DYNAMICALLY RECONFIGURABLE SYSTEMS RESEARCH

VASON P. SRINI

Computer Science Division and Electronics Research Lab.
University of California, Berkeley, CA 94720

NSF Final Report, Grant No. DCR -8508344
August 1988

SUMMARY

Computer systems that are capable of undergoing changes in the semantics or the interconnection of their modules in a dynamic way, called dynamic reconfiguration, are considered. The problems that must be addressed to change the semantics of a module are discussed. The change to the semantics of a module or their interconnection can be based on local information, global information, or a combination of both. Ways to effect changes when conflicts exist between local and global information are also discussed. Algorithms for reconfiguring the modules when data dependency constraints are present have been developed. Changing the semantics or the interconnection structure of a module may induce changes in other modules. Systematic ways to deduce induced changes have been developed. Protocols have also been developed for communicating reconfiguration information between modules.

A methodology based on the dataflow principles has been devised for designing reconfigurable systems. The nodes in the dataflow graph can store state information. These nodes are used to represent global and local controllers. The design of a sample operating system has been outlined using the dataflow methodology. Since interprocess communication is one of the key issues in reconfiguration, a multiprocessor architecture has been developed to support this. A separate synchronization memory is used in the multiprocessor for storing status information, process table, join table, and other data structures needed for interprocess communication. Simulation results show that fast interprocess communication is achievable with the synchronizing memory.

The report contains four chapters. The issues in reconfiguration and some applications are described in Chapter 1. The steps that must be taken to reconfigure a system are outlined in Chapter 2. It is based on an extended dataflow methodology which has been published as a paper. The application of the methodology to design a distributed operating system is described in Chapter 3. A system architecture capable of supporting reconfiguration is also shown in Chapter 3. The architecture support for fast interprocess communication (IPC) is described in Chapter 4. Dynamic memory management and process management for the parallel execution of Prolog programs on the proposed system architecture are used to illustrate the fast IPC.

Table of Contents

Chapter 1. ISSUES IN RECONFIGURATION

1. INTRODUCTION

Reconfiguration is an important part of computer systems containing multiple processors. Commercial systems such as Honeywell 68/80 running MULTICS, and Burroughs 6700/7700 running MCP support reconfiguration to provide graceful degradation and to improve performance. But reconfiguration in the above commercial systems is carried out using a statically specified set of configurations and ad hoc techniques. This research is primarily concerned with the problem of dynamically reconfiguring a multiple processor system comprising a large number of processors. Reconfiguration is done by changing the mixture of programs that are allowed to run on processors, and changing the strategies and policies used by the OS. The problem situation is called a dynamically reconfigurable operating system (DROS).

There are three key areas in a DROS. They are dynamic assignment of tasks to processors, interprocess communication, and memory management. The thrust of the research is in devising efficient ways to do interprocess communication. This is needed to communicate state information quickly during reconfiguration. We have developed a multiprocessor architecture that supports fast interprocess communication.

Recent research activities in reconfigurable computer systems can be classified into four categories. They are:

a. reconfigurable computer architecture,
b. reconfigurable software,
c. reconfigurable interconnection structure, and
d. reconfigurable program assignment (dynamic assignment of tasks to processors).

A summary of the activities in the above four categories is now presented.

Reconfigurable computer architectures have been proposed by Reddi [1] and Lipovski [2]. Some of the parts of the architecture proposed by Lipovski, using a Banyan network, have been implemented at the University of Texas, Austin, Texas. Reconfigurable semiconductor wafers containing a few hundreds of LSI processors and memories have been proposed by Hsia [3]. A reconfigurable microprocessor based system, called the dynamic computer, has been proposed by the Karteshevs [4-7]. The design of a reconfigurable distributed computer hardware with potential application to ballistic missile defense has been carried out by Vick [8,9] and Davis [10].

Design of reconfigurable software has been carried out in parallel with the above architecture development. Changing modules in a database systems "on the fly" has been discussed by Fabry [11]. The design of an experimental operating system for PDP 11/40E, where the representation of abstract data types can be replugged is discussed by Goullon [12]. All components of the operating system except the kernel are designed to be repluggable. This replugging approach is aimed at experimenting with different abstract data type representations having an invariant specification or leads to a compatible specification.

Computer systems with reconfigurable interconnection structures are commercially available. Some examples are MULTICS [13,14] running on Honeywell 68/80 processors, MCP running on B6700/7700 [15], Tandem_16 [16], and BTI-8000 [17]. A network of computers with reconfigurable interconnection structure is also under development at SUNY, Buffalo [18].

Dynamic assignment of programs (tasks) to processors using network flow algorithms has been extensively studied by Stone [19-21] and by other researchers [18,22-25]. One of the primary goals in Stone's work is to minimize the total of execution costs and interprocessor communication costs under the assumption that the assignment algorithms know the behavior of the programs. The assignment of programs to processors when one or more components fail is studied by Kim [26] and others [27,28].

Although considerable progress has been made towards understanding the issues in the above four categories of reconfigurable computer systems, a unified approach to design and implement reconfigurable systems is not available. Reconfiguration requires the monitoring of program behavior, reusable resource usage, and component (hardware and software modules) failures. There is an overhead incurred in

monitoring and the extent to which this overhead can be reduced is not well known. The dynamic assignment of tasks from the various programs to processors so that the computer system is globally stable [20] is still an open problem.

Almost all computer systems have some facility to do static reconfiguration under operating system control. There are also operating systems based on virtual machines, e.g. IBM's VM370, that allow different versions of an operating system to be concurrently debugged on a single processor or a multiprocessor (shared memory). The problem is that reconfiguration takes time and there are occasional surprises to programmers trying to execute previously well behaved programs. For example, consider a computer system with a single processor and several I/O processors (channels). If one or more disk units fail in the system, and a portion of a multivolume file resides in a failed disk unit, then the program manipulating the file cannot be executed although the operator has reconfigured the computer using the remaining healthy disk units. The reason is that the operating system allows logical names for files. A mapping from logical names to physical devices is performed statically either by the programmer (e.g. VAX/VMS) or by the operating system (e.g. Multics [14]) during the definition of the logical name.

This research proposes a dataflow graph based methodology for designing a DROS subject to real-time constraints and/or data dependency constraints. The proposed methodology is used to identify some of the key problems in the design of a DROS. This is shown in Section 2. Several application areas where DROS and the principles employed in DROS will be useful are outlined in Section 3.

2. PROBLEMS IN A DROS

2.1 DATAFLOW GRAPH MODEL FOR DROS

A DROS comprises several asynchronous processes which cooperate to control the various resources of the computer system. Each resource has a set of attributes and the values of some of the attributes can change from time to time. For example, a buffer pool is a resource and its key attributes are size, type of data it can store, and the buffer allocation/liberation algorithm. The size of the buffer pool changes as buffers are allocated to requesting processes. Different allocation/liberation algorithms are needed to handle requests from the file system and processes responsible for interprocess communication.

In order to accurately state the problems in a DROS, it is represented using a dataflow graph. A dataflow graph is based on the principles of Petri Nets [29-31] and basic dataflow graphs developed by Dennis [32] and others [33-37]. Dataflow is a model of computation where functions can begin execution when required data becomes available. The execution of any function is free of side-effects, that is a set of input values may be consumed and a set of output values may be produced. Thus dataflow is characterized by asynchrony, functionality, and concurrency. Although there are other models such as Hoare's cooperating sequential process [38], they are not considered because of the presence of variables that are treated as storage locations. Another reason is that it is easier to detect deadlock, flooding, and starvation using a dataflow graph [35].

A dataflow graph is a labeled, undirected or partially directed graph consisting of a set of nodes, arcs, and a tagging scheme. The nodes represent communicating activities with internal state in a DROS including local control elements and global control elements. The arcs represent data paths for communicating data items and state information. Each node has several attributes including a distinct name, an operation, and a set of I/O specifications listing its input and output arcs. Each arc has several attributes including an arc name, arc type, and the maximum number of data items (tokens) the arc can carry. Each node has local memory to save the values for node and arc attributes. A model configuration is an assignment of values to the attributes of nodes and arcs. Each token in a dataflow graph carries a tag so that several distinct calculations can be processed in the proper order without interference between distinct calculations. The distinct calculations result from the multiple instances of a task such as a loop, a recursive function call, and the elements of a stream [35].

In the design of DROS, the arcs carrying state information, and the nodes representing local and global control elements are important items. Dashed arcs, called snoop arcs, represent the paths carrying state information to the control elements. Special kinds of nodes are used in a dataflow graph to represent control elements. These nodes are called configuration specifying node (CSN), self reconfiguring node (SRN), and reconfiguration request node (RRN). A CSN is intended for representing a global control

element. It has local memory to store the state information. Note that local memory can be simulated in dataflow by feeding back tokens. A CSN can assign values to the attributes of nodes and arcs connected to it using global information. The SRNs are intended for representing local control elements. It has local memory to store state information. The SRNs assign values to the attributes of nodes and arcs using local information. The RRNs are intended for representing the elements in a DROS that can request changes to the attribute values. Using the above special nodes and the general nodes of a dataflow graph it is possible to represent any DROS, where reconfiguration refers to changing the values for the attributes of nodes and arcs. The dataflow graphs with snoop arcs, CSNs, SRNs, RRNs, and general nodes are called as extended dataflow graphs (EDFGs).

A detailed description of EDFGs is given in [39]. A complex computer system such as the Cray-1S processor has been modeled and its architecture analyzed for improving performance using EDFGs. The execution of the nodes in the EDFGs has been simulated using GPSS V [40]. The timing statistics obtained for a class of programs supplied as input to the Cray-1S model is identical to the execution time on the Cray-1S processor. The proposed changes to the Cray-1S architecture have been incorporated in the Cray-XMP processor. The usefulness of a dataflow methodology for designing OS is further explored in Chapter 3.

2.2 ISSUES IN A DROS

The key issues in a DROS are identifying the components to be reconfigured, and resolving the questions **when to reconfigure, what to reconfigure, and how to reconfigure.** The specific research questions are formulated using the EDFG representation of a DROS. The problems involved in reconfiguration are:

a. identifying nodes whose I/O arc specifications or operations have to be changed,
b. determining a new I/O arc specification for each of these nodes,
c. determining a new operation for each of these nodes,
d. determining the set of arcs that should not have any tokens on them so that I/O arc specifications to nodes can be changed,
e. determining values for the rest of the arc attributes,
f. determining values for the rest of the node attributes, and
g. formulating strategies that allow reconfiguration to take place in stages when there are tokens on arcs in the calculated set of arcs mentioned

above.

Reconfiguration can be based on global information, local information, or a combination of both. The latter approach introduces complications because of potential conflicts between the global and local requirements. The timeliness of a reconfiguration may be important because of data dependency requirements and realtime requirements. Simultaneous reconfiguration requests from several RRN nodes further complicate the above problems along with handling exception conditions during a reconfiguration. Simultaneous reconfiguration requests can be to a single CSN or to distinct CSNs. These requests can induce conflicting I/O arc specifications for one or more nodes.

A set of solutions to the above problems are described in chapter 2. The practical applications of reconfiguration are outlined in the next section.

3. APPLICATIONS OF DROS

The need for dynamic reconfiguration is present in almost all computer systems. The extent to which reconfiguration is permitted in current computer systems is limited. Reconfiguration is performed manually and statically using ad hoc techniques. However, dynamic reconfiguration of the hardware and software in a timely manner is required in several applications. Some examples are expert systems, ballistic missile defense, air traffic control and passenger reservation systems, computational aerodynamics,

weather model analysis, and intensive care units of hospitals.

3.1 EXPERT SYSTEMS

The software modules used in the knowledge base of expert systems require reconfiguration to the module functions as new information arrives at the databases or new rules are added. Currently, these changes are made manually and statically, a time consuming activity and prone to errors. One important area in expert systems is belief revision [42] as new knowledge is obtained. If the belief revision part is designed using the principles employed in a DROS then the semantics of the modules in it can be dynamically changed. Systems using a knowledge-based approach to fault tolerance [43] employs a diagnostic blackboard and a number of knowledge sources. The knowledge sources communicate with the diagnostic blackboard and are initially evaluated by some preset parameters. Later the evaluation is based on information gathered from different parts of the system under diagnosis. The modules forming the evaluation process can use the principles and strategies developed for DROS.

3.2 BMD SYSTEM

In the case of ballistic missile defense (BMD) system employing endoatmospheric engagement, the computer system has approximately 30 seconds to engage, detect, track, and destroy incoming missiles [10]. Within the 30 seconds, the hardware and the software of the BMD computer has to be reconfigured to handle the variable number of data streams entering the system. Although research efforts are pursued to design hardware capable of undergoing reconfiguration within 30 seconds [2,5,9,10,41], there is not much effort going on in designing reconfigurable software systems outside the BMD-ATC group. The DROS approach using EDFG can greatly assist the tracking and targeting problems in the BMD application. The tracking and targeting problems in BMD and how it can be represented using EDFGs is now outlined.

A few thousand radar beams scan the surveillance volume searching for objects entering the volume. Once objects are detected, special radar beams have to be interleaved to verify detection, precision tracking, target classification, and several other functions. The strategy to be used in interleaving the radar beams and the steps to be used in processing the incoming signals depends on the characteristics of the incoming data. There can be a large number of combinations of target shapes, characteristics, and threat values entering the surveillance volume. From this large number, a small number of targets have to be identified that presents the ultimate threat.

The signals received by each radar beam and the signal processing performed on each of the beams can be viewed as an asynchronous activity. It can be represented as a general node or as a SRN if the radar beams are special beams. A neighborhood function can be used to partition the radar beams. For each neighborhood a coordinator can be defined for target detection, classification, and tracking. Since the coordinators use global information in their decision making, they are represented as CSNs. If the CSNs are connected so they can communicate with each other, then precision tracking and the threat target identification can be carried out by one or more CSNs.

The resulting EDFG for the BMD system undergoes reconfiguration in a dynamic manner using the information from several CSNs. The dynamic reconfiguration facilitates target identification and classification using a database. It also facilitates selecting a small number of potential threat targets and precision tracking them.

3.3 AEROSPACE SYSTEM

The projected increase in air traffic during the 1980's, and the increased demands from pilots and passengers on the safety of air traffic will greatly increase the data and computations on the air traffic control computers. These computers use modular redundancy and backup databases to provide hardware fault tolerance. Employing these redundancy techniques to provide fault tolerance in future air traffic control computers will be cost prohibitive. However, a combination of redundancy and reconfiguration of hardware/software might offer cost effective solutions.

The processor and storage requirements of computational aerodynamics has been described by Chapman [44]. To compute values for some of the key measures involved in the flow of fluid over a practical three dimensional wing-body configuration, computer systems employing tens of thousands of processors and main memory with capacity in excess of 30 trillion words are required [42]. Fault tolerance using

redundancy techniques is not feasible in such computer systems. However, the computer system will gracefully degrade when hardware/software faults occur provided it has been designed using a DROS. Recently, a design of a computer system [45], called flow model processor (FMP), employing 512 processors has been proposed to NASA Ames for computational aerodynamics. Although fault tolerance and automatic recovery for software are discussed in the FMP proposal, it is not clear that the reconfiguration characteristics of the FMP can be predicted in a reasonable manner. For example, what is the delay involved in switching a spare processor to replace a faulty processor? What happens to the program currently using all processors? If software fails, when is it detected? How are the recovery procedures invoked? How is it guaranteed that the program and data of the currently running program are not corrupted by the software faults? The above questions cannot be answered with any degree of accuracy and the primary reason is that the FMP has not been designed using a DROS. The design and implementation approaches employed in a DROS will be useful in the design of future computer systems for computational aerodynamics.

3.4 RESTRUCTURABLE VLSI

Recent developments in silicon fabrication technology using electron beams and x-rays for projecting circuit patterns onto silicon wafers have made possible the development of very large scale integrated circuits (VLSI) with a million devices per die by 1990. Although VLSI chips are more reliable than LSI chips and discrete components, there is still a nontrivial failure rate. It is not economical to discard VLSI chips when some of the circuits are faulty. For example, a 1 Mega bit memory chip with faults in a 64 K bit part is certainly useful except for the faulty portion of the chip. If the chip design has been a priori reconfigurable, then every time faults occur in the chip, the chip will reconfigure itself to isolate the faulty part. Reconfiguration will also facilitate changing the functionality of a VLSI chip containing electrically erasable and programmable memory elements. Recently, a restructurable VLSI processor design has been proposed by Budzinski [46] incorporating some of the features needed for reconfiguration. The strategies useful in DROS will also be useful in the design of reconfigurable VLSI chips and wafer scale integrated circuit chips [3, 47].

4. REFERENCES

1. S. S. Reddi, and E. A. Feustal, "A Restructurable Computer System", IEEE Transactions on Computer (TC), Vol. C-27, No. 1, Jan. 1978, pp. 1-20.

2. J. Lipovski, "On a Varystructured Array of Microprocessors", IEEE TC Vol, C-26, No. 2, Feb. 1977, pp. 125-137.

3. Y. Hsia, G. C. C. Chang, and F. D. Erwin, "Adaptive Wafer Scale Integration", Proc. of 1979 Intl. Conf. on Solid State Devices, Tokyo, Japan, Aug. 1979.

4. S. I. Kartashev, and S. P. Kartashev, "Dynamic Architectures: Problems and Solution", Computer Magazine, July 1978, pp. 26-40.

5. S. I. Kartashev, and S. P. Kartashev, "Problems of Designing Supersystems with Dynamic Architectures" IEEE TC Vol. C-29, No. 12, Dec. 1980, pp. 1114-1132.

6. S. I. Kartashev, and S. P. Kartashev, "Microcomputer System with Dynamic Architecture", IEEE TC, Vol. C-28, No. 10, Oct. 1979, pp. 704-721.

7. S. I. Kartashev, S. P. Kartashev, and C. V. Ramamoorthy, "Adaptation Properties for Dynamic Architectures", Proc. 1979 AFIPS Conf. Vol. 48, AFIPS Press, Montvale, NJ, pp. 543-556.

8. C. R. Vick, "A Dynamically Reconfigurable Distributed Computing System", Ph.D. dissertation, Auburn Univ., Auburn, Alabama, 1979.

9. C. R. Vick, S. P. Kartashev, and S. I. Kartashev, "Adaptable Architectures for Supersystems",

Computer Magazine, Nov. 1980, pp. 17-35.

