Old Dominion University Research Foundation

DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING
COLLEGE OF ENGINEERING & TECHNOLOGY
OLD DOMINION UNIVERSITY
NORFOLK, VIRGINIA 23529

ATAMM ENHANCEMENT AND MULTIPROCESSOR PERFORMANCE EVALUATION

By

John W. Stoughton, Principal Investigator
Roland R. Mielke, Co-Principal Investigator

Sukhamoy Som, Research Associate
Rodrigo Obando, Graduate Research Assistant
Mahyar R. Malekpour, Graduate Research Assistant
Robert L. Jones III, Graduate Research Assistant
Brij Mohan V. Mandala, Graduate Research Assistant

Year End Report for 1990

Prepared for
National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23665

Under
Research Grant NCC1-136
Paul J. Hayes, Technical Monitor
ISD-Information Processing Technology Branch

June 1991

DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING
COLLEGE OF ENGINEERING & TECHNOLOGY
OLD DOMINION UNIVERSITY
NORFOLK, VIRGINIA 23529

ATAMM ENHANCEMENT AND MULTIPROCESSOR PERFORMANCE EVALUATION

By

John W. Stoughton, Principal Investigator
Roland R. Mielke, Co-Principal Investigator

Sukhamoy Som, Research Associate
Rodrigo Obando, Graduate Research Assistant
Mahyar R. Malekpour, Graduate Research Assistant
Robert L. Jones III, Graduate Research Assistant
Brij Mohan V. Mandala, Graduate Research Assistant

Year End Report for 1990

Prepared for
National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23665

Under
Research Grant NCC1-136
Paul J. Hayes, Technical Monitor
ISD-Information Processing Technology Branch

Submitted by the
Old Dominion University Research Foundation
P.O. Box 6369
Norfolk, Virginia 23508-0369

June 1991

ATAMM ENHANCEMENT AND MULTIPROCESSOR PERFORMANCE EVALUATION

TABLE OF CONTENTS

CHAPTER ONE

INTRODUCTION

The purpose of this document is to describe progress of research to refine and evaluate a new multicomputing system philosophy in a VHSIC technology based multicomputer system during the period March 16, 1989 to December 31, 1990. This research supports ongoing investigations being conducted at NASA Langley Research Center concerning the insertion of VHSIC technology to potential future aerospace applications. This is the year ending report for calendar year 1990 on the research performed under Cooperative Agreement NCC1-136.

During the past four years, the authors and colleagues have conducted research concerning the development of strategies for concurrent processing of complex algorithms. A significant result of this work has been the development of a multicomputer operating strategy for executing large-grained, decision-free algorithms on data flow architectures. The operating strategy is expressed as a model for concurrent processing called ATAMM for Algorithm To Architecture Mapping Model [1, 2]. The model is significant because it identifies the control dialogue and data flow required to implement a decomposed algorithm in a data flow architecture, and because it provides a context for analytically predicting system time performance.

The focus of the present research is to develop and evaluate an ATAMM Multicomputer Operating System (AMOS) for use in a VHSIC multicomputer architecture [3]. The target system is the Advanced Development Model (ADM) as shown in Figure 1 which was developed by the Air Force at Westinghouse Electric Corporation. The ADM system consists of four MIL-STD-1750A processors interconnected by a PI bus and an IEEE 488 bus. The interface of the ADM system with the external world is carried out by an unit called 1553B. In the testbed, a Microvax II is used for initialization and debugging of processors and an IBM PC/386 is used for input and output activities. The ADM system is to operate as a multicomputer environment for execution of complex decomposed algorithms such as are found in command, control and signal processing applications. A detailed description of the ADM system will be presented in Chapter Four.

During the report period, the ATAMM Multicomputer Operating System for the ADM was defined and a detailed description of the operating system was delivered to Westinghouse for implementation. The development of a set of software support tools for the ADM system was also initiated. Future work will include installation of AMOS on the ADM, and then testing and evaluation of the complete system. The purpose of this report is to document the specification of AMOS for ADM. In Chapter Two, the ATAMM model is reviewed and the performance of algorithms executing under the ATAMM rules is presented. In Chapter Three, enhancements to ATAMM developed for the ADM implementation of AMOS are described. Included are descriptions for the strategies for real-time, on line performance control and fault

tolerance. The detailed specification for AMOS and input/output communication software are presented in Chapter Four. In Chapter Five, three software support tools developed for performance analysis in the ATAMM environment are described. The report concludes with a summary of the status of research in Chapter Six.

The use of brand names in this report is for completeness, and does not indicate NASA endorsement.

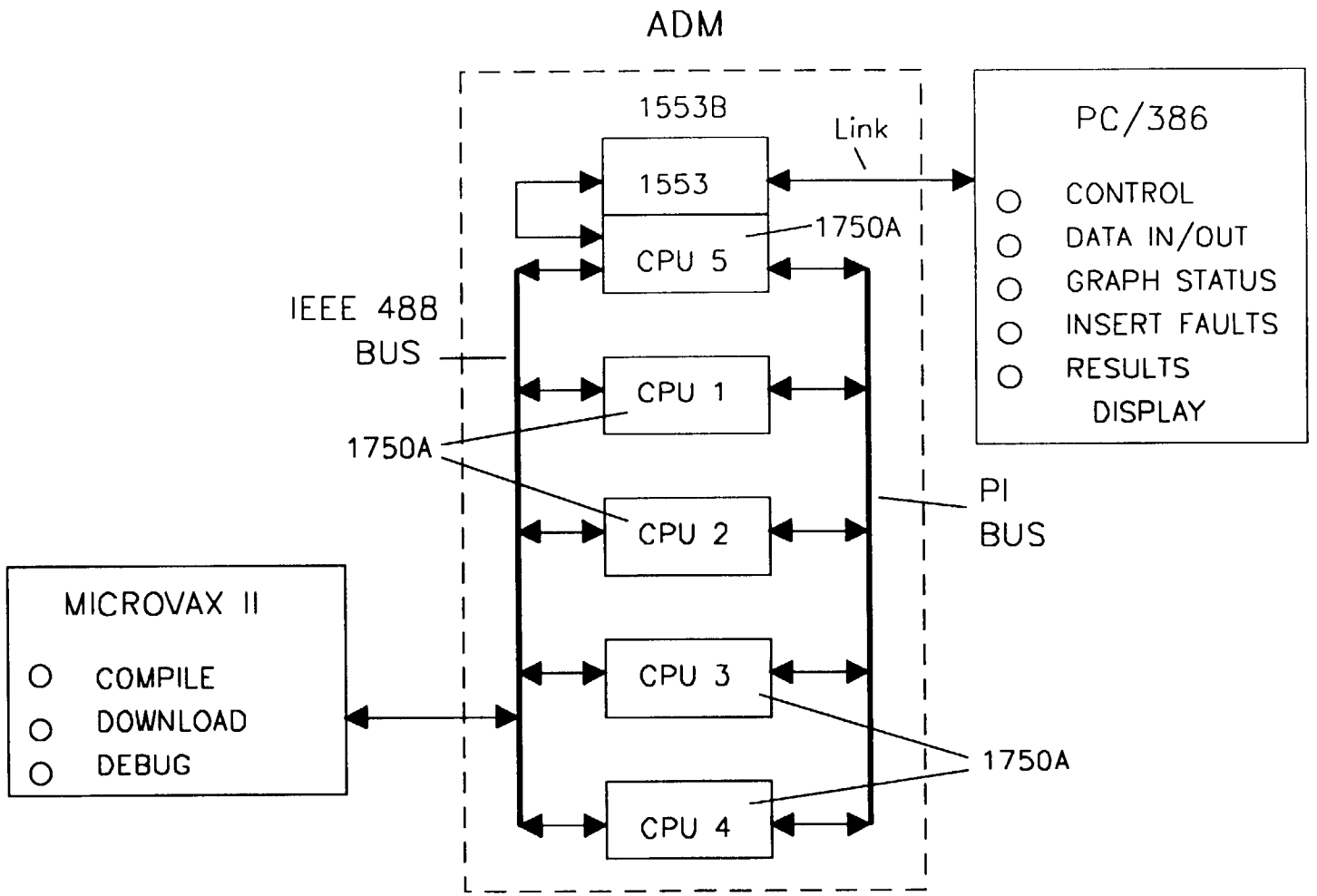


Figure 1. A testbed for the Advanced Development Model (ADM).

CHAPTER TWO

THE ATAMM MODEL

2.0 Introduction

The ATAMM model is reviewed briefly in this chapter. The definition of ATAMM is presented and illustrated by example in Section 2.1. In Section 2.2, the time performance of algorithms executing according to the ATAMM rules described. Strategies are developed for generating operating conditions for predictable performance based on the number of available computing resources.

2.1 Model Description

The ATAMM model consists of a set of Petri Net marked graphs which incorporates general specifications of communication and processing associated with the implementation of a decomposed, large-grained algorithm in a data-flow architecture. In this section, the execution of a computational problem is represented by the ATAMM model. Some familiarity with Petri Nets and marked graphs is assumed [4]. A more detailed description of the ATAMM model and its characteristics are found in [5, 6].

An algorithm marked graph (AMG) is a marked graph which represents a specific algorithm decomposition. Transitions and places are represented as nodes (vertices) and directed edges respectively. Vertices of the algorithm marked graph are

in a one-to-one correspondence with each occurrence of an algorithm operation. The transition times represent the computational times of the respective algorithm operations. The algorithm marked graph contains an edge (i, j) directed from vertex i to vertex j if the output of vertex i is an input for vertex j. Edge (i, j) is marked with a token if an output from vertex i is available as an input to vertex j. Source transitions and sink transitions for input and output signals are represented as squares.

To illustrate the construction of an algorithm marked graph, consider the problem of computing the output of a discrete linear, time invariant system given a sequence of inputs to the system. Let the system be described by the state equation

$$x(k) = Ax(k-1) + Bu(k)$$

and the output equation

$$y(k) = Cx(k),$$

where x is a p-vector, u is a m-vector, and y is a r-vector. The algorithm operations are defined as matrix multiplication and vector addition, and the natural algorithm decomposition resulting from the state equation description is selected. The algorithm marked graph for this decomposed algorithm is shown in Figure 2. The initial marking indicates that initial condition data are available.

The algorithm marked graph is a useful tool for representing decomposed algorithms and for displaying data flow within an algorithm. However, the algorithm marked graph does not display procedures that a computing structure must manifest in order to perform the computing task. In addition, the issues of control, time performance, and resource management are not apparent in this graph. These

important aspects of concurrent processing are included in the ATAMM model through the definition of two additional graphs. These additional graphs are defined in the following.

The node marked graph (NMG) is a Petri Net representation of the performance of an algorithm operation by a functional unit. Three primary activities, reading of input data from global memory, processing of input data to compute output data, and writing of output data to global memory, are represented as transitions (vertices) in the NMG. Data and control flow paths are represented as places (edges), and the presence of signals is notated by tokens marking appropriate edges. The conditions for firing the process and write transitions of the NMG are as defined for a general Petri Net, while the read transition has one additional condition for firing. In addition to having a token present on each incoming signal edge, a functional unit must be available in a queue of available functional units for assignment to the algorithm operation before the read node can fire. Once assigned, the functional unit is used to implement the read, process, and write operations before being returned to a queue of available functional units. The initial marking for an NMG consists of a single token in the Process Ready place. The NMG model is shown in Figure 3.

A computational marked graph (CMG) is constructed from the AMG and the NMG by the following rules:

1) Source and sink nodes in the algorithm marked graph
   are represented by source and sink nodes in the
   CMG.

2) Nodes corresponding to algorithm operations in the algorithm marked graph are represented by NMGs in the CMG.
3) Edges in the algorithm marked graph are represented by edge pairs, one forward directed for data flow and one backward directed for control flow, in the CMG.

A forward directed edge goes from a predecessor write transition to a successor read or sink transition. Forward edges are also shown as part of the NMG in Figure 2 where they are labeled OF and IF edges of the predecessor and successor transitions respectively. A backward directed edge goes from a successor read transition to a predecessor read or source transition. Backward edges are also shown as part of the NMG where they are labeled OE and IE edges of predecessor and successor transitions respectively. The initial marking for the edge pair consists of a single token in the forward directed place if data are available, or a single token in the backward directed place if data are not available. In order to illustrate the construction of a computational marked graph, the CMG corresponding to the algorithm marked graph of Figure 1 is shown in Figure 4.

