data items removed from the queue upon initiation of the operation associated with n_i

4) T_p the minimum number of items in the queue for the initiation of n_i to be enabled

T_p can be thought of as a threshold for the 'firing' of the operation associated with n_i. When the latter is initiated, W_p words are removed from the queue; when it completes, U_p words are added to each arc directed out of n_i. An *execution* of the graph is defined as a series of sets S_1, S_2 , consisting of node numbers. The nodes in each set represent those which were initiated at the ith step. Karp and Miller add another stipulation to make their model more amenable to analysis: that any node which becomes fireable is in fact initiated within a finite amount of time. A computation which adheres to the latter stipulation is called a *proper execution* of the graph.

Karp & Miller prove a strong form of determinacy for computations represented within their model: that the succession of data items appearing on a queue associated with any node during the course of a computation, will be the same for every computation with the same graph and the same initial state. This is true

despite the fact that no particular delay is associated with the operations of any node Other properties of the model, such as whether a computation terminates and conditions for boundedness of the queue lengths are investigated by Karp & Miller However, there is no provision for decision operations in their model

Adams' model

Adams' model [3] includes an elaborate specification of the data types allowed on arcs A grammar is used to generate the permissible types and the result is that data structuring at a very general level is facilitated However, this data structure handling is facilitated only in a very formal sense the model is not concerned with the problems of actually moving such structures on arcs For our purposes we can dispense with this aspect of the model and treat the data items on arcs as uniformly consisting of a word of some sort We shall denote this data word with the symbol 'w' As before, the model employs a graphical representation with nodes corresponding to operators and arcs to data exchange The model has various interpretations depending on what set of *primitive nodes* are chosen, essentially what the permitted basic operators will be The set of primitive nodes is divided into two classes, called 'r' and 's' nodes They are defined as follows

1) The 'r' nodes have an ordered set of inputs and an ordered set of outputs Associated with each node is a function f which is strict in all of its inputs, that is, a data word must be present at each of them in order for the node to be enabled
2) The 's' nodes have an ordered set of inputs and outputs as above In

addition, however, each of their inputs has associated with it one of two states *locked* or *unlocked*. The function f associated with the 's' node is strict only in the inputs arriving on unlocked nodes. That is, if a word is presented on each unlocked input, the node becomes enabled. In addition a function g is associated with 's' node which determines the next state of each of the inputs, whether locked or unlocked, depending on the current state and the value being presented on the currently unlocked inputs. That is, for a node with h inputs $g(S_1, S_2, \ldots, S_h, w_1, w_2, \ldots, w_h) \rightarrow (S'_1, S'_2, \ldots, S'_h)$, where S_i has the value 'locked' or 'unlocked'. Alternatively one can say that f is strict in all of its inputs even for the 's' node if we allow the symbol nil in the domain of the input set and stipulate that when an input is locked, it is defined as presenting the input nil.

The 'locking' feature of Adams model allows decision constructs which vary the structure of the program dynamically. Adams also has provision for recursion in his model. The model defines a node called a *procedure node* with which is associated a defining computation graph called a *graph procedure*. When the conditions for the enabling of a procedure node are present, the corresponding graph procedure is invoked with the same inputs. A *graph program* P is an ordered pair (G,X) where G is a graph procedure and X is a set containing all graph procedures that could be invoked in the course of executing G. When a procedure node is encountered, the model specifies that a copy of the corresponding graph procedure is made and the inputs to the procedure node are transferred to the graph procedure. When the graph procedure terminates, the copy is deleted and its outputs transferred to the output arcs of the procedure.

node Arbitrarily nested recursion is facilitated in such a scheme No implementation issues are considered in the model proper.

Adams proves that any computation represented in his model is determinate He also shows that his model can represent any computable function by showing how it can be made to simulate a Turing machine

3. The Basic Execution Discipline

In what follows we will be drawing on the material developed in section 1.

A processor has local high speed memory for a *main word* or MW as well as for a *main boolean* or MB. The former will correspond to the elementary data element (word size) of the system while the latter refers to boolean tokens which the program may generate for controlling decision instructions. Both items are program data in the sense that they are generated by the program itself in accordance with the meaning of the program. This is in contrast to data generated by the implementation e.g. ACK tokens which have no meaning to the program.

Whenever a processor is assigned to execute a chain of instructions we will refer to the dynamic state of its execution on that processor as an *instantiation* of that chain. The MW and MB together form the *instantiation context* of that instantiation. The instantiation context is held continuously in the processor high speed memory as long as the instantiation is active. As detailed in section 1, an instantiation may be unable to continue if certain conditions are not satisfied by the next instruction fetched from its chain. In that event it will be suspended and enter an inactive state. Its context must be saved for subsequent reactivation. Two possibilities for the context save area were discussed earlier. For the purposes of this section we shall assume that the context is saved in the instruction which caused the deactivation. Thus, in general, instructions will have an MW/MB field. Many instructions will not require the MB field -- this will

be the case whenever the instruction operates on the MW alone and is unrelated to any boolean values Other instruction types will not even require an MW field, as we shall see

In the course of executing a chain, binary instructions will require an operand in addition to the MW/MB already in the processor The additional operand required by a binary instruction, whether it be a data word or boolean control token, will be referred to as the *2nd word* or 2W and *2nd boolean* or 2B respectively Thus binary instructions will require a 2W/2B field The processor will fetch the instruction and 2W/2B operand, and will apply the operation code to the latter and the MW/MB already in its local context register The context register is used much like an accumulator in simple one address instruction conventional architectures Note that because we are emphasizing the low level parallelism and fine granularity of data flow, each instruction is an elementary operation Therefore, while the processor context holds both a main word and a boolean field the second operand field of an instruction will always contain one or the other but not both Different instructions types make use of the two kinds of program data and each type will have only the operand it requires sent to it

An instantiation context will be activated whenever a chain is newly enabled, i e started at its first instruction This will occur in two ways An I/O facility will bring MW/MB data into the system from the outside world Each such inputed MW/MB will cause the activation of an instantiation of the chain which processes it We assume the I/O facility itself will properly activate the instantiations and will not be further concerned with I/O functions in this thesis Other than I/O, a chain will be newly instantiated when, in the course of program execution, an

already active chain sends an operand to an instruction node which has no predecessors If the instruction is unary, the operand sent will comprise the MW for the instantiation If the instruction is binary, two active chains will provide an operand each before the new chain can be activated, one will be designated the MW operand and the other -- a 2W operand

However, as described in section 1, any requisite 2W/2B operands are assumed by the execution mechanism to be in the insructions when the latter are fetched If this proves not to be the case the instantiation will have to be suspended and its context saved A *context mark* (CM) will be set to indicate the presence of a suspended instantiation context in the MW/MB fields When the required 2W/2B operand subsequently arrives, the CM will be checked If it is set, the context will be reinstantiated, sequential execution proceeds with the same instruction The presence of ACKnowledgements, to be detailed later, may also be required to activate or re-activate an instantiation of a chain This protocol will apply uniformly regardless of whether a new instantiation of a chain is being activated or a previously active one is being reactivated from a suspended state Unary operations, whether at the beginning or middle of a chain, cannot result in the suspension of a chain instantiation If a unary operation begins a chain, the storing of the MW itself will cause the instantiation to be activated

4. ACKnowledgements and Determinacy

ACK's have two functions in data flow systems The more straightforward is as a synchronization device for controlling the use of limited buffer space In the Dennis data flow machine, each arc is interpreted as having a capacity of 1, i e it

can hold one instance of the output of the node which feeds it If this node fires twice before its successsor fires once, the contents of the single token buffer represented by the arc between them will be overwritten by the second firing and the first token will be lost To avoid this, each instruction node is required to return an ACK to its predecessor node indicating to the latter that it has consumed the token 'residing' in the arc which connects them Instructions will, in general, not be enabled until ACK s have been returned by all successor instructions

The need for ACK's in the above role is eliminated in SDF for successive nodes on the same chain because each instantiation of a chain is executed by a different processor which retains the instantiation context for the duration of the chain's execution Each has a local context register and hence there is no possibility of overwriting tokens In effect, there are as many buffers as instantiations

For 2W/2B operands however, this does not hold This is because all instantiations of a path must fetch 2W/2B operands from the same field in a given instruction, i e they must all share a buffer of capacity 1 Thus if an instruction a on chain A sends a 2W operand to an instruction b on chain B and two instantiation of the the former execute before one of the latter, we have the same problem as in a standard data flow interpretation with arc capacity of 1 Accordingly, we shall employ an ACK for 2W/2B operands Thus any instruction which consumes a 2W/2B will return an ACK to the source instruction Likewise any instruction which provides a 2W/2B will check for the presence of an ACK before executing Note however, that if instructions are restricted to unary and binary, only one such ACK can ever be required, the other operand is held in the instantiation

context register

ACKs have another function in a static data flow interpretation preserving determinacy by enforcing FIFO processing of successive token sets For static program segments i e those which do not contain decision elements of any kind we have seen earlier that a data flow model in which the arcs are interpreted as unbounded FIFO queues will produce outputs that are a determinate function of successive input sets [24] Restricting the capacity of arcs to 1 yields a special case of the FIFO queue and hence amounts to a method of insuring determinacy For dynamic program constructs the situation is not as straightforward While any schema representable within Adams' model is determinate it may not make progress as a computation or the outputs which it generates may not be meaningfully related to its inputs The data flow base languages which have been developed thus far generally restrict the use of dynamic program elements to certain predefined constructs These will include a conditional and an iteration device of some kind These constructs will be so designed that they behave with respect to the surrounding code as a functional operator which is strict in all of its inputs and outputs That is they will not initiate any computation unless all inputs are present and further all outputs will be generated for every instantiation As such they can be treated like any other 'static' operator and their inclusion in a computation schema will not compromise determinacy or liveness To illustrate the subtleties involved we will consider two such basic data flow configurations a generalization of the conditional and the loop These will be shown in some detail because they will serve as a springboard from which to develop the analogous constructs in SDF

Consider figure 4 1 The sections marked e1 and e2 are arbitrary well formed data flow schemata, the only proviso is that they have identical inputs and outputs The conditional chooses one of these program segments for processing the inputs depending on the value of the boolean token generated by the *decision expression* , an arbitrary boolean valued function The decision function takes as arguments some subset of the total inputs to the conditional The first group of m inputs out of the total n inputs to the conditional is shown in the diagram as participating in the evaluation of the decision expression The generated decision token is routed to the boolean input of the n *switch nodes* Each switch is responsible for 'switching' one of the n inputs to the corresponding input on the chosen alternate e1 or e2 -- as directed by the decision token The *merge* node is actually an abreviation for n merge nodes, each with two data inputs, one boolean control input and a single output The control input is fed the same token which controls the input switches, and it used by the merge to decide which of its two data input tokens is to pass unchanged to its output It is intuitively clear, and has been demonstrated formally, that the tokens which appear at the output of the merge will correspond in order to the sets of tokens arriving at the inputs of the conditional This will be true even if successive input sets are allowed to instantiate the conditional concurrently To illustrate consider two successive sets of tokens T1 and T2 arriving at the inputs of the conditional and assume that the decision tokens generated by each result in the execution of schemata e1 and e2 respectively Assume that e1 executed with inputs T1 takes longer to execute than does e2 with inputs T2 The output of e2 corresponding to inputs T2 will not be permitted to exit the conditional before the output of e1 on inputs T1 The order of boolean tokens arriving at the merge will guarantee that the two outputs are emitted in order

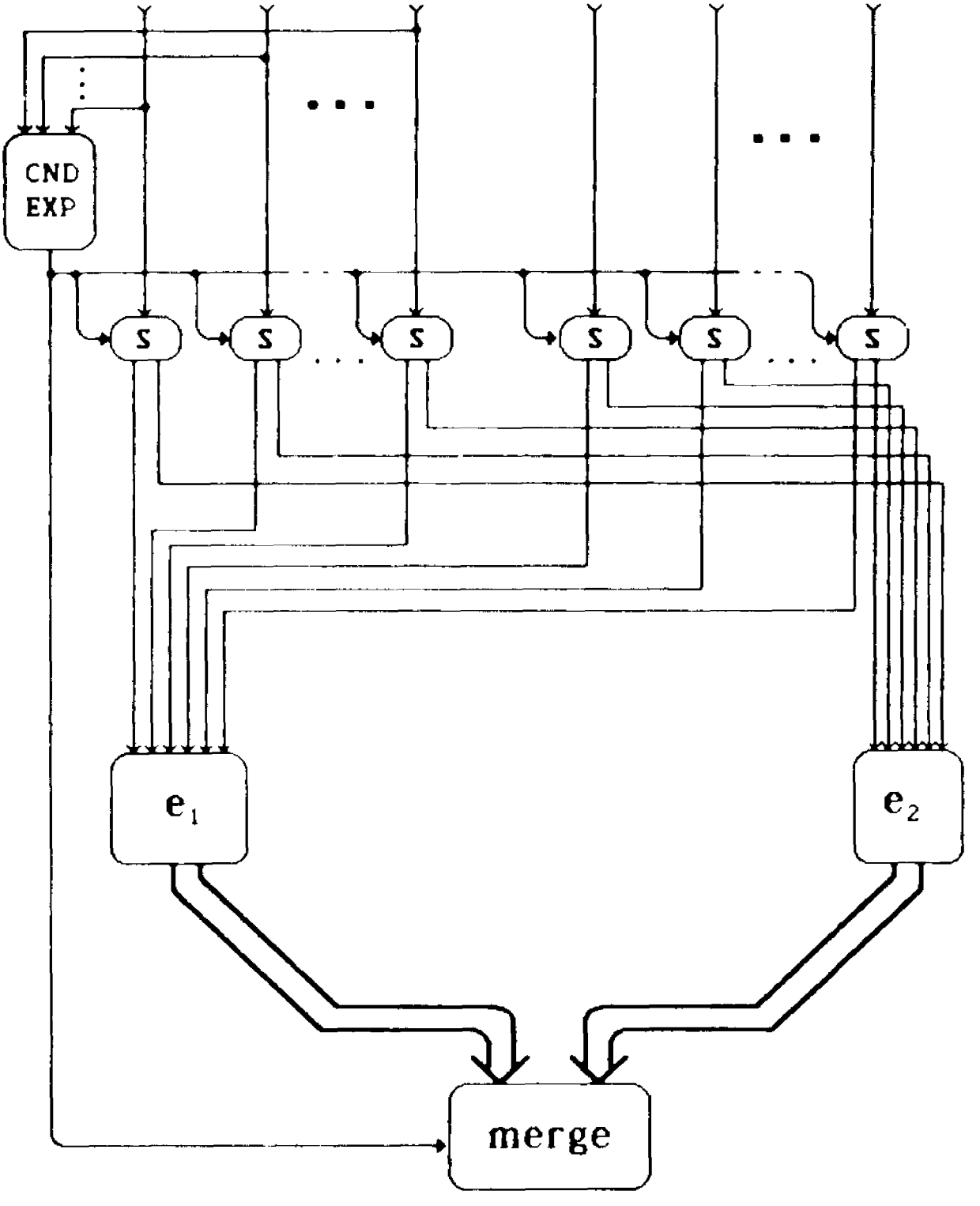


Fig 4.1 Generalized DF Conditional

Note that the schema of figure 4 1 allows the values used to make the decision to be identical to, unrelated or partially overlapped with those used in the alternates e1 and e2. If they are unrelated, some of the inputs to the conditional are simply ignored in the schemata e1 and e2. Likewise, the inputs affected by e1 and e2 need not be the same, the construct simply stipulates that all inputs affected under either boolean condition be input to the conditional. Inputs not changed by one of the alternate schemata will be passed through identity operators.