10. C. G. Davis, and R. L. Couch, "Ballistic Missile Defense: A Supercomputer Challenge", Computer Magazine, Nov. 1980, pp. 37-46.

11. R. S. Fabry, "How to design a system in which modules can be changed on the fly", Proc. 2nd Intl. Conf. on Software Engineering, 1976, pp. 470-476.

12. H. Goullon, R. Isle, and K. Lohr, "Dynamic Restructuring in an Experimental Operating System", IEEE Transactions on Software Engineering, (TSE) Vol. SE-4, No. 4, July 1978, pp. 298-307.

13. E. I. Organick, The Multics System: An Examination of its Structure, MIT Press, Cambridge, MA, 1972.

14. R. R. Schell, Dynamic Reconfiguration in a Modular Computer System, Ph.D. Thesis, MIT, Cambridge, MA, June 1971. NTIS Report AD 725 859, Springfield, VA 22151.

15. Burroughs Corporation, B6700 Information Processing Systems, Hardware Reference Manual, 1972, Burroughs Corporation, Detroit, Michigan 48232.

16. J. A. Katzman, "A Fault-Tolerant Computing System", Proceedings of the Hawaii International Conference on System Sciences, Jan. 1978, pp. 85-102.

17. BTI 8000 System Fundamentals, Version 1.0, April 1981, Sunnyvale, CA.

18. L. D. Wittie, and A. M. Van Tilborg, "MICROS, A Distributed Operating System for MICRONET, A Reconfigurable Network Computer", IEEE TC, Vol. C-29, No. 12, Dec. 1980, pp. 1133-1144.

19. H. S. Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithms", IEEE TSE, Vol. SE-3, No. 1, Jan. 1977, pp. 85-93.

20. H. S. Stone, and S. H. Bokhari, "Control of Distributed Processes", Computer Magazine, July 1978, pp. 97-106.

21. G. S. Rao, H. S. Stone, and T. C. Hu, "Assignment of Tasks in a Distributed Processor System with Limited Memory", IEEE TC, Vol. C-28, No. 4, April 1979, pp. 291-299.

22. Y. C. Chow, and W. H. Kohler, "Models for Dynamic Load Balancing in a Heterogeneous Multiple Processor System", IEEE TC, Vol C-28, No. 5, May 1979, pp 354-361.

23. J. K. Ousterhout, D. A. Scelza, and P. S. Sindhu, "Medusa: An Experiment in Distributed Operating System Structure", Comm of the ACM, Vol. 23, No. 2, Feb. 1980, pp. 92-105.

24. C. C. Price, "The Assignment of Computational Tasks Among Processors in a Distributed System", Proc. 1981 AFIPS Conf., Vol. 50, AFIPS Press, Montvale, NJ, pp. 291-296.

25. I. N. Chen, P. Y. Chen, and T. Feng, "Associative Processing of Network Flow Problems", IEEE TC, Vol. C-28, No. 3, March 1979, pp. 184-190.

26. K. H. Kim, and C. V. Ramamoorthy, "Failure-tolerant Parallel Programming and its Supporting System Architecture", Proc. 1976 AFIPS Conf., Vol. 45, AFIPS Press, Montvale, NJ, pp. 413-423.

27. T. F. Gannon, and S. D. Shapiro, "An optimal Approach to Fault Tolerant Software Systems Design", IEEE TSE, Vol. SE-4, No. 5, Sept. 1978, pp. 390-409.

28. A. Avizienis, "Fault Tolerance by Means of External Monitoring of Computer Systems", Proc. 1981 AFIPS Conf., Vol. 50, AFIPS Press, Montvale, NJ, pp. 27-40.

29. C. A. Petri, "Communications in Automaton", University of Bonn, 1062. Translation: C. F. Greene, Supp. 1 to TR RADC-TR-65-337, Vol. 1, Rome Air Development Center, Griffiss Air Force Base, New York, 1965.

30. J. L. Peterson, "Petri Nets", ACM Computing Surveys, Vol. 9, No. 3, Sept. 1977, pp. 223-252.

31. C. V. Ramamoorthy, and G. S. Ho, "Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets", IEEE TSE, Vol. SE-6, No. 5, Sept. 1980, pp. 440-449.

32. J. B. Dennis, and J. B. Fosseen, "Introduction to data flow schemas", Computation Structures Group Memo: 81, Project MAC, MIT, Cambridge, MA, 1973.

33. D. A. Adams, "A computation model with data flow sequencing", TR-117, CS Dept., School of Humanities and Sciences, Stanford University, Dec. 1968.

34. R. M. Karp, and R. E. Miller, "Parallel program schemata", J. Computer System Sciences, Vol. 3, No. 2, May 1969, pp. 147-195.

35. Arvind, and K. P. Gostelow, "The Id Report: An asynchronous language and computing machine", Dept. of Computer and Information Science, TR-114, Sept. 1978, UC Irvine, CA.

36. A. L. Davis, "The architecture and system method of DDM1: A recursively structured data driven machine", Proc. of 5th Annual Symposium on Computer Architecture, New York, 1978, pp. 218-215.

37. S. P. Landry, and B. D. Shriver, "A dataflow simulation research environment", Workshop on data Driven Languages and Machines, Toulouse, France, Feb. 1979, pp. x 1-15.

38. C. A. R. Hoare, "Communicating sequential processes", Communications of the ACM, Vol. 21, No. 8, Aug. 1978, pp. 666-677.

39. V. P. Srini, and B. D. Shriver, "A Methodology for Designing and Modeling Reconfigurable Systems", International Journal of Computer and Information Sciences, Oct. 1984.

40. V. P. Srini, and J. F. Asenjo, "Analysis of Cray-1S architecture", Proc. of the 10th Annual Intl. Symposium on Computer Architecture, Stockholm, June 1983, pp. 194-206.

41. B. D. Rathi, A. R. Tripathi, and G. J. Lipovski, "Hardwired Resource Allocators for Reconfigurable Architectures", Proceedings of 1980 Intl. Conf. on Parallel Processing, Boyne Highlands, Michigan, Aug. 1980, pp. 109-117.

42. M. Stefik, et al, "The organization of expert systems: A prescriptive tutorial", Technical Report VLSI-82-1, Xerox-PARC, Palo Alto, CA, Jan. 1982.

43. E. Hudlicka, "Diagnosing Problem-Solving System Behavior", Ph.D Dissertation, COINS Tech. Report 86-03, Univ. of Mass., Computer and Information Science Dept., Amherst, MA, Feb. 1986.

44. D. R. Chapman, "Computational Aerodynamics Development and Outlook", Dryden Lectureship in Research, 17th Aerospace Sciences Meeting, New Orleans, Jan. 1979. NASA Technical Report 79-0129. Also, AIAA Journal, Vol. 17, Dec. 1979, pp. 1293-1314.

45. S. F. Lundstrom, and G. H. Barnes, "A Controllable MIMD Architecture", Proceedings of 1980 Intl. Conference on Parallel Processing, Boyne Highlands, Michigan, Aug. 1980,, pp. 19-27.

46. R. L. Budzinski, J. Linn, and S. Thatte, "A Restructurable Integrated Circuit for Implementing Programmable Digital Systems", Proceedings of 2nd Caltech Conference on Very Large Scale Integration, Pasadena, CA, Jan. 1981.

47. J. I. Raffel, et al., "A Wafer-Scale Digital Integrator using Restructurable VLSI", IEEE Journal of Solid-State Circuits, Vol. sc-20, No. 1, Feb. 1988, pp 399 - 406.

Chapter 2 STEPS IN RECONFIGURING A SYSTEM

1. DATA AQUISITION PROCESS

Reconfiguration in a computer system can be user initiated or system initiated. With the EDFG representation this means the communication of information from RRNs to CSNs using snoop arcs or communication within SRNs. This communication must take place in a timely manner. Since requests for reconfigurations cannot be predicted ahead of time, there is the possibility that several requests for reconfiguration might reach a CSN at the same time. This greatly complicates the communication of information between nodes. Efficient ways to communicate reconfiguration information and to identify what reconfigurations can be potentially performed have to be devised. A CSN might want additional information from RRNs and/or other nodes connected to it before determining whether the attributes of a node can be changed. This requires protocols for communication between nodes and efficient implementation techniques for the protocols. The protocols are described in the paper by Srini [1]. Gantt charts of the protocols are described in this section for four cases.

The Gantt chart for the data acquisition process in a system containing a single CSN is shown in Figure 1. An RRN initiates reconfiguration. The CSN interacts with concerned nodes to obtain enough information for starting reconfiguration.

The Gantt chart for the multiple CSN situation is shown in Figure 2. The CSN that receives the reconfiguration request has to interact with other CSNs. The interaction may involve several cycles of message communication between CSNs.

If the system has a single SRN the interaction with other nodes is fairly simple and it is shown in Figure 3.

The Gantt chart for a system containing a CSN and an SRN is shown in Figure 4. The CSN gathers information from the nodes and then interacts with the SRN. If there are conflicts with the information received from SRN, the CSN held value is used in the reconfiguration.

An architecture that supports fast interprocess communication is shown in chapter 4.

2. RECONFIGURATION DRIVE ALGORITHM

A single request for changing the attribute values of a node in an EDFG can induce changes in the attribute values of nodes connected to it. The set of all nodes and arcs whose attribute values have to be changed and the new values for the attributes have to be computed. Algorithms that perform the above are called reconfiguration drive algorithms. Algorithms have been developed when single reconfiguration request to a CSN, and multiple reconfiguration requests to a CSN are present. These are described in the paper by Srini [1]. The realtime constraints and conflicting requirements of different reconfiguration requests can greatly complicate the implementation of these algorithms. The effects of these constraints on the algorithms can be reduced if the communication between the nodes is fast.

3. RECONFIGURATION STRATEGIES

A reconfiguration strategy is a set of state transitions that will eventually transform an EDFG to a reconfigurable state. A simple strategy that CSNs can use is to set the state of some nodes to suspended and allow others to fire until a reconfigurable state is reached. The CSNs make the required changes in I/O arc specifications and operation to all nodes and changes to the attributes of arcs. There is no guarantee, however, that a reconfigurable state will ever be reached. There are two additional problems with this strategy: time and performance. Some reconfiguration must take place within a certain time limit after receiving the request. Otherwise, the results might be catastrophic or the effects of reconfiguration might be useless. For example, a computer system might be expected to keep a certain throughput rate and maintain a certain percentage for the utilization of resources. If the performance level cannot be maintained, then response time degrades which in turn affects the realtime needs.

Reconfiguration strategies have been developed [1] for the following cases when realtime constraints and data dependency constraints are present.

a. Global information and a single CSN.

b. Global information and multiple CSNs.

c. Local information and global information with multiple CSNs.

The Gantt chart for the reconfiguration process when there is a single CSN is shown in Figure 5. The first step is to pick a path in the reconfiguration drive tree using a strategy such as minimum path length [1]. Reconfiguration can then proceed in one step, as shown in Figure 5.a, if there are no obstructions. This is the fastest way to achieve reconfiguration and must be used when realtime constrains are present. However, obstructions do occur in the path in some systems and in such cases reconfiguration is carried out in multiple steps (C1, C2, etc.), as shown in Figure 5.b. During each step a part of the system is reconfigured. The system is allowed to function so that some of the obstructions in the reconfiguration path can be removed. The time needed to do reconfiguration is unpredictable. So, this approach is not applicable when realtime constraints are present.

If multiple CSNs (with or with out local information) are present in a system then the reconfiguration process is carried out using the single step or the multiple step approach. The Gantt charts for the reconfiguration process are shown in Figure 6. The one step approach must be used when realtime constraints are present.

4. REFERENCE

1. V. P. Srini, and B. D. Shriver, "A Methodology for Designing and Modeling Reconfigurable Systems", International Journal of Computer and Information Sciences, Oct. 1984.