The complete ATAMM model consists of the algorithm marked graph, the node marked graph, and the computational marked graph. A pictorial display of the components of the ATAMM model are shown in Figure 5.

Graph execution based on the ATAMM rules has several useful and important properties [5]. Execution is live, reachable, safe, deadlock-free, and consistent [6]. Liveness indicates that all transitions in the CMG are firable from the initial marking, whereas reachability ensures that CMG will generate an output for each input. Safeness guarantees that output of an algorithm operation will not be overwritten before it is picked up by a successor algorithm operation or sink. This property is a result of including backward control places in the CMG and is necessary for safe periodic operation. The necessary and sufficient condition for avoidance of deadlock in the graph play is to ensure that once assigned, a functional unit always is able to complete execution of an algorithm operation. A computation can not enter deadlock because no read transition is executed unless the output edges of the corresponding NMG are empty and a functional unit is available. The consistency property implies that computations are repeated periodically when input are applied periodically.

2.2 Time Performance

In this section, the time performance of algorithms implemented in data flow architectures according to the ATAMM rules is investigated. First, performance measures for computing speed and throughput are defined. It is shown that the ATAMM model is useful for analytically calculating bounds for these measures. Then, graph play is described and used to determine resource requirements necessary to achieve a specified time performance. Finally, the ATAMM performance plane is defined. This diagram displays possible operating strategies with resource

requirements as a parameter. Using this display, a system operator is able to specify quantitatively system time performance.

2.2.1. Performance Measures

Two measures of time performance, TBIO and TBO, are defined in this section. The performance measure TBIO (time between input and output) is the elapsed computing time between an algorithm input and the corresponding algorithm output. Therefore, TBIO is an indicator of computing speed. It is shown in [7] that the algorithm imposed lower bound for TBIO, denoted $TBIO_{LB}$, is given by the sum of transition times for nodes contained in the longest directed path from the input source to the output sink in the AMG.

The performance measure TBO (time between outputs) is the elapsed computing time between successive algorithm outputs when the AMG is operating periodically at steady-state. Therefore, the inverse of TBO is an indicator of throughput frequency. It is shown in [7] that the algorithm imposed lower bound for TBO is given by the largest time per token of all directed circuits in the CMG. A second bound on TBO is imposed by the availability of resources. It is shown in [6] that the resource imposed lower bound for TBO is TCE/R where TCE (total computing time) is the sum of transition times for all nodes in the AMG and R is the number of available functional units. The lower bound for TBO, denoted TBO_{LB}, is the greater of the algorithm bound and the resource bound.

To illustrate the calculation of these performance bounds, consider as an example the AMG shown in Figure 6 and the corresponding CMG shown in Figure 7. The AMG contains four directed paths from the input source to the output sink. These paths, identified by included transitions, are (1, 2, 6, 7), (1, 3, 6, 7), (1, 4, 6, 7) and (1, 5, 6, 7). The sum of transition times of nodes in each path is 7 so that $TBIO_{LB} = 7$. The largest time per token of any directed circuit in the CMG is 2. There are several directed circuits which yield this result; one such directed circuit is the circuit containing the read, process and write transitions of node 6 and the read transition of node 7. Therefore, $TBO_{LB} = 2$.

2.2.2 Graph Play and Resource Requirements

Two diagrams which display graph play and are useful for determining the number of resources needed to achieve specified performance measures are defined next. The SGP (single graph play) diagram is a diagram which displays the execution of each node of the AMG as a function of time. The diagram is constructed for a single input data packet under the assumption that unlimited resources are available to play the graph. Node activity is denoted by a solid line and the symbols (<, >) are used to indicate the beginning and end of execution. When several nodes are active at the same time, lines indicating node activity are stacked vertically so that computing concurrency is apparent. The SGP diagram for the AMG of Figure 6 is shown in Figure 8.

The resource requirements to execute a single data packet are obtained by counting the number of active nodes during each time interval in the SGP diagram. The peak resource requirement is denoted by R_{min} and represents the minimum number of resources necessary to achieve operation at TBIO = $TBIO_{LB}$. For the AMG in Figure 6, $R_{min} = 4$ is the minimum number of resources necessary to execute the graph with TBIO = $TBIO_{LB} = 7$.

The TGP (total graph play) diagram is a diagram which displays the execution of each graph node when the graph is operating periodically in steady-state with period TBO. As with SGP, the diagram is constructed under the assumption that unlimited resources are available to play the graph, and a different diagram results for each value of TBO. The TGP diagram is drawn using information from SGP. SGP is divided into segments of width TBO, and these segments are overlaid to form TGP. Each segment from SGP represents a new input data packet. Data packets are numbered sequentially so that the packet numbered i+1 is the data packet which is input to the graph TBO time units after the packet numbered i. To illustrate the construction of this diagram, TGP for the AMG of Figure 6 is shown in Figure 9.

The resource requirements to execute multiple data packets injected with period TBO are obtained by counting the number of active nodes during each time interval in the TGP diagram. The peak resource requirement necessary to execute the graph periodically with period greater than or equal to TBO is denoted by R_{max}. R_{max} is determined by finding the largest resource requirement in all TGP diagrams drawn for injection intervals greater than or equal to TBO. For example, the TGP diagram

drawn for TBO = TBO_{LB} = 2 shown in Figure 9 indicates that a minimum of 7 resources is required. If this same TGP diagram is drawn for all values of TBO > 2, it can be shown that the required number of resources remains less than 7. Therefore, R_{max} to achieve TBO = 2 for the AMG shown in Figure 6 is equal to 7.

2.2.3. ATAMM Performance Plane

For a given algorithm decomposition, the parameters TBIO, TBO and R define an operating point for ATAMM. The display of all operating points on a graph of TBO versus TBIO with R indicated as a parameter is called the ATAMM performance plane. The ATAMM performance plane, illustrated in Figure 10, is extremely useful for selecting system operating strategies. The use of this diagram is described in this section.

The best system performance is achieved by operating at point B where TBIO = $TBIO_{LB}$ and TBO = TBO_{LB}. The resource requirement associated with this operating point is the value of R_{max} computed from the TGP diagram drawn for $TBIO_{LB}$ and TBO_{LB}. Operation at point B is obtained by the use of injection control as shown in Figure 11. Injection control is a control procedure which limits the maximum rate at which new input data packets can be injected. When presented with continuously available input data packets, the natural behavior of a data flow architecture results in operation where data packets are accepted as rapidly as available resources and the input transition permit. This leads to operation at a steady-state operating point where TBO = TBO_{LB} but TBIO > $TBIO_{LB}$. This occurs because the pipeline from input to

output becomes congested with extra data packets which must wait for free resources to be processed. Injection control eliminates data packet congestion and thus preserves operation at $TBIO_{LB}$.

When there are not sufficient resources to operate at point B, the operating point must be shifted to a new location having a smaller resource requirement. Using injection control procedures, it is possible to shift the operating point vertically along line B-V. This strategy preserves TBIO while degrading throughput performance. Such a strategy is useful for real-time control and signal processing applications where maintaining high computing speed is very important. Operating points on line B-V for lower resource requirements are calculated from the TGP diagram by increasing TBO until the number of active nodes in any time interval decreases by one from the previous operating point. These operating points are implemented by adjusting the minimum input injection control interval. As an example, consider the AMG shown in Figure 6. Operation at TBIO = 7 and TBO = 2 requires 7 resources. By increasing TBO to 3, the number of required resources decreases to 5. This can be observed by increasing the value of TBO in the TGP diagram of Figure 9 until the number of concurrently active nodes decreases. Increasing TBO to 5 further reduces the resource requirement to 4 resources. These operating points are displayed in the ATAMM performance plane as shown in Figure 12.

It is also possible to shift the operating point horizontally along the line B-H to reduce resource requirements. This strategy preserves TBO while degrading computing speed performance. Such a strategy is useful for number crunching

applications where maintaining throughput is important. Operating points on line B-H for lower resource requirements are obtained by adding control edges to the original AMG. A control edge is an AMG place which imposes a precedence relation among two transitions, but does not imply data dependency. When such an edge is added to an AMG so that the longest directed path from the input source to the output sink is increased, the resulting new graph has an increased TBIO value but still describes the same algorithm.

The addition of a control edge can create new directed circuits having increased time per token values so that TBO is also increased. This potential problem is avoided by adding dummy nodes to the AMG. A dummy node is an AMG transition which implements an identity operation and requires zero computation time. The dummy node serves as a buffer to provide additional storage for the output data of a graph node. Implementation of a dummy node is a memory operation and thus does not require a resource. Using the dummy node, it is possible to increase the token count on circuits formed by adding control edges, thus preserving the value of TBO in the original graph. Control edges and dummy nodes also can be used to improve performance bounds and to balance resource requirements. Operating point design using control edges and dummy nodes is explained in more detail in [8].

To illustrate shifting the operating point horizontally, consider again the AMG shown in Figure 6. Adding a control edge directed from node 3 to node 4 creates a new directed path from input source to output sink which contains nodes (1, 3, 4, 6, 7). Therefore, $TBIO_{LB}$ for the new graph is equal to 8. However, the control edge

also creates a new directed circuit containing the read, process and write transitions of nodes 1 and 3, and the read transition of node 4. This directed circuit has a time per token value of 3 so that TBO_{LB} is increased to 3. The time per token value of this circuit is reduced by adding a dummy node to the edge directed from node 1 to node 4. The new AMG and the corresponding CMG are shown in Figures 13 and 14, respectively. A second control edge and dummy node are also added in Figure 13 for the purpose of reducing the peak resource requirement. The SGP diagram and the TGP diagram for TBO = 2 are shown in Figures 15. The new operating point having TBIO = 8, TBO = 2, and R = 5 is shown on the performance plane diagram in Figure 12. Also shown are additional operating points on the constant TBIO = 8 line which are implemented by injection control as described previously.

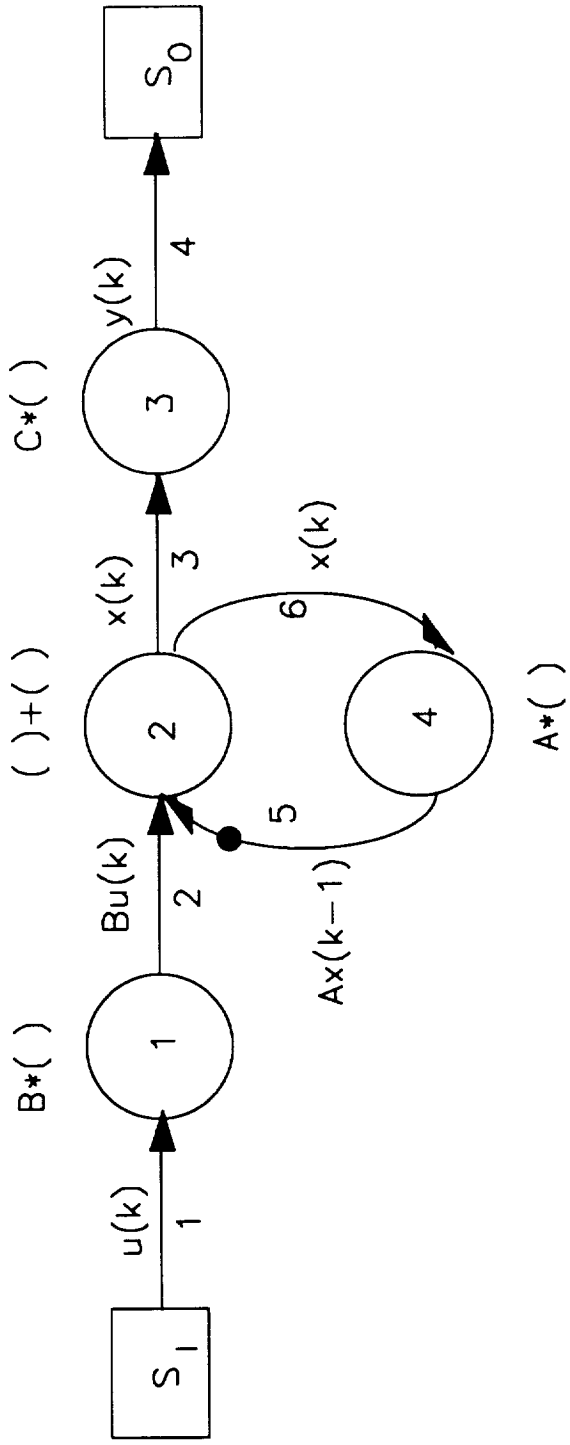
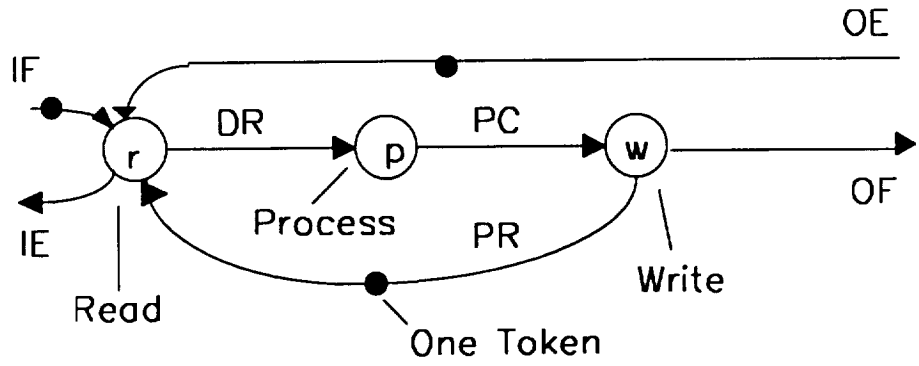


Figure 2. Algorithm marked graph for discrete system equation.