For the conditional, an appropriately restricted construct permits multiple concurrent instantiations while still preserving determinacy and liveness. For iteration however this is not the case. Figure 4.2 shows our generalization of the data flow iteration construct. If we trace the flow of data beginning just after the 'G' nodes, the construct will be seen to include a structure similar to the conditional. Inputs to the loop are applied to the switches, and, as before, a subset of them are applied to the decision schema. The latter generates a boolean token in accordance with whether or not a next iteration is to be executed. This token is fed to the switches which route the data either directly to the loop's successors or to the *loop body*, an arbitrary well formed data flow schema with n inputs and outputs. The outputs of the loop body are fed to one of the *gate* inputs. The function of the gates is to distinguish between new input to the loop and *feedback* inputs -- those which are returned from a previous iteration. In addition to generating a boolean token for the switches, the decision schema will provide the same token to each of the gates. Thus when a next iteration is elected, the outputs of the loop body are also chosen by the gates as the next inputs to the decision expression rather than any new loop inputs which may be available at the other inputs to the gates. When a decision has been made to exit the loop, the

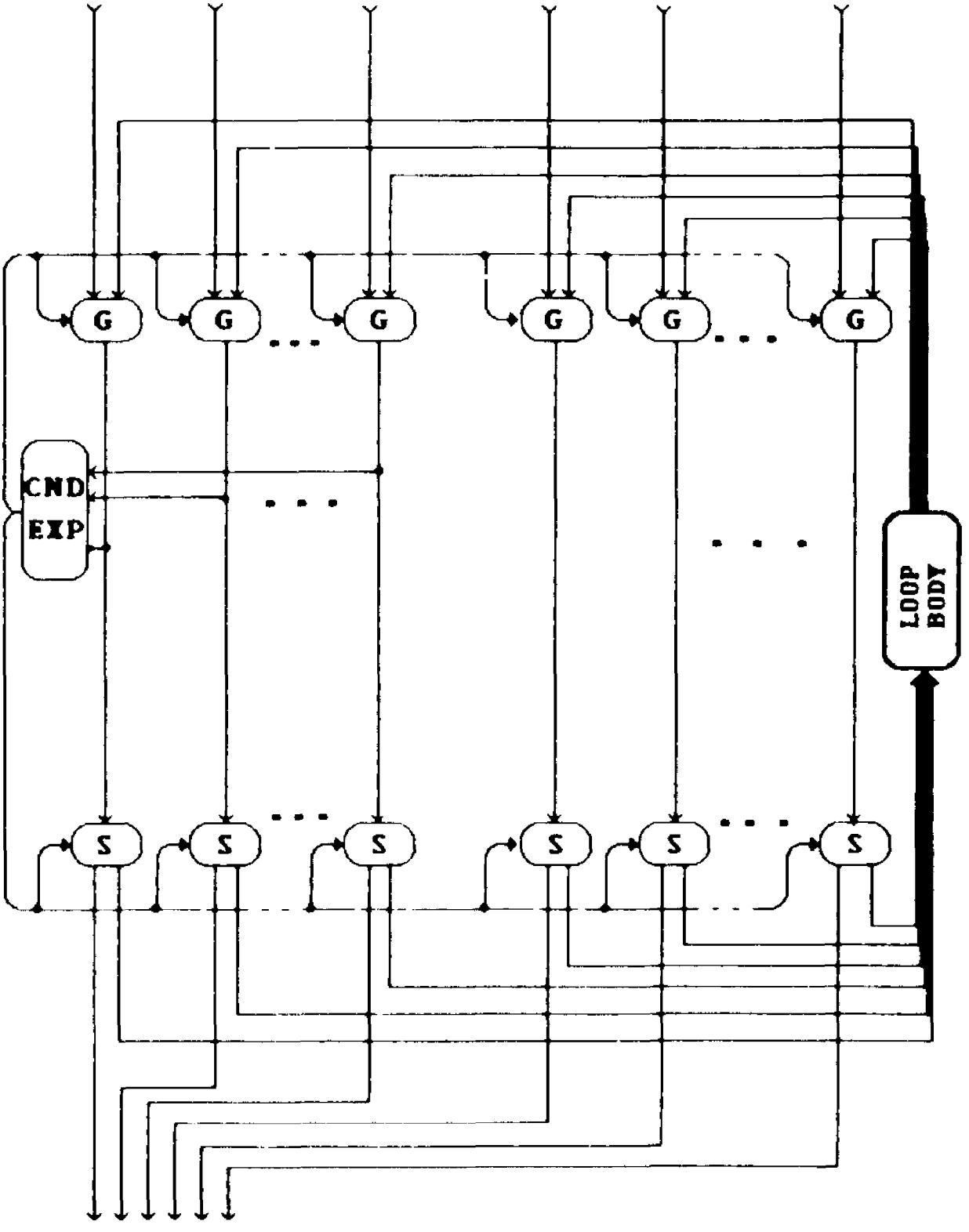


Fig 4.2 Generalized DF Loop

same boolean value which causes the switches to route tokens out of the loop will permit the gates to pass new tokens waiting at the inputs to the loop In figure 4.2 the gate nodes together with their boolean control inputs insure that no confusion arises between new inputs to the loop and feedback inputs from previous iterations If the two type of inputs were simply merged indeterminately, the behavior of the loop would be unpredictable for successive sets of inputs or, more accurately, it would depend on the timing of those inputs and the delay of processing them In short the loop's outputs would not constitute a determinate function of its inputs This arrangement implies that only one instantiation of the loop may be permitted at a time Thus pipelining between successive stages of the loop body are not possible considerable oportunities for parallelism are not exploited

The reduced parallelism resulting from the elimination of pipelining in these constructs has been addressed in the literature in two ways For the static machine, recursion has been suggested as a way to invoke multiple instantiations of the same code without compromising determinacy [34] The various pointers maintained between activation records keeps track of which operands belong to which instantiations Tail recursion becomes the basic iteration device The other approach is that of the dynamic machine Tagging is used to give a unique name to different operand sets thus permitting different instantitations of the same code to be processed concurrently without interferring with one another For the remainder of our development here we continue with the assumption of a static machine and the restricted parallelism which it allows for dynamic program constructs Nevertheless, as we shall see, SDF is not as restrictive in this regard as conventional data flow

How does determinacy fare in SDF? The following discussion is intended to provide an intuitive perspective on the issues which are dealt with in the subsequent formal devlopment. Consider two successive sets of inputs *a* and *b* to a simple static path, i.e. a chain (we shall give a formal definition of a static path shortly). Both activate instantiations of the chain on different processors, say A and B respectively. Let the chain consist of *n* instruction nodes I1, I2, ..., I*n* where the first *k* instructions are unary and the rest are binary. All instantiations must fetch and execute the same copy of the code, and must fetch 2W/2B operands, where required for a given instruction, from the same field. For the first *k* instructions, it is clear that no confusion can arise between operand sets, each processor operates strictly on the instantiation context in its own local register. It is equally clear however, that the processors may complete those *k* instructions in a different order than they were activated. The consequences of this timing disparity will become evident as soon as a binary instruction is encountered. If processor B completes *k* instructions and fetches instruction I(*k* +1) before processor A does, it will consume the 2W/2B operand it finds there -- an operand which arrived at I(*k* +1) first and hence was intended for the instantiation of that instruction to be executed by processor A. Even if all instructions are binary, instantiations can get out of synch. This will occur for example if the 2W/2B field in I1 is replenished before processor A completes execution of that instruction. This will enable processor B to fetch and execute I1 and possibly complete it before processor A does. The result will be that the 2W/2B operands required for the execution of I2 will be consumed out of order.

Our approach will be to guarantee FIFO consumption of 2W/2B operands by using

ACK's between successive nodes on a chain That is, instruction I1 will not be permitted to execute without the presence of an ACKnowledgement from its successor, I2 The latter will provide the ACK as part of its execution sequence, thus indirectly indicating that any 2W/2B has been consumed Thus processor B could not even begin execution of I1 before processor A has actually consumed a 2W/2B from I2, because the ACK required to enable I1 was consumed by processor A and will not be replenished except by A itself when it executes I2

We shall designate the ACKnowledgement exchanged by nodes on the same chain ACK* to distinguish it from the ordinary ACK used to synchronize the forwarding of 2W/2B s between nodes on different chains This emphasizes that the ACK* does not have the buffer capacity controlling function of the ordinary ACK, it is strictly a determinacy enforcement device Note that the ACK* is not an ACKing of 2W/2B s -- that is the function of the ACK Rather it indicates to the node at which it arrives that a previous instantiation has executed the successor node It can be thought of as ACKing the MW/MB s held in each instantiations processor and thus serves to maintain the correct order among the main context words of the instantiations The *effect* of maintaining this order is insure that 2W/2B's are consumed in FIFO order Thus even a succession of unary instructions will be required to exchange ACK* s because if different instantiations of such a succession are permitted to execute completely asynchronously they may complete out of order and subsequently consume 2W/2B's out of order

5. Formal Development:

5.1 The Instructions

Table 4.1 collects the different instruction fields and their functions as discussed above

Table 4.1 -- Instruction Fields

mnemonic	meaning	comment
OP	operation code	includes instruction length specification. instruction length will vary
MW	main word	components of
MB	main boolean	instantiation context
2W	second word	additional operand
2B	second boolean	for binary operations
CM	context mark	indicates presence of MW or MB
ACK	acknowledgement	from recipient of a 2W/2B on another chain
ACK*	acknowledgement	from successor on the same chain
DEST	destination address	to which a token is to be forwarded

DEST addresses are specified as triple (c, i, t), where c is the destination chain address, i is the address of the instruction within the chain and t indicates the type of token being forwarded (the c is analogous to p in the triple (p, i, t) used in section 2) They type indication is used to determine which field in the destination instruction the token will be written to There are three types an instruction may contain exactly one of each They are

1) t = 2W/2B indicates a program datum either a word or boolean value A node may contain one or the other, but not both Alternatively if an operand is to be used to instantiate the first instruction on a chain, the DEST type will indicate MW/MB as the target field

2) t = A indicates that an ACK should be sent to the DEST address Will be present for binary operations only

3) t = A* indicates that an ACK* shold be sent to the DEST address -- usually the predecessor instruction on this chain Required for all nodes except inside loops (discussed later)

Table 4 2 lists the basic instruction types which we shall employ in our SDF programming system First an example of each instruction type is given This is followed by the different field types where an X indicates that the field is used by the instruction DEST field types used by the instruction are listed together in one

Table 4.2 -- Instruction Types

Instruction type	Example	fields MW	MB	2W	2B	CM	ACK	ACK*	DEST	Action
unary operator -- boolean	logical NOT	X	X	--	--	X	[X]	X	[B],A*	u(mb) -> mb -> Dest-B
unary operator -- word	2's cmplmnt	X	X	--	--	X	[X]	X	[W],A*	u(mw) -> mw -> Dest-W
binary operator -- boolean	logical AND	X	X	--	X	X	[X]	X	[B],A,A*	b(mb,2b)->mb -> Dest-W
binary operator -- word	addition	X	X	X	--	X	[X]	X	[W],A,A*	b(mw,2w)->mw -> Dest-W
unary comparator	If mw < Ø	X	--	--	--	X	[X]	X	[B],A*	u(mw) -> mb ->Dest-B
binary comparator	If mw > 2w	X	--	X	--	X	[X]	X	[B],A,A*	u(mw,2w) ->mb -> Dest-B
B - splitter	------	X	X	--	--	X	X	X	B,A*	mb - > Dest-B
W - splitter	------	X	X	--	--	X	X	X	W,A*	mw - > Dest-W
unconditional jump	------	--	--	--	--	--	--	--	-----	jump address -> PC
branch on Condition (1)	------	X	X	--	--	X	[X]	X	[W],A*	If condition (mb) then mw -> Dest-W else branch adrss->PC
branch on Condition (2)	------	X	--	--	X	X	[X]	X	[W],A*	If condition (2b) etc. [as in BC1]
terminate	------	--	--	--	--	--	--	--	-----	cancels current context and frees processor

column The last column shows the effect of the instruction's execution Bracketed fields may or may not be included in the specified instruction; if unbracketed, the instruction will always include the field The notation u(x, y) -> z used in the Action column denotes that the result of the unary operation of the specified instruction applied to the arguments x and y, is written to z Analagously, b(x, y) -> z represents the effect of the application of a binary operation -> Dest-*x* means that the value is to be sent to the destination of type *x* from among the DEST fields in the specified instruction

In connection with table 4 2 note the following

1) All instruction types are shown requiring both the ACK* and DEST-A* fields reflecting the adherence of all nodes along a chain to the ACK* protocol However, the first instruction on a chain will not have a DEST-A* field, as there is no chain predecessor to whom the ACK* should be returned The MW/MB operand supplied from another chain is treated like all interchain token exchange and is acknowledged with an ordinary ACK Similarly, the last node on a chain has no ACK* field as there is no successor to provide it

2) In general, all instructions types will require the context save fields MW/MB and CM insofar as they must receive an ACK* from their chain successor before executing The exceptional case will be inside of loops (detailed later), and certain instructions discussed in (7) below, which avoid the ACK* protocol In such cases, *unary* instructions will not require the context save fields since the processor contains all the requisite data in its local registers and therefore there is nothing which could cause its suspension at this

instruction

3) Dest-A fields are required by all binary instructions to return an indication that they have consumed their 2W/2B operands

4) Any instruction which sends an operand to another chain requires the ACK field to ascertain that the destination instruction has consumed the previous operand and is therefore free to receive another

5) Comparator instructions test data word(s) for some condition and leave the boolean result of the test in their MB fields in effect a condition code The latter may be sent as a 2B operand to other chains as usual

6) Splitters send either an MW or MB to Dest-W or Dest-B unchanged They are used to distribute operands to more than two (chain successor and one other) destinations A sequence of splitters can be used to distribute operands to an arbitrary number of destinations In what follows we shall abreviate such a sequence with a single distributor node shown with multiple outputs It is to be understood as representing as many splitters as necessary -- see figure 4 3