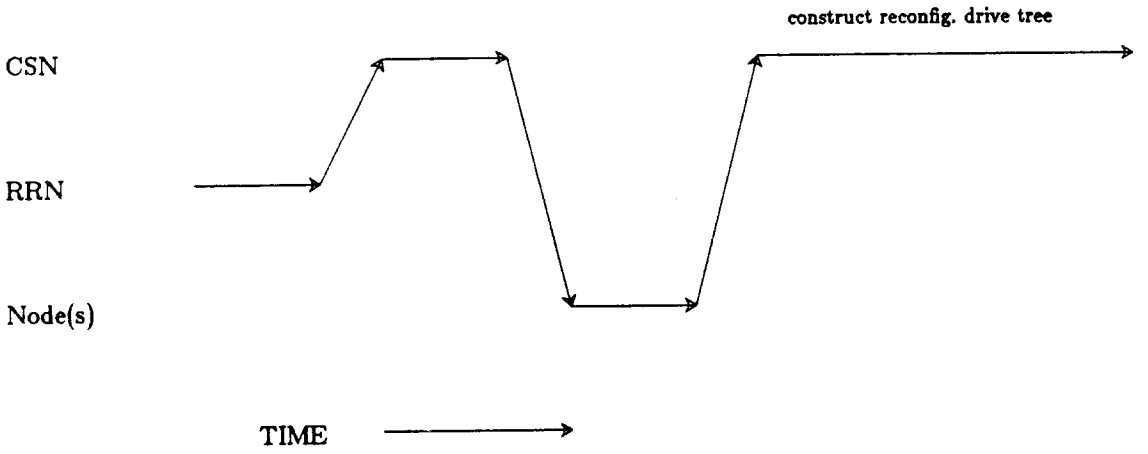


Figure 1. Single CSN

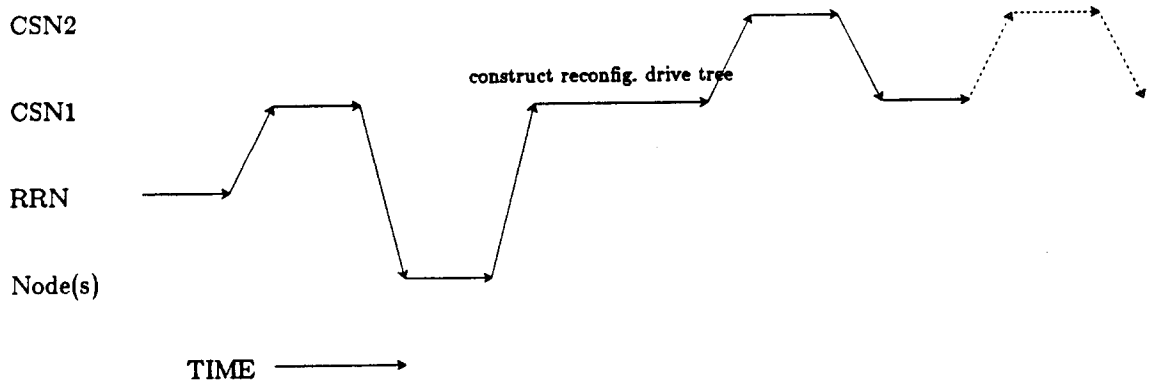


Figure 2. Multiple CSNs

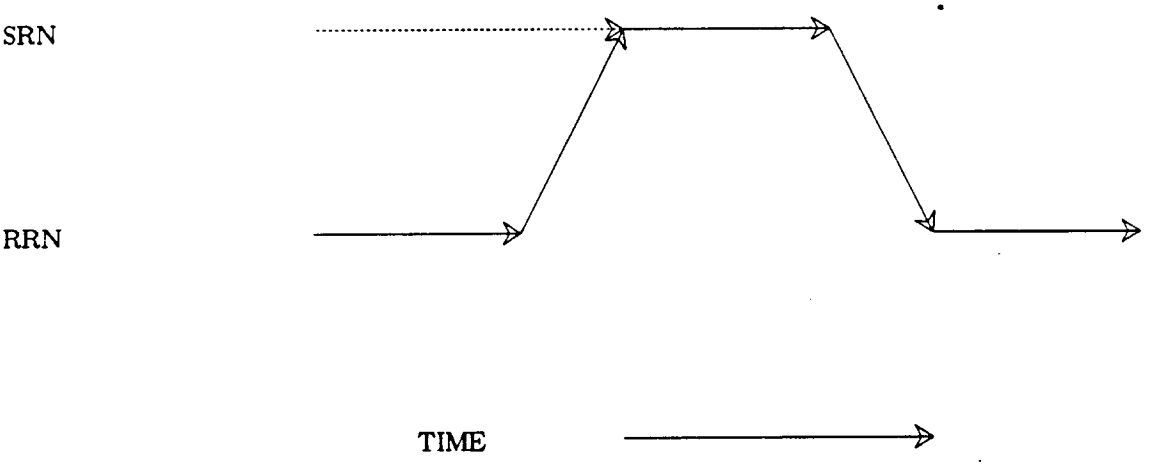


Figure 3. SRN

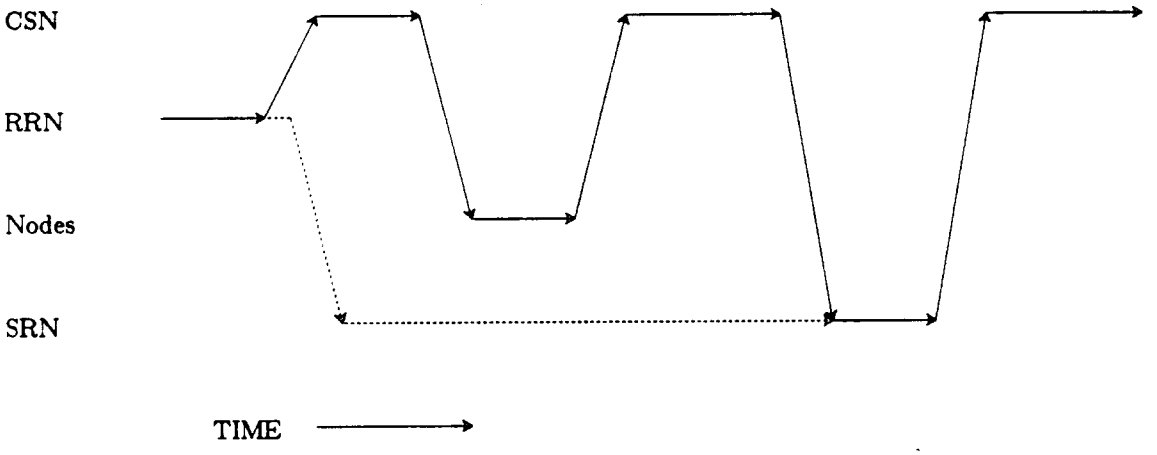


Figure 4. SRN and CSN

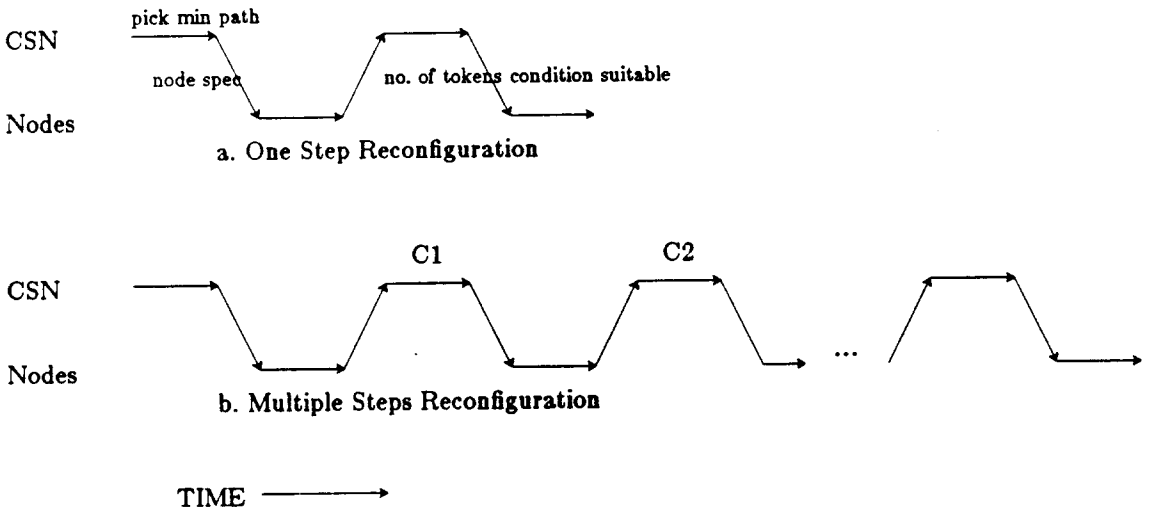


Figure 5. Reconfiguration for Single CSN

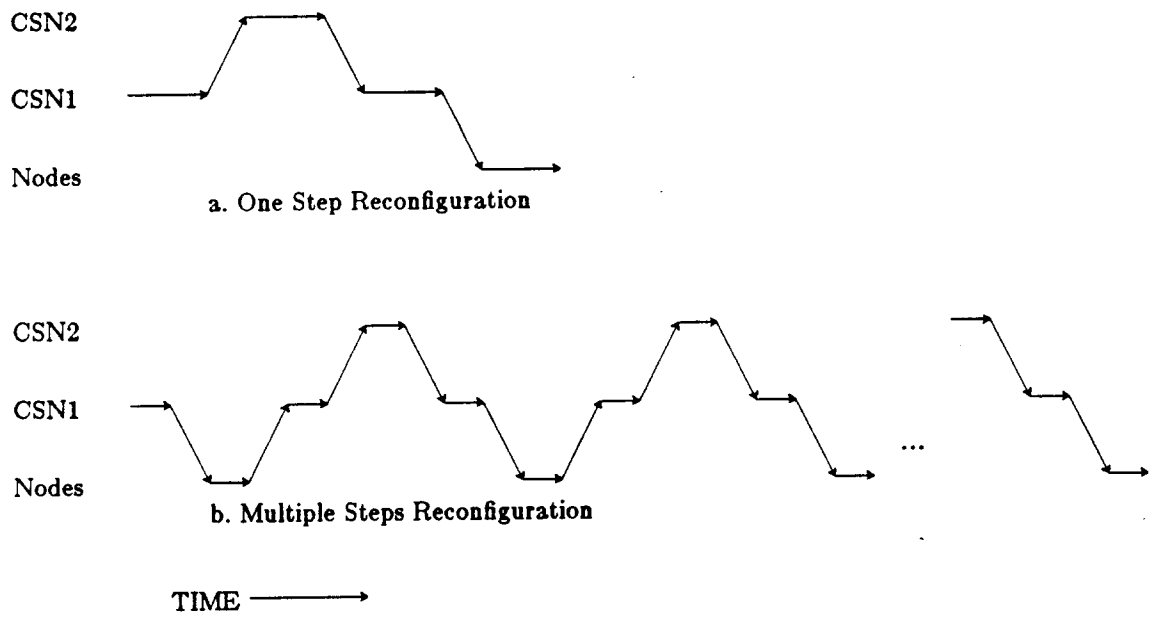


Figure 6. Reconfiguration for Multiple CSNs

Chapter 3. OPERATING SYSTEM DESIGN USING DATAFLOW GRAPHS

1. MOTIVATION

The design of a distributed operating system is used to show the usefulness of dataflow graphs in OS design. The modules of the DOS can make use of reconfiguration strategies discussed earlier. Distributed operating systems (DOS) have been designed and implemented by several research groups. Some examples are Medusa [1] and StarOS [2] for Cm* [3], Roscoe [4] for the University of Wisconsin's system, Micros [5] for a network of LSI processors at SUNY Buffalo, and HXDP of Honeywell [6]. The kernel functions in the above DOS provide an execution environment for programs based on the imperative model of computation [7,8]. The purpose of this work is not to report on yet another DOS, but to outline a reconfigurable DOS design using the principles of dataflow [9 -12].

A distributed computer system (DCS) is characterized by multiplicity of physical and logical resources, decentralized control, and cooperative autonomy among resources [13]. The logical and physical resources communicate by passing messages. The message communication can be implemented using shared memory techniques. The DCS usually decomposes programs into asynchronous tasks and executes them on the available processors. The programs for DCS can be prepared using an imperative model of computation [7,8], a dataflow model of computation [12] or logic programming. The concurrency in programs that can be detected by a compiler is limited if the programs are written using an imperative model of computation. Logic programming languages exhibit concurrency at the clause level and compile time detection of AND parallelism and OR parallelism is feasible. Dataflow programs allow concurrency to be detected at the lowest level (instruction level). This is because in a dataflow model of computation, functions are executed when the required input becomes available.

A conceptual organization of a DCS used in this work is shown in Figure 1.a. The details of a single level using shared memory and a single global address space is shown in Figure 1.b. The level memory in Figure 1.a corresponds to the synchronizing memory in Figure 1.b. It is used for write shared data (multiple writers and readers). The synchronization bus and the snooping caches also support the interconnection between processor nodes. They represent IN1 and IN3 in Figure 1.a. The system memory in the DCS has N partitions, one for each level. Each partition corresponds to the crossbar memory in Figure 1.b. Read shared, private data, and code block of a level are stored in a partition. The host node in Figure 1.b is responsible for the communication between levels (c.f. IN4 in Figure 1.a) and also user interface to the system memory. This architecture is under development at Berkeley as a part of the Aquarius project (c.f. [35 - 37]).

Since a DOS forms part of a DCS with distributed control, data, and physical resources, the sharing of data and resources is no longer the primary concern of the OS. As a result, the OS design and implementation need not be based on an imperative model of computation [7,8] where a variable is identical to a storage location. The absence of centralized control and the fact that no one entity in the DCS knows the status of the entire system means that the DOS can be designed as a collection of asynchronous activities which communicate occasionally be passing messages. One can immediately note that the conceptual and initial design phases of a DOS can be carried out using dataflow graphs [12,14]. A basic dataflow graph [11] comprises a collection of nodes connected by directed arcs. Each node has an operation which can be a single instruction or a collection of instructions, namely a block. Nodes fire and communicate tokens by sending and receiving tokens on output arcs and input arcs respectively. What is not obvious in the applicability of dataflow principles in the implementation of DOS. Lauer and Needham [15] have conjectured that an implementation of an OS can be categorized to be either a message-oriented system or a procedure-oriented system. If the tokens in dataflow are treated as messages, enabling conditions for nodes as synchronizing conditions, and arcs as data paths, then a new message based approach to implementing DOS that is free of side-effects is possible (exceptions are CSN and SRN type of nodes). The above observation and the architecture of a DCS that combines distribution and copying semantics of data to execute programs based on the dataflow model of computation have motivated the development of a dataflow graph methodology for designing and implementing DOS.

This chapter is divided into six sections. Three approaches for designing DOS are described in Section 2. The organization of a DOS is discussed in Section 3. A processor architecture for doing fast interprocess communication is described in Section 4. The design of a DOS kernel using the dataflow methodology is discussed in Section 5. The implications of the processor architecture in designing a DOS is

discussed in Section 6.

2. DESIGN APPROACHES

There are at least three approaches for designing DOS based on process interaction facilities. They are procedure call [16] or monitors [17,18], message passing [19,20], and dataflow [12,14]. The procedure call approach is a synchronous form of interaction. The message passing and dataflow are asynchronous approaches.

Using the procedure call approach, a DOS is designed as a collection of sequential processes [18,21] with interaction between processes using procedure calls. A process, X, wishing to interact with another process, Y, does so by calling a procedure in Y with parameter passing using values or references. X waits until a return is performed by the procedure in Y before continuing with its execution. A process usually wants to interact with another process since they may be sharing data objects. To facilitate the orderly sharing of data among processes using procedure calls, the data abstraction, monitor [17], has been invented by Hoare, Brinch Hansen, and Dijkstra. (Simula's class, CLU's cluster, ALPHARD's form, and Hewitt's actors are other data abstractions). The monitor approach facilitates interaction between processes by changing shared data to which they have been given access rights at the time of initialization. So, compile time checking can be performed for access right violations. Several DOSs have been designed using the procedure call/monitor approach. Micros [5] is one example that uses monitors for process interaction. Distributed systems based on Mesa [22] also use monitors for process interaction. The DOSs StarOS [2], and Medusa [1] for Cm*, and Roscoe [4] use procedure call for process interaction. But the number of procedure calls has been reduced by limiting the calls to the absolutely necessary functions. Design of DOS for realtime systems using only procedure call has been proposed by Brinch Hansen [16] and an experimental DOS outlined in [23].

A DOS using the message passing approach comprises a collection of sequential processes that communicate by sending and receiving messages. Processes do not act on each other (interact) since they do not share data objects. Instead, processes communicate. A process, X, wishing to communicate with another process, Y, does so by sending a message to Y. After sending the message, process X can proceed with its calculation unless an acknowledge or a receive reply statement follows the send message. The asynchronous nature of message communication permits processes in different processors to execute simultaneously. It also facilitates the separation of DOS components which simplifies the implementation and maintenance of DOS. Several of the DOSs [1,2,4,5] use messages for interprocess communication. Medusa [1] has used message communication for the most part except in places where sharing (process interaction) is absolutely necessary. An experimental network-based DOS using Smalltalk [24,25] at Xerox PARC is an example using only message passing.

A DOS using the dataflow approach comprises a collection of functions that are data driven. Data is communicated using self-identifying messages (tokens). A function may be a primitive or a collection of primitives and other functions. There is no sequencing imposed on the execution of a primitive or function except those due to data dependency. Resource managers will be specified as CSN nodes or SRN nodes in the dataflow approach. These nodes have local memory to save state information. The dataflow approach has all the advantages of the message passing approach plus exploiting concurrency at the primitives level. Two operating systems [26,27] have been designed and simulated on a dataflow simulator [28] running under MULTICS to demonstrate that the dataflow approach is a viable one. The sample operating system described in [29] has been designed using basic dataflow graphs [11] in [26]. A virtual machine monitor has been designed using dataflow graphs in [27]. The design and simulation exercises showed three problem areas where extensions to basic dataflow graphs are needed before using dataflow approach in designing DOS. The problem areas are static and directed graph structure, inability to dynamically change the semantics of nodes, and excessive copying of tokens in generating data structures. The base language for the high level dataflow language Id [12] with dynamic nodes, the APPLY primitive, and the incremental structure memory organization (I-structure) in [30] provide one set of solutions to the above problems. We provide a solution with emphasis on reconfiguration, that is dynamic changes to the semantics of nodes and arc direction, using the logic programming language Prolog.

3. DOS ORGANIZATION

There are two approaches for organizing a DOS. The first approach replicates the DOS kernel in each processor. The second approach distributes the functions of the DOS to processors, where each processor has the facilities for handling absolutely essential functions such as interrupts, process interaction and communication.

Network-based distributed systems are one class of examples using the first approach, where the kernel is most of the DOS. DOSs such as Micros [5] and Roscoe [4] are also examples of the first approach but the kernel is not most of the DOS. In Micros, the kernel consists of runtime support for concurrent Pascal [18], support for message communication, and the key functions. In Roscoe [4], the kernel comprises link and message communication support, interrupt handler, timer manager, process management using a resource manager, and routing tables for messages.

Medusa [1] is a good example of the second approach to organize a DOS. StarOS [2] is another example. Medusa requires each processor in Cm* to contain an interrupt handler, task multiplexer, and device handler. Most of the DOS functions in Medusa are in five utilities that are distributed over several processors.

4. PROCESSOR ARCHITECTURE

A processor architecture with primitives for sending and receiving messages, acknowledging messages, and facilities for not receiving messages when space is not available is needed for interprocess communication using messages in the dataflow approach. The protocols discussed in chapter 2 for doing reconfiguration require five to seven message communications between modules to communicate the state information. If reconfiguration is to be done efficiently the interprocess communication must be done quickly.

There are very few commercially available processors [31,32] that support messages for communication between processes within a processor, as shown in [15]. However, shared memory techniques can be used to efficiently communicate messages between processes in different processors. This form of message communication is used in several DOSs. For example, LSI-11s have been interconnected using a special hardware called Kmap [3] in Cm* so that messages can be efficiently communicated between processes on different processors in Medusa [1] and StarOS [2]. The Micronet architecture [5] uses a frontend processor in each node to send and receive messages from processes in other nodes. The shared memory is used in the VAX-11/780 [33] to communicate messages between processes in different processors.

We have developed a shared memory multiprocessor architecture (shown in Figure 1.b) to support fast interprocess communication. It is intended for executing Prolog programs. The architecture contains a synchronizing memory and a crossbar memory. The synchronizing memory contains process table entries for creating and manipulating processes, join table entries for synchronizing the execution of processes, and other data structures needed for executing Prolog. Locks used in accessing write-sharable data items are also in the synchronizing memory. The crossbar memory contains the data space and most of the code space of processes created in the execution of a Prolog program. There is a single data address space for all processes. The processes use the address space in a cactus stack like structure. There is also a single code address space for all processes.

Each processing node in Figure 1.b has three processors. The parallel Prolog processor (PPP) executes application programs written in Prolog. The numeric processor (NP) executes functions requiring floating point operations and integer operations. The supervisor processor (SP) executes functions of the operating system. There is no loss of generality in using Prolog since programs written in Fortran, C, and Pascal can be called from Prolog programs using foreign functions.

5. DOS DESIGN EXAMPLE

The design of the kernel functions of a DOS using dataflow graphs (EDFGs) is presented in this section. Although it is possible to take any operating system and design it using EDFG (as shown in [26,27]), we have chosen a DOS kernel for several reasons. One reason is to show how reconfiguration can be done. The DOS kernel functions implied by the the architecture in Figure 1 and the parallel execution of Prolog

programs [34,35] are table initiation, decomposition, allocation, token management, memory management, measurement, and diagnostics. The kernel functions execute on SPs. The kernel functions communicate with several tables kept in the synchronizing memory. An EDFG of the kernel is shown in Figure 2. Several of the nodes in Figure 2 can be implemented using Prolog. The code associated with the nodes can be changed at runtime using the assert/retract mechanism of Prolog. This allows dynamic changes to policies and strategies used by the resource manager.

5.1 OVERVIEW OF KERNEL FUNCTIONS AND TABLES

Nodes representing the kernel functions and tables are briefly described and compared to parts of contemporary operating systems.

The table initiation function conveys information in tables to other functions of the kernel. It also supplies tabular form of EDFGs to dynamic nodes in EDFGs. The table initiation function is similar to "process control" part of contemporary operating system such as UNIX, MULTICS, HYDRA, MCP, OS, VM370, and VMS.

The decomposition function detects parallelism implicit in Prolog programs by identifying nodes that can be fired simultaneously. It also dynamically expands nodes, i.e. creates multiple instances of a node. This function uses the enabling conditions, represented as a firing semantics set (FSS), associated with each node to determine whether the node can be enabled. Contemporary operating systems and existing "distributed operating systems" [1-6] do not have a function similar to that of decomposition. This is because program decomposition is done statically by the programmer and detected by the language processor.

The allocation function assigns nodes to processors, maintains the status of processors, and reassigns nodes as the computation dictates. It uses the information gathered by the measurement function and heuristics in reassigning nodes to processors. This function is similar to the "scheduling" and "reconfiguration" function in contemporary operating systems.

The token management function constructs and distributes tokens. The distribution activity consists of sending tokens to processors, receiving tokens from processors, updating tables, and sending tokens that contain nodes to decomposition functions. The token management function is similar to the "interprocess communication" part of contemporary operating systems.

The memory management function determines when to make copies of tokens and when to allow multiple reads on tokens. It also maintains tables, facilitates the movement of tokens to output devices, and the usual functions of memory management. It also facilitates node reassignment by communicating with the allocation function and the diagnostics function. This function is similar to the "memory management" function in operating systems that support virtual memory concept.

The measurement function collects metering information from nodes, arcs, processors, and devices. It computes statistics on the number of times a node has fired, average firing time, average arc capacity, and utilization of resources. The measurement function is used by the allocation function in assigning nodes to processors so that token movement is reduced. The measurement function is similar to the measurement facilities in MULTICS, VMS, OS and VM370.

The diagnostics function detects and locates faulty processors, and facilitates the reassignment of nodes to healthy processors. This function provides graceful degradation in the performance of a distributed computer system when hardware and software faults appear. Very few commercially available operating systems have functions similar to the diagnostics function.

The NODE_TABLE contains values for the static attributes of nodes in EDFGs that are to be executed. The NODE_INFO_TABLE contains values for the attributes of nodes that are dynamic and implementation related. The ARC_TABLE contains values for the attributes of arcs and implementation details. The PROCESSOR_NODE_CORRESPONDENCE_TABLE contains the status of processors and the nodes assigned to them.