NMG EDGE LABELS

IF	Input Buffer Full
IE	Input Buffer Empty
DR	Data Read
PC	Process Complete
PR	Process Ready
OE	Output Buffer Empty
OF	Output Buffer Full

Figure 3. ATAMM node marked graph model.

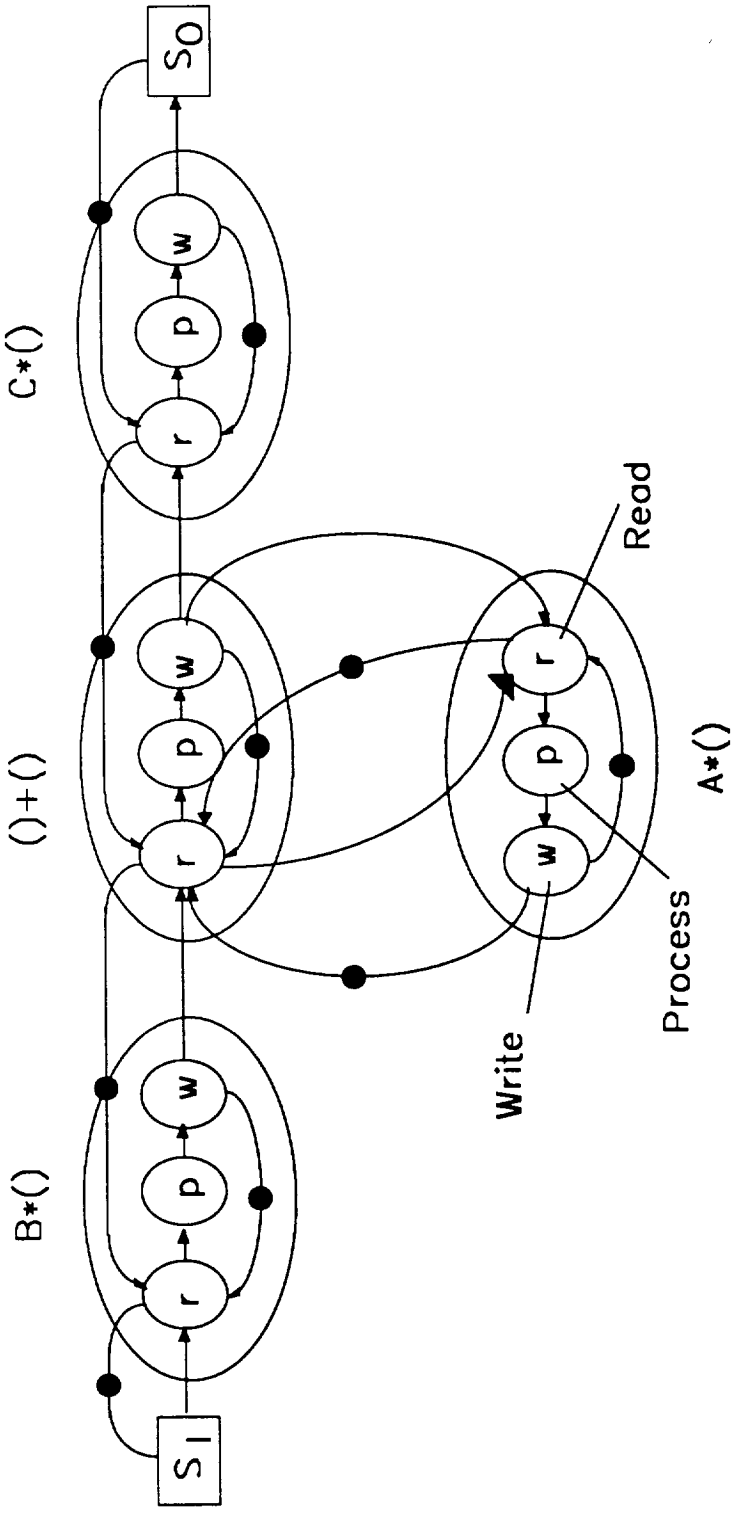


Figure 4. ATAMM computational marked graph model for discrete system equation.

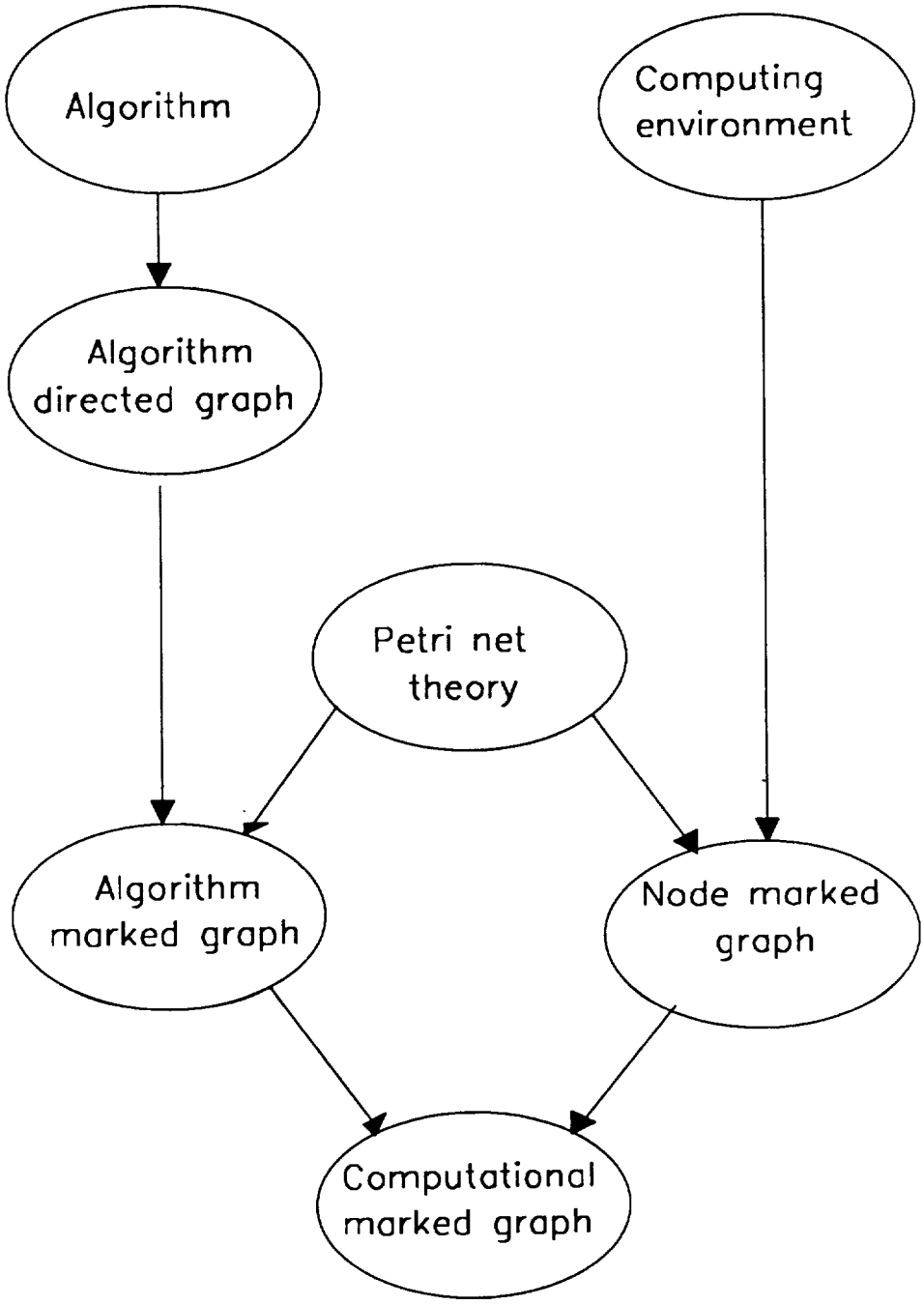


Figure 5. ATAMM model components.

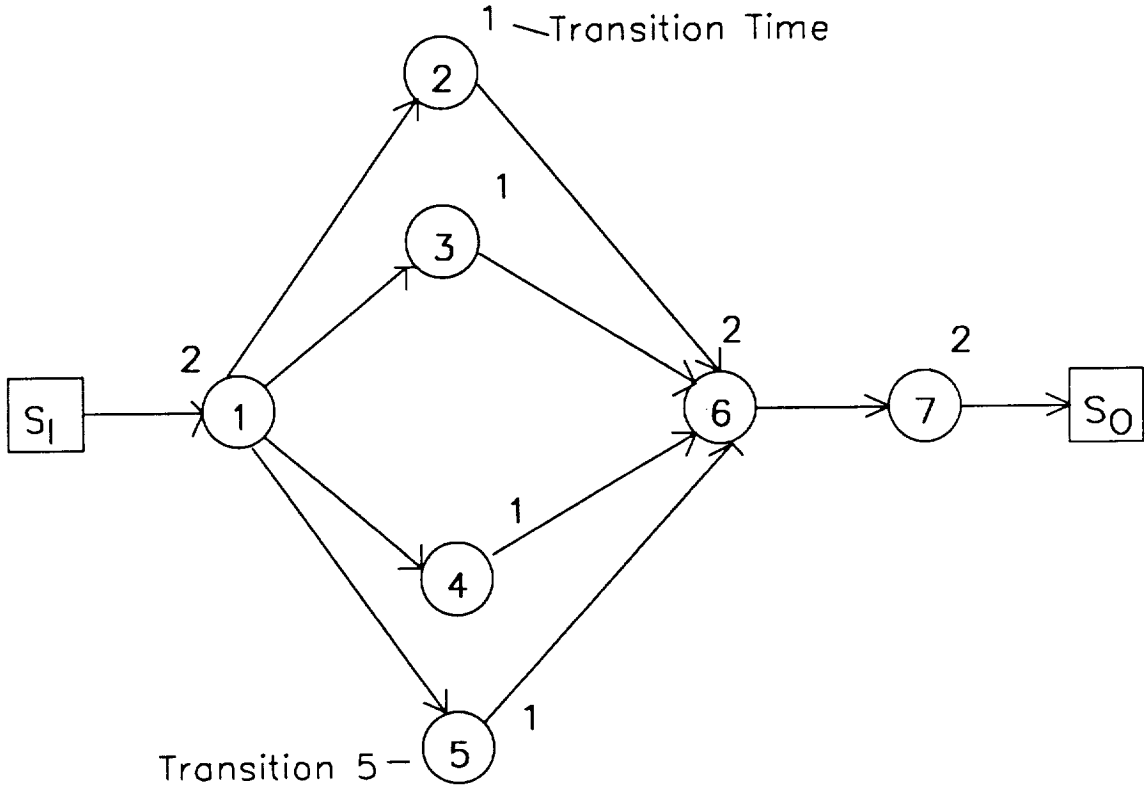


Figure 6. AMG for performance example.