7) Unconditional jumps are permitted only to instructions *within the same chain* No ACK* or Dest-A* is shown for these instructions as they require no inputs and have no effect on data It is therefore plausible to assume that all instantiations will experience identical delay in executing them Other (unary) instruction types may qualify in the same way In such cases, the instructions immediately before and after the region of constant delay will be

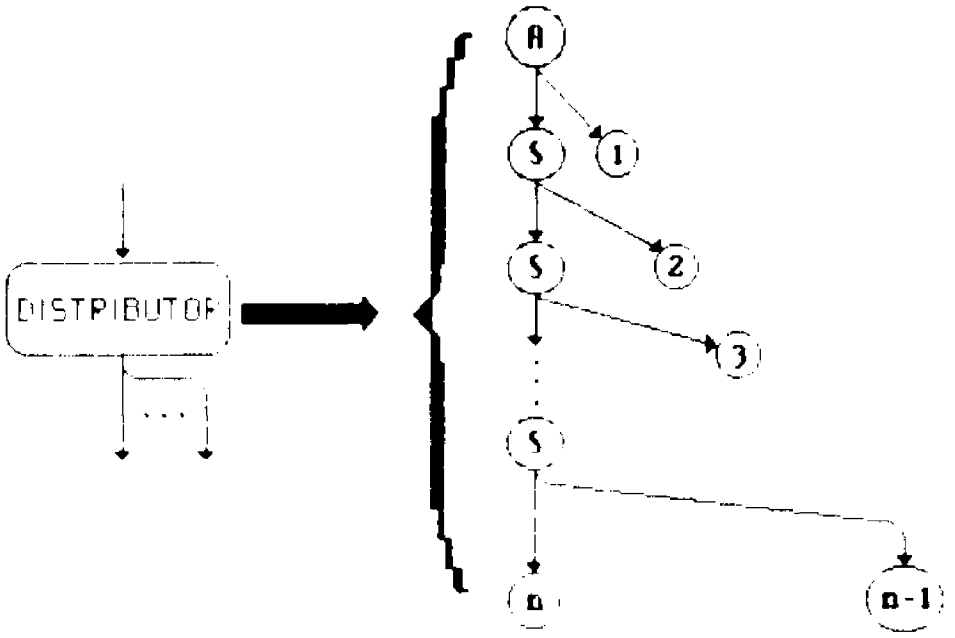


Fig. 4.3

The distributor is understood to represent the expansion at the right. The node marked 'A' generates the token to be distributed. The nodes numbered '1', '2', ..., 'n' are the recipients of the distributed token. The 'S' nodes are splitters

required to exchange the ACF* protocol

8) The Branch on Condition (BC) instructions use a boolean value to decide whether or not to take the branch again -- only within a chain. Type (1) BC assumes the boolean value has been computed by its chain predecessor. It therfore simply accesses its own condition code (MB). Type (2) BC accepts a 2B from some other node. Both types may optionally send their MW's to a Dest-W *if the branch is not taken*. In other words, the BC may incorporate the action of a W-splitter on a conditional basis.

9) Instruction nodes will have all of their requisite ACK and ACK* fields set, as part of their initial state

10) As with most formal graph models of computation, we shall assume that once an instruction is enabled, i e all requisite fields have been filled, it will execute within a finite but unspecified amount of time

We now consider the three basic SDF programming constructs in turn

5.2 The Static Path.

Any data flow graph is 'interpreted' to some extent by virtue of being called by that name its nodes represent particular operations driven by the flow of data which is represented by the arcs The operations represented by the nodes are also generally assumed to be functional and hence determinate The notion of an interpretation has usually been associated in the literature with some particular implementation of the arcs -- specifically how the storage production and consumption of multiple operands which may accumulate for a node is implemented

Definition: By an *uninterpreted* data flow graph we shall mean one which is uninterpreted beyond the minimal sense described above In particular, it will have no protocol specified for the handling of multiple operands

A *static data flow graph* is a well formed data flow schema, all of whose nodes represent functional operators which are single valued and strict in all of their inputs That is, it will contain no gates, switches or other flow control nodes The notion of 'well formed' has been treated extensively in the literature When control nodes are excluded, the notion of 'well formed' boils down to simply insuring that, other than the schema's specified inputs and outputs, all node inputs and outputs are connected to arcs supplying tokens of the appropriate type For the moment, we require only this simplified version of 'well formed' However, for different models proposed in the literature the *interpretation* of a static data flow graph will differ in certain details We shall adopt one particular such interpretation for our purposes -- specified below

An SDF *static path* (abbreviated SP) is a static data flow graph which has been partitioned by the application of Algorithm 3 1 Equivalently, it is a set of chains which include no branch instructions connected arbitrarily via COM arcs, provided that the underlying graph is optimally partitioned by the chains in the sense of Theorem 3 1 The static path is further restricted as follows Inputs may be directed only to the first node on a chain, outputs may be taken only from the last node of a chain Other than such I/O, all arcs must be connected to a node on both ends

The last restriction in the definition of a static path is a convention which will prove useful in later proofs We can define *dummy nodes* which act like an identity function, they have no effect on data Thus if we wish to take an output from a node somewhere in the middle of a static path's chain, we can simply start a new chain with a dummy node and have its MW supplied by the node from

which we wish to take an output The dummy node will be the last node on a chain and hence taking an output from it will be permitted under the definition of a static path

The *in-degree* of a static path is equal to the number of inputs to the path, the *out-degree* to the number of outputs Thus a simple static path has in/out-degree of 1; a compound path will have in/out-degree ≥1 By including chains consisting of dummy nodes which take inputs or provide outputs the in/out-degree of a static path may be increased arbitrarily

A *sub-static path* (subSP) of a static path P with underlying graph G, is defined for any subgraph G' of G as the chains of P restricted to the nodes in G' Where otherwise understood we shall omit reference to the underlying graph G Note that the chains of a subSP of a static path P may represent a suboptimal partition of its nodes even though P is optimally partitioned Thus a subSP of some static path may not satisfy the definition of a static path

To show that a static path represents a determinate computation we will make use of basic results established in the literature Karp and Miller's original model for parallel computations would suffice as a basis for deriving our results for the static path Setting $U_p = W_p = T_p = 1$ for all nodes p and $A_p = 0$ for all nodes other than input nodes in their model would yield the equivalent of a static data flow graph as we have defined it However, their original model does not include a decision control mechanism which we shall require when developing the conditional and loop contructs while their later model [25] is unnecessarily general for our purposes We shall therefore draw on Adams' model which does

provide the necessary control nodes

For the static path we do not require any control nodes We can therefore consider Adams' model restricted to 'r' nodes That is, all node inputs are 'unlocked' and nodes are considered firable when an input datum is available on every input We can also dispense with Adams' protocol for procedure invocation since at this point we are dealing with the static path which consists only of primitive nodes A 'program graph' in the sense that Adams' uses the term restricted in this way qualifies as a static data flow graph as we have defined it and we shall be refering to this particular interpretation when using the term static data flow graph in the sequel

For the static data flow graph the following results have been shown

Proposition 1 Let G be an uninterpreted static data flow graph with n inputs and m outputs Then the m outputs of G represent a determinate function of a *single* set of n inputs

Proposition 2 Let G be as above and let the arcs in G be interpreted as unbouded FIFO queues Then successive sets of the m outputs of G representa determinate function of successive sets of n inputs

Note that in Adams' model Proposition (1) is simply a special case of Proposition (2) [4] . Other authors have proved the results separately We shall require both

Lemma 4.1: An SDF static path P represents a determinate function of a *single* set of inputs Further, any subSP of the static path P represents a determinate function of a single set of inputs

proof: An SDF static path is a static data flow graph which has been partitioned by an application of Algorithm 3 1 The partitioning leaves the topology of the underlying graph unchanged Hence proposition 1, which applies to an uninterpreted DF graph, will apply equally well to the partitioned version, i e to the SDF static path Further, the ACK protocols introduced by the SDF interpretation are equivalent to an uninterpreted graph for a single set of inputs This is because all nodes have any requisite ACKs as part of their initial state and hence will not be delayed because of the ACK requirements Only a second or later set of inputs would require the generation of new ACKs The same reasoning applies equally well to any subSP of a static path □

Proposition (2) is not as easily extended to an SDF SP The handling of multiple inputs is specified by an *interpretation* of the underlying graph and here SDF differs significantly from the usual dataflow interpretations Specifically, Adams' model specifies FIFO queues for the links between nodes With each primitive node p is associated an Initiation Queue IQ_p which also has a FIFO discipline Each execution activity occasioned by a node p is associated with an identifier which is also placed on IQ_p These executions are initiated in accordance with successive input operand sets arriving on the FIFO queues Since we have eliminated procedure activation from Adams' model in the static data flow graph, we can simplify matters by thinking of the static data flow graph as representing a

machine Each node p represents a hub of computational activity Various instances of the operation specified by p may be going on concurrently The interpretation specifies only that a) each such hub is fed by a FIFO queue from each node which provides its input operands, and b) each hub places its results on outgoing FIFO queues in the same order as corresponding input sets were received That is results are placed on outgoing arcs in the order in which their identifiers appear on IQ_p

For an SDF static path however FIFO queues cannot be thought of as characterizing the arcs connecting nodes on a chain because each such node is not necessarily an independent locus of activity Several nodes comprising a chain segment may be invoked with the same activation The identifier associated with that activation is as usual also associated with the data token input to the first node on the segment However, successive versions of the token, as it is operated upon by the nodes on the segment, will also be associated with the same identifier simply by being retained in the activating processor Hence there is no ambiguity associated with data travelling along the arcs on a chain and the notion of FIFOing such data is not meaningful

While this would appear to be a simplifying condition over the static data flow graph, we nevertheless cannot apply Proposition (2) directly to the SDF static path Indeed the notion of an 'input' to an arbitrary node, as it is used in the static data flow graph, is not readily understandable in SDF, since a series of nodes may be executed as a single activity

We will describe for the SDF static path a formal analog of the notion of a 'node' and its inputs and outputs as it is defined for ordinary static data flow graphs, and show that its behavior satisfies the conditions for Proposition (2) to apply We shall require the following notation Denote by P a static path Let the chains in P be $c_1, c_2, \ldots, c_n$, and denote the instruction nodes on c_i by $I_{i1}, I_{i2}, \ldots, I_{i,mi}$ Where the distinction is clear we shall also use I_{ik} to denote the corresponding node of the underlying static data flow graph Let MW_{ik} be the main word input to I_{ik} We shall assume the MB to be included in the MW so as not to clutter the notation Thus the words $MW_{11}, MW_{21}, \ldots, MW_{n1}$ are the n input words to the n chains Let $2W_{ik}$ be the second operand required by I_{ik} from some other I_{rs}, $r \neq i$ and $s < k$ Define the *inputs* to a node I_{ik} in an SDF static path as follows

1) for a COM arc directed into I_{ik} the associated 2W operand is an input.

2) for a SEQ arc directed into I_{ik} the associated MW operand is an input

Note that in the case of the MW this 'input' is strictly a formal notion since it may physically be an intermediate value within a single activity Now let successive instantiations of the instruction I_{ik} be represented by superscripting Thus we will refer to the successive *events* I^l_{ik} and I^{l+1}_{ik} To be precise, such events will

refer to the *completion* time of the jth and $j+1$st instantiations respectively of the instruction I_{ik}. Similarly, the main words consumed by the events I^j_{ik} and I^{j+1}_{ik} will be denoted MW^j_{ik} and MW^{j+1}_{ik}, and likewise $2W^j_{ik}$ and $2W^{j+1}_{ik}$ for the 2W's consumed by the same instructions.

Proposition (2) says that if nodes are fed by FIFO queues and further that they place their outputs on arcs in the order that corresponding inputs were consumed then the graph represents a determinate computation. However, it is not strictly necessary that actual FIFO queues be employed for Proposition 2 to hold. What we have in mind can be seen by defining the *creation time* $t_c(MW^j_{ik}/2W^j_{ik})$ of the data items $MW^j_{ik}/2W^j_{ik}$ as follows:

1) if the datum is an initial input then its creation time is equal to its position in the sequence of inputs on the same arc.

2) otherwise $t_c(MW^j_{ik}) = h$, where the event $I^h_{i,k-1}$ generated MW^j_{ik} as its output and $t_c(2W^j_{ik}) = h$, where the event I^h_{rs} generated MW^j_{ik} as its output for some $r \neq i$ and $s < k$.

Thus t_c is not an absolute time scale but rather an ordering of the completion of events I^j_{ik} in time. We can now rephrase Proposition 2 as follows:

If 1) data are consumed in the order created

and 2) results are created in the order that corresponding inputs are consumed

then the computation represented by a static data flow graph is determinate

We can imagine these conditions implemented without the use of FIFO queues as follows Each datum is appended a label when created designating its creation time Arcs represent memory facilities dedicated for use by the nodes to which the arc is incident The producer node deposits results in this memory at arbitrary locations As long as the memory contains at least one datum, the consumer node searches for the one with lowest label value and fetches it as an operand This arrangement is entirely equivalent to that of Adams and Karp & Miller and we can now use this version of Proposition 2 without futher justification

We now show that nodes in an SDF static path statisfy the conditions of Proposition 2 For nodes fed by arcs which are inputs to the graph as a whole, we need only show that condition (2) of the revised proposition holds because the creation times of input data are part of the initial conditions For other nodes however, in order to capture the essence of both conditions, i e that data tokens are both created and consumed in order, we shall have to consider a *pair* of adjacent nodes in SDF -- since the definition of the creation time of a datum MW^j_{ik} / $2W^j_{ik}$ depends on the predecessor of I_{ik}

The result is via a pair of lemmas

Lemma 4.2: Let I^j_i be instantiation j of instruction i on some chain in a static path P The ACK* consumed by the event I^j_i will be replenished only by the event I^j_{i+1}

proof: By strong induction on the instantiation number I_i has an ACK* as part of its initial state which is consumed by the event I^1_i Now only an execution of I_{i+1} can provide an ACK* for I_i But no event I^j_{i+1}, $j>1$, will ever occur before I^1_{i+1} because I_i must always execute before I_{i+1} and there is no ACK* at I_i with which any jth instantiation of I_i could execute Hence only I^1_{i+1} can provide the ACK* This is the base case