The above tables and others are initially supplied with values by the table initiation function. Each table has a manager for receiving tokens, adding entries to the table, sending tokens to other functions, and maintaining the table in memory. Each table and its associated manager are represented as a node. Although these nodes need not be part of the kernel, they are included in Figure 2 for the sake of completeness.

The EDFG in Figure 2 is a high level description of a DOS kernel. Each node corresponds to a kernel function or a table and has a firing semantics set (FSS) specifying the conditions under which the node can be enabled for firing. The FSS of nodes corresponding to the kernel functions are shown in Figure 3. At any time several of these nodes can be firing simultaneously provided the FSS conditions can be met. The arcs connecting nodes and the arc attributes show the communication paths (arc direction) between kernel functions, message formats (token type), buffer space (arc capacity), number of full buffers (current number of tokens), and average time to move buffers (arc latency time). The details of a kernel function are in the operation attribute of the node corresponding to the kernel function. The details can be specified as an EDFG. This activity can be repeated any number of times until the nodes have operations that are primitives. The above property of EDFG is called the hierarchical decomposition structure. Each step at which the operations of nodes in an EDFG are specified as EDFGs is called a refinement of the EDFG. Several of the nodes corresponding to the kernel functions in Figure 2 are refined one time to show how EDFGs can describe the details of the kernel. The refinement also shows the modules of the DOS kernel functions and the resource managers of the DOS. After this refinement, the details of nonprimitive nodes are described in words. This is done primarily to make the report readable. The data type of most of the tokens flowing between nodes is a structure. In this paper, tokens are synonymous to messages.

5.2 DECOMPOSER

The DECOMPOSER of the kernel identifies nodes in an EDFG that can be executed in parallel. A refinement of DECOMPOSER is shown in Figure 4. The DECOMPOSER comprises an INITIATOR, NODE_RECEIVER, up to m ANALYZERS, m >= 1, an ANALYZER_MANAGER, ACTIVATOR, and several primitive nodes with the operation FUNNEL. The node ANALYZER_MANAGER is a CSN and the ANALYZER nodes are RRNs.

INITIATOR

The INITIATOR node is responsible for starting the decomposition function and the beginning of the execution of any EDFG that entered the system using the TABLE_INITIATOR. For every node in an EDFG, the INITIATOR enters into the NODE_INFO_TABLE the user supplied details relating to node operation, I/O arc specifications and associated FSS, and tagging scheme to be used by the nodes in the EDFG. It removes nodes from the INPUT_NODE_TABLE, sends a message to the MEMORY_MANAGER to update NODE_NAME_TABLE and ARC_NUMBER_TABLE after receiving input and initial tokens, and sends the nodes to NODE_RECEIVER. The INITIATOR also computes values for implementation related parameters such as current and maximum enabling counts on input or output arcs of nodes in EDFGs and enters them into NODE_INFO_TABLE.

NODE_RECEIVER

The NODE_RECEIVER receives on its input arcs tokens containing nodes sent by the TOKEN_MANAGER and INITIATOR. It also receives tokens containing information on the space availability of output arcs. The nodes received by a NODE_RECEIVER are those that have received at least one token from the execution of some nodes, nodes that have received tokens from the external environment, or nodes with initial token. The NODE_RECEIVER sends each of the received tokens to the ANALYZER_MANAGER. The NODE_RECEIVER continues to receive tokens as long as its input arc capacities are not exceeded. When the arc capacities are reached, tokens are not sent to the DECOMPOSER by TOKEN_MANAGER.

ANALYZER_MANAGER

The ANALYZER_MANAGER is a CSN. It communicates with a number of ANALYZERs, and ANALYZER_STATUS_TABLE and the PENDING_NODES. The ANALYZER_MANAGER attempts to assign an ANALYZER to each node sent by the NODE_RECEIVER or the PENDING_NODES. It performs the assignment by communicating with the ANALYZER_STATUS_TABLE. The status of each ANALYZER is maintained by the ANALYZER_STATUS_TABLE node. If the

ANALYZER_MANAGER sends a token requesting an ANALYZER then the ANALYZER_STATUS_TABLE sends the capability [2] to an available (free) ANALYZER provided one exists. The nodes that require space on their output arcs before they can be enabled for firing are sent to PENDING_NODES. The ANALYZER_MANAGER sends to PENDING_NODES tokens that are received from NODE_RECEIVER containing information on the space availability of output arcs of nodes. Nodes in PENDING_NODES that have received space on their output arcs are sent to the ANALYZER.

The ANALYZER_MANAGER also changes the operation performed by an ANALYZER. For each ANALYZER, there are two operations implementing two different analysis strategies. The ANALYZER_MANAGER sends the operation corresponding to strategy 1 as the value for the operation attribute of an ANALYZER. On receiving a reconfiguration request from an ANALYZER the ANALYZER_MANAGER sends Strategy 2 as the value for the operation attribute of ANALYZER.

ANALYZER

Each ANALYZER performs an analysis on the node supplied to it to determine whether that node can be enabled for firing or the node has to be expanded (Dynamic nodes). For a dynamic node the ANALYZER obtains the capability to the file containing the EDFG and sends the capability to the TABLE_INITIATOR. The ANALYZER sends a message to the TOKEN_MANAGER to collect the details of the node from the NODE_TABLE and keeps it in the level memory. The ANALYZER then gathers the dynamic node attributes from the NODE_INFO_TABLE. It performs the analysis using one of the following approaches specified as the value for the operation attribute of the ANALYZER.

STRATEGY 1

This strategy is used if the node to be analyzed employs the standard firing rule, that is a token must be available on all input arcs.

The ANALYZER checks the current enabling count on input arcs of the node it is working on. If the count is not zero, that is one or more input arcs do not have tokens, the ANALYZER ends the analysis. Otherwise, if the enabling count on the output arcs is not zero, that is one or more output arcs have no space, the node is entered into a waiting line in the PENDING_NODES. This is done by sending a token to the ANALYZER_MANAGER. The ANALYZER then ends analysis.

STRATEGY 2

This strategy is used if the node to be analyzed employs the nonstandard firing rule. A pattern matching scheme is used to check the enabling conditions of the node.

An input/output pattern of a node shows the input arcs that must have tokens and the output arcs that must have space available to enable the node for firing. For a node there may be a set of such input/output patterns, called FSS.

If the pattern of tokens on the input arcs of a node and the pattern of available space on the output arcs of the node matches an entry in the FSS, then the ANALYZER recognizes the node as an enabled one. The enabled node and its input arcs are sent to the ACTIVATOR. Else, the ANALYZER sends a token containing the node to the ANALYZER_MANAGER for entry into the waiting line in PENDING_NODES and ends analysis.

ACTIVATOR

The ACTIVATOR receives the node and input arc descriptions sent by ANALYZERS. For each node, the ACTIVATOR sends the node name and input arc numbers to the ALLOCATOR.

5.3 ALLOCATOR

The ALLOCATOR assigns nodes in EDFGs to processors by communicating with the SCHEDULER. The nodes are in the tokens sent by the ACTIVATOR in the DECOMPOSER. The SCHEDULER maintains the PROCESSOR_NODE_CORRESPONDENCE_TABLE. On receiving a token from the ALLOCATOR, the SCHEDULER communicates with the PROCESSOR_NODE_CORRESPONDENCE_TABLE to reserve an available processor and if successful it sends back the processor id to the ALLOCATOR. If no healthy processor is available at the level, the SCHEDULER sends the token to the ALLOCATOR at the next level for possible assignment to a processor. The states of a processor can be healthy and available, healthy and non-available, or faulty. Initially, the PROCESSOR_NODE_CORRESPONDENCE_TABLE has the processor id's of all the healthy processors in the column for processor id and no entry for node names. The states of healthy processors are initially available. Periodic diagnostics are run on the processor by making a transition to the healthy non-available states.

The ALLOCATOR sends the node name, tag, input arc numbers of the node, and a processor id to the TOKEN_MANAGER so that tokens can be supplied to the processor. The time at which a processor starts executing its assigned operation is determined by the interconnection network between the processors, and the TOKEN_MANAGER. The time duration for which a node remains assigned to a processor depends on the nature of the node. The nodes in a loop, recursive EDFG, and nodes receiving streams of tokens will usually remain assigned to processors as long as some of the designated arcs contain tokens. The mechanism that facilitates the above is an involved one and its description is beyond the scope of this report. Note that a node is assigned to a processor until it completes firing or a fault is detected in the processor. The presence of a fault in a processor results in reassigning the node to a healthy and available processor. This is done using a communication protocol between the processor and the ALLOCATOR, MEMORY_MANAGER, and DIAGNOSTICS_PACKAGE. The modules of the DOS kernel functions that perform the automatic reassignment of nodes is described in subsequent sections.

Another approach to design the ALLOCATOR is by using heuristics that reduce the token movement between processors. This approach requires empirical data before devising the heuristics.

5.4 TOKEN_MANAGER

A TOKEN_MANAGER retrieves tokens from memory using the descriptor in the ARC_TABLE, NODE_TABLE, or NODE_INFO_TABLE. It constructs tokens and sends tokens to processors. It stores tokens generated by processors using the descriptors in the ARC_TABLE, NODE_TABLE, and NODE_INFO_TABLE in memory. A refinement of the TOKEN_MANAGER is shown in Figure 5. A TOKEN_MANAGER consists of a COLLECTOR, several CONSTRUCTORS, a CONSTRUCTOR_MANAGER, several COMMUNICATORS, and a COMMUNICATOR_MANAGER.

COLLECTOR

The COLLECTOR function receives node names from the ANALYZER function of the DECOMPOSER. For each node description received, the details are retrieved from NODE_INFO_TABLE by using the MEMORY_MANAGER, if the details are not already in the memory.

CONSTRUCTOR

Each CONSTRUCTOR retrieves tokens from memory using ARC TABLE, organizes the tokens, and sends the tokens to the designated processors. The CONSTRUCTORs are maintained by a CONSTRUCTOR_MANAGER which assigns to each token received from the ALLOCATOR an available CONSTRUCTOR. The token received from the ALLOCATOR consists of node name, tag, input arc numbers, and a processor id.

A CONSTRUCTOR removes tokens with the proper tag from memory using the arc descriptions in the ARC_TABLE. The arcs correspond to the list of input arc numbers in the token received from the ALLOCATOR. The CONSTRUCTOR decreases the current number of tokens on the arcs by one, and updates the implementation details of arcs. The CONSTRUCTOR also decrements the current enabling count on output arcs of nodes acting as source for the above mentioned arcs. A CONSTRUCTOR prepares

a token, called instruction packet, containing the instructions of the node and data for executing the instructions. It sends the instruction packet to the interconnection network and then changes its state to available for assignment by the CONSTRUCTOR_MANAGER to another token received from the ALLOCATOR.

COMMUNICATOR

A COMMUNICATOR updates ARC_TABLE and NODE_INFO_TABLE with the results generated by processors. It also communicates the effects of the results to the DECOMPOSER. It also communicates with the PROCESSOR_NODE_CORRESPONDENCE_TABLE so that nodes in recursive EDFGs and nodes receiving streams of tokens [12] will remain assigned to the processors. Results that have to be communicated to the external environment (output) are temporarily stored in the memory using the OUTPUT_ARC_TABLE.

COMMUNICATORs are maintained by a COMMUNICATOR_MANAGER that assigns to each token received from the interconnection network for processors an available COMMUNICATOR. The COMMUNICATOR_MANAGER is a CSN. Using snoop arcs it gathers the current number of tokens on the input arcs of NODE_RECEIVER module in the DECOMPOSER. If space is not available for tokens in the input arcs mentioned above, then the COMMUNICATOR_MANAGER does not assign tokens to COMMUNICATOR until space becomes available.

5.5 DIAGNOSTIC_PACKAGE

A DIAGNOSTIC_PACKAGE receives information on faulty processors, and initiates the node reassignment process. It also removes the faulty processors from the PROCESSOR_NODE_CORRESPONDENCE_TABLE, schedules faulty processors for repair, and adds repaired processors to the PROCESSOR_NODE_CORRESPONDENCE_TABLE. A refinement of DIAGNOSTIC_PACKAGE is shown in Figure 6 and it consists of an EXERCISER, a RECOGNIZER, a LOCATOR, and a CONFIGURATOR.

EXERCISER

The EXERCISER periodically exercises processors by sending messages containing diagnostic programs. These programs are executed by the processor when it is not doing any useful computation (idle). The diagnostic programs are usually not in level memory and are brought in by using the MEMORY_MANAGER. During the periodic diagnosis, the status of a processor is set to non-available in the PROCESSOR_NODE_CORRESPONDENCE_TABLE.

RECOGNIZER

The RECOGNIZER receives fault information sent by processors, and analyzes the tokens sent by idle processors after they have executed diagnostic programs. If the data in the token sent by a processor indicates faults, then the status of the processor is changed to faulty by communicating with the PROCESSOR_NODE_CORRESPONDENCE_TABLE. If a node has already been assigned to the processor then the RECOGNIZER communicates with the ALLOCATOR and the MEMORY_MANAGER to reassign the node to an available processor. The RECOGNIZER sends a token to the faulty processor indicating the change in processor and a token to the LOCATOR containing the id of the faulty processor. It is assumed that only a small fraction of the processors at any level are faulty at any time. Note that the EXERCISER and the RECOGNIZER prevent the propagation of faults in a processor to other processors. They also prevent errors encountered during a node firing from affecting nodes at other levels. In this way, each level acts as a check point in the execution of EDFGs. The EXERCISER and the RECOGNIZER at the various levels make the organization of processors appear as a fault tolerant computer system in executing EDFGs.

LOCATOR

The LOCATOR attempts to locate faults in the faulty processor and schedules repairs off line. The repaired processors are brought into service by entering their id's in the PROCESSOR_NODE_CORRESPONDENCE_TABLE. If faults are not locatable the faulty processor is marked for replacement. The LOCATOR also maintains a log of all the repairs performed on processors in the system memory. The MEMORY_MANAGER is used to update this log and to output the log.

CONFIGURATOR

The CONFIGURATOR detects faulty SPs, removes faulty SPs, schedules them for repair, and maintains a SP_TABLE. The SP_TABLE is an extension of a PROCESSOR_NODE_CORRESPONDENCE_TABLE with a column for SP numbers and status, a column for the kernel function names, and a column for the utilization and characteristics of the SPs.

At start-up time, the CONFIGURATOR allocates functions of the kernel to SPs and makes entries in the SP_TABLE. Since each of the functions of the kernel itself is an EDFG, the nodes of the kernel functions are assigned to SPs. The SPs are periodically exercised by sending tokens containing diagnostic programs. These programs are executed by the SPs after completing their present task and before starting the next task. After executing a diagnostic program, a SP sends a token containing the result of the diagnostic program to the CONFIGURATOR.

The CONFIGURATOR receives the token and if the data in the message indicates faults, then the status of the SP is changed to faulty. The node is reassigned to a spare SP, if one is available, or allowed to time share a SP that is not heavily used. The faulty SP is then scheduled for repair. If no fault is detected by the CONFIGURATOR, the SP is allowed to resume its operation. This is accomplished by sending a "clear" token to the SP. A repaired SP is brought back into operation by changing the status of the SP to healthy in the SP TABLE.

A SP sends an "error" token to the CONFIGURATOR if a fault is detected during the execution of an operation. The CONFIGURATOR changes the status of the SP to faulty, assigns the node to a healthy SP, and sends a token containing the node and the data to the healthy SP.

6. IMPLICATIONS OF THE PROPOSED ARCHITECTURE

The architecture in Figure 1 has several implications on the field of DOS design. The implications can be grouped into the following areas:

a. Mechanisms,

b. Modularity,

c. Robustness,

d. Protection, and

e. Organization.

A brief discussion on each of the areas is included.

The DOS kernel outlined in Section 5 shows that token communication is the key issue and not processor sharing since each processor executes an assigned task to completion unless faults occur. So, task allocation strategies that reduce token traffic have to be devised. The distribution of data and the copying semantics of data complicates memory management. Heuristics that decide when to replicate data and when to allow multiple reads have to be devised. These and other related issues such as measurement are expected to play key roles in future DOS design.

Each processor in the proposed multiprocessor architecture executes nodes with operations at the instruction level or block level in a data driven manner. So, the architecture implies modularity in DOS design. Changes to a DOS can be readily carried out by replacing one or more modules since the architecture supports single binding to a variable in a Prolog clause.

The proposed architecture implies robustness. Each processor reassigns nodes to a healthy processor if a fault is detected. The diagnostics function, discussed in Section 5, provides the necessary protocol to do the reassignment. If one or more SPs fail, the result will be a degradation in DOS performance and not a system crash.

Since operating system programs execute on SPs, the execution of application programs cannot corrupt the operating system, and vice versa. So, protection issues will play a minor role in DOS design since the architecture implies protection.

7. REFERENCES

1. J. K. Ousterhout, D. A. Scelza, and P. S. Sindhu, "Medusa: An experiment in distributed operating system structure", Communications of the ACM (CACM), Vol. 23, No. 2, Feb. 1980, pp 92-105.

2. A. K. Jones, et al, "StarOS, a multiprocessor operating system for the support of task forces", Proceedings of the 7th Annual Conference on Operating Systems, 1979, pp 117-127.

3. R. J. Swan, S. H. Fuller, and D. P. Siewiorek, "Cm* - A modular, multi-microprocessor", Proceedings of the AFIPS Conference, AFIPS Press, Vol. 46, June 1977, pp 637-644.

4. M. H. Solomon, and R. A. Finkel, "The Roscoe distributed operating system", Proceedings of the 7th Annual Conference On Operating Systems, 1979, pp 108-114.

5. L. D. Wittie, and A. M. Van Tilborg, "MICROS, A distributed operating system for MICRONET, a reconfigurable network computer", IEEE Transactions on Computers, Vol. C-29, No. 12, Dec. 1980, pp 1133-1143.

6. E. D. Jensen, "The Honeywell experimental distributed processor - An overview", Computer Magazine, Vol. 11, No. 1, Jan 1978, pp 28-37.

7. A. Van Wijngaarden, et al, "Revised report on the algorithmic language Algol 68", ACM SIGPLAN notices, Vol. 12, No. 5, May 1977, pp 1-70.

8. J. Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs", CACM, Vol. 21, No. 8, Aug. 1978, pp 613-641.

9. D. A. Adams, "A computation model with data flow sequencing", TR-117, Dec. 1968, CS Dept., School of Humanities and Sciences, Stanford University, CA.