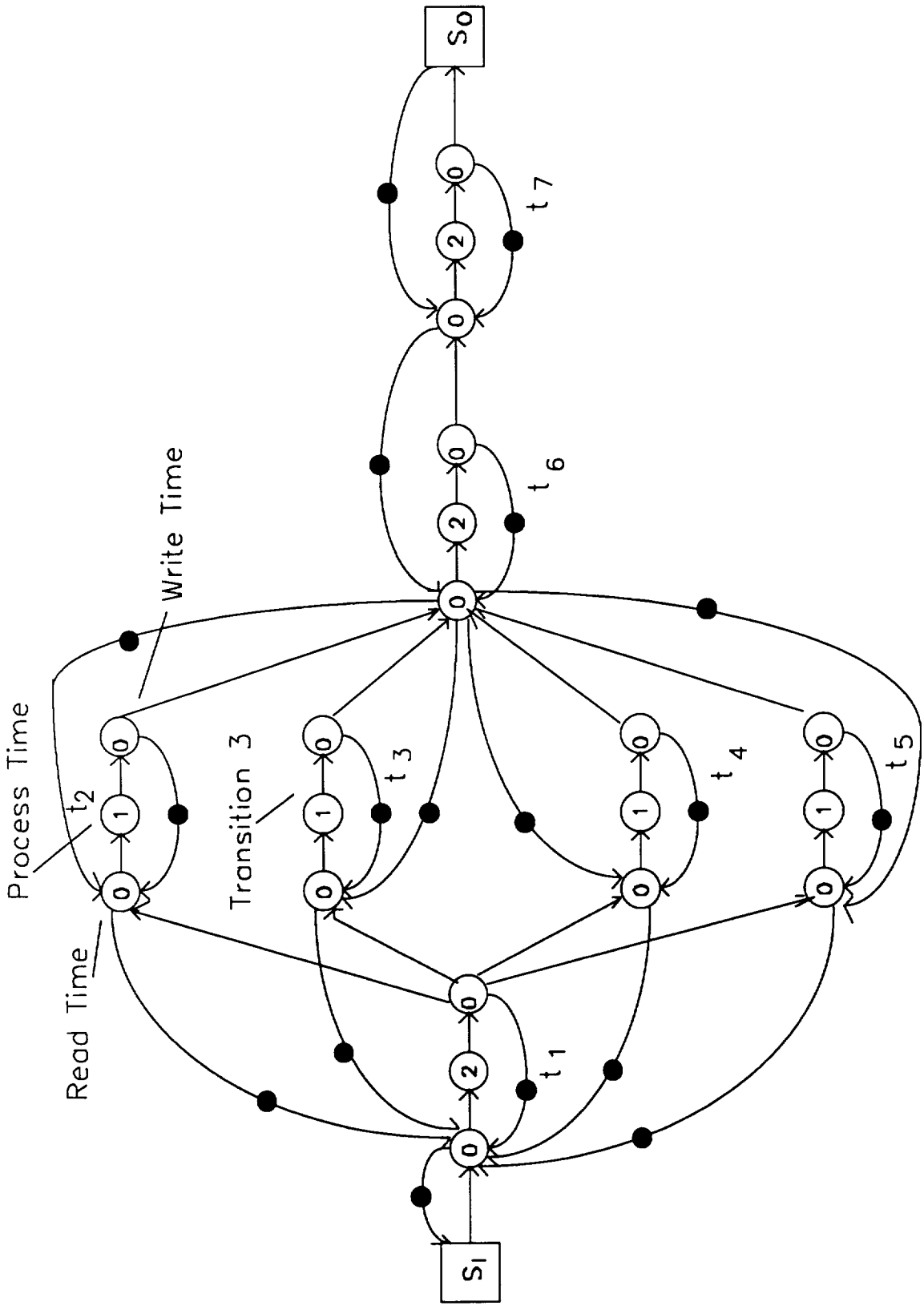


Figure 7. CMG for Figure 6.

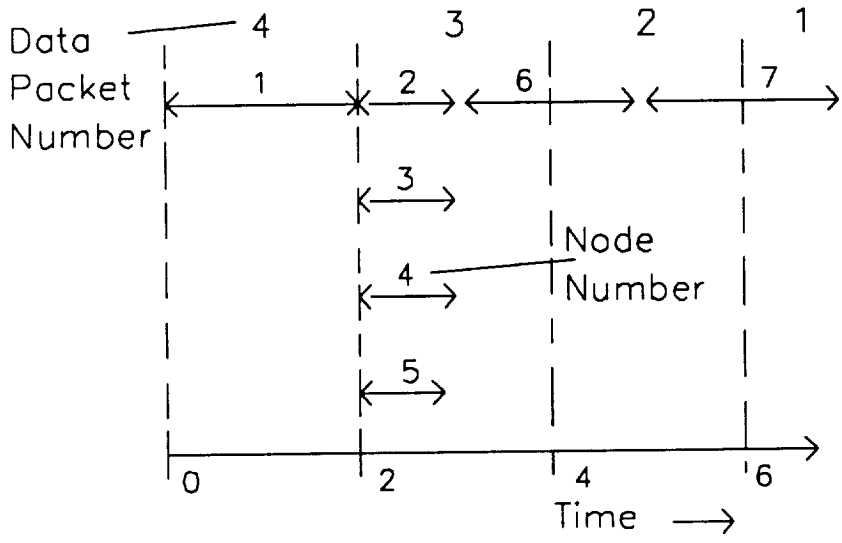


Figure 8. SGP diagram for Figure 6.

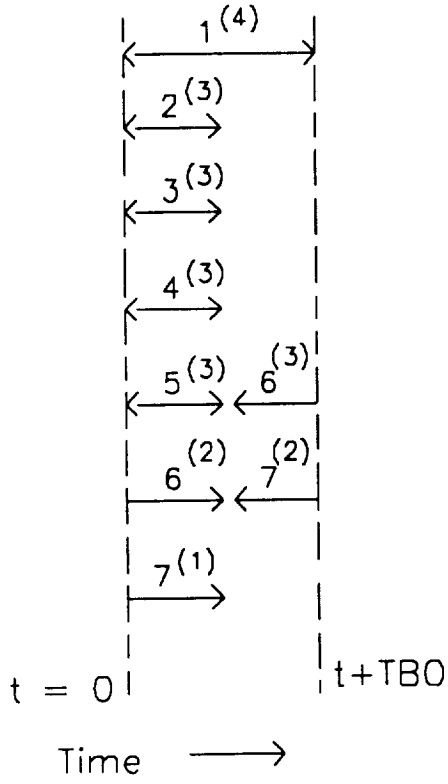


Figure 9. TGP diagram for Figure 8 with TBO = 2.

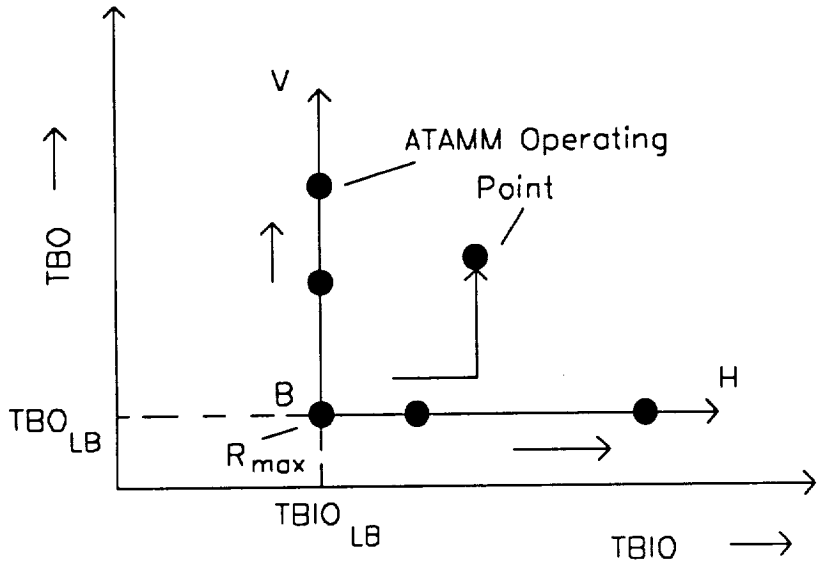


Figure 10. ATAMM performance plane.

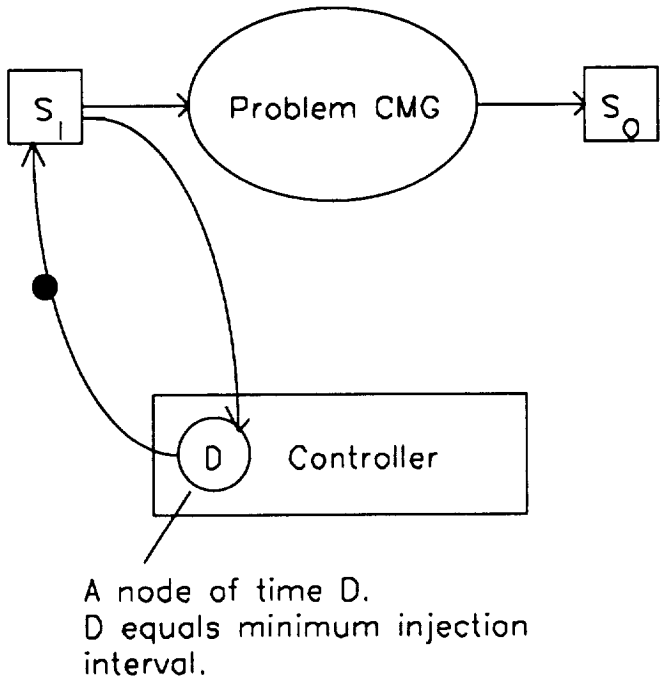


Figure 11. Injection control implementation.

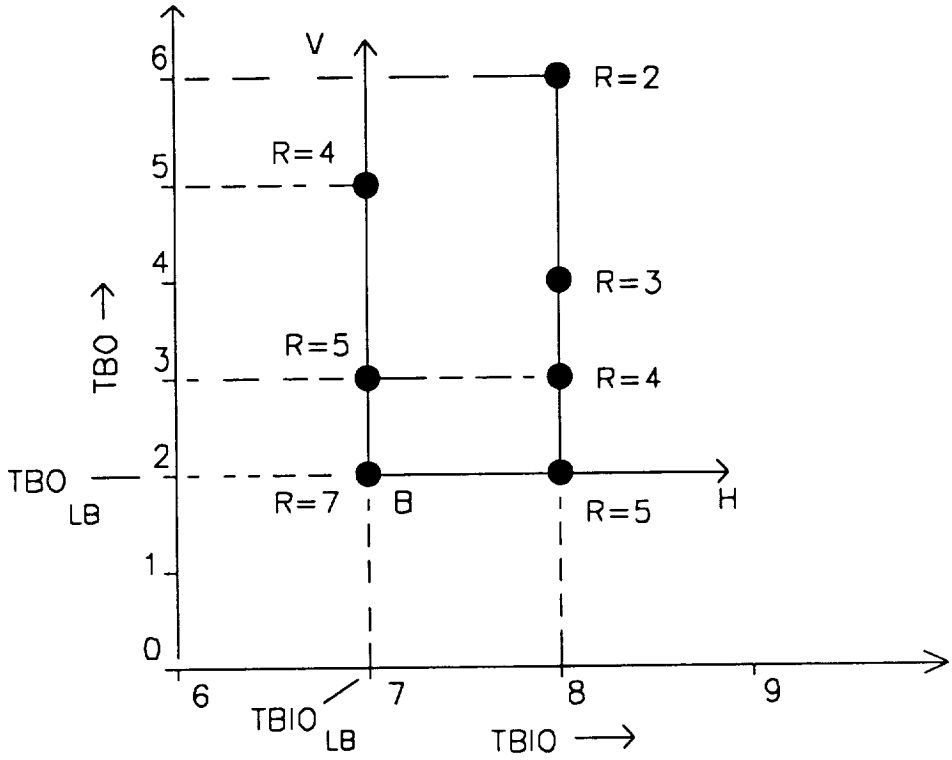


Figure 12. ATAMM operating points for Figures 6 and 13.