Now we assume the lemma holds for instantiations $j \le k$ and show that it holds for the $k+1$st instantiation as well That is, we wish to show that only the event I^{k+1}_{i+1} will replenish the ACK* consumed by I^{k+1}_i We consider all the possible events which could conceivably replenish the ACK* consumed by I^{k+1}_i As before, it must be some instantiation of I_{i+1} Any event I^r_{i+1}, $r>k+1$, cannot possibly replenish the ACK* consumed by I^{k+1}_i because all instantiations $>k+1$ execute I_i after I^{k+1}_i by definition and hence could not have executed I_i until the ACK* consumed by I^{k+1}_i had already been replenished It follows that they

themselves could not have replenished it

Thus the ACK* in question must be replenished by some event I^{j}_{i+1}, $j \leq k+1$ However, for every $j \leq k$, it must be that the ACK* provided by I^{r}_{i+1}, $r \leq j$, cannot replenish the one used by I^{j+1}_{i} To see this, consider that by the induction assumption only I^{j}_{i+1} can replenish ACK* consumed by I^{j}_{i} Also I^{j+1}_{i} directly succeeds I^{j}_{i} by definition Hence if there were a situation in which I^{r}_{i+1} had replenished the ACK* consumed by I^{j+1}_{i}, this would imply in particular that the ACK* used by I^{r}_{i} had been replenished else I^{j+1}_{i} could not have occured But this replenishment took place before the event I^{r}_{i+1} since the latter, in replenishing the ACK* consumed by I^{j+1}_{i}, could only occur after I^{j+1}_{i} But this contradicts the induction assumption

Hence in particular no event I^{j}_{i+1} $j \leq k$ could replenish the ACK* consumed by the event I^{k+1}_{i} Since we have already established that the same is true of any I^{r}_{i+1}, $r > k+1$, it follows that only I^{k+1}_{i+1} qualifies to replenish the ACK* consumed by I^{k+1}_{i} The lemma follows □

Let I_{ik} and I_{st} be any pair of adjacent nodes in an SDF static path P, such that the latter is a successor of the former Let A^{j}_{ik} be the jth set of inputs consumed by

I_{ik} and B^j_{ik} the jth output of I_{st} Note that the superscript for A^j_{ik} specifies the order of consumption of inputs, not the completion time of the event which consumes them -- as it does for MW/2W's Let f_{ik} denote the mapping implemented by I_{ik} and similarly use f_{st} for I_{st} These mappings are determinate by assumption Define $f(A_{ik}) = f_{st}(f_{ik}(A_{ik}))$, i.e the composition of the two mappings for a *single* argument A_{ik} Note that this mapping is not necessarily a function since I_{st} may require an input not provided by I_{ik} and hence $f(A^j_{ik})$ may have different values, depending on the alternate input to I_{st} In particular $f(A^j_{ik})$ may be undefined if that input never arrives We are not concerned to deal directly with the alternate input of I_{st} because, since I_{ik} and I_{st} can be any pair of adjacent nodes, anything proved for one input of I_{st} and its predecessor I_{ik} will also hold for the other input and *its* predecessor However, we can denote the alternate input's jth arrival by a place holder, ∂^j Then f becomes a function of two arguments $f(A^j_{ik}, \partial^j)$

Denote by F the mapping of input *sequences* to output sequences implemented by the composite action of I_{ik} and I_{st} That is

$$[B^1_{st}, B^2_{st}, \ldots] = F[(A^1_{ik}, \partial^1), (A^2_{ik}, \partial^2), \ldots]$$

Lemma 4.3: $F[(A^1_{ik}, \partial^1), (A^2_{ik}, \partial^2), \ldots] = [f(A^1_{ik}, \partial^1), f(A^2_{ik}, \partial^2), \ldots]$

where (A^j_{ik}, ∂^j) is a valid input set for all j

proof: First note that I_{ik}, I_{st}, their connecting arc and the input arcs included in A_{ik} and ∂, together constitute a subSP of the SDF static path P By Lemma 4.1 the output of this subSP is determinate and we have that $B^j_{st} = f(A^j_{ik}, \partial^j)$

Now there are 2 cases to consider

case 1 The arc connecting I_{ik} and I_{st} is a COM arc In this case the operand transmitted from I_{ik} to I_{st} is $2W_{st}$ Consider the first pair of successive input sets A^1_{ik} and A^2_{ik} to I_{ik}, and assume that A^1_{ik} has been consumed For I^2_{ik} to execute an ACK must be returned to I_{ik} from I_{st} indicating that the latter has consumed $2W^1_{st}$ and can accept another Since I_{st} will not return an ACK until it has executed it follows that an event I^1_{st} must complete before I^2_{ik} can begin Hence the first 2 outputs of I_{st} can only be $f(A^1_{ik}, \partial^1)$ and $f(A^2_{ik}, \partial^2)$

Now assume the lemma is true for j successive input sets, i e

$$F[(A^1_{ik}, \partial^1), (A^2_{ik}, \partial^2), \ldots, (A^j_{ik}, \partial^j)] = [f(A^1_{ik}, \partial^1), f(A^2_{ik}, \partial^2), \ldots, f(A^j_{ik}, \partial^j)]$$

By exactly the same argument, the event I^{j+1}_{ik} cannot begin before I^{j}_{st} completes and similarly event I^{j+2}_{ik} cannot begin before I^{j+1}_{st} completes Hence $f(A^{j+1}_{ik}, \partial^{j+1})$ will follow $f(A^{j}_{ik}, \partial^{j})$ as output of I_{st} and the lemma follows by induction Note that if I^{1}_{st} is the first instruction on a chain $(t=1)$, the operand transmitted may be MW^{j}_{s1} but the same ACK rules apply and the argument is identical

<u>case 2</u> I_{st} is a chain successor of I_{ik} i e $s=i, t=k+1$ In this case the result of executing I_{ik} is kept in the executing processor's Home registers and the contents thereof become the MW_{st} to be used in the execution of I_{st} Let A^{j}_{ik} and A^{j+1}_{ik} be any pair of successive inputs to I_{ik} and assume that I^{j}_{ik} has begun execution In order for I^{j}_{ik} to begin, and ACK^{*} must have been consumed By Lemma 4 1 this ACK^{*} can be replenished only by I^{j}_{st} Hence I^{j+1}_{ik} cannot begin before I^{j}_{st} completes The same induction as was used in case 1 now applies with the result that

$$F[(A^{1}_{ik}, \partial^{1}), (A^{2}_{ik}, \partial^{2}), \ldots] = [f(A^{1}_{ik}, \partial^{1}), f(A^{2}_{ik}, \partial^{2}), \ldots]. \quad \square$$

Lemma 4.4: An SDF static path represents a determinate computation

proof: We show determinacy by satisfying the conditions of Proposition 2, i e that data are consumed and produced in order Using the same notation as above

for a pair of adjacent nodes, these conditions may be stated formally:

(4.1) If $t_c(MW^j_{st}) < t_c(MW^h_{st})$ then $t_c(f_2(MW^j_{st}, \partial^j)) < t_c(f_2(MW^h_{st}, \partial^h))$

or, since the superscript of MW^j_{st} refers to the completion time of the event I_{st} which consumes it, the right side of (4.1) is equivalent to simply $j < h$.

Now assume to the contrary, i.e. that $f_2(MW^h_{st}, \partial^h)$ was created before $f_2(MW^j_{st}, \partial^j)$ or $h < j$ and the output sequence from I_{st} is [. . . $f(A^h_{ik}, \partial^?)$, . . . $f(A^j_{ik}, \partial^?)$, . . .]. By Lemma 4.3, the input sequence to I_{ik} must be [. . . , A^h_{ik}, . . . , A^j_{ik}, . . .]. Our assumption then implies that the instruction I_{ik} initiated execution on the input set A^j_{ik} before completing execution on a previous input set A^h_{ik}. This in turn implies that an ACK* was available for a second instantiation of I_{ik} even while the first had not progressed to the next instruction which, by Lemma 4.2, is impossible. If I_{ik} is the last node on a chain, the ACK which must be returned by its successor on another chain similarly guarantees that no second instantiation can be initiated until a previous one has completed. One successor, either on the same or a different chain, must exist, namely I_{st}. □

We have demonstrated the determinacy of the SDF static path. Note however, that we have not demonstrated liveness nor is it necessary to assume it in Lemmas 4.3

and 4.4. We only needed to show that nodes adhere to FIFO processing on their inputs, i.e. that *if* outputs are produced, they correspond to successive inputs. It is still possible that an input will fail to arrive or will be somehow invalid and cause the computation to hang up. The ACK* which must be returned to I_{st} by its successor has also not been considered in the proofs; this could also hold up the execution of I_{st} and hence the production of outputs. Strictly speaking therefore, the right side of the assertion in Lemma 4.3 should be interpreted as *any prefix* of the indicated sequence of outputs. As the reader may verify, Lemma 4.4 still holds with this interpretation.

In this thesis we shall use the term *live* in the following way.

Definition: Let P be an SDF program construct with input set A and assume that P is known to generate a determinate output set B for a *single* input set A. Let this mapping be denoted f. P is called live if, for a sequence of inputs $[A^1, A^2, \ldots]$, the output sequence $[f(A^1), f(A^2), \ldots]$ is always generated.

Thus liveness as we are using it is only defined where determinacy holds as well. A construct can be determinate and not live but not vice versa.

The property of *reusability* has also been investigated in connection with graph modelled computations. Reusability refers to the property of a graph whereby no unused tokens are left after a terminal state is achieved. Such residual tokens would easily result from a computation which hangs up. On the other hand, if we could work only with computation structures that were guaranteed to make consistent progress, i.e. consume and process all input presented, it would be

much more straightforward to design programs that are clean, that is, leave no tokens behind Thus the two concepts, while distinct, are related

It is easy to show that liveness and reusability hold for the SDF static path As long as graphs are well-formed in the simple sense of correctly matched input-output types, it is clear that all inputs will be completely consumed because nodes fire only when all inputs present data and all such data are anihilated when the node fires Formally, once we have shown that Adams' model is applicable to the SDF static path via Lemmas 4 1 through 4 4, any properties of that model will apply to SDF as well While Adams himself does not address the issues of liveness and reusability, his model -- when restricted to the static data flow graph -- is equivalent others (e g [24]) for which these properties have been shown to hold It is clear that the expressive power of the static path is at least as great as that of the data flow static path -- the underlying graph on which it is based An interesting sidelight is the possiblity that the linear flavor of the chains from which an SDF static path is constructed, may facilitate a more programmer friendly environment for software development than the essentially amorphous data flow graphs In this sense, the SDF static path may have a greater expressive power

5.3. The Conditional

We turn now to the SDF *conditional*, derived from the generalized data flow conditional of figure 4.1. The conditional is comprised of a *conditional boolean generator* (CBG) and a *conditional alternator* (CA). Refer to figure 4.4. The CBG makes the determination of which of 2 static paths are to be executed, based on some function of L of the conditional's n inputs. The boolean decider token is shown travelling on the leftmost chain in the figure. If more than 2 chains are involved, a distributor is appended to the chain carrying the boolean decider so that it may be routed to all chains. The conditional alternator directs execution to the appropriate SP based on the decider token. It is formed by prefixing a BC instruction to each of the input chains of one of the SP's -- call it SP1. The output of the distributor is used by the BC's to either continue execution with SP1 or branch to the corresponding 1st instruction on the chains of SP2, the alternate SP. After completing the code in SP1, each of the chains executes an unconditional branch to the exit point of that chain from the conditional -- shown in the diagram as J2. In other words, each chain executes in the normal fashion along its length except that one of 2 sections it contains is skipped -- as dictated by the boolean decider.

At the exit of the conditional each chain executes a special instruction called a *conditional return ACK** (CRA*), the purpose of which is to insure determinacy of the outputs of the conditional. It achieves this by returning an ACK* to only one of its two predecessors at a time, that is, either to the last instruction in SP1 or the last instruction in SP2 on any given chain. The destination of this ACK* is

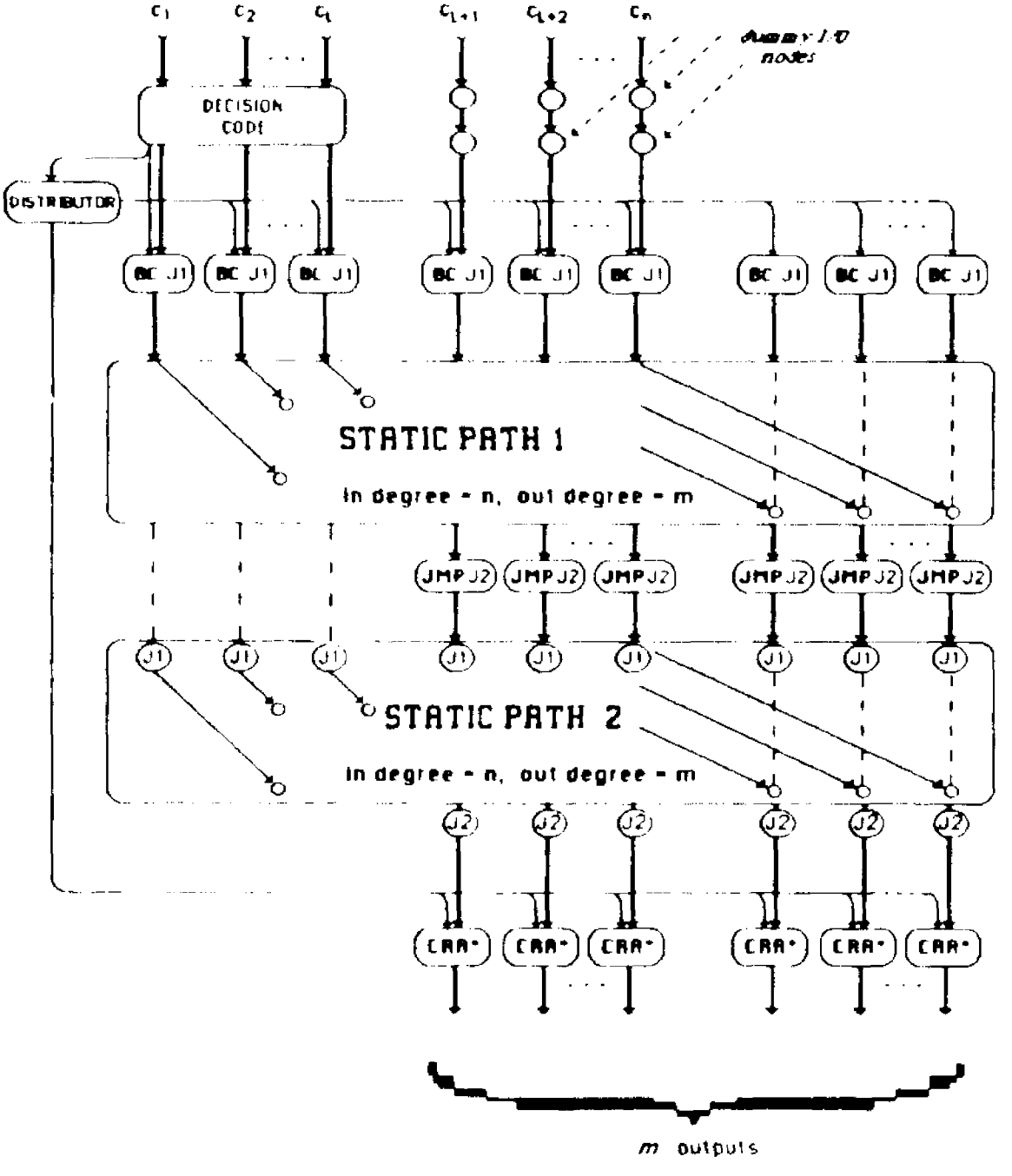


Fig. 4.1 The SDF Conditional

determined by a control input to the CRA*, the same input which feeds the BC on that chain The operation of the CRA* is defined carefully below