10. R. M. Karp, and R. E. Miller, "Parallel program schemata", J. Computer System Sciences, Vol. 3, No. 2, May 1969, pp 147-195.

11. J. B. Dennis, and J. B. Fosseen, "Introduction to data flow schemas", Computation Structures Group Memo: 81, Project MAC, MIT, Cambridge, MA, 1973.

12. Arvind, and K. P. Gostelow, "The Id Report: An asynchronous language and computing machine", Dept. of Computer and Information Science, TR-114, Sept. 1978, UC Irvine, CA.

13. P. H. Enslow Jr., "What is a distributed data processing system?", Computer, Jan. 1978, pp 13-21.

14. V. P. Srini, and B. D. Shriver, "Abstract dataflow protocol for communication in distributed computer systems", Proceeding of COMPCON 1980, Washington D.C., Sept. 1980, pp 321-332.

15. H. C. Lauer, and R. M. Needham, "On the duality of operating system structures", Proceedings of

Second International Symposium on Operating Systems, IRIA, Oct. 1978.

16. P. Brinch Hansen, "Distributed Processes: A concurrent programming concept", Communications of the ACM (CACM), Vol. 21, No. 11, Nov. 1978, pp 934-941.

17. C. A. R. Hoare, "Monitors: An operating system structuring concept", Vol. 17, No. 10, Oct. 1974, pp 549-557.

18. P. Brinch Hansen, The Architecture of Concurrent Programs, Prentice-Hall Inc., Englewood Cliffs, N.J., 1977.

19. J. A. Feldman, "High level programming for distributed computing", CACM, Vol. 22, No. 6, June 1979, pp 353-368.

20. P. Brinch Hansen, "The nucleus of a multiprogramming system", CACM, Vol. 13, No. 4, April 1970, pp 279-288.

21. C. A. R. Hoare, "Communicating sequential processes", CACM, Vol. 21, No. 8, Aug. 1978, pp 666-677.

22. B. W. Lampson, "Experience with processes and monitors in Mesa", CACM, Vol. 23, No. 2, Feb. 1980, pp 105-117.

23. P. Brinch Hansen, DOS Using Distributed Processes - Personal Communication, 1979.

24. D. H. Ingalls, "The Smalltalk-76 programming system design and implementation", Proc. of the 5th Annual ACM Symposium on Principles of Programming Languages, Jan. 1978, Tucson, Arizona.

25. Xerox Learning Research Group, "Smalltalk-80 System", Byte, Vol. 6, No. 8, Aug. 1981, pp 36-48.

26. M. R. Shekarchi, "The realization of a sample operating system in a dataflow programming language", MS Project, CS Dept., University of Southwestern Louisiana, Lafayette, LA, Aug. 1978.

27. D. T. Nguyen, "A dataflow model and semantic specification for the transformational unit; virtual access control unit and virtual machine monitor", Technical Report, CS Dept., University of Southwestern Louisiana, March 1979.

28. S. P. Landry, and B. D. Shriver, "A dataflow simulation research environment", Workshop on Data Driven Languages and Machines, Toulouse, France, Feb. 1979, pp x 1-15.

29. S. E. Madnick, and J. J. Donavan, Operating Systems, McGraw Hill Inc., New York 1974.

30. Arvind, and R. E. Thomas, "I-Structures: An efficient datatype for functional languages", MIT-LCS Report, Sept. 1980, MIT, Cambridge, MA.

31. D. R. Appelt, "Making it compatible and better: Designing a new high-end computer", Electronics, Oct. 1979, pp 131-136.

32. GEC 400 Computer, Central Processor Unit Nucleus, GEC Computers Limited, Hertfordshire, UK, Dec. 1977.

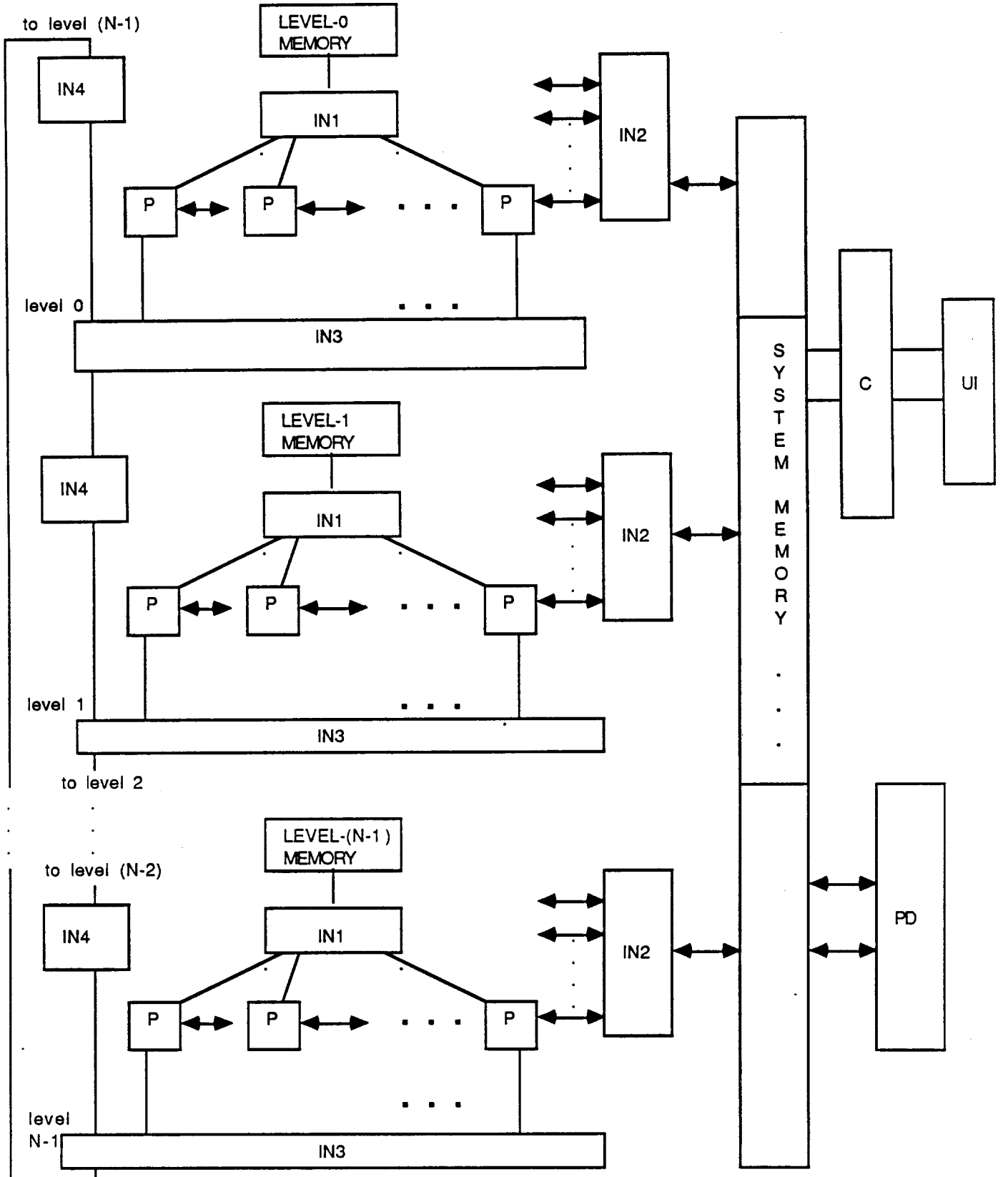
33. Digital Equipment Corp., VAX Hardware Handbook, Digital, Maynard, MA, 1981.

34. B. S. Fagin, and A. M. Despain, "Performance Studies of a Parallel Prolog Architecture", Proceedings of the 14th Intl. Symposium on Computer Architecture, Pittsburgh, PA, June 1987.

35. B. S. Fagin, "A Parallel Execution Model for Prolog", Ph.D Thesis, Technical Report No. UCB/CSD 87/380, Nov. 1987, CS Div., Univ. of California, Berkeley.

36. A. M. Despain, Y. N. Patt, V. P. Srini, et al., "Aquarius", Computer Architecture News, March 1987.

37. T. M. Nguyen, V. P. Srini, and A. M. Despain, "A Two-Tier Architecture for High-Performance Multiprocessor Systems", Proceedings of the International Conference on Supercomputers, St. Malo, France, July 1988.



IN - intereconnection network

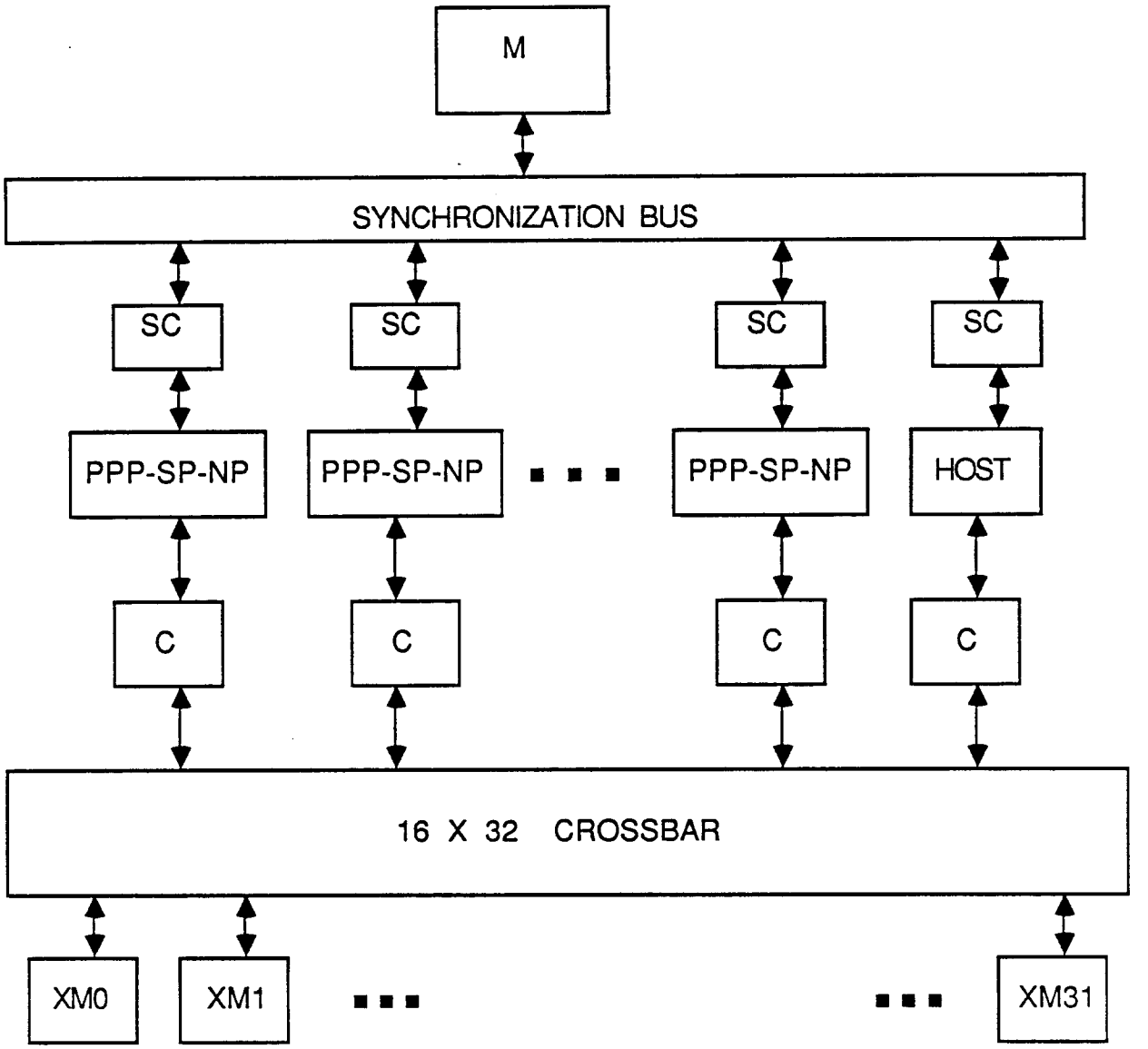
PD - Peripheral devices

UI - user interface

C - Compiler

P - processor

Figure 1.a
Organization of processors for executing EDFG



LEGEND

M ---- 64 M-Bytes of Synchronization Memory.

SC -- SNOOP CACHE, 4 M-Bytes of Data.

C ---- CACHE, 4 M-Bytes of Data and
4 M-Bytes of Instruction.

XMj -- 16 M-Byte Memory Module.

PPP-SP-NP -- Parallel Prolog Processor,
Supervisor Processor,
Numeric Processor.

Figure 1b Details of a Level

Legend :

IN1 - Interconnection network to level memory

IN2 - Interconnection network to system memory

IN3 - Interconnection network for processors

TM (i-1) - TOKEN_MANAGER at level (i-1)

TM (i+1) - TOKEN_MANAGER at level (i+1)

EXTL - Externally initiated diagnostics

IPL - Siganl to initiate the execution of kernel programs

OUTPUT - Operator console output

D (i+1) - Decomposer at level (i+1)

PD - Peripheral devices

 — Configuration specifying node

 — General node

 — Snoop arc

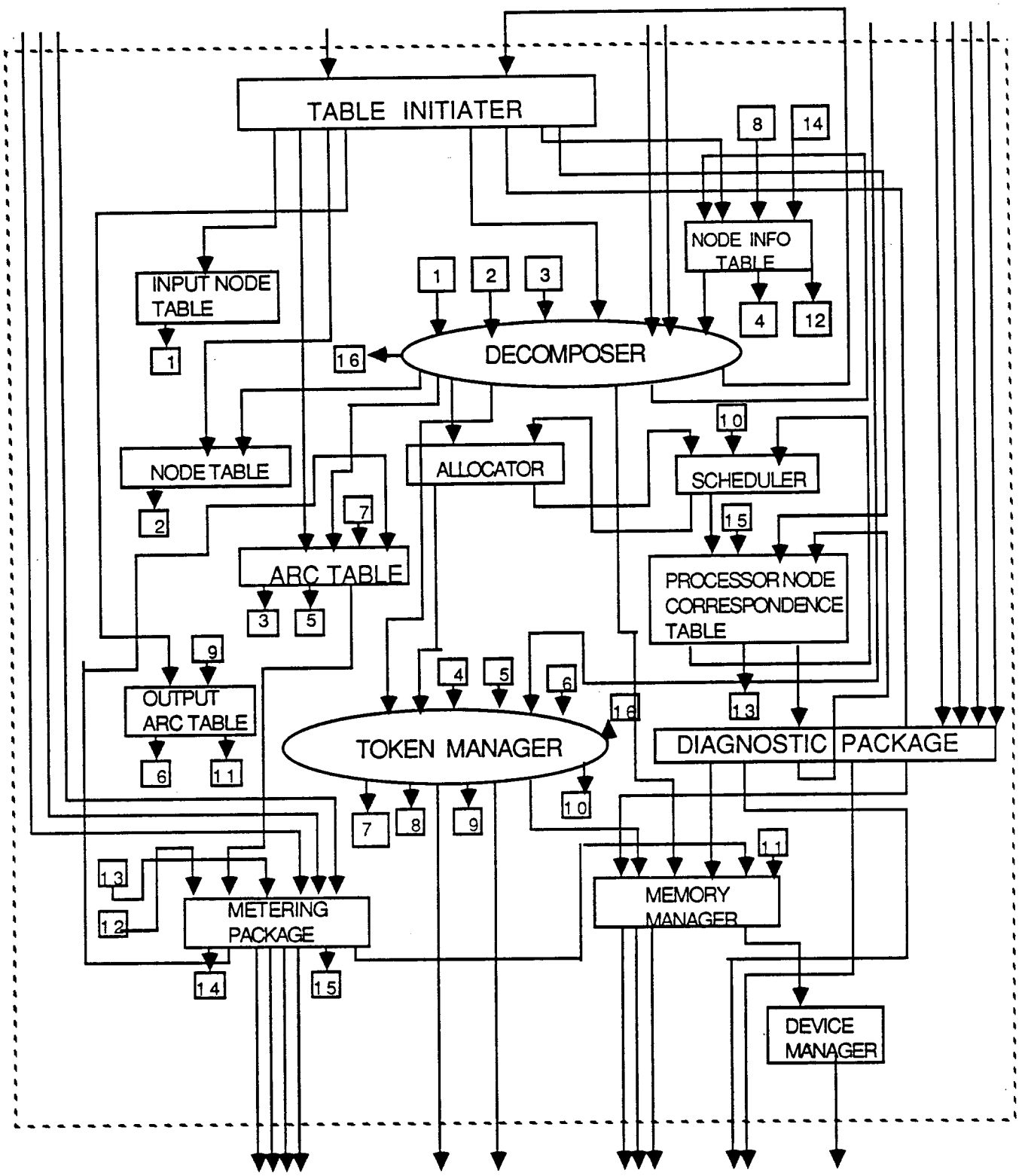


FIG 2 KERNEL FUNCTIONS

NOTATION

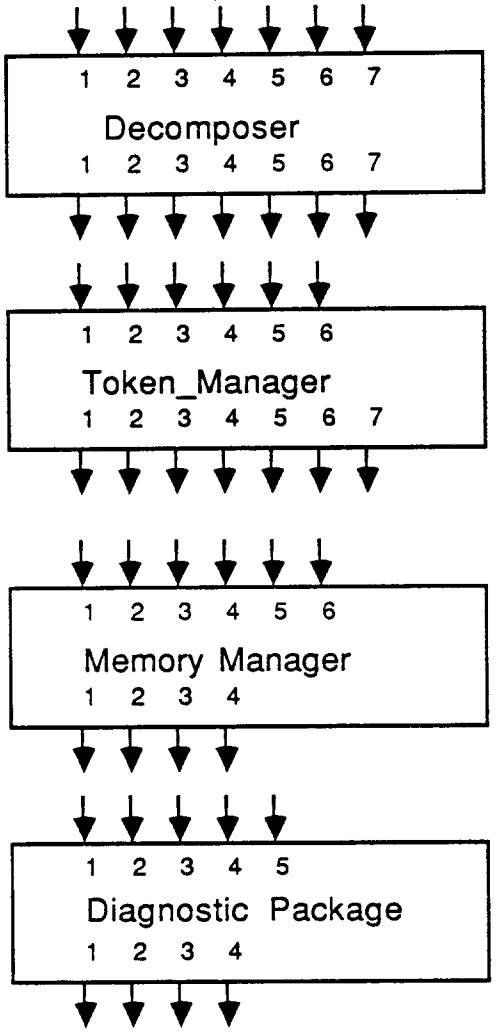
[] - List of input arcs that must receive tokens after node firing has been initiated or

list of output arcs that must have space available before node firing is completed