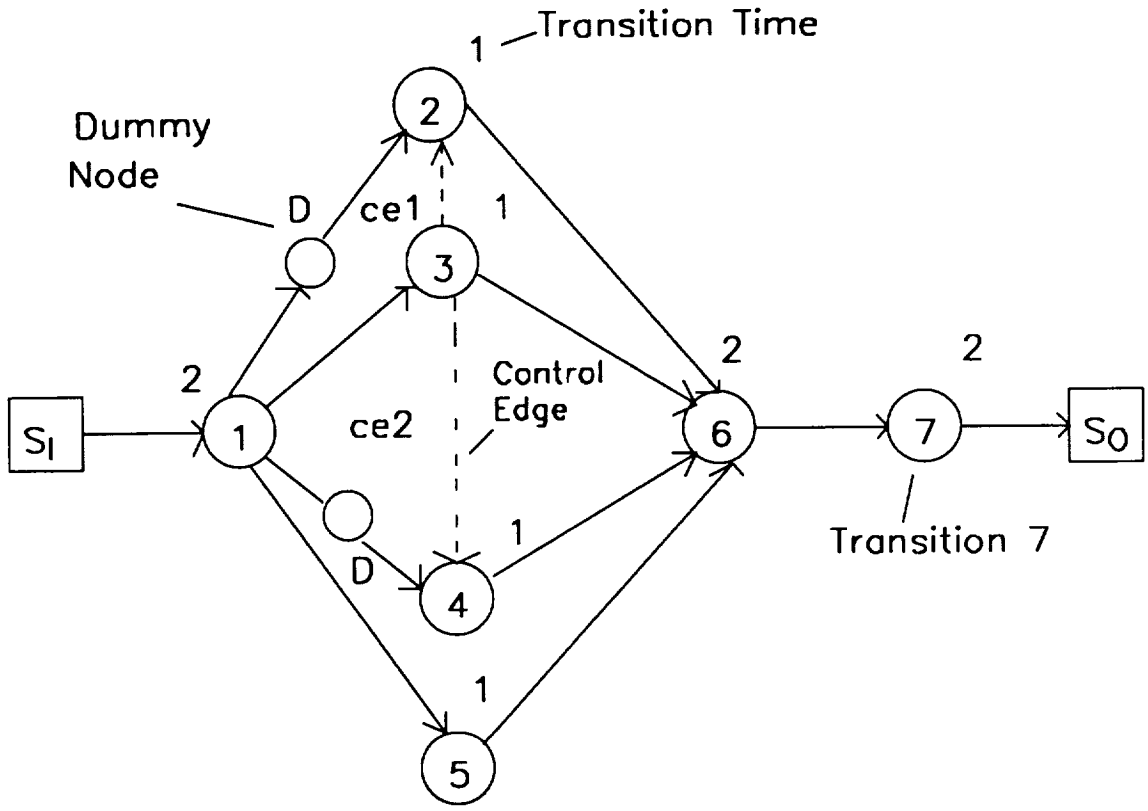


Figure 13. Modified AMG for Figure 6.

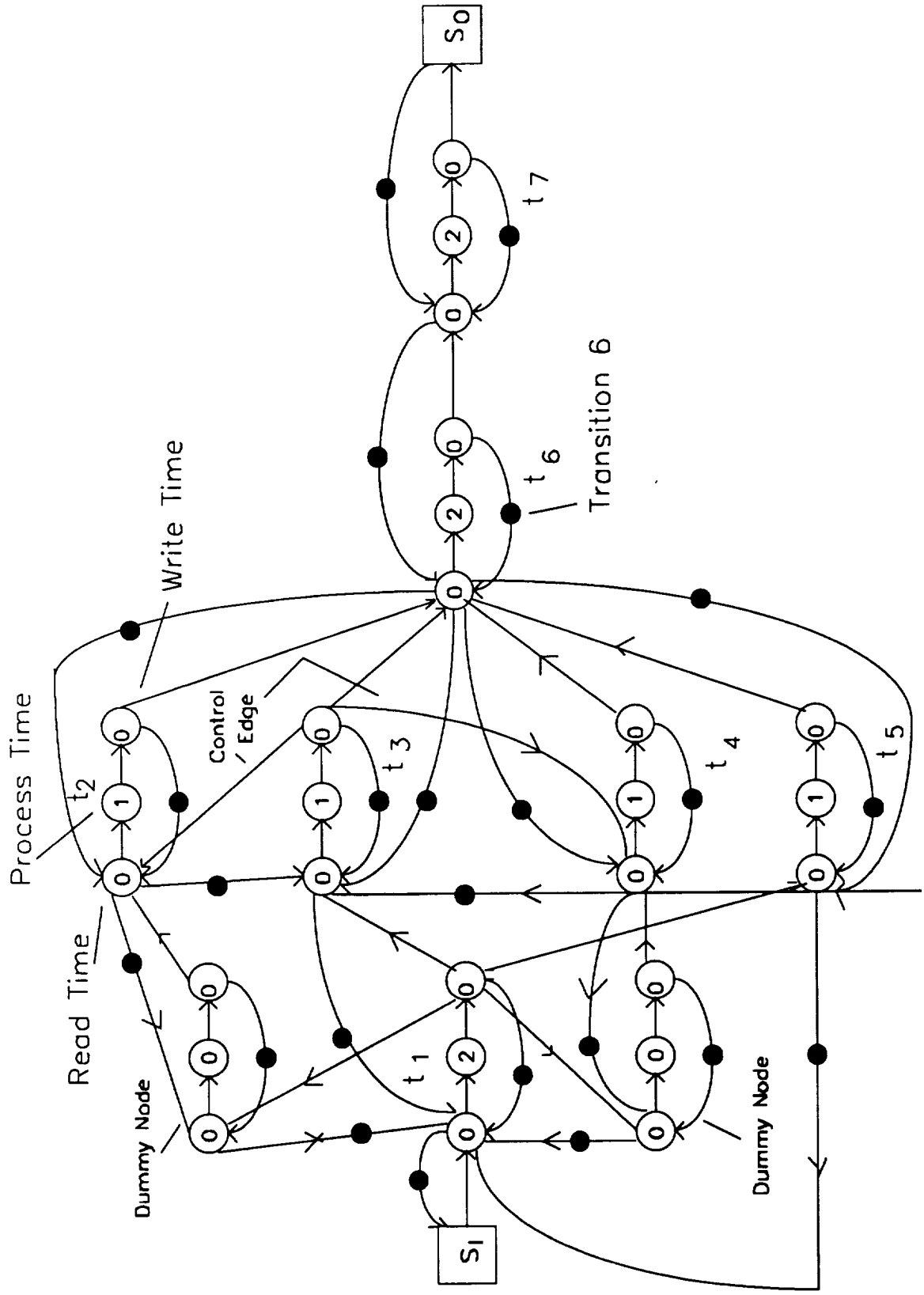
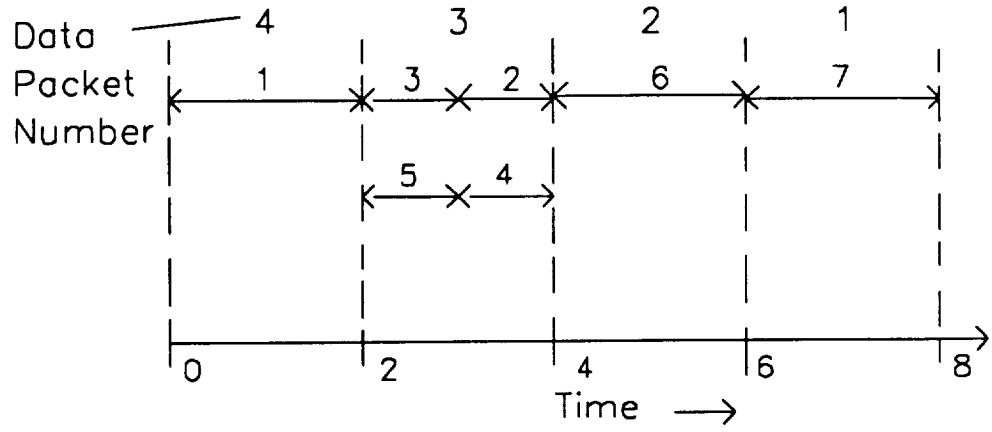
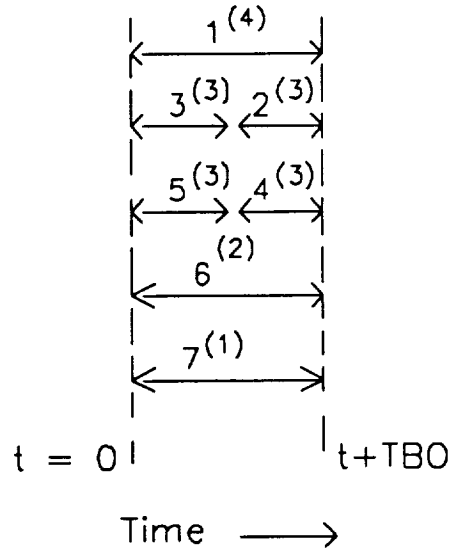


Figure 14. Modified CMG for Figure 13.



(a)



(b)

Figure 15. For the AMG of Figure 13, (a) SGP diagram. (b) TGP diagram for TBO=2.

CHAPTER THREE

ATAMM ENHANCEMENTS

3.0 Introduction

In this chapter, enhancements to ATAMM developed for the ADM system are presented. First, a method to control the time performance of the system based on knowledge of the number of available processors is presented in Section 3.1. This allows a user to specify system performance using the ATAMM performance plane information given in Chapter 1. Then, new strategies for achieving fault tolerance in the ADM system are described in Section 3.2. Included is the development of a procedure for implementing triple mode redundancy (TMR) and a methodology for dealing with processor failure during self test.

3.1. Real-Time Control Strategy

Included in AMOS for the ADM is the capability for on-line real-time control of the system time performance. The ATAMM performance plane described in the previous chapter displays all possible operating points, with the resource requirement necessary to achieve the operating point shown as a parameter. A set of actual operating points is selected by the user by identifying one operating point for each resource number, R through 1. Each such point specifies the system time performance, that is TBIO and TBO, for a particular number of available resources.

The set of actual operating points selected in this way is compiled in a table called the operating point table. The operating point table forms the control law for implementing the real-time feedback control strategy for ADM. The calculation of performance bounds and construction of the SGP, TGP and performance plane diagrams has been automated in a software package called Design Tool. The software is being developed to operate on IBM PC/386 compatibles in the Microsoft Windows environment.

AMOS is designed to monitor continuously the number of available functional units. At any instant, the number of available resources is used to identify an operating point through the operating point table. If the number of resources changes, then a corresponding new operating point is identified. System operation at the new operating point is achieved by adjusting the injection control time interval, and by modification of the AMG through the addition or deletion of control edges and dummy nodes.

In the ADM system, the operating system counts the number of functional units available and communicates this number to the IBM PC/386 where the operating point table is stored. Using a simple table look-up procedure, an operating point is identified and the graph structure and injection rate necessary to realize this operation are specified. This information is communicated back to the operating system where the graph structure is changed and the input rate adjusted. Therefore, the entire feedback control process is integrated with the ATAMM operating system. The control methodology is shown in Figure 16.

3.2 Fault Tolerant Strategies

The present section is intended to summarize the Fault-Tolerant Strategies used to enhance the ATAMM strategy. The section is divided into two subsections: TMR Implementation and Fault Detection & Recovery.

The TMR implementation in ATAMM is performed at the graph level. The transformation of a graph to incorporate the TMR strategy is explained in detail. Fault detection and recovery are outlined in the last subsection.

3.2.1 TMR Implementation

TMR (Triple Modular Redundancy) is one of many Fault-Detection-and-Correction techniques that can be applied in the design of a reliable computing system. The philosophy behind TMR is to triplicate a given work or task to detect and correct faults. The detection is based on the comparison of the results of the multiple outcomes or outputs of the triplicated task. The correction of the fault is accomplished by selecting one out of the three outputs as the correct one. If there is an error in any of the three sets, the other two will be identical, hence the latter are assumed to be correct. This scheme is used to detect up to two faults but it can only correct one.