Since each of the alternates in the conditional is a static path, multiple instantiations may follow one another through their successive operations In principle then we should be able to allow multiple instantiations of the conditional concurrently However, the CRA* instructions must return an ACK to their distributor Hence as soon as the first control token is processed by the distributor no further operand sets can get past the BC s until the that first instantiation exits the conditional The net effect is that only one instantiation of the conditional at a time will be active To alleviate this bottleneck we shall buffer the control token traffic between the distributor and the CRA*s In an actual machine we would expect the buffer to be as large as the number of instantiations which can be concurrently active in the larger of the two static paths -- as limited by the ACK* protocol This approach is related to that investigated in [10]

Every output of the conditional must arbitrate between two corresponding outputs of the static paths via a CRA* Since the number of outputs may be greater then the number of inputs, there may exist chains in the SP s which provide output but do not take input from one of the conditional's n inputs Instead, their input MW is provided by some other chain within the SP s To unify our treatment we shall artificially transform chains which provide output into ones which provide input as well We accomplish this by prefixing a dummy BC instruction to all such chains (see figure 4 4) The only input to these BC s is the boolean decider output

of the distributor A special op-code designation tells the processor to initiate an instantiation of the chain with a *delayed context* In other words, the processor will fetch and execute the BC, whose effect is simply to choose the next instruction to be fetched, *without the presence of a CM* The context registers of the processor will not be set until the next instruction, when the arrival of the MW will establish the instantiation's context The activation of a new chain which would normally result from the arrival of the MW to the latter instruction, is suppressed -- the activation has already been invoked by the arrival of the decider token at the BC The rightmost group of chains in figure 4 4 are shown as obtaining their MW context from somewhere inside the static paths SP1 and SP2

Note that there may also be chains which take their MW from one of the conditional's *n* inputs but do not provide an output of the conditional Instead they provide one or more 2W operands to other chains within SP1 and/or SP2 and then terminate The leftmost chains in figure 4 4 are shown terminating within the conditional after delivering 2W operands somewhere inside SP1/SP2

We now proceed formally

Definition: The *conditional return ACK'* instruction is defined as follows

1) The format of the CRA* instruction is

field	function
a) MW/MB/CM	context save area
b) 2B	control input field
c) Bit Buffer (BB)	FIFO buffer of control input
d) DEST A	address to send ACK for control input
e) DEST1 A*	address of one possible predecessor
f) DEST2 A*	address of alternate possible predecessor
g) RSTRT	bit indicating instruction needs restarting
h) ACK*	field for ACK* from successor

Items (a), (d) and (h) have the usual meaning. (a) is required in case an ACK* is not forthcoming from the CRA*'s successor and the context must be saved. The ACK in (d) is the usual one for COM arcs. However, as will be seen below, the CRA* becomes active and executes some function whenever a control input arrives, even if the CM is not set, i.e. the chain on which it occurs is not executable. In particular, the ACK for the control input is returned immediately upon receipt in most cases even if the CRA* is not being fetched by a processor executing its chain. Items (e) and (f) are the two predecessor paths between which the control input must choose; it is essentially through the return of these ACK*'s that the CRA* achieves its function. The function of the RSTRT bit will be made clear below.

The initial state of the CRA* includes an ACK* and RSTRT bit. Further, in

contrast to the usual convention, the two predecessors of the CRA* *do not* have ACK* as part of their initial condition

2) The following two procedures constitute the basic action of the CRA* instruction. The first is invoked in response to the arrival of a control token, the second represents the action of the CRA* proper in the course of execution of its chain

```
Procedure PROCESS_CONTROL_TOKEN (CTRL_TKN)
  Begin
  If RSTRT Then Begin
                send ACK* to DEST1 or DEST2 based on CTRL_TKN
                reset RSTRT
                End

          Else  shift 2B into BB

  If BB not full Then return ACK to DEST_A
  End       // end of PROCESS_CONTROL_TOKEN//
```

```
Procedure EXEC_CRA*.

  Begin
  If BB not empty Then  Begin
                        send ACK* to DEST1 or DEST2 based on next BB
                        update BB
                        If BB was full Then send ACK to DEST_A
                        End

                  Else  set RSTRT

  End       // end of EXEC_CRA*//
```

The action of the CRA* instruction may now be defined as follows

input	action
2B	PROCESS_CONTROL_TOKEN(2B)
MW (chain exec)	If not ACK* Then Begin save context in MW/MB fields set CM release processor End Else EXEC_CRA*.
ACK*	If CM Then Begin EXEC_CRA*. reset CM End

The procedures PROCESS_CONTROL_TOKEN and EXEC_CRA* are a producer and consumer of the bounded buffer BB, hence provision must be made to insure that their actions are mutually exclusive in time

3) Both inputs of the CRA* instruction are defined to be SEQ arcs. This designation is consistent in the sense that we still have only one SEQ input permitted to any node because the inputs of the CRA* are used in a mutually exclusive way.

4) The CRA* is defined to be fireable when an input triple $(\partial_c, \partial_0, \partial_1)$ is presented to its control '0' and '1' inputs respectively, where '0' and '1' denote the alternate data inputs. In addition to the usual type restrictions which hold for data lines in the static path, we permit the token nil on either but not both of the inputs '0' or '1'. The token nil is a formalization of the notion that no input is also an 'input', i.e. that one or the other of the data inputs is not required in order for the CRA* to be fireable. Specifically, $(\partial_c, \partial_0, \partial_1) \in \{(F, MW, nil), (T, nil, MW)\}$. The notation $\sim(j)$ will be used to designate the complement of the jth data input line to a CRA*, that is input number '0' if j is '1', and input number '1' if j is '0'.

Note that (3) applies to the BC instruction as well. For a given instantiation, its two outputs are mutually exclusive, hence the designation of its output arcs as being of the SEQ type is appropriate.

Definition: A *conditional boolean generator* (CBG) is an SDF static path with equal in-degree and out-degree, at least one of whose outputs includes a boolean

value.

A *conditional alternator* (CA) consists of a pair of SDF static paths SP1 and SP2, which are combined as follows

1) All chains in SP1 are prefixed a BC instruction which jumps to the first instruction of the corresponding chain in SP2 The chain which includes the instruction which generates the decider token will be prefixed a BC type (1) The remaining chains will receive a BC type (2) (see table 4.2) Chains which are activated inside their static path have the chain activation function of their first node supressed This function is transferred to a BC which is prefixed to the first instruction the BC activates the chain in *delayed content* mode

2) All chains in SP2 have a CRA* appended after their last instruction The ACK*'s of these instructions are directed to their predecessors in SP2 and the last instruction on the corresponding chain in SP1

3) All chains in SP1 have an unconditional JMP appended to their last instruction The JMP is directed to the corresponding CRA* inserted in (2) above

The CA is said to have in-degree n and out-degree m

An SDF *conditional* consists of a CBG with L inputs and outputs and a CA with

in-degree n and out-degree m such that $L \leq n$ L of the n inputs to the CA are provided by the CBG The output of the CBG carrying the boolean token is routed to the control input of all BC instructions in the CA via a distributor, as well as to the control input of all the CRA* s The remaining $n - L$ inputs to the CA come from outside the conditional The conditional is said to have in-degree n and out-degree m

We must now show that the conditional represents a determinate computation As in the previous section, our approach will be to show how the conditional may be represented within the Adams model In the context of dynamic constructs such as the conditional however the issues of liveness and reusability become significant Once the BC and CRA* instructions are introduced Adams' model does not --by itself -- guarantee either property Even if the type values of connected inputs and outputs are matched, deadlocks and hang-up states are possible It is easy to see that even a single input set could cause a computation to hang up for example if the MW arriving at a CRA* is presented at the input other than the one dictated by the control token For SDF we will permit the use of the BC and CRA* instructions only in the conditional construct as defined above We will show that when so restricted the properties of determinacy and liveness hold The matter of reusability will not be directly addressed

Definition: A sequence of data tokens presented to the three inputs of a CRA* instruction is said to have a *consistent description* $[A^1, A^2, \ldots, A^j, \ldots]$, if $A^j \in \{(F, MW, nil), (T, nil, MW,)\}$, for all j Note that it is the introduction of

implemented by the CBG for a single input set X. That is, $f\emptyset(x_1, x_2, \ldots, x_n) = (y_1, y_2, \ldots, y_n)$ Let the name CBG itself denote the mapping of a single input set X_L to True or False as implemented by its decision code -- that is, $CBG(X_L) = D$, where D represents the value of the decider token. Let F∅ denote the mapping of input *sequences* to output sequences implemented by the CBG as a whole. Thus $F\emptyset[X^1, X^2, \ldots] = [(Y^1, D^1), (Y^2, D^2) \ldots]$ Similarly let F1 represent the mapping of input sequences to output sequences implemented by the conditional alternator Thus $F1[(Y^1, D^1), (Y^2, D^2), \ldots] = [Z^1, Z^2, \ldots]$ where D is the decider variable Note that the chain which generates D also participates in X, the situation implies simply that both an MW and MB are active at the same time on that chain

By Lemmas 4 5 and 4 4 we can infer that

(4 6) $$F\emptyset[X^1, X^2, \ldots] = [(f\emptyset(X^1), CBG(X^1_L)), (f\emptyset(X^2), CBG(X^2_L)), \ldots]$$

and hence that the CBG is both determinate and live

Let f1 denote the mapping of inputs to outputs implemented by SP1 for a *single* input set and similarly f2 for SP2 Define a function valued function $\partial: X_L \mapsto (f1, f2)$ as follows

$$\partial(X_L) = f1 \text{ if } CBG(X_L) = \text{true}$$

and f2 otherwise

Thus we can write $\partial(\bar{X}^i_L)(Y^i) = Z^i$ We will use $\partial_j(X_L)$ to denote that part of $\partial(X_L)$ which produces the output z_j

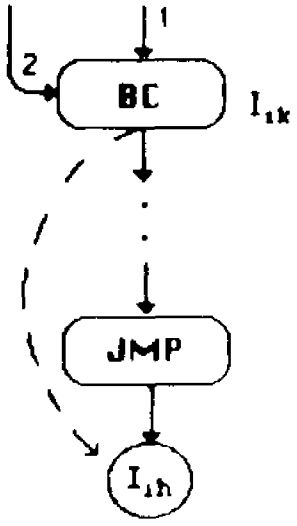
Lemma 4.6: For any execution of an SDF conditional there exists a description of the input sequences to all CRA* s which is consistent

proof: By the definition of the conditional we have that each pair of chains which end in the SEQ inputs to a CRA* originates at a BC Instruction whose control token is also fed to the CRA* By Lemma 4 4 and the discussion above the CBG is determinate and live Thus for any valid input sequence to the conditional $[X^1, X^2, \ldots]$ we must have inputs $[(f\emptyset(X^1), CBG(X^1_L)), (f\emptyset(X^2), CBG(X^2_L)) \ldots]$ to the BC instructions Hence if a first control token arrives a BC then it must arrive at all BC's and further a first data input must have arrived at all BC's as well Conversely if a data input arrives at a BC such an input must have arrived at all BC's as will have control tokens By the definition of the BC instruction we have that a full input set will be presented to the static path corresponding to $\partial(X^1_L)$ Also by Lemma 4 4 the static paths SP1 and SP2 are determinate and live and hence $\partial_i(X^1_L)(Y^1)$ will arrive at input number $CBG(X^1_L)$ of the CRA* on chain c_i before any other input arrives on the same input arc Letting the

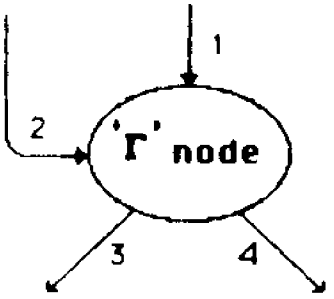
argument to input number $\sim(CBG(X^1_L))$ of the CRA* be nil, we have that the first inputs to any CRA* may be described $A^1 \in ((F, MW, nil), (T, nil, MW,))$ The lemma follows inductively by assuming it to hold for a description with j control inputs and applying exactly the same argument to show that there exists a description with $j+1$ control inputs which is also consistent □

Next we show how the BC and CRA* instructions can be represented as nodes in Adams' model For each instruction we construct an 'r' or 's' node of the type described by the model We then designate inputs and outputs for the SDF instruction in question, and show how the action of the instruction maps these inputs to outputs in accordance with the 'f' and 'g' functions associated with the corresponding 'r' or 's' nodes

For the BC instruction, this is relatively straightforward because the BC is defined as fireable only when a datum of appropriate type is presented at every input We can therefore model the BC as an 'r' node in Adams' schema Figure 4 5a shows an instruction I_{ik} which is a BC and which arbitrates between successors $I_{i,k+1}$ and I_{ih} As prescribed by Adams' model, the input arcs of 'r' nodes are unlocked by definition i e a datum must be presented on each of them in order for the node to be enabled We need merely extend the set of valid output tokens to include nil For the BC instruction, a nil output is equivalent to simply not instantiating the corresponding successor instruction We will define the 'inputs' to a BC instruction to be the MW_{ik} (data input) and $2B_{ik}$ (control input) The



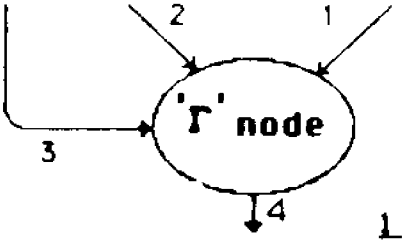
The BC Instruction



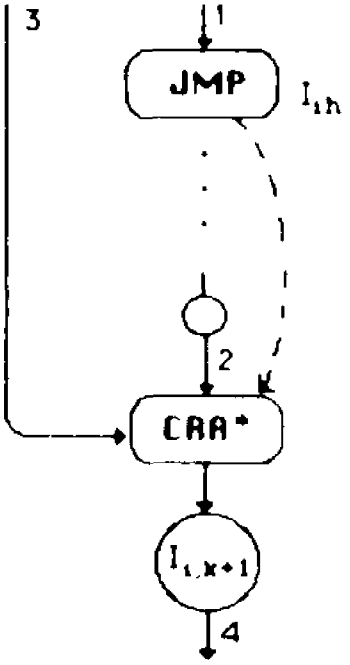
1 f(w, T) ---> (w, nil)