() - List of input arcs that must have tokens available for initiating node firing or

list of output arcs that must have space available for initiating node firing

& - Seperates conditions on input arcs (appears on left hand side of &) from output arcs (appears on right hand side of &)



{ (1) & (1|2), (2|3) & (6)
(4|5|6) & (6) , (7) & (3) }

{ (1) & (6), (2, [3,4]), (5|6) }

{ (1|2|3|4|5|6) }

{ (1) & (1,2), (2) & (3), (3), (4|5) }

Figure 3 Firing Semantics Sets of some of the nodes in Figure 2

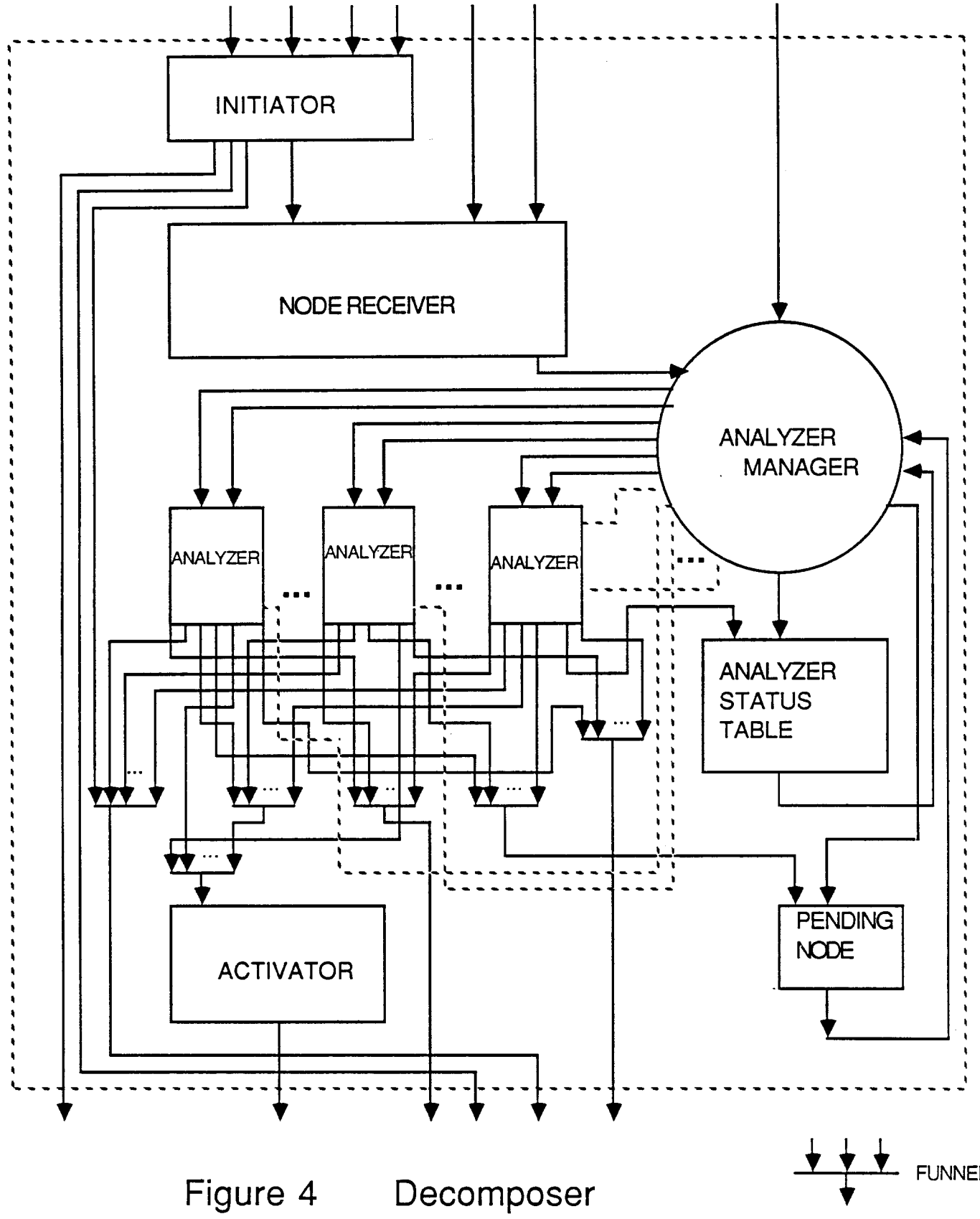


Figure 4 Decomposer

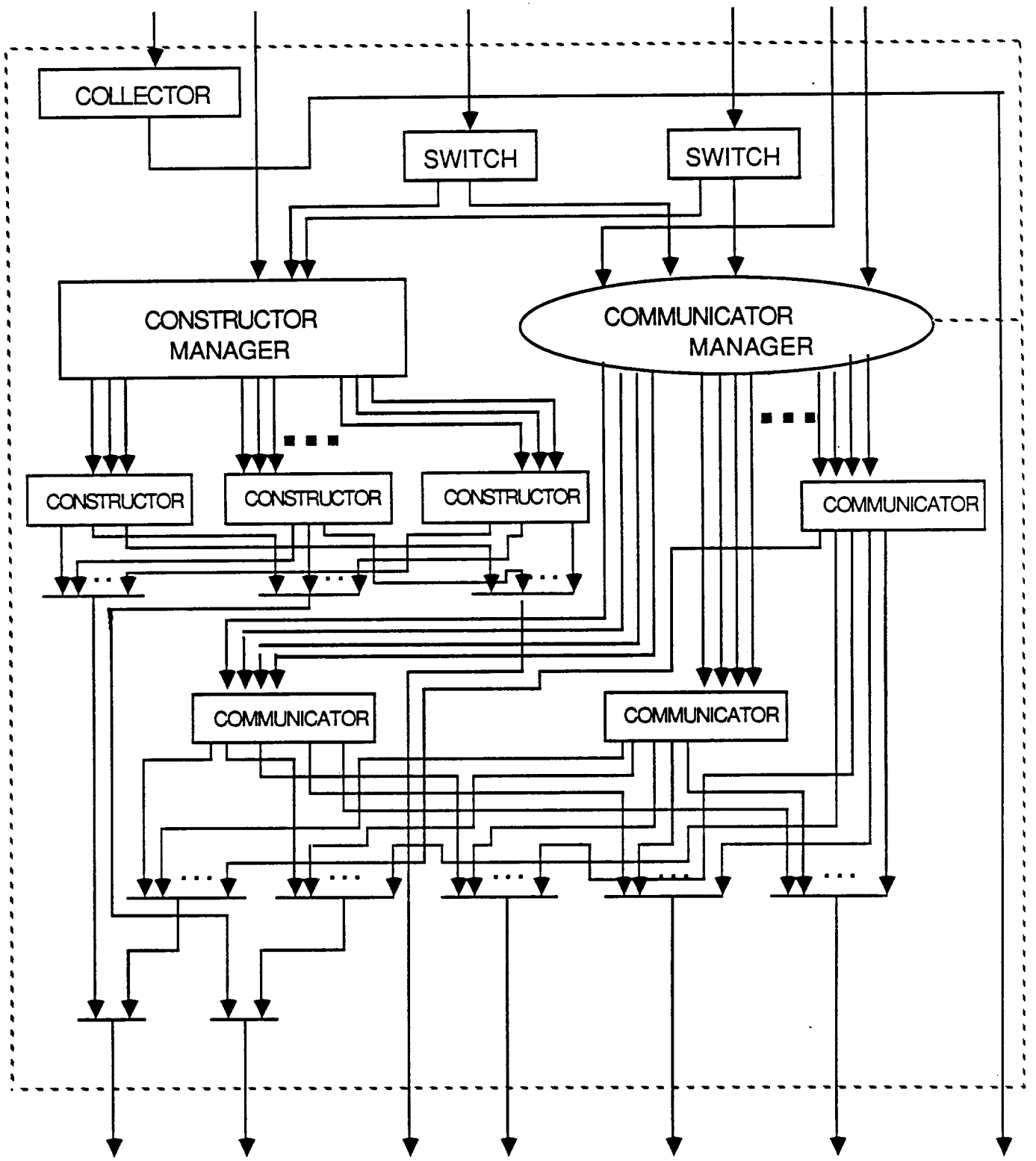


Figure 5 Token Manager

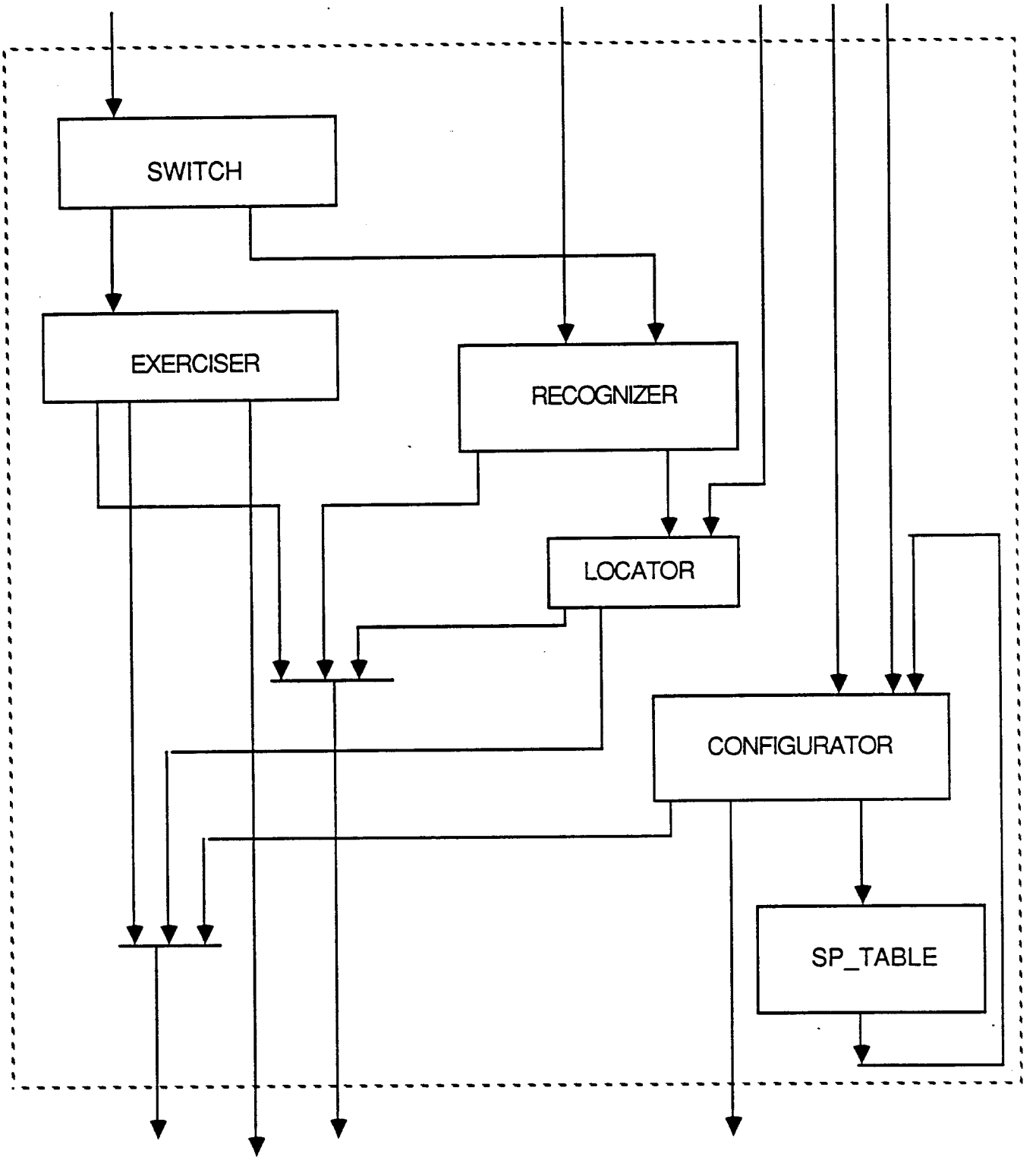


Figure 6 Diagnostic Package

Chapter 4. INTERPROCESS COMMUNICATION

The reconfiguration process described in Chapter 2 requires fast interaction between resource managers (CSNs) and various parts (nodes) of a system. In this chapter we show how the architecture in Figure 1.b of Chapter 3 can be used to achieve fast interprocess communication (IPC). The execution of Prolog programs with AND/OR parallelism on the architecture is used to describe the mechanisms that are employed to achieve fast IPC.

1. ADDRESS SPACE FOR PROCESSES

The architecture provides a data space and a code space for a Prolog program. Each space has 2**28 words. The data space is shared by all the AND processes and OR processes created during the execution of the Prolog program. The data space is divided into heap, stack, PDL, trail, and global heap (H2). The cactus stack of Figure 1 shows the sharing of the address space by the processes.

The global heap contains most of the information used in IPC. It stores process table entries, private memory of each process (hash windows), locks, join table, processor table, and other system tables. If the global heap can be accessed quickly by the processes then IPC will be fast. Assuming that the global heap space will be small for a Prolog program compared to the stack space or heap space, one good place to store it is the synchronizing memory. The probability of finding the item in the synchronization cache will be higher since no other space has to reside in the cache. To validate the assumption some experiments were performed on a simulator using Prolog programs. This is described in the next section.

2. MEMORY REFERENCES TO GLOBAL HEAP

A simulator for the parallel execution of Prolog programs with AND processes and OR processes, called PPP simulator [1], was developed with a single cycle memory and fixed amount of space for each process. The PPP simulator was used to show that for a set of Prolog benchmark programs the memory references to the global heap is small compared to the total number of memory references to data space. A brief description of the programs used is given in Table 1. The PPP simulator was modified to gather statistics on memory references to the stack, heap, trail, and global heap. The frequency with which memory reads and writes are done for the benchmark programs is shown in Table 2. The percentage of the read and write references to the global heap are shown in Tables 3 and 4 respectively. We have separated the read and write references to get some understanding on the memory writes from the cache to the synchronizing memory.

The benchmark programs have been run with AND processes and OR processes selectively invoked. The suffixes A and O indicate the features that are included in the simulated execution. The results in Table 2 show that when only AND processes are created for a program the percentage of the references to H2 is not very large. However, when OR processes are created, the percentage of references to H2 increases significantly. This is due to the increase in the number of references to process table entries and hash windows. The hash window of an OR process contains the bindings that can be used by other processes. Since the AND processes of an OR process share the hash window, the number of references goes up when the windows are searched. This is shown in Table 3. The sequential searching of hash windows and process table entries in the simulator also contributes to the high percentage of memory references.

3. PROCESSOR-CACHE INTERFACE

To support fast IPC a combination of hardware, microcode, and software mechanisms are used. One architecture proposal for the processor node of Figure 1.b in Chapter 3 is discussed in this section.

A processor node contains three processors, caches, and interfaces to the synchronization bus and crossbar as shown in Figure 2. The VLSI-PPP coprocessor will execute the AND/OR processes of Prolog. This coprocessor will be an extension of the VLSI-PLM [2, 3]. The supervisor processor (SP) will execute some of the key resource management functions such as memory management and Prolog builtins such as read and write. One of the commercially available microprocessors such as Motorola 68030 can be used as an SP. The numeric processor (NP) will execute floating point and vector operations used in Prolog programs. The architecture proposed by Yung [4] will be modified to function as an NP.

The interface signals needed for communication between the three coprocessors, synchronizing cache, and the crossbar cache are shown in Figure 2. These signals and the protocol used in communication are preliminary and may change as we make progress in the system building activity. The interface signals are used to show IPC.

3.1 DYNAMIC MEMORY MANAGEMENT

The protocol for communicating state information between the coprocessors for obtaining a page of memory is shown in Figure 3 to illustrate some of the hardware flags and signals used in IPC. The basic assumption is that the VLSI-PPP requests for a page of memory for use as stack or heap when page boundary is reached. This is shown by the setting of ENDP flag. The signals ENDOFPAGE and NEXTPAGE are used to obtain a page. If a page is not available the clock for the address unit of VLSI-PPP is frozen. The protocol is shown in Figure 3.a. The interface between the three coprocessors and the caches, called CONDUCTOR, uses the protocol in Figure 3.b to acquire a page. The PAGEAVAIL flag indicates that the CONDUCTOR has the next available page. If this flag is set and the ENDOFPAGE signal comes from VLSI-PPP the CONDUCTOR can immediately supply the page. Otherwise, the WAIT flag is set. It forces the request from VLSI-PPP to wait until a page is supplied by the SP. The CONDUCTOR protocol is designed to provide an available page without any delay. The SPBUSY indicates that the SP is busy and cannot be requested.

The task of obtaining an available page from a free page list is done by the SP. The protocol used by the SP to supply a page to the CONDUCTOR is shown in Figure 3.c. The PAGEFOUND flag indicates that a page is available and can be supplied immediately. If a page is not available the routine for obtaining an available page is started. The DONE flag is set when this routine succeeds in obtaining a page.

3.2 PROCESS CREATION

The protocol for creating a process requires an understanding of the the architecture of VLSI-PPP which will be based on the VLSI-PLM chip [2, 3]. It is assumed that the reader is familier with the details of the VLSI-PLM chip. The CONDUCTOR has a set of 8 registers, called communication registers (COMMREG), that shadows the argument registers of the VLSI-PLM. The COMMREG is loaded when the put instructions preceding the call_p instruction of the VLSI-PPP is executed.