The implementation of TMR in ATAMM is achieved at the graph level. A graph without the application of TMR is said to be a simplex graph. A simplex graph is any normal graph defined to be executed under the rules of ATAMM. A TMR graph is a graph that implements the TMR strategy in all of its nodes. A TMR graph

can be expressed as a transformation of a simplex graph. A simplex graph can be transformed into a TMR graph by triplicating every node in it and by triplicating every output edge in every resultant node. The distinction of the three nodes per original node that are created with this process is made through the use of colors. The first node is labelled red, the second green, and the third blue. To refer to a given node in a TMR graph it is necessary to use not only its task number but its color. After the triplication is finished, the second operation is the connection of the output edges to their corresponding nodes. Let us assume nodes A and B are connected as shown in Figure 17(a). After the transformation there will be a connection from red node A to all three nodes B, another from green node A to all three nodes B and lastly one from blue node A to all three nodes B. The resulting TMR graph is shown in Figure 17(b).

The procedure explained above can be expressed mathematically in the following manner. A given simplex graph with n number of combined nodes, sources and sinks has square connection matrix for the AMG

$$\mathbf{AMG} = [c_{ij}] \quad i,j = 1, \ldots, n$$

where

$c_{ij} = 1$ if there is a directed edge from the i^{th} node to the j^{th} node.

$c_{ij} = 0$ otherwise.

A vector of m square matrices is

[M1 M2 M3 ... Mm]

In general, the connection matrix for a CMG is defined as the sum of two matrices, **EC** and **IC**. These two matrices are: the external connection of the nodes of the AMG and the internal connection of the AMG. These two matrices can only be defined in terms of the NMG. The connection matrix of the NMG that is being used in the definition of ATAMM can now be defined. The NMG is composed of three transitions, namely: read, process and write. The connection matrix that defines the internal connection of these transitions is

$$\mathbf{NMG} = [c_{ij}] \; i, j = 1,...,3$$

where

$c_{ij} = 1$ for the pairs (i, j) = (1, 2), (2, 3), (3, 1),

$c_{ij} = 0$ otherwise.

where 1 corresponds to the read transition, 2 to the process transition and 3 to the write transition.

Once the NMG is given, the external connection matrix **EC** can be defined. If nodes A has a directed edge to node B in an AMG, then the write transition of node A

has a directed edge to the read transition of node B. Also there is a directed edge from the read transition of node B to the read transition of node A. Therefore, the external connection matrix **EC** is defined as

$$\mathbf{EC} = [0\ 0\ I]^T\ [\mathbf{AMG}]\ [I\ 0\ 0] + [I\ 0\ 0]^T[\mathbf{AMG}]^T[I\ 0\ 0]$$

which is a 3n x 3n matrix where n is the number of nodes, sinks, and sources in the AMG. **I** is a n x n identity matrix and **0** is a n x n zero matrix.

The internal connection matrix **IC** is defined as follows

$$\mathbf{IC} = [I\ 0\ 0]^T[I][0\ I\ 0] +$$
$$[0\ I\ 0]^T[I][0\ 0\ I] +$$
$$[0\ 0\ I]^T[I][I\ 0\ 0]$$

then

$$\mathbf{CMG} = \mathbf{EC} + \mathbf{IC}$$

or

$$\mathbf{CMG} = \begin{bmatrix} \mathbf{AMG}^T & \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{I} \\ \mathbf{AMG+I} & \mathbf{0} & \mathbf{0} \end{bmatrix}$$

where **CMG** is a 3n x 3n connection matrix and n is the number of nodes, sinks, and sources in the **AMG**.

The connection matrix $\mathbf{AMG_3}$ of a TMR graph is expressed in terms of the connection matrix **AMG** of a simplex graph. This is

$$\mathbf{AMG_3} = [\mathbf{I}\ \mathbf{I}\ \mathbf{I}]^T\ \mathbf{AMG}\ [\mathbf{I}\ \mathbf{I}\ \mathbf{I}]$$

where $\mathbf{AMG_3}$ is a 3n x 3n connection matrix. The CMG connection matrix of a TMR graph is

$$\mathbf{CMG} = \begin{bmatrix} \mathbf{AMG_3}^T & \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{I} \\ \mathbf{AMG_3}+\mathbf{I} & \mathbf{0} & \mathbf{0} \end{bmatrix}$$

Each colored node (red, green and blue) reads three sets of the original simplex data sets. Each of these data sets comes from a colored node from the predecessor nodes. The first operation that a node has to execute is the comparison of the three data sets. This comparison is performed by a unit called voter. The voter compares all data sets and determines if there is any error on any of the sets. The output of the voter is divided in two parts. The first part is a data set to be used in the task of the node. The second part corresponds to an error report. There are three possible outcomes in the error report, they are: there is no error in the data sets; there is a

recoverable error in the data sets; and there is a fatal error. The first refers to the case where there is no difference among all three data sets. Any data set is then used as input to the node. The second error refers to the case where there was one set in disagreement with the other two. Any data set of the two in agreement is then used as the input set to the node. The color of the node that produced the erroneous data can be part of the error output. The third error refers to the case where there were not two data sets in agreement. That is, there is more than one error. In this instance any data set can be used since there is no way to determine which is in error or which is correct. This is a fatal error since this error propagates erroneous data throughout the graph. This error should flag an exemption to the operating system to take a corrective action.

3.2.2 Fault Detection and Recovery

Fault detection in ATAMM can be implemented at the graph manager [3] level. The graph manager is the part of the implementation of ATAMM that runs the graph. It scans the graph seeking enabled nodes and assigns resources that execute them. The graph manager assigns a resource to a computing node. The resource executes the given operation. After this resource has finished and delivered the output data to the appropriate data edges, the resource can be tested before it returns to be available to the system. This can be a self-test of the resource. The result of this test is then passed to the graph manager. Based on the test result, a decision is made whether the resource is able to continue being used by the system or has to be discarded from it.

Recovery is fulfilled by discarding the resource that reported an error during its test. The resource is not allowed back into the system by not been available for assignment by the graph manager. It is clear that the type of fault that can be handled is the one that can be detected by the resource itself. This type of fault includes faults in the subsystems of the resources that do not directly intervene in the execution of instructions, e.g., ALU operations, I/O operations, etc. This summarizes the fault detection and recovery of ATAMM.

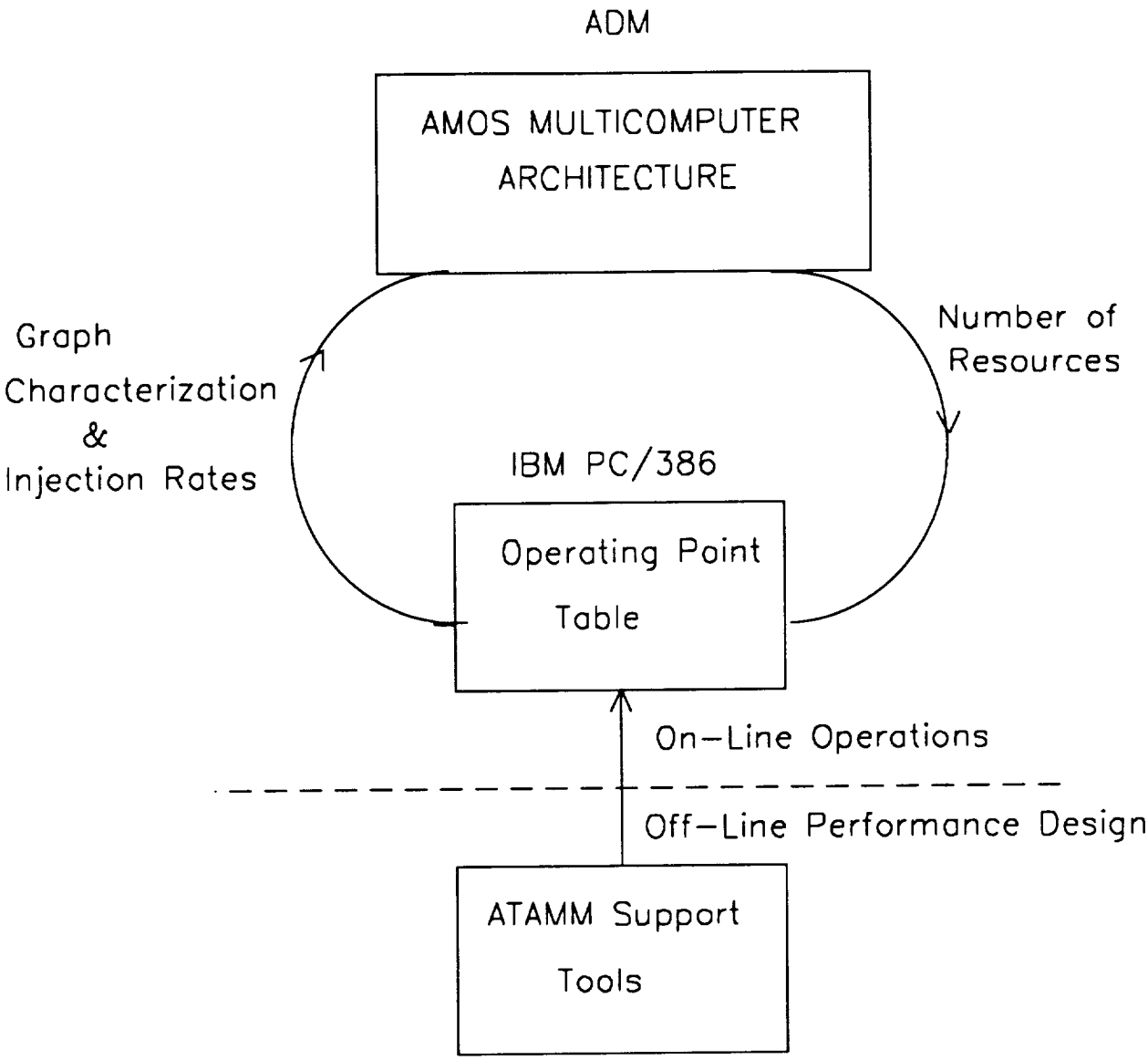
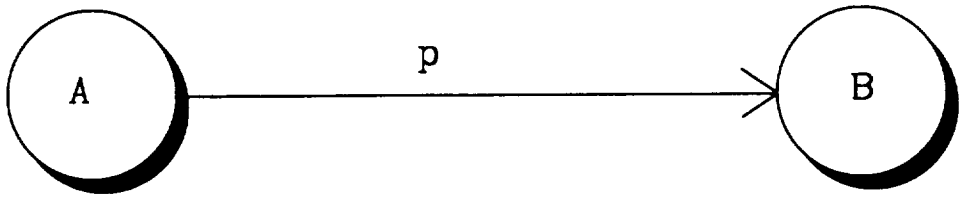
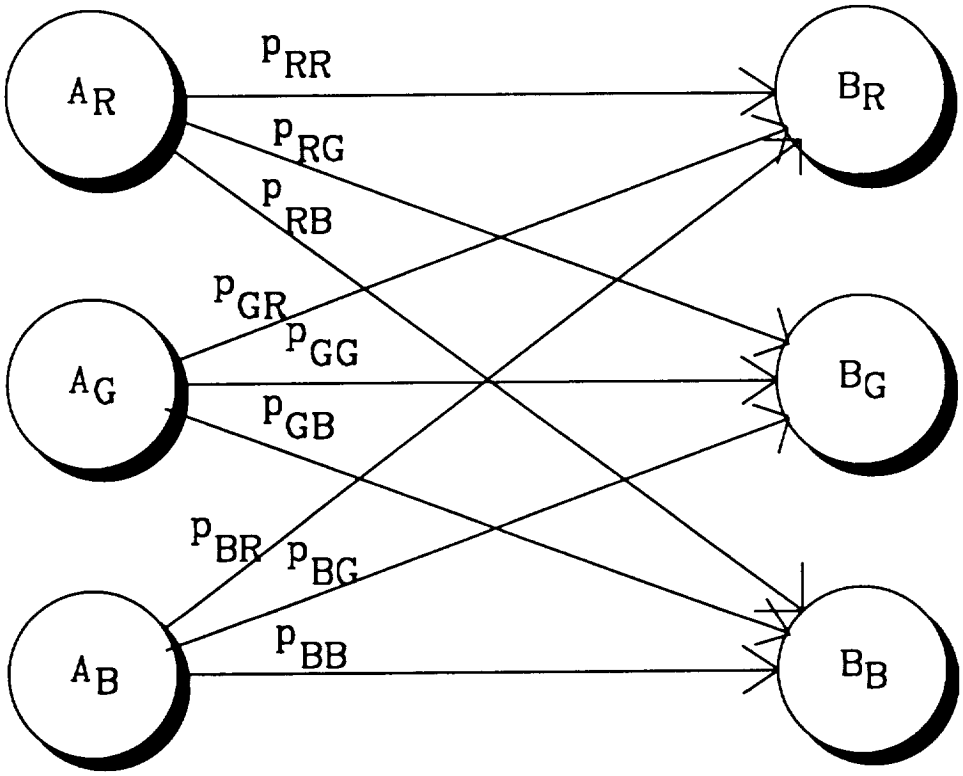


Figure 16. Control strategy for ADM system.