2 f(w, F) ---> (nil, w)

(a) (b) (c)



The CRA* Instruction



1 g (L, L, U, nil, nil, T) ---> (U, L, L)

f (nil, nil, T) ---> (nil)

2 g (L, L, U, nil, nil, F) ---> (L, U, L)

f (nil, nil, F) ---> (nil)

3 g (U, L, L, w, nil, nil) ---> (L, L, U)

f (w, nil, nil) ---> (w)

4 g (L, U, L, nil, w, nil) ---> (L, L, U)

f (nil, w, nil) ---> (w)

Fig 4.5 Modelling The BC and CRA*

(a) is the SDF construct, (b) the equivalent node in Adams' model and (c) gives the 'f' and g functions for the latter

output will come from the set (($MW_{i,k+1}$, nil), (nil, MW_{ih})) depending on the value of the control input $2B_{ik}$ With this definition of inputs and outputs, it is easy to verify that the action of the BC instruction corresponds to the definition of the 'f' function associated with the 'r' node shown in the figure

In figure 4 5b we show an 's' node which models the function of the CRA* instruction Since one or the other of the instructions inputs are not required for a given instantiation, we need to use the input 'locking' feature in Adams' model -- hence the use of an 's' node The 's' node is simply a controlled merge, its associated 'f' and 'g' functions are shown in the figure Control and data tokens are processed alternately First the control input is unlocked while both data inputs are locked The node fires upon receipt of a control token and the effect of this firing is to unlock one or the other of the data inputs and re-lock the control input the output is unaffected, i e has value nil When a datum arrives at the unlocked input the node fires by passing it unchanged to the output At the same time both data inputs are again locked and the control input unlocked, in preparation for the next cycle The corresponding SDF construct consists of a CRA* and a JMP instruction which preceeds one of its inputs -- this corresponds to the way the CRA* is used in the conditional In the figure I_{ik} is the CRA* while I_{ih} is the JMP which precedes one of its inputs We define the 'inputs' and 'output' for this construct to be $2B_{ik}$ (control input), MW_{ik} and MW_{ih} (alternate data inputs) and $MW_{i,k+1}$ (output) We will call the combined CRA* and JMP construct an *SDF merge*

Since we are employing an 's' node we must have definitions of 'locked' and 'unlocked' input arcs for the corresponding SDF construct We shall use the following

A SEQ arc input to an SDF merge is defined to be *locked* when its predecessor along that arc has no ACK*, and *unlocked* otherwise

Our definition of 'locked' is consistent with Adams' since if no ACK* is available in the predecessor instruction along a given input arc that instruction cannot execute and hence will present no MW to its successor Since the MW is how we defined data input to the SDF merge, we have that nil is always presented to the merge at an input which is 'locked In Adams' model an 's' node is understood to 'know' which inputs are locked and that data need be presented only at the unlocked ones in order for the node to fire In the SDF merge the same behavior follows from the fact that the instruction will only be instantiated by the execution of one of the chains which end in a data input arc Thus its predecessor along that arc must have had an ACK* and hence is unlocked by definition Only one MW context is used in the execution of the merge and therefore a word which might be presented at the other data input arc will be ignored In particular, if the latter is 'locked', the instruction remains fireable

For the control input we take

A COM arc input to an SDF merge is defined to be *locked* if any SEQ arcs input to the same are unlocked, and *unlocked* otherwise

In SDF we can easily make this definition work in a manner consistent with that of Adams model because of the restriction that instructions are instantiated only in the course of chain execution In other words, since the active mode is chain execution we make the state of COM arcs a slave to the events which are occasioned by the data on SEQ arcs Specifically, the default state of the COM input to an SDF merge is 'locked' No proper execution of the CRA* instruction will occur as a result of the arrival of a control token (The updating of the BB is a special housekeeping function associated with the CRA* but does not correspond to an instantiation of the instruction) In accordance with our definition, this means that at least one of the SEQ arc inputs of the merge are 'unlocked', i e its predecessor has an ACK* available and hence may be instantiated At such time that the CRA* is executed in the course of executing that chain, the ACK* will be consumed thus rendering the SEQ inputs 'locked' By our definition, the COM arc is now 'unlocked' However, as part of its execution sequence, the CRA* will fetch the next control token and send an ACK* to the appropriate SEQ input predecessor Thus in one execution cycle, the control input is consumed, its arc is again 'locked' and one of the SEQ inputs is 'unlocked' for the next instantiation For the 's' node with which we are modelling the SDF merge, this sequence is equivalent to the node functional transitions (for e g the case where input number '0' is initially unlocked)

(4.7) g (L, U, L, nil, w, nil) ---> (U, L, L, T/F, nil, nil) ---> (L, U/L, U/L,)

f (nil, w nil) ---> (w), (T/F, nil, nil) ---> (nil)

where 'w' represents a data word The first 'g' state corresponds to having an ACK* in one of the SEQ arc predecessors, hence that input is 'unlocked' while the control input is 'locked' The point where that ACK* is consumed but before the CRA* executes corresponds to the second 'g' state, the SEQ input arc has been 'locked' the control input is therefore 'unlocked' by definition and the next BB datum is considered to be presented at the latter input Finally, the last 'g' state parallels the result of executing the CRA*, one of the SEQ inputs has been 'unlocked' as a result of consuming a control token and the control input is now again, by definition, 'locked' The 'f' transition is simply the passing of the data word to the output in the course of execution of the CRA*, the second transition is a dummy state change which makes the definition of the 'f' function total but results in no output

The above scenario assumes that one of the SEQ inputs to the CRA* is initially unlocked If however an instantiation of the CRA* finds the BB empty, neither of the SEQ inputs will become unlocked Regardless of any control tokens which may subsequently arrive, the CRA* instruction will remain deadlocked because there are no ACK*'s available to its predecessors -- it will never execute To avoid this, the RSTRT bit will be set when executing the CRA* if no control tokens are available The RSTRT bit indicates that an exception must be made to the rule that only a processor executing the appropriate chain can instantiate the CRA*

Instead, when a first control token subsequently arrives, it will invoke a partial execution of the CRA*, that is, it will cause an ACK* to be returned to the appropriate SEQ input predecessor The RSTRT bit is also reset It is easy to verify that this sequence is precisely analogous to the 'f' and 'g' transitions (4 7)

We have illustrated how the action of the BC and CRA* instructions (as used in the SDF merge) can be modelled by an appropriate node in Adams' schema In order for the properties of Adams' model to hold however, propositions (1) and (2) from the previous section must apply to the use of these instructions in a computation That is they must be shown to exchange inputs and outputs with their neighbor nodes in the manner of FIFO queues (or equivalently, that they create outputs in the order that corresponding inputs were created) For the static path, this was demonstrated via Lemmas 4 2 - 4 4 We shall extend these results to the BC and CRA* instructions The determinacy characteristic of Adams' model will then follow By further restricting the use of the BC and CRA* to the SDF conditional we will be able to use Lemma 4 6 to show that the resulting SDF computation is also live

First we extend Lemma 4 3 to include the BC and CRA* instructions The contention of the lemma was that, for two adjacent instructions I_{ik} and I_{sl}

$$(4.8) \qquad F[(A^1_{ik}\ \partial^1),(A^2_{ik}\ \partial^2),\ \ldots\ ] = [f(A^1_{ik}\ \partial^1), f(A^2_{ik}\ \partial^2),\ \ldots\ ],$$

where the $(A^j_{ik}\ \partial^j)$ are valid input sets to the instruction pair for all j

Lemma 4.7: Let I_{jk} and I_{st} be two adjacent instructions in an SDF computation exactly as in Lemma 4.3 except that one of them may be a BC instruction or an SDF merge. Then (4.8) holds.

proof: If I_{st} is a BC instruction, the dynamic aspect of the BC does not effect the interaction between I_{jk} and I_{st} since the ACK* protocol applies as usual and both nodes fire only when data are presented at all inputs. Hence Lemma 4.3 holds directly. The same is true if I_{jk} is an SDF merge.

If I_{jk} is the BC, only SEQ arcs are permitted at its output and hence the connecting arc between the two instructions must be SEQ. We now augment the set of valid inputs to and outputs from I_{st} to include nil. Thus if the BC is instantiated such that its 'other' successor is chosen, I_{st} will be defined to consume a set of nil inputs and produce a nil output. With this convention it is clear that we can always have $B_{st} = f(A_{jk}, \delta)$, i.e. the configuration I_{jk} and I_{st} constitutes a determinate computation on a single set of inputs.

If I_{st} is an SDF merge, the alternate input δ to I_{st} is actually a pair $(\delta_0, \delta_{\sim(b)})$, where b is the number of the merge data input to which I_{jk} is connected. For a

single input set $(A, (\partial_0, \partial_{\neg(b)}))$ to the instruction pair embedded in the conditional, either $A = nil$ or $\partial_{\neg(b)} = nil$ and further, the control token value will match the non-nil input number (T=1, F=0), again by Lemma 4.6. Hence I_{st} will eventually fire and we have that $f_{st}(f_{ik}(A_{ik}), (\partial_0, \partial_{\neg(b)})) = B_{st}$ or $B_{st} = f(A_{ik}, \partial)$. The same argument applies to the case where I_{ik} and I_{st} are connected via a COM arc by setting $\partial = (\partial_0, \partial_1)$.

Now we consider the two cases -- I_{ik} is a BC or I_{st} is an SDF merge -- in turn, and prove first that (4.8) holds for the first two inputs A^1_{ik} and A^2_{ik} to the instruction pair I_{ik} and I_{st}.

<u>case 1)</u> I_{ik} is a BC instruction. Note first that since ∂ is only a place holder, we are only concerned to show the correspondence of B^j_{st} to A^j_{ik}. Thus if the 'other' successor is chosen on A^1_{ik} but I_{st} is chosen on A^2_{ik}, the correspondence holds vacuously, we simply define an instantiation of I_{st} with nil inputs and outputs to occur before $f(A^2_{ik})$ is produced. Similarly, if I_{st} is chosen for A^1_{ik} but not for A^2_{ik}, we define the dummy event as occuring after $f(A^1_{ik})$ is produced. And of course we can have [nil, nil, ...] as the output sequence if I_{st} is not chosen for either

input For the remaining case, i.e. where the successor I_{st} of the BC is instantiated for both inputs note that while the BC instruction has the option of instantiating either successor, it is nevertheless true that it cannot itself be executed until at least one of them returns an ACK* (except the first instantiation) Therefore, if I_{st} is instantiated on both first inputs Lemma 4 2 applies whence Lemma 4 3 applies verbatim

case 2) I_{st} is an SDF merge Here we have two subcases depending on whether the connecting arc is SEQ or COM

case 2a: The connecting arc is SEQ By assumption the inputs $(A_{ik}, (\partial_c, \partial_{\neg(b)}))$ are valid hence one of the following four descriptions of the first two inputs $(A^1_{ik}, (\partial^1_c, \partial^1_{\neg(b)}))$ and $(A^2_{ik}, (\partial^1_c, \partial^2_{\neg(b)}))$ must hold

a) $A^1_{ik} = A^2_{ik} = nil$ $\partial^1_{\neg(b)} = \partial^2_{\neg(b)} = non\text{-}nil$ and $\partial^1_c = \partial^2_c = \neg(b)$

b) $A^1_{ik} = nil$, $A^2_{ik} = non\text{-}nil$, $\partial^1_{\neg(b)} = non\text{-}nil$, $\partial^2_{\neg(b)} = nil$, $\partial^1_c = \neg(b)$ and $\partial^2_c = b$

c) A^1_{ik} = non-nil, A^2_{ik} = nil, $\partial^1_{\neg(b)}$ = nil, $\partial^2_{\neg(b)}$ = non-nil, $\partial^1_o = b$, and $\partial^2_c = \neg(b)$

d) $A^1_{ik} = A^2_{ik}$ = non-nil, $\partial^1_{\neg(b)} = \partial^2_{\neg(b)}$ = nil and $\partial^1_o = \partial^2_o = b$

As in case (1), we are only concerned to show the correspondence of B^j_{st} to A^j_{ik}. For (a) through (c) therefore, the correspondence holds vacuously since we can insert nil values for A^j_{ik} in appropriate positions so that a non- nil A_{ik} input, where present, will correspond to the correct B_{st}. For (d), we again note that while the SDF merge has a choice as to which predecessor an ACK* will be returned, it is nevertheless true that no predecessor can fire until such an ACK* is available. Thus the proof of Lemma 4.2 applies to I_{ik} and I_{st} and it follows that I^2_{ik} will not be enabled until I^1_{st} completes. Thus B^2_{st} must follow B^1_{st} and hence (4.8) holds for (d) as well.

case 2b: The connecting arc is COM. No nil tokens travel on the COM arc, a datum is always generated by I_{ik} and is always consumed by I_{st}.

that the ACK* protocol between them is different. However, this protocol is precisely a FIFO queue discipline and therefore Adams model applies directly, there is no need to prove (4.8)

The lemma now follows inductively by applying exactly the same reasoning, for each of the cases (1) and (2a) to the $j+1$st input and output, assuming it holds for j inputs and outputs □

Not that it was not necessary to limit the use of the BC and SDF merge to the conditional. The lemma only tells however that if valid inputs are assumed, and if outputs are generated, then those outputs will be as indicated in (4.8). For the next result however this restriction to the conditional is required

Lemma 4.9: An SDF conditional is determinate and live

proof: The proof of Lemma 4.4 applies directly to any configuration of adjacent instructions I_{it} and I_{st} for which (4.8) is true. Hence the equivalence to a FIFO discipline holds for the input/output relationship between all nodes in a conditional. Since the instructions themselves have been shown to be representable as 'r' or 's' nodes within Adams' model the properties of the model apply -- whence the conditional is determinate. Note that we still have not made use of the restriction to the conditional construct

To show liveness, consider any instruction pair I_{ik} and I_{st} where I_{st} is an SDF merge in a conditional and the arc connecting them is SEQ. By Lemma 4.6 the input sets to I_{st} have a consistent description. Thus the merge will generate a valid output for each valid input. Further the sequence of inputs $(A^j_{ik}, (\partial^j_0, \partial^j_{\neg(b)}))$ for all j, to the configuration I_{ik} and I_{st} must be valid. The inputs $(\partial^j_c, \partial^j_{\neg(b)})$ are themselves inputs to the CRA* and A_{ik} must also have consistently received valid inputs else it could not have provided the same to input number b of the CRA*. By Lemma 4.7 the output of the conditional arbitrated by this particular SDF merge must be $[f(A^1_{ik}, \partial^1), f(A^2_{ik}, \partial^2), \ldots]$, i.e. the computation will make progress will make progress and generate outputs in the order of corresponding inputs for every input consumed. Since I_{st} can be any SDF merge at the output of the conditional it follows that the conditional as a whole is live. □