The protocol for the VLSI-PPP to load the COMMREG and communicate the processor create request to SP is shown in Figure 4.a. The SHADOW flag of the VLSI-PPP indicates when the COMMREG can be loaded. If it is set then the loading of the argument registers AX[0] - AX[7] is shadowed. The signal LOADSHADOW tells the CONDUCTOR to load one of the COMMREG. The address of the register is supplied on the SHADOWADDR bus. On seeing a call_p instruction the VLSI-PPP sends the PROC_CREATE request to the CONDUCTOR (which in turn sends it to SP) and the number of arguments on SHADOWADDR bus. The setting and resetting of the SHADOW flag are done using SETSHADOW and RESETSHADOW signals. If the arguments cannot be communicated before the PROC_CREATE signal then a microroutine will be called to explicitly copy the contents of the AX registers to COMMREG.

The protocol needed by the CONDUCTOR to support process creation is shown in Figure 4.b. It is assumed that each register in the COMMREG has a valid bit. This bit is set when the corresponding register is loaded and reset when the register is read out. On receiving a PROC_CREATE signal the CONDUCTOR checks the COMMREG to make sure that all the arguments are available. If so then it resets SHADOW flag in the VLSI-PPP until the SP has actually created a process and the arguments have been sent to the processor that will run the process. The SHADOW flag is set and the VLSI-PPP can start loading the COMMREG. Note that while the SHADOW flag is reset the VLSI-PPP can continue to run programs and load the registers AX[0] - AX[7] but they will not be shadowed in the CONDUCTOR.

The SP's protocol for supporting process creation is shown in Figure 4.c. It starts executing the routine that creates a process on receiving the PROC_CREATE_SP signal from the CONDUCTOR.

4. REFERENCES

1. B. S. Fagin, and A. M. Despain, "Performance Studies of a Parallel Prolog Architecture", Proceedings

of the 14th Intl. Symposium on Computer Architecture, Pittsburgh, PA, June 1987.

2. V. P. Srini, et al , "A CMOS Chip for Prolog" Proceedings of the Intl. Conference on Computer Design, Rye Brook, New York, Oct. 1987, pp 605 - 610.

3. V. P. Srini, et al, "Design and Implementation of a CMOS Chip for Prolog" Technical Report No. UCB/CSD 88/412, March 1988, CS Division, University of California, Berkeley.

4. R. Yung, "The Aquarius Numeric Processor", Technical Report No. UCB/CSD 88/418, June 1988, CS Division, University of California, Berkeley.

Table 1. Description of Benchmark Programs

Benchmark	Lines of PPP code	Description
ckt4	366	circuit design of 2-input Mux
deep_bak	160	tests deep backtracking
diff	447	symbolic differentiation (several)
divide	270	symbolic differentiation [1/(x^n)]
hang	272	a course schedule for students
knight	464	knight's tour of chess board
maze	725	finding a path through a maze
mumath	218	Hofstader's mu math (theorem prover)
queens	289	safely placing queens on chessboard
query	525	a small database search
times	271	symbolic differentation [x^n]

Tabe 2. Memory Reads and Writes

Benchmark	Heap	Stack	Trail	PDL	Sync
deep_bakA	3.8%	77.9%	0.0%	0.0%	18.3%
mazeO	7.7%	66.2%	0.3%	1.1%	24.6%
knight0O	20.0%	55.7%	0.9%	2.9%	20.4%
diffA	34.7%	51.2%	1.1%	0.0%	13.0%
ckt4A	30.9%	49.5%	8.3%	0.0%	11.3%
mumathO	12.9%	43.1%	0.5%	0.6%	43.0%
queensO	13.6%	39.4%	1.0%	0.0%	46.0%
ckt4O	22.6%	33.7%	5.9%	0.0%	37.8%
queryO	26.8%	31.1%	6.8%	0.0%	35.3%
times10A	12.8%	30.2%	0.8%	0.0%	56.2%
divide10A	16.0%	29.4%	0.7%	0.0%	53.9%

MEMORY WRITES

Benchmark	Heap	Stack	Trail	PDL	Sync
deep_bakA	2.5%	14.1%	0.0%	0.0%	9.9%
mazeO	1.3%	10.9%	0.2%	0.7%	6.8%
knight0O	6.0%	20.2%	0.6%	1.8%	2.0%
diffA	22.8%	31.2%	1.1%	0.0%	7.3%
ckt4A	12.6%	13.7%	4.1%	0.0%	5.6%
mumathO	5.1%	20.1%	0.2%	0.4%	16.2%
queensO	6.0%	16.4%	0.5%	0.0%	17.6%
ckt4O	9.2%	8.8%	2.9%	0.0%	1.2%
queryO	11.2%	5.8%	3.4%	0.0%	7.8%
times10A	8.9%	19.1%	0.8%	0.0%	28.8%
divide10A	11.6%	18.3%	0.7%	0.0%	27.6%

Table 3. Breakdown of Global Heap (H2) Reads

H2 READS (% over H2 R+W)			
Benchmark	Window	Join Tbl	Proc Tbl
deep_bakA	0 0.0%	20 2.1%	418 44.0%
mazeO	28343 50.9%	0 0.0%	12007 21.6%
knight0O	17505 82.1%	0 0.0%	1705 8.0%
diffA	0 0.0%	4 1.2%	146 42.8%
ckt4A	0 0.0%	378 0.5%	36150 49.5%
mumathO	9055 31.2%	0 0.0%	9029 31.1%
queensO	3957 32.3%	0 0.0%	3601 29.4%
ckt4O	431916 93.7%	0 0.0%	14603 3.2%
queryO	19045 72.3%	0 0.0%	1505 5.7%
times10A	0 0.0%	18 1.4%	614 47.4%
divide10A	0 0.0%	18 1.4%	614 47.4%

Table 4. Breakdown of H2 Writes

H2 WRITES (% over H2 R+W)			
Benchmark	Window	Join Tbl	Proc Tbl
deep_bakA	0 0.0%	10 1.1%	503 52.9%
mazeO	1198 2.2%	0 0.0%	14123 25.4%
knight0O	137 0.6%	0 0.0%	1970 9.2%
diffA	0 0.0%	3 0.9%	188 55.1%
ckt4A	0 0.0%	281 0.4%	36238 49.6%
mumathO	568 2.0%	0 0.0%	10383 35.8%
queensO	538 4.4%	0 0.0%	4150 33.9%
ckt4O	2158 0.5%	0 0.0%	12455 2.7%
queryO	4100 15.6%	0 0.0%	1692 6.4%
times10A	0 0.0%	9 0.7%	654 50.5%
divide10A	0 0.0%	9 0.7%	654 50.5%