(a)



(b)

Figure 17. (a) Simplex and (b) TMR AMG node pair representation.

CHAPTER FOUR

ADM IMPLEMENTATION OF ATAMM

4.0 Introduction

In this chapter, the adaptation of the ATAMM model to the ADM system is described. The architecture of the ADM system is discussed in Section 4.1. In Section 4.2, the major components of the ATAMM Multicomputer Operating System (AMOS) are identified and AMOS operations are explained using a state diagram description. Then the software of 1553B and IBM PC/386 are described in Section 4.3 for input/output communication and real-time control.

4.1. ADM Architecture

A VHSIC ATAMM data flow architecture, called the Advanced Development Model (ADM), is under development [3]. For convenience, the ADM architecture is shown again in Figure 18. This system consists of four identical VHSIC 1750A processors which communicate over a dual PI bus. Each 1750A has a physical memory of size 256K which is used for storing all codes and data. Also connected to the PI bus is a 1553B which serves as a gateway for input and output data flow from an IBM PC/386. There are one 1553 communication module and one 1750A in a 1553B. The total physical memory size is 128K. Communications over the PI bus are accomplished by broadcasting and use of direct memory access and requires

exclusive control over a single PI bus semaphore. A processor or 1553B must grab the PI bus semaphore before communicating over the PI bus. All processors also communicate over an IEEE-488 bus to a Microvax computer which is used to download application programs and files for debugging activities. The 1553B module is connected to the IBM PC/386 by a single line communication link. Data are transferred between the 1553B module and the IBM PC/386 by synchronous communications. It is possible to perform logical operations in 1553B and communications over the PI bus and communication link concurrently. In addition to input and output, the communication link is used for fault injection, fault recovery, modification of the algorithm graph in real-time, and passing information back to IBM PC/386 for testing purposes. The 1553B acts as a source and sink for the algorithm marked graph and thus is capable of controlling the input injection rate to the 1750A processors and collecting output from the PI bus.

4.2 AMOS Description

The ATAMM Multicomputer Operating System (AMOS) is the operating system of the ADM hardware and its operation is based upon ATAMM rules. First, fundamental principles of AMOS are described. Second, a detailed description of AMOS data structures are presented. Third, an example is used to illustrate the operations of the operating system. Finally, the operations performed by a functional unit are elaborated.

4.2.1 Operating System Principles

AMOS is the logical interface among three components, the AMG, the graph manager, and a resource queue consisting of a set of identical functional units or resources as shown in Figure 19. The AMG represents the computational problem. The graph manager updates and monitors the status of the algorithm marked graph. When an AMG node is enabled, the graph manager assigns a functional unit from the queue of available functional units to perform the corresponding algorithm operation. The functional unit is the component which executes all three node marked graph (NMG) transitions of each AMG node. The functional unit communicates with the graph manager to update the status of the AMG and other functional units to read and write data. The graph manager and functional units communicate by the PI bus through a resident operating system called Modified Kernel Operating System.

Each of the 1750A is a resource or functional unit of the AMOS. The graph manager and AMG also are distributed among all functional units. All the 1750As maintain an identical copy of the graph manager, MKOS, AMG, application codes, and data for the AMG nodes. The distribution of system components on the ADM is shown pictorially in Figure 20. However, only the graph manager residing in the functional unit on top of the queue of available functional units can fire an enabled AMG node. In order to ensure that all functional units have an identical copy of the graph data structure, the graph manager of a functional unit grabs the PI bus semaphore before changing the graph data structure. A broadcast is issued to all other 1750As and the 1553B to update the respective graph data structures. After waiting

for a predetermined time interval to allow the updating to complete, the functional unit releases the PI bus for other communication. This distribution of activities has the advantage of increasing the number of functional units in the system and at the same time improving the potential for achieving a higher degree of fault tolerance to functional unit failure.

The operation of a functional unit is represented by the state diagram shown in Figure 21. Initially, all functional units awake in the state labeled Idle. A functional unit remains in this state until its identifier appears at the top of the resource queue. When this occurs, the functional unit undergoes a state transition to the Examine Graph state. In this state, the functional unit actively monitors the status of the AMG until an algorithm node becomes enabled. When an enabled algorithm node is identified, the functional unit assigns itself to perform the algorithm operation, grabs the PI bus, and undergoes another state transition to the Execute state. During this state transition, the functional unit identifier is removed from the top of the resource queue and a bus communication F announcing that an algorithm operation has been initiated is broadcast on the bus. Then, the functional unit releases the PI bus. The functional unit remains in the Execute state until the algorithm operation is complete. At the completion of the algorithm operation, the functional unit grabs the PI bus semaphore and initiates a second bus communication D which includes a broadcast of the algorithm operation output data to all other functional unit global memories. At this time the functional unit again changes state to the Self Test state and releases the PI bus. The Self Test state corresponds to a diagnostic check of the functional unit.

After a successful self test, the functional unit returns to the initial Idle state. This state transition is accompanied by a grabbing of the PI bus semaphore, a third bus broadcast communication R announcing that the functional unit identifier should be returned to the bottom of the resource queue, and a release of PI bus. While in any state, the functional unit may be interrupted to update its graph data structures and resource queues following F, D, or R broadcasts from other functional units.

4.2.2 Data Structures

The data structures of AMOS consist of two arrays, BLOCKS and EDGES, that hold all of the information regarding nodes and edges of an algorithm marked graph. Also, there is a table, PRIORITY, that holds the precedence order of the nodes of the algorithm graph. In addition, there are four queues, QUEUE, WORK, DIAG, and RECOV that hold information about current status of functional units. The QUEUE is a FIFO queue of available and unassigned functional units, WORK is a pool of assigned functional units, DIAG is a pool of functional units in a diagnostic state, and RECOV is a pool of functional units to be recovered by the system. In this section a detailed description of these data structures are presented.

Every functional unit (1750As) has an instance of AMOS. After every F, D, or R events, the graph and resource structures are updated by the individual 1750As separately. The variables BLOCKS, EDGES, PRIORITY, QUEUE, and etc. are defined as arrays. Although these variables are defined as arrays, they are treated as

linked lists, i.e., the linked list is implemented using array indices. The linked list structure reflects the dynamic structure inherent in this architecture model.

A block is a node of AMG. In TMR mode, it is a set of three colored-nodes, red, green, and blue and in SIMPLEX mode a set of only one node. Its primary use is in TMR mode. FIRING is a global variable that holds the identification code (ID) of the block being fired. It is used to ensure that all of the colored-nodes of the block are fired before firing the next block. If there is no block being fired, then it is set to zero. MODE, a global variable, indicates the mode of operation and is initially set by the user to SIMPLEX, 1, or TMR, 3. In TMR mode, when the number of functional units drops to less than three, AMOS will change the value of MODE to SIMPLEX to reflect the decrease in the number of functioning resources. BLOCKS is an array of N elements with components BLOCKS[j], the range of j is from 0 to N, where N represents the number of nodes in the AMG graph. EDGES is an array of M elements with components EDGES[k], the range of k is from 0 to M, where M represents total number of edges in the AMG graph. QUEUE, WORK, DIAG, and RECOV are arrays of size equal to the maximum number of available functional units at the start up. These arrays are described in the following paragraphs.

BLOCKS: BLOCKS[j] is an element of the array BLOCKS and holds all information about a block. BLOCKS[j] consists of nine variables which are explained below. FUNCTION_ID is an integer representing the task ID or a pointer pointing to the application program. ID is a three element array which holds the identification code of functional units assigned to the colored-nodes of the block. ID is used to keep track

of functional units for future recovery purposes. BUSY_CTR is a counter that holds the number of functional units working on the block. It is incremented after every F-transition command and decremented after every D-transition command. Another variable, DONE_CTR is a counter that holds the number of functional units released from the block. It is used to check if a block can be enabled. It is set to zero when the block is enabled and is incremented by every D-transition. ENABLE_CTR is a counter that holds the number of enabled colored-nodes that have not yet fired. When the block is firable the ENABLE_CTR is set to the MODE of operation. It is decremented after every colored-node of a block is fired (F-transition). INPUTS is an array of pointers having components INPUTS[i], where the range of i is from 0 to 2. INPUTS[i] is the header pointer pointing to a linked list of input (incoming data) edges to the ith colored-node. Another variable, OUTPUTS is an array of pointers having components OUTPUTS[i], where the range of i is from 0 to 2. OUTPUTS[i] is the header pointer pointing to a linked list of output (outgoing data) edges originating from the ith colored-node. (It implicitly represents all backward control edges from all successor nodes to this node.) Figure 22 is a pictorial representation of these two linked lists. IN_SUMMARY is an array of integers with components IN_SUMMARY[i], where the range of i is from 0 to 2. IN_SUMMARY[i] is a summary of INPUTS[i] and is an integer having a value equal to the number of input edges of the ith colored-node when all have data and is zero otherwise. OUT_SUMMARY is an array of integers with components OUT_SUMMARY[i], where the range of i is from 0 to 2. OUT_SUMMARY[i] is a summary of

OUTPUTS[i] and is an integer having value equal to the number of outgoing edges originating from the ith colored-node when all are empty and is zero otherwise. A block is enabled under the following conditions:

1. DONE_CTR = MODE,
2. All IN_SUMMARY[i]s, i = 0..2, are non-zero, and
3. All OUT_SUMMARY[i]s, i = 0..2, are non-zero.

EDGES: EDGES[k] is an element of the array EDGES and holds all information about an edge. EDGES[k] consists of eleven variables which are described in the following. EDGE_QUEUE is a circular linked list that holds addresses of the memory locations where the data are stored. The addresses are accessible to the INITIAL and TERMINAL blocks to write and read data, respectively. For future recovery purposes the length of the queue, L, is one more than the SEGMENTS or, number of Dummy nodes plus two. Structure of each element of the EDGE_QUEUE consists of three elements; a) LABEL is a pointer to the beginning of the data container, b) ID holds the identification code of the functional unit which wrote the data into that data container, and c) NEXT is a pointer to the next element of the EDGE_QUEUE. SEGMENTS is an integer equal to the number of dummy nodes on the edge plus one. It is used to check capacity of the EDGE_QUEUE of the edge. If SEGMENTS is equal to ITEMS, then EDGE_QUEUE is full and no more data can be written into it. ITEMS is a counter indicating the number of data items on the edge. The range of ITEMS is from zero to SEGMENTS. It is incremented, by the INITIAL node, every time new data are written on the edge. It is decremented, by the TERMINAL node,

every time OUTPUT_WIDTH becomes zero. INITIAL holds the block number of the origin of the edge. It is used to update the graph and can also be used to check the integrity of the graph. TERMINAL holds the block number of the destination of the edge. It is used to update the graph. EDGE_COLOR indicates the color of the INITIAL node of the edge. It is also used to update the graph. The value of color is identified as 1 for red, 2 for green, and 3 for blue. OUTPUT_WIDTH, a counter, is set to MODE when its present value is zero and ITEMS is non-zero. It is decremented by one for each F-transition of the TERMINAL block. TERMINAL_PTR is a pointer to the element of the EDGE_QUEUE where the TERMINAL node reads data. It is updated every time OUTPUT_WIDTH becomes zero. Updating TERMINAL_PTR means that it should be pointing to the next element of the EDGE_QUEUE. Updating is performed by the TERMINAL node. INITIAL_PTR is a pointer to the element of the EDGE_QUEUE where the INITIAL node writes data. It is updated every time an output is written to the edge. Updating INITIAL_PTR means that it should be pointing to the next element of the EDGE_QUEUE. Updating is performed by the INITIAL node. NEXT_INPUT is a pointer to the next edge which is an input edge to the TERMINAL block. NEXT_OUTPUT is a pointer to the next edge which is an output edge of the INITIAL block. NEXT_INPUT and NEXT_OUTPUT are used to examine all of the input and output edges of a block, respectively.