5.4. The Loop

The iteration construct which we will develop for SDF is patterned after the basic

WHILE (condition) DO BEGIN (loop body) END

programming structure. The parallel version of this logic has a data flow implementation which was presented earlier in figure 4.2. The viability of this construct, that is, its determinacy and liveness, can be demonstrated in relatively straightforward manner, considerably more straightforward than the development in the previous section which was required for the conditional. This is because the loop, unfortunately, is by far more severely constrained than the conditional with respect to the parallelism that it permits, hence there is less to prove. Specifically, the loop can only be instantiated for one set of inputs at a time. Parallelism is permitted within the loop, both in the 'condition' part and the 'loop body', between operations that are not directly dependent on one another. However, a second set of inputs cannot be permitted to instantiate the loop until a previous one has completely terminated, otherwise there would be no way to distinguish operands from the different instantiations. This problem plagues all 'static' dataflow interpretations as well. The 'colouring' of different instantiations is the term used in the literature for some technique whereby operations will be constrained only to sets of operands all of which are associated with the same iteration number of the same instantiation. The most developed of these techniques is the Unraveling Interpreter of Arvind and Gostelow. We shall briefly examine the relevance of this 'dynamic' colouring technique to SDF in a

subsequent section For the programming system being developed here however, we shall remain with the 'static' approach and our loop will permit only single instantiation at a time

The general SDF loop construct is shown in figure 4.6 The decision code is a static path which implements a boolean valued function of some subset of the n inputs to the loop Each chain in the loop body is prefixed by a special version of the BC instruction which we will call the *Loop Branch on Condition* (LBC), its operation will be defined carefully below The LBC is controlled by the decider token generated by the decision static path As for the conditional, if there are more than two chains in the loop body, a distributor will route the decider token output to the instructions on each chain The LBC will cause its own executing processor to instantiate either a) that part of its chain which is within the loop body, that is, continue with the instruction immediately following the LBC or b) the first instruction on its chain after the loop In effect, it will cause the loop to either go around again or terminate Each chain segment included in the loop body ends with an unconditional JMP to the first instruction on that chain which is within the loop If the data word which is the MW for the chain does not participate in the decision as to whether or not to continue the loop, the JMP will return to the LBC instruction on the chain If it is required for the decision, the JMP will be directed to the first instruction on that chain in the decision static path Thus the loop condition is always tested first and, if found not to hold, will send a 'False' boolean token to the LBC instructions The executing processor for each chain will skip the loop body segment and continue with the next instruction on the

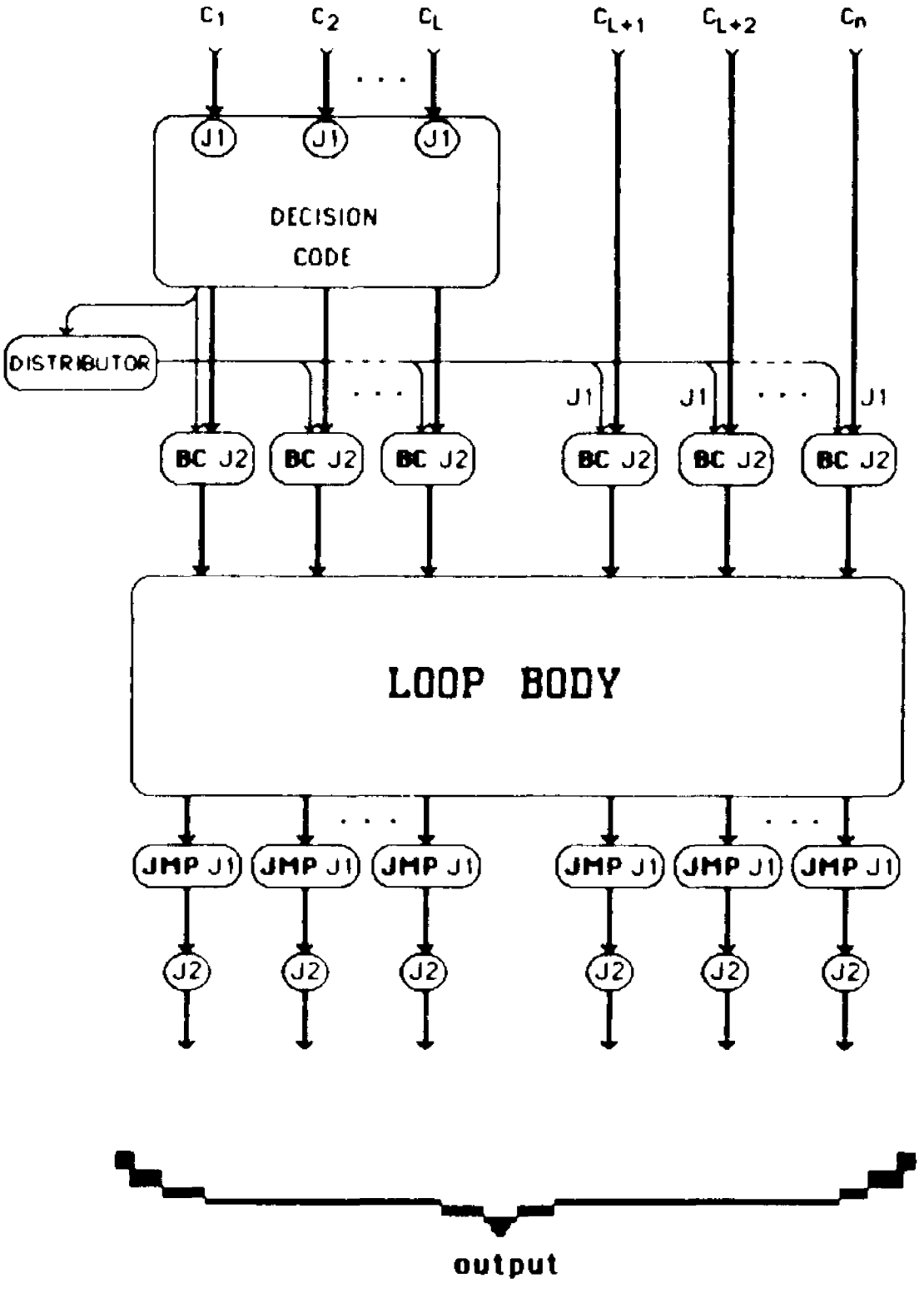


Fig. 4.6 The SDF Loop

chain Note that the loop body must have an identical number of inputs and outputs, in contrast to the conditional which does not share this constraint This reflects the nature of any iteration construct, in the sense that every value participating in the loop is being updated in some way at each iteration and hence must be both input and output to the loop body An exception to this is a constant which might be employed by the computation in the loop body Since the operators in most data flow models are 'memoryless' the generally accepted approach for dealing with constants in iteration constructs is to force them to circulate so that they are in effect presented anew at each iteration In practice, a deviation from the strict functionalism of data flow is permitted for the sake of efficiency the constants are simply made a read-only field in the instructions which use them This technique is also applicable to SDF

To insure that only one instantiation of the loop is active at a time, we suspend the ACK* protocol on all chains within the loop From an implementation perspective, this means that every instruction on a chain beginning with the first one included in the decision static path, through the last instruction before the unconditional JMP, will have no ACK* field and no DEST A* field If a chain does not participate in the loop test, the first instruction in which the ACK* protocol is suspended is the one immediately following the LBC The LBC itself can be thought of as the arbitrator for the loop to the code around it As long as the loop condition tests positive and iterations are continuing, the LBC is simply another instruction within the loop body, and hence the ACK* protocol does not apply When the loop eventually tests negative, it is the responsibility of the LBC to implement the ACK* protocol for the loop as a whole vis a vis the surrounding

code. That is, it must a) wait for an ACK* from the instruction on its chain immediately following the JMP, in case the latter instruction has been held up and is holding the saved context of a previous instantiation, and b) return an ACK* to the instruction on its chain immediately preceding the first one within the loop, thus permitting a next instantiation of the loop. Thus the LBC on a chain which does not participate in the decision code, will return an ACK* -- when the loop terminates -- to its immediate predecessor. The LBC on a chain which passes through the decision static path will return the ACK* to the instruction immediately preceeding the first one in the decision code. In all cases the LBC will wait for an ACK* -- again, only when the loop terminates -- from the instruction on its chain immediately following the unconditional JMP. Thus the LBC will require the MW/MB/CM fields to save the instantiation context, in case the ACK* is not present when exiting the loop.

Definition: The LBC instruction is defined as follows.

1) The format of the LBC instruction is the same as the BC type (1) and type (2). Type (1) is driven by the boolean datum which is generated by the immediately preceeding instruction on its chain, it is found in the MB field. No 2B field is required for type (1). Type (2) uses a 2B field rather than an MB field and waits for the control token to arrive from another chain.

2) The action of each of the two type of LBC is defined by a procedure For type (1)

```
Procedure LOOPTEST1 (CNTRL_TKN)
        Begin
        If CTRL_TKN = FALSE
           Then If NOT ACK*
                   Then  Begin
                         save context in MW/MB fields
                         set CM, release processor
                         End
                    Else  Goto  BRANCH ADDRESS
        End
```

and the procedure representing the action of type (2) is

```
Procedure LOOPTEST2 (CNTRL_TKN).
        Begin
        If NOT 2B
                Then Begin
                        save context in MW field
                        set CM release processor
                        End
                Else If CTRL_TKN = FALSE
                        Then If NOT ACK*
                                Then    Begin
                                        save context in MW field
                                        set CM release processor
                                        End
                                Else    Goto BRANCH ADDRESS
        End
```

We can now define the action of the LBC instructions -- for each type of arriving operand -- as follows For the LBC type (1) we have

input	action
MW (chain exec)	LOOPTEST1(MB).
ACK*	If CM Then LOOPTEST1(MB)

and for the LBC type (2)

input	action
MW (chain exec)	LOOPTEST2(2B)
2B	If CM Then LOOPTEST2(2B)
ACK*	If (CM AND 2B) Then LOOPTEST2(2B)

The type (1) LBC will be used on the chain which also contains the instruction which actually generates the boolean decision token -- shown in figure 4.6 as the leftmost chain. All the remaining chains will employ a type (2) LBC.

Lemma 4.9: The SDF Loop is determinate and live.

proof: If a first set of inputs has been received by a chain c_i of the loop, this implies that the initial ACK* in the predecessor instruction of the loop on c_i has been consumed and therefore that that predecessor cannot be re-instantiated until its ACK* is replenished. By the definition of the SDF Loop, none of the instructions in the loop return an ACK* to any predecessor except the LBC and then only when it receives a False control token. Hence no second set of inputs will be permitted to instantiate the first instruction of c_i within the loop, until

the first instantiation finds the loop condition false and thereby results in the LBC returning an ACK* to the loop's predecessor on c_i. Thus we can assume that as long as the first instantiation is executing within the loop on c_i, no second instantiation of the loop on c_i will be invoked. Note however, that this does not imply that only one instantiation at a time will be executing anywhere in the loop as a whole. Chain c_i may complete execution within the loop for a particular instantiation while chain c_j may still be active for the same instantiation. Chain c_i will permit a next instantiation to begin even while c_j is still active on the previous one.

If in fact it were true that only one instantiation of the loop *as a whole* could be active at once, we could 'unravel' the iterations of the loop into an equivalent SDF program comprising a series of sections each consisting of the decision code followed by a distributor and LBC instructions followed by a copy of the Loop Body (see figure 4.7). By Lemma 4.5, the decision code can be taken to be an SDF static path of in and out-degree n. As long as the control token arriving at the LBC is 'True', it behaves exactly like an ordinary BC instruction except that no ACK* protocol applies. From the previous section we know that the BC can be represented as an 'r' node in Adams' model. Further, since we are assuming that only one set of inputs is being processed within the entire loop instantiation at a time, the issue of FIFO queues as the discipline for input/output between adjacent nodes is irrelevant. The fact that the LBC can be modelled as an 'r' node is sufficient to legitimately employ it in a graph computation, and the properties of

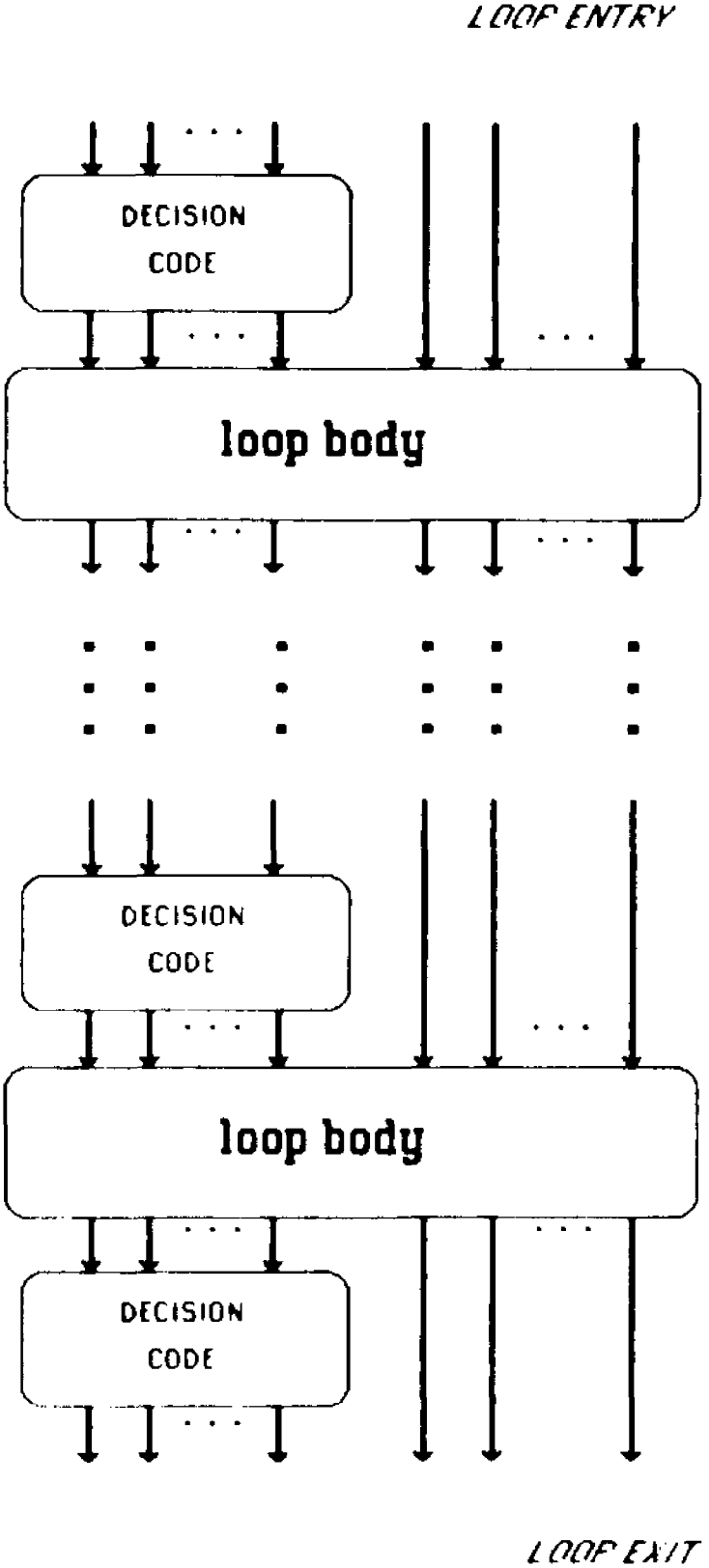


Fig 4.7 'Unravelling' of an SDL Loop

path to which only a single set of inputs at a time is being applied By Lemma 4.1, the computation is determinate It is also live, as has been shown in general for static graph modelled computations which are well formed in the sense of matched inputs and outputs of adjacent nodes (see previous section)