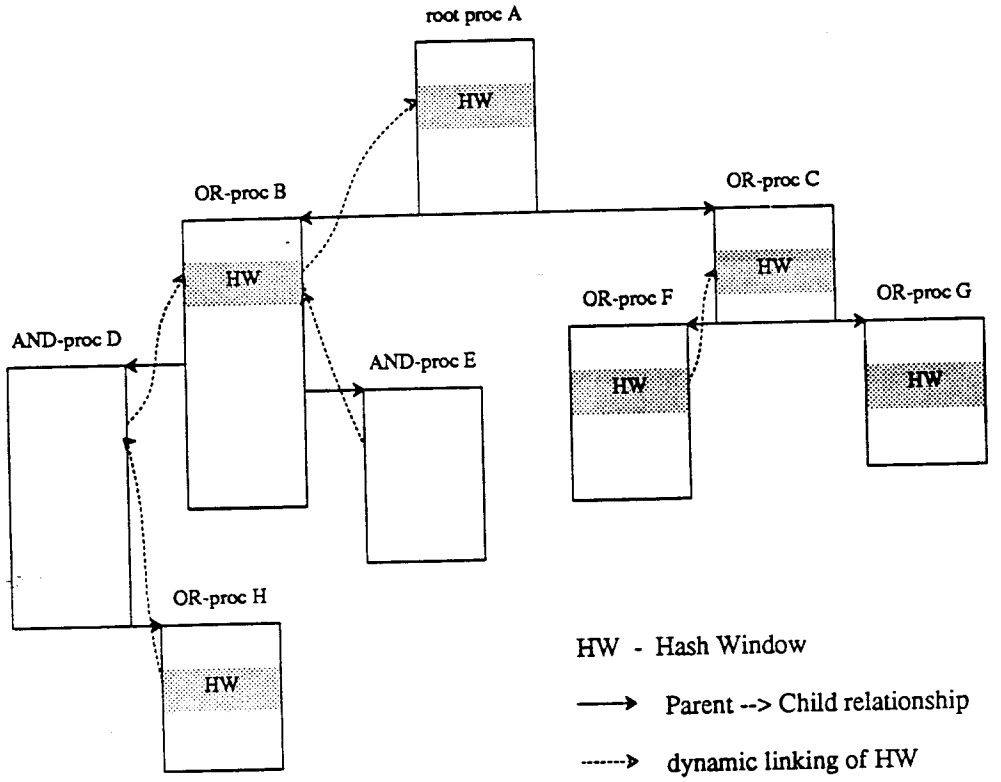


Figure 1. Cactus Stack Model of AND/OR Prolog Processes

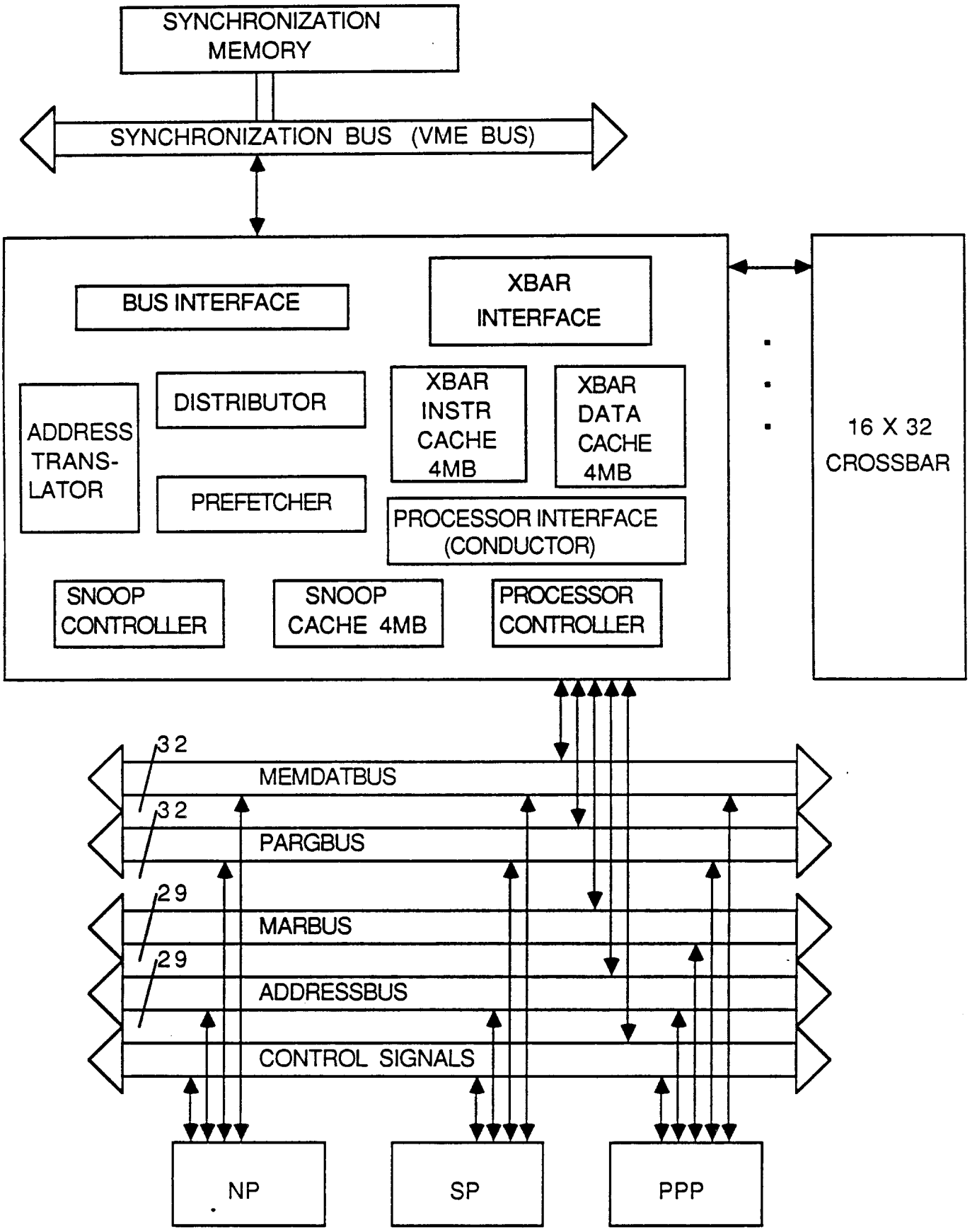


Figure 2 -- A Processor Node

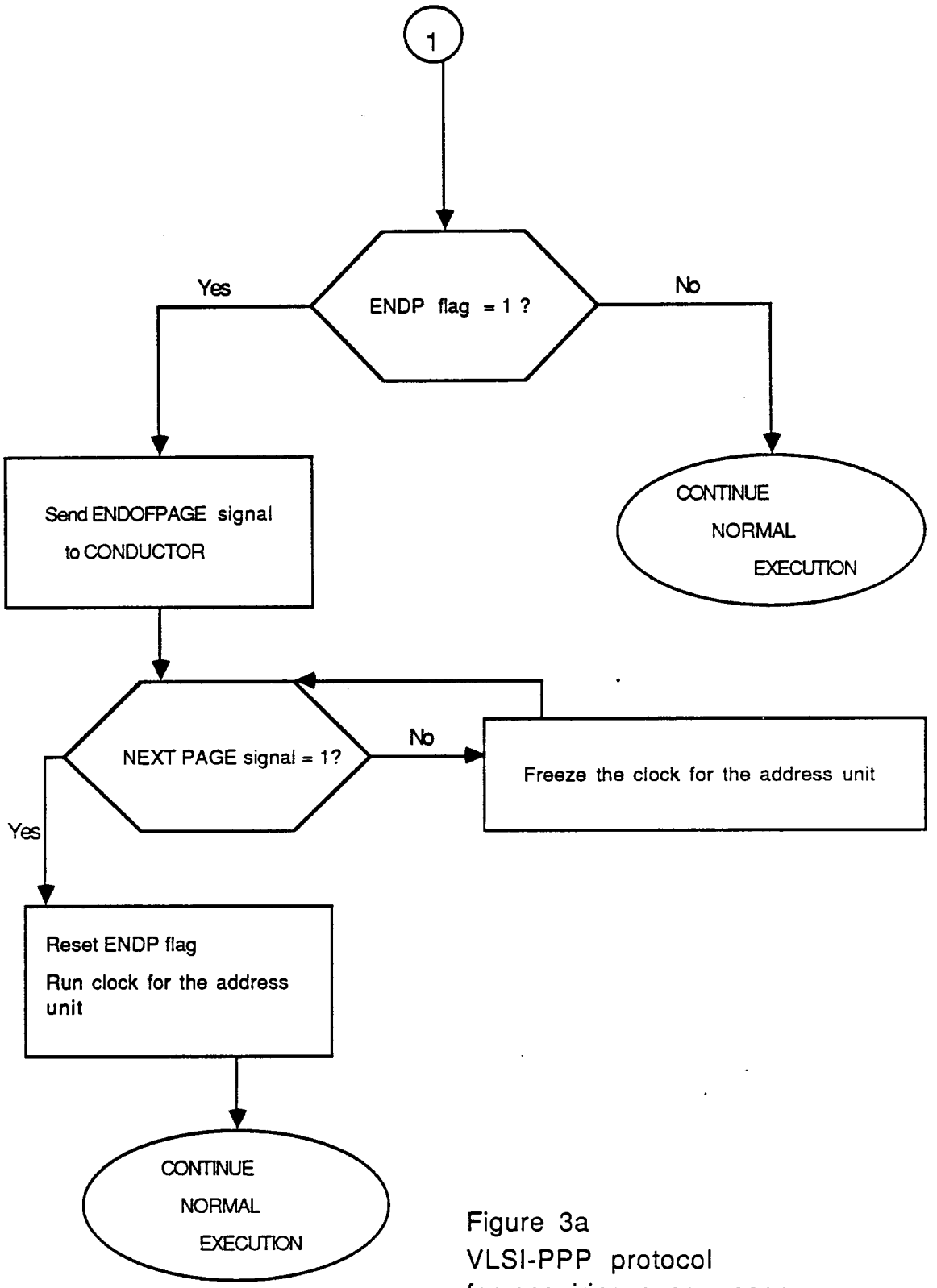


Figure 3a
VLSI-PPP protocol
for acquiring a new page

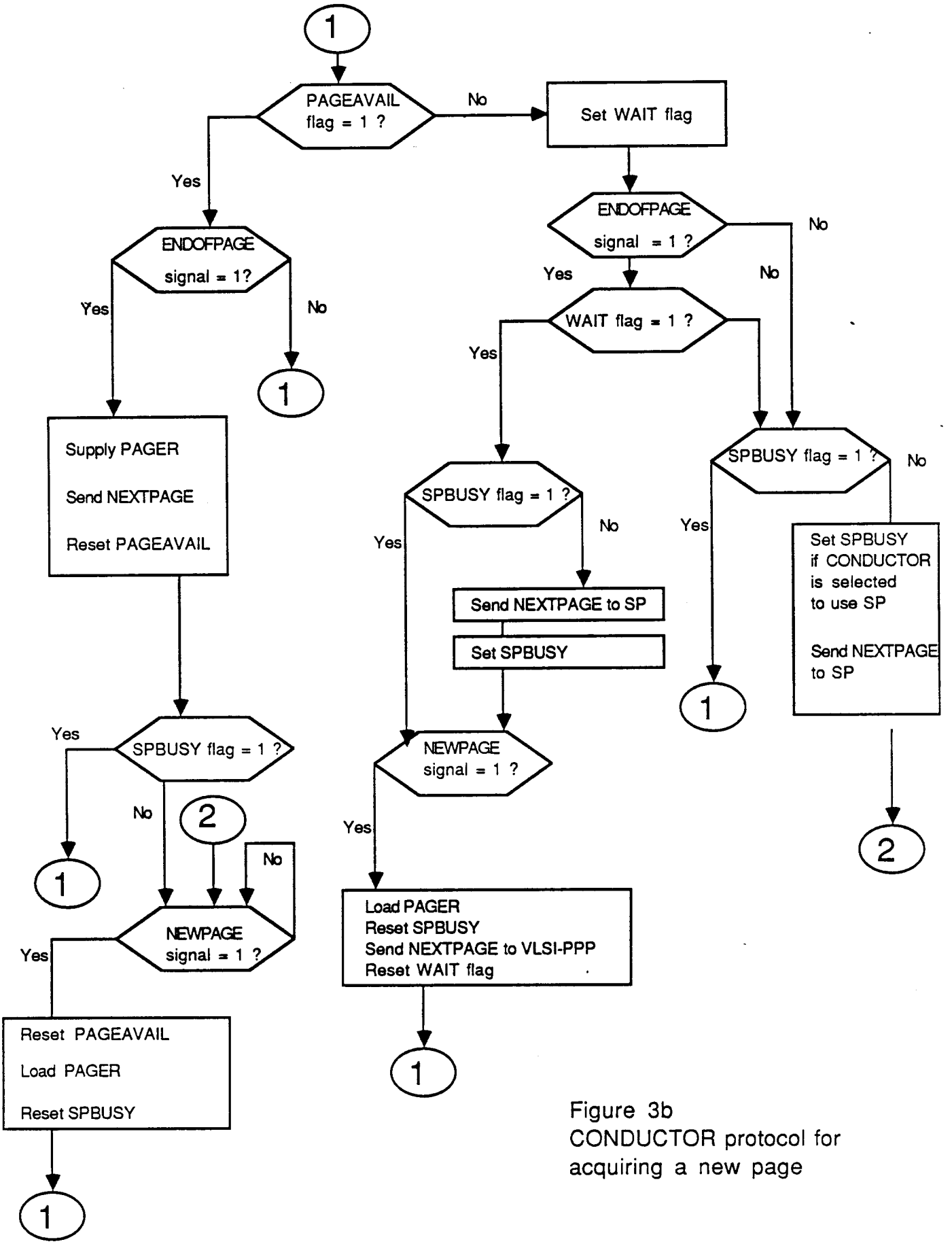


Figure 3b
CONDUCTOR protocol for acquiring a new page

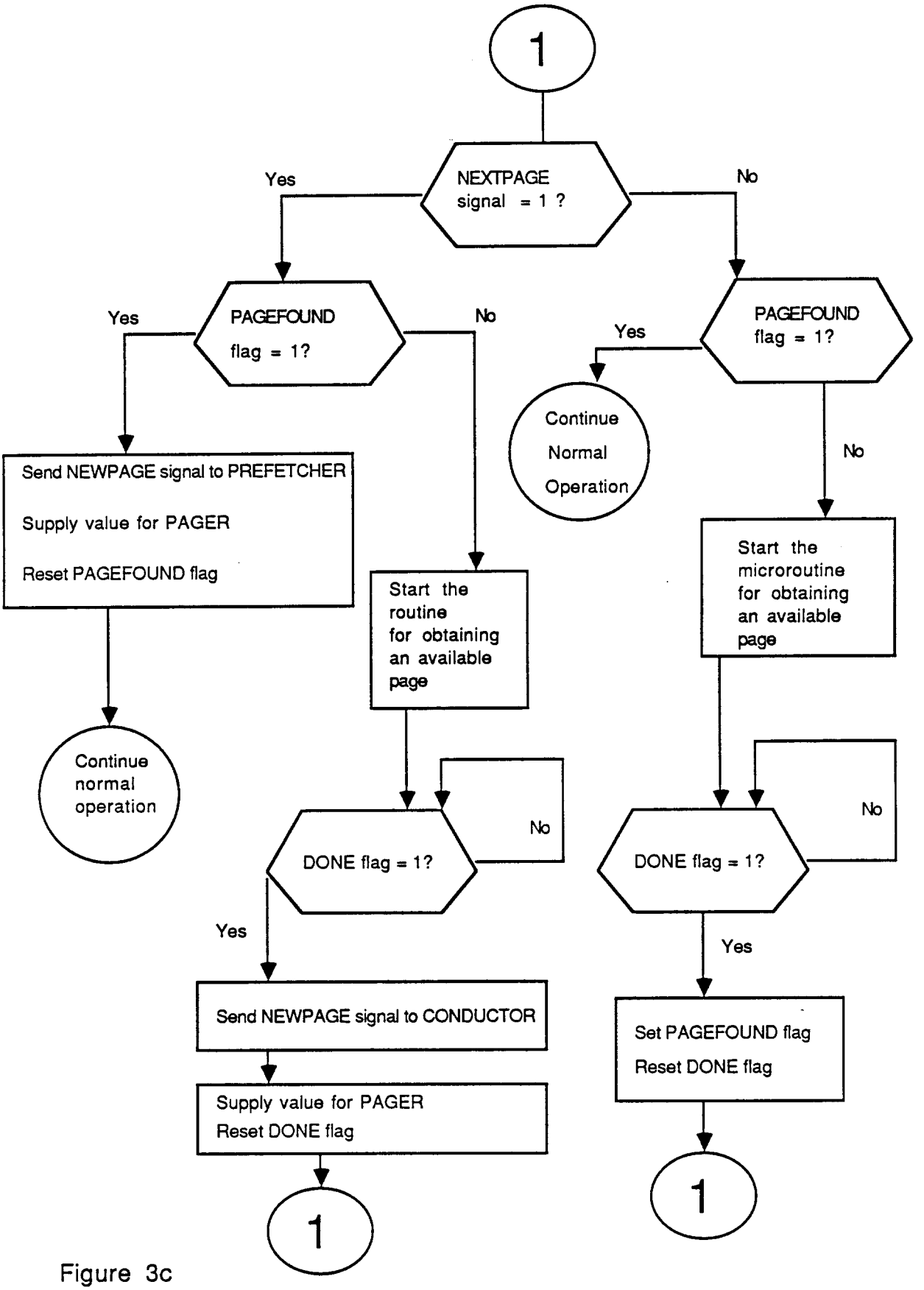


Figure 3c
SP/CONDUCTOR protocol for supplying an available page

PROCESS CREATION USING COMM REGISTERS

FLAGS : SHADOW flag

CALL-P INSTRUCT

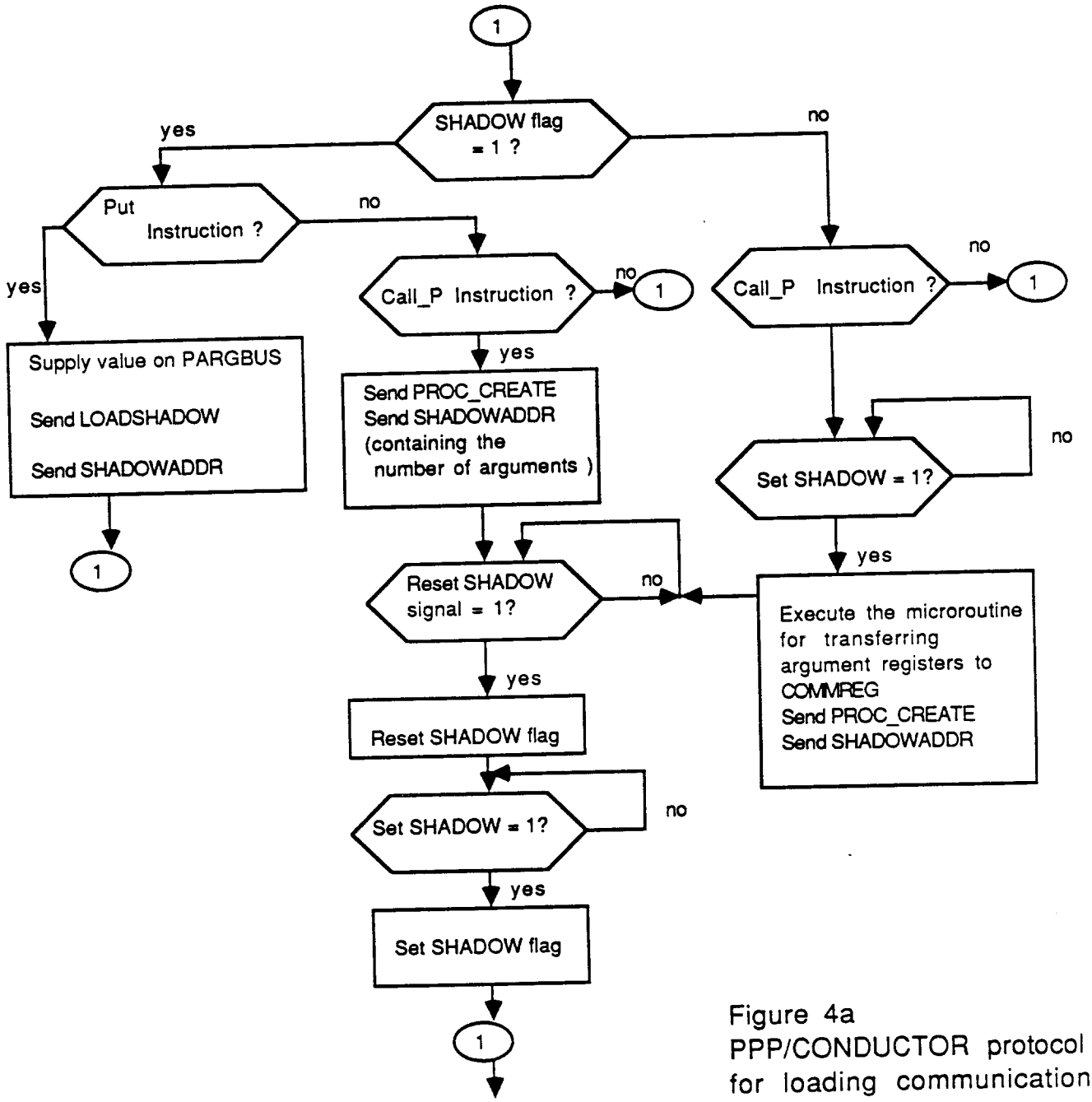


Figure 4a
PPP/CONDUCTOR protocol for loading communication registers

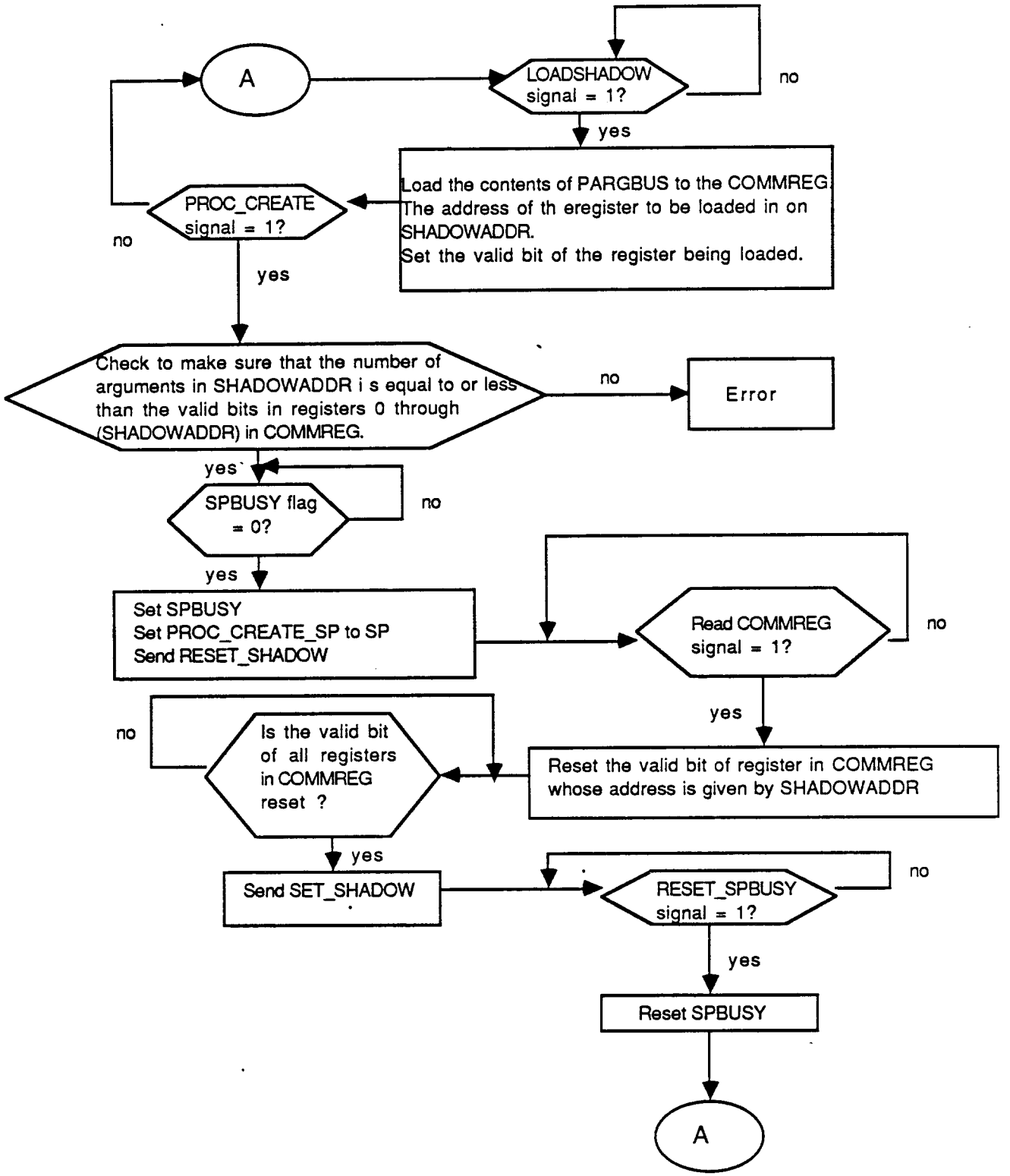


Figure 4b
CONDUCTOR protocol for supporting process creation

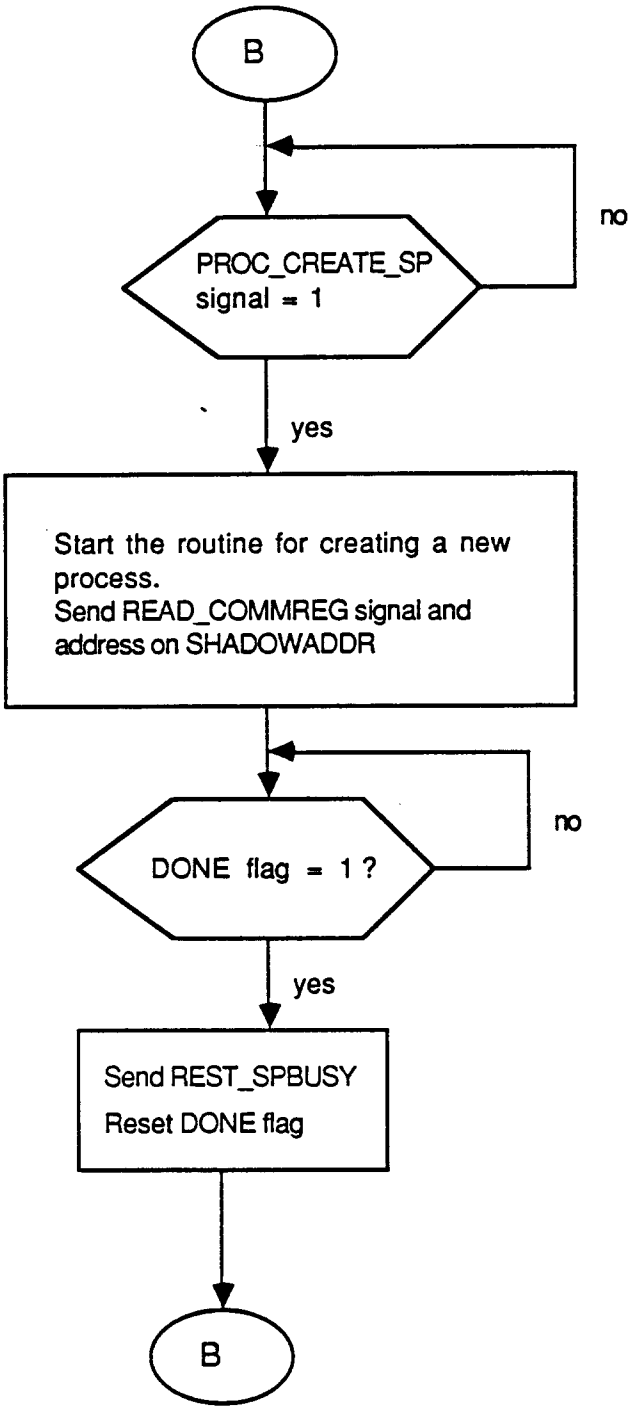


Figure 4c
SP protocol for supporting process creation

5. Chapter 5. CONCLUSION

This research initiative has established directions in three areas. The first area pertains to methodology. An extended dataflow methodology (EDFG) has been used to represent reconfigurable computer systems such as Cray-1S (c.f Chapter 2 [40]). The EDFG also has potential application in distributed operating systems as shown in Chapter 3. The design of a complete DROS using EDFG, its simulation, and implementation are areas of future research. The PARET simulation system [1] is one example of a simulation system that uses EDFG. The graphical interface and the flexibility of the simulator will allow DROS's and multiprocessor systems to be simulated at various levels of detail.

The second area is programming language. The logic programming language Prolog has features that are useful for specifying modules of the reconfigurable system. Logical variables of Prolog can have values bound at runtime using unification. Backtracking can be used to explore alternatives. The side-effect operations assert and retract can be used to change code at runtime. Although a DROS has not been written in Prolog and simulated as a part of this research, some directions have been established for future research in DROS using Prolog.

The third area is systems architecture for fast interprocess communication. A system with a single global address space and a shared memory is expected to play a key role. The two-tier shared memory architecture supports fast communication of synchronization information. The use of caches with a lock state [2] should reduce the contention on the synchronizing bus when locked cache blocks are involved. The processor architecture in Chapter 4 is one proposal to support fast IPC. It is currently under design and simulation as a part of the Aquarius project at Berkeley. The completion of the processor and system architecture will allow reconfigurable systems to be designed and evaluated using Prolog.

REFERENCES

1. K. M. Nichols, and J. T. Edmark, "Modeling Multicomputer Systems with PARET", Computer, Vo. 21, No. 5, May 1988, pp 39 - 48.

2. P. Bitar, and A. M. Despain, "Multiprocessor Cache Synchronization", Proceedings of the 13th International Symposium on Computer Architecture, Tokyo, Japan, June 1988, pp 424 - 433.