Of course, as we have seen above, it is not necessarily the case that only one instantiation at a time will be active in the Loop as a whole only that a single instantiation will be active in each chain at a time Nevertheless there will be some chain which will complete the first instantiation of the loop before all others -- call it c_{t1} and its time of completion t_1 Until time t_1, only one instantiation is active in the entire loop because every chain is occupied with that instantiation Between t_1 and t_2, all chains but c_{t1} are still occupied with the first instantiation Further, c_{t1} must have deposited any 2W or 2B operands in nodes onother chains which it supplies before terminating Therefore, c_{t1} bears no relationship to the first instantiation of the Loop after time t_2 Similarly, after t_2 the chains c_{t1} and c_{t2} are irrelevant to the remaining computation in the first instantiation of the loop Thus we can 'prune' the static path dynamically More precisely, with the completion time t_i of each chain c_{ti}, we associate a subSP S_i of the sequence shown in figure 4 7 consisting of all chains except $c_{t1}, c_{t2}, \ldots, c_{ti}$ For $i < j$, we have that S_j is a sub SP of S_i and further, at time t_i, no part of S_i which is not in S_j can contain any tokens destined for instructions in S_j since by

time t_j the chains $c_{t1}, c_{t2}, \ldots c_{ti}$ have completed all execution relevant to this instantiation It follows that all remaining activity associated with the first intantiation at time t_j will be constrained to S_j By induction, there will always exist some subSP of the Loop Body to which all activity of the first instantiation is constrained and within which only that instantiation of the loop is active -- until every chain has terminated the instantiation By Lemma 4 1, this subSP also represents a determinate function of a single set of inputs and as before, is also live

If we now assume that the loop is determinate and live for j successive sets of inputs the same argument will give us that determinate outputs will be generated for $j+1$ successive inputs and the lemma will follow by induction □

6. The SDF Programming System

We now have three basic building blocks with which to construct computation schemas in SDF the static path the conditional and the loop Each will yield a determinate live output sequence for any valid input sequence It is clear from the foregoing that we can join any pair of valid constructs *a* and *b* together by simply connecting the outputs of *a* to the inputs of *b* Since the outputs of *a* are determinate and live *b* will be receiving valid inputs hence the outputs of *b* will also be determinate and live Clearly this process can be repeated indefinitely and arbitrarily complex programs may be constructed We state this result as a theorem

Theorem 4.1: Any computational schema consisting of SDF static paths conditionals and loops where each pair of constructs is joined by connecting some subset of the outputs of one to some subset of the inputs of the other represents a computation which is determinate and live

It is important to note however that while each static path within an arbitrary SDF schema will be optimized in the sense of Algorithm 3 1 (because we have so defined it) it will nevertheless be true that in general the resulting partition of the schema as a whole will be suboptimal This is simply because the outputs of a static path are not necessarily emitted at the same time and hence another static path which it feeds may not be supplied all of its inputs at the same time We shall briefly examine this and other problems in the next section

Conclusion and Further Research

1. Problems with the SDF Model

The main difficulty with the SDF computation model is that of finding an optimal partition Although Algorithm 3 1 yields an optimal partition for certain restricted cases much work remains before a computationally practical method of partitioning can be said to have been achieved Firstly, the algorithm is applied globally to a single graph As mentioned in the previous section, if several such optimized SDF schemata are joined to form a larger whole, the resultant schema may be suboptimal In practice, it would be prohibitive to apply Algorithm 3 1 globally to large programs What is needed is a 'modular' algorithm for partitioning one which can be used to join a static path b to a static path a which has already been optimally partitioned in such a way that it is not necessary to recalculate the entire partition Even with such a divide and conquer technique, our programming system has a serious drawback Since the body of the loop as well as the alternates of the conditional are defined as static paths, there is no provision for nesting these constructs Note however, that this restriction was imposed only to allow the optimal partitioning of the static paths The problem is that once nesting is permitted, the length of any chain -- in terms of execution time -- becomes a function of dynamic program behavior A compilation based partitioning scheme therefore cannot be effective, it only sees the static structure of the computation If we did not insist on the particular partitioning

effected by Algorithm 3 1 the properties of determinacy and liveness would still hold We could then define a 'well formed' SDF program recursively, as consisting of 1) a static path 2) a conditional or 3) a loop -- as before However, the conditional's alternates could now be defined as consisting of either a) a static path -- as before, or b) a 'well formed' SDF program Similary, the loop would remain as we have defined it but the loop body would also be either a static path or a 'well formed' SDF program Arbitrary nesting would thus be facilitated This structuring method is similar to that used in Rumbaugh's data flow system [31]

To optimize the partitioning into chains of program segments whose delay varies dynamically we would require a partitioning mechanism which is also dynamically variable To illustrate consider two paths A and B which converge at an operator α, where the relative delays of A and B vary dynamically A processor executing a predecessor of α would decide to continue on to α only if its companion path was ahead of it in making progress to α The latter condition insures that the processor can continue toward α knowing that the companion path will have already supplied the necessary second operand for α by the time it gets there If the condition does not hold, the processor will deposit its result at α via a COM arc and will terminate its current path, yielding to the processor progressing down the companion path for the actual execution of α The question of whether or not such information can be obtained in a sufficiently localized manner for a sufficiently general class of programming situations would represent an ambitious research undertaking

The delay of COM arcs themselves has also been ignored in the definition of

optimality used in Theorem 3.1. This cost will be increasingly significant as larger numbers of processors are contemplated. At a minimum, some kind of heuristic for incorporating this cost into the optimization technique would be required for a practical system.

2. Further development of SDF.

The author envisions two particular lines of development which could substantially enhance the effectiveness of SDF.

The first involves a dynamic version of the execution discipline and correspondingly, an architecture which employs tagging to effectively coordinate the movement of tokens and the invocation of operators. This general approach is found in e.g. [1]. Arbitrary numbers of instantiations of a given operator are permitted concurrently. The ACK protocol between successive operators along a path is eliminated. Instead, each instantiation, called an activity, appends a unique tag to the result tokens it generates. The tag contains a static as well as a dynamic component. The static component specifies the address of the instruction and the operand within that instruction for which the token is destined. The dynamic component specifies a recursive context and an iteration number if a loop is in progress. The machine which implements such a computation must enforce the rule that only tokens with matching tags may be used to invoke a given activity. In practice, this is done with some form of associative store. Tokens from different activities may be generated in any order

and are deposited in the associative store Whenever two tokens with matching tags are detected by the store's controller, the corresponding activity is invoked

In SDF, one could logically associate a distinct associative store with each computation path This permits significant locality of matching activity, the store could be much smaller and the search time correspondingly shorter than in the usual dynamic machine In addition the absence of ACKnowledgement token traffic would likely render even more significant the advantage of sequential execution of computation paths employed in SDF

The second direction of further development relates to data structuring -- an important topic which has not been addressed in this thesis

In conventional computation systems whose semantics include a notion of memory, fast access to e g arrays is possible because one can easily translate an index into a memory location While the semantics of many functional languages includes parallel data structure operations (e g the FORALL), the implementation of these operations is not nearly so straightforward because only values are permitted as the objects in a computation, no memory locations may be updated Such operations are generally viewed at the implementation level in terms of specialized operators -- 'black boxes' -- which take the entire array as one of their operands and either append or extract an element -- yielding a logically distinct new version of the structure

With SDF one could conceivably take a group of computation paths, each of which

relates to what would normally be considered a single array element, and consider their aggregate as a structure. For example, an array of data to which is applied a series of vector operations can be viewed as a set of parallel computation paths, each consisting of the same operations but relating to a different datum. The thrust of this approach would be to retain the lowest level of data flow intact, i.e. only primitive tokens are permitted to flow on arcs, structures never appear at this level. Data/path structuring is a higher level of abstraction superimposed on the basic data flow level. Computation paths which are grouped together in this way may be permitted to execute with a certain degree of synchrony

References

[1] Ackerman, W. B., *A Structure Memory for Dataflow Computers*, M.S. thesis, MIT Department of Computer Science, Report No. LCS/TR-186, Cambridge, Mass., August 1977.

[2] Ackerman, W. B. and J. B. Dennis, "*VAL -- A Value Oriented Algorithmic Language Preliminary Reference Manual*," MIT Laboratory for Computer Science Technical Report TR-218, MIT, Cambridge, Mass., June 1979.

[3] Adams, D. A., *A Computational Model with Data Flow Sequencing*, Technical Report CS117, Computer Science Dept., Stanford University, Palo Alto, California, 1968.

[4] Adams, D. A., "A Model for Parallel Computations," *Parallel Processor Systems, Technologies and Applications*, 1970.

[5] Arvind and R. H. Thomas, *I Structures: An Efficient Data Type for Functional Languages*, Laboratory for Computer Science, Technical Memo 178, MIT, Cambridge, Mass. Sept. 1978.

[6] Arvind and K. P. Gostelow, "A Computer Capable of Exchanging Processors for Time," *Proc. IFIP Congress 1977*, Toronto, Canada, pp. 849-853.

[7] Arvind, K. P. Gostelow and W. Plouffe, *An Asynchronous Programming Language and Computing Machine*, Technical Report TR 114a, University of California at Irvine, California, Dec. 1980.

[8] Backus, J. W., "Can Programming be Liberated from the Von Neumann Style? A Functional Style and its Algebra of Programs," *Communications of the ACM*, Vol 21, pp. 613-644, August 1978.

[9] Bredt, T. H., *A Survey of Models for Parallel Computing*, Technical Report No. 8, Digital Systems Laboratory, Computer Science Department, Stanford University, Stanford, California, August 1970.

[10] Brock, J. D. and L. B. Montz, "*Translation and Optimization of Data Flow Programs*," Proceedings of the 1979 International Conference on Parallel Processing, August, 1979.

[11] Coffman, E. G. and P. J. Denning, *Operating Systems Theory*, Prentice Hall, 1973.

[12] Davis, A. L., "The Architecture and System Method of DDM1: A Recursively Structured Data Driven Machine," *Proc. 5th Ann. Symp. Computer Architecture*, 1978, pp. 210-215.

[13] Dennis, J. B., "Programming Generality, Parallelism and Computer Architecture," *Information Processing 68*, North-Holland Publishing Co., 1969, pp. 482-492.

[14] Dennis, J. B., "First Version of a Dataflow Procedure Language," *Lecture Notes in Computer Science*, Vol. 19, Springer-Verlag, 1974, pp. 362-376.

[15] Dennis, J. B., "Data Flow Supercomputers," *Computer*, Vol. 13, No. 11, November 1980, pp. 48-56.

[16] Dennis, J. B., C. K. Leung and D. P. Misunas, "A Highly Parallel Processor Using a Dataflow Machine Language," CSG Memo 134-1, Laboratory for Computer Science, MIT, June 1979.

[17] Even, S., *Graph Algorithms*, Computer Science Press, 1979

[18] Gajski, D. D., D. J. Kuck and D. A. Padua, "Dependence Driven Computation," *Proc. COMPCON Spring*, Feb. 1981, pp. 168-172.

[19] Gajski, D. D., D. A. Panda, D. J. Kuck and R. H. Kuhn, "A Second Opinion on Dataflow Machines and Languages," *Computer*, Feb. 1982, pp. 58-70.

[20] Gaudiot, J. L. and M. D. Ercegovac, "A Scheme for Handling Arrays in Dataflow Systems," *3rd Int'l Conf. Distributed Computing Systems*, Hollywood, Florida, Oct. 1982.

[21] Gottlieb, A., R. Grishman, C. P. Kruskal, K. P. McAuligge, L. Rudolph and M. Snir, "The NYU Ultracomputer -- Designing an MIMD Shared Memory Parallel Computer," *IEEE Transactions on Computers*, Vol. C-32, No. 2, Feb. 1983, pp. 175-189.

[22] Hwang, K. and S. P. Su, "*Priority Scheduling in Event-Driven Dataflow Computers*," TR-EE-83-86, Purdue University, Indiana, Dec. 1983.

[23] Hwang, K. and F. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, 1984

[24] Karp, R. M. and R. E. Miller, "Properties of a Model for Parallel Computation: Determinacy, Termination, Queueing," *SIAM Journal of Applied Math.*, Vol. 14, No. 6, November 1966, pp. 1390-1141.

[25] Karp, R. M. and R. E. Miller, "Parallel Program Schemata," *Journal of Computer and System Sciences 3*, Vol. 2, May 1969, pp. 147-195

[26] Kosinski, P. R., *A Data Flow Programming Language*, IBM Research Report RC-4264, March 1973.

[27] Kung, H. T. and C. E. Leiserson, "Systolic Arrays (for VLSI)," *Proceedings of the Symposium on Sparse Matrix Computations and their Applications*, November 2-3, 1978, pp. 256-282

[28] Luconi, F. L., *Asynchronous Computational Structures*" MAC-TR-49, MIT, Cambridge, Mass., 1968

[29] Requa, J. E. and J. R. McGraw, "The Piecewise Data Flow Architecture: Architectural Concepts," *IEEE Transactions on Computers*, Vol. C-32, No. 5, May 1983, pp. 425-438.

[30] Rodriguez, J. D., *A Graph Model for Parallel Computation*, Technical Report TR-64, Project MAC, MIT, Cambridge, Mass., 1969.

[31] Rumbaugh, J., "A Data Flow Multiprocessor," *IEEE Transactions on Computers*, Vol. C-26, No. 2, Feb. 1977, pp. 138-146.

[32] Swamy, M. N. S. and K. Thulasiraman, *Graphs, Networks and Algorithms*, Wiley Interscience Press, 1981.

[33] Watson, I. and J. Gurd, "A Practical Dataflow Computer," *Computer*, Feb. 1982, pp. 51-57.

[34] Wong, K-S., *An Astract Implementation for a Generalized Dataflow Language*, Ph.D. thesis, MIT, Canbridge, Mass., May 1979.