QUEUE: QUEUE is a FIFO queue holding information about available and unassigned functional units. Each element of the QUEUE is a record of three

components ID, COLOR, and NEXT. ID holds the identification code of an available functional unit. COLOR is a variable containing the color of the colored-node of the enabled block that the functional unit will process. COLOR carries relevant information only when it belongs to one of the top MODE elements of the QUEUE. The COLOR value is assigned according to the position of the functional unit in the top of the QUEUE; first red, second green, and third blue. NEXT holds the index of the next element of the QUEUE. It is used to treat QUEUE as a linked list. If NEXT is zero, then there are no more elements in the list. The first element of QUEUE is used as a dummy head node of the linked list and to keep track of content of the array. Note that the COLOR field of the first element holds the number of functional units in the array.

WORK, DIAG: WORK and DIAG have the same structure as QUEUE but are treated differently. WORK is a pool holding identification codes of all functional units which have been processing nodes. DIAG is a pool holding IDs of functional units which are in a diagnostic state.

PRIORITY: It is an array holding block numbers. The position in the array determines the block's priority. The block at the first element is the block with the highest priority in the graph.

4.2.3 Example

This example is provided to give more insight to the data structures of AMOS. Figure 23 is part of a graph considered for this example. In this example, the focus is

on the node labeled E and all of the changes regarding these nodes are depicted in the following figures. The mode of operation is SIMPLEX. Figure 24 is a pictorial representation of the data structure and contents of QUEUE, WORK, DIAG, EDGES, and BLOCKS. The initial contents of EDGES[i], the range of i is from 0 to 3, are shown in Figure 25. Figure 26 depicts the structure of EDGE_QUEUEs of all the edges. Note that the edge from E to C has a dummy node and thus the length of its EDGE_QUEUE is one more than other edges. The read and write pointers of the edges are also shown in this figure. The initial contents of block E are shown in Figure 27. When block E is enabled, the ENABLE_CTR is set to the MODE and IN_SUMMARY is cleared as shown in Figure 28. The functional unit assigned to the block E is transferred from the QUEUE to WORK and starts reading inputs to the block E. After reading the input on the AE edge, ITEMS of AE edge is decremented and the read pointer of the block E concerning this edge, E_Read, is advanced. The NEXT_INPUT field of AE edge provides the block E with the information about next input to the block as in Figure 29. After reading the input on the BE edge, ITEMS of BE edge is decremented and the read pointer of the block E concerning this edge, E_Read, is advanced. The NEXT_INPUT field of BE edge provides the block E with the information about next input to the block. The value of that pointer for this graph is now Nil indicating the end of reading process for the block E as shown in Figure 30. While processing the application program as described in Figure 31, there are no changes in the data structure of block E. After writing to edge ED, ITEMS of ED edge is incremented. The NEXT_OUTPUT field of ED edge provides the block E

with information about next output edge of the block. Also the write pointer of the ED edge, E_Write, is advanced so that the block E can write to the new place next time as shown in Figure 32. After writing to edge EC, ITEMS of EC edge is incremented. Also the write pointer of the EC edge, E_Write, is advanced so that the block E can write to the new place next time. The NEXT_OUTPUT field of EC edge provides the block E with information about next output edge of the block. A pointer with a value of Nil indicates the end of the process. At this point, the graph is updated before broadcasting as described in Figure 33. After writing the output data and broadcasting the updated graph, the functional unit migrates from the WORK to DIAG to perform a self test. Note that the DONE_CTR is set to the MODE of operation as in Figure 34.

4.2.4 Functional Unit Operations

Every functional unit has an instance of AMOS. Although every functional unit has knowledge of status of other processors, it acts as a stand alone entity. Every functional unit goes through a sequence of operations as shown in Figure 35. These operations together form the states of the resource state diagram of Figure 21. Implementation of AMOS is best understood by examining these operations of functional units.

Idle: When in Idle, the functional unit examines the QUEUE continuously. First QUEUE is checked to determine if there are at least as many functional units in the QUEUE as the MODE of the system by examining QUEUE[0].ID. Second, QUEUE

is checked to determine if QUEUE[i].ID is the same as its own ID, where the range of i is from 0 to MODE. If the self identification is successful, the search is then made for an enabled block based on priorities assigned to the blocks. An enabled block is detected by examining IN_SUMMARY[i], OUT_SUMMARY[i], and the DONE_CTR. The variables IN_SUMMARY[i] and OUT_SUMMARY[i] are, in this state, assigned proper values by examining all input and output edges of the block. This search continues until an enabled block is found. Having a block to execute, the functional unit selects a colored-node, based on its position in the QUEUE, to fire. The global variable FIRING is set to the ID of the block. After FIRING holds a valid ID, the functional unit selects the appropriate colored-node of that block to execute and changes state to On_Hold_Read state.

On_Hold_Read: While in On_Hold_Read state, the functional unit is constantly trying to get control of a communication channel. The reason being that the functional unit must inform all other functional units before the execution of the AMG node begins. Duration of this state depends on the traffic and communication channel protocol.

Update and Read: After establishing a communication link, the functional unit conducts a second search for enablement of nodes with higher priorities than the previously enabled nodes. Selecting a node with the highest priority, the functional unit migrates from the QUEUE to the pool of working functional units(WORK). The variable BUSY_CTR is incremented and ENABLE_CTR is decremented. It then updates its copy of the graph and instructs all other processors and 1553B to do the same. This broadcast is called a F event. The communication channel is then

released. Reading of input data begins after releasing the channel. After reading every input, the variable OUTPUT_WIDTH of the corresponding edge is decremented. If the current value of OUTPUT_WIDTH is zero, the pointer TERMINAL_PTR of that edge is advanced and the variable ITEMS is decremented. In TMR mode, the functional unit votes on the three sets of inputs and chooses the correct set for processing.

Process: In this state, the functional unit executes the application program. To do so, control is passed to the application program. Upon completion of the task, control is passed back to AMOS. Duration of this state is the same as execution time of the application program.

On_Hold_Write: To write the generated outputs, the functional unit has to get control of a communication channel. Duration of this state depends on the traffic and communication channel protocol.

Update And Write: After establishing a communication link, the functional unit identification is removed from the WORK queue to the diagnostics queue (DIAG). The DONE_CTR is incremented and the BUSY_CTR is decremented. In this state the functional unit writes the output data to the memory locations associated with the appropriate edge. The variable ITEMS of the corresponding edge is incremented and the pointer INITIAL_PTR of that edge is advanced. The functional unit updates its copy of the graph and then broadcasts data and instruction to update graph structure in other processors and 1553B. This broadcast is called a D event. If an error is detected in the Read state, the color of the node and ID of the functional unit

responsible for the error are sent along with the D event. The communication channel is then released.

Test: In this state the functional unit performs a self test. Upon completion, the functional unit requests for a channel. Duration of this state depends on the test routine.

On_Hold_Update: To let the system know about its availability to undertake a task, the functional unit needs to grab a communication channel.

Update: After establishing a communication link, the functional unit identification is removed from the DIAG queue and placed in QUEUE, if the self test was successful. Otherwise, it simply removes itself from the diagnostics queue. In any case, the functional unit broadcasts the updated resource queues (this broadcast is called a R event) and releases the communication channel.

4.3 1553B Software

Four major tasks are performed by the code of 1553B. First, communication between 1553B and IBM PC/386 are controlled by the code of 1553B. Second, source and sink for a computational problem are implemented in the 1553B. Third, all FDTs (FDT stands for Fire, Data, Time) are received from 1750As and time tagged. The format for the FDT is described in Figure 36. A binary coding is used and each FDT is 8 words (128 bit) long. The meaning and size of binary codes of the FDT are also described in the above figure. Fourth, the code of 1553B is used to control the time

between successive inputs (TBI) to 1750As, to pass control words from the IBM PC/386 to 1750As, and to pass back status information of 1750As to the IBM PC/386.

Both 1553B and IBM PC/386 have a transmit and a receive buffer. The contents of the transmit buffer of the 1553B is written onto the receive buffer of the IBM PC/386 periodically. Similarly, the content of the transmit buffer of the IBM PC/386 is transmitted to the receive buffer of the 1553B at the same periodic rate. The maximum size of the transmit and the receive buffers are 32 addresses where each address points to a 32 word (each word is 16 bit long) data. The contents of the transmit and the receive buffers are described in Figure 37. The transmit buffer of the 1553B and the receive buffer of the IBM PC/386 are divided in output buffer, FDT buffer, status AMOS buffer, and status 1553 buffer. The first word of the output and FDT buffers indicates that length of output data and the number of FDTs, respectively. The size of the output and FDT buffers are specified during initialization depending on the maximum requirement. The status AMOS buffer is a 32 word long buffer indicating status of AMOS which is updated at every broadcast by 1750As. The contents of the status 1553 buffer are an input flag, an output flag, a FDT flag, a control flag, an error code, and an error flag. The error flag the and error code are used to indicate overflow in either FDT or output queues in the 1553B. All other flags are needed for handshaking purposes. A flag in transmit and receive buffers is indicated by (T) and (R), respectively. For example, the input flag (T) of the 1553B is representing the input flag in the transmit buffer of 1553B. The transmit buffer of the IBM PC/386 and the receive buffer of the 1553B are organized as an input buffer, a

control AMOS buffer, and a flag buffer. The size of the input buffer is specified during initialization. The first word of the input buffer is an indicator for the length of input data. Input and output queues of size 2 and a FDT queue of size 64 are maintained for temporary storage of inputs, outputs, and FDTs. The input queue is used to store the input after collecting it from the input buffer. The output and FDT queues are used to store the output and FDTs before transferring them to their respective buffers. The control AMOS buffer is 32 word long and contains commands to 1750As. The flags contain an input flag, an output flag, a FDT flag, a control flag, and the minimum injection interval (T). Control AMOS and T are updated by a control block and a table containing choices for injection interval T by the IBM PC/386 software.

Although communication between 1553B and IBM PC/386 is periodic, flags are used to prevent overwriting on top of data which is not yet read and also reading of the same data more than once. For every type of data, there are four flags. For example, there are output (T), output (R) in 1553B and output (T), output (R) in AT for the output data. The following rules for interpreting and changing these flags ensures safeness in communication. All the flags are to be initially reset.

Transmitting Side (1553B or IBM PC/386): If flags in the transmit and the receive buffer are the same, the receiver has picked up previous data. It is safe to place new data into the transmit buffer. The corresponding (T) flag in the transmit buffer is toggled to indicate to the receiver that a new data has been placed. If the (T) and (R)

flags are not the same, the receiver has not acknowledged that the previous data was picked up.

Receiving Side (1553B or IBM PC/386): If flags in the transmit and the receive buffer are the same, the data in the buffer has not changed. If the corresponding (T) and (R) flags in the receive buffer are not the same, the data in the buffer is different from the previous one. Data is read and the corresponding (T) flag is toggled to indicate the same to the transmitter.

As an example, suppose an algorithm output is to be sent to the PC. The output (T) and output (R) flags are compared in the 1553B. If the flags are the same, new output data is deposited in the output buffer of the 1553B and its output (T) flag is toggled. In the next periodic communication, the contents of the output buffer and the output flag (T) are transferred to its respective locations in the PC side. When comparing the output (T) and (R) flags in PC, the two flags will be found unequal. Hence the code in PC will be able to detect that a new output has arrived. The output is read and the output (T) flag in the PC is toggled which will be transmitted to the 1553B as an acknowledgment.

The main routine for 1553B is a continuous loop routine consisting of three tasks which are performed by subroutines A, B, and C. A detailed flow chart is shown in Figure 38 (a) through (d). After an initialization process in the main routine, a continuous loop is executed whose first step is execution of subroutine A. In subroutine A shown in Figure 38(b), it is checked whether any information needs to be transmitted to or received from the IBM PC/386. If the input flag in the transmit and

the receive buffer of the 1553B are not the same (which indicates arrival of a new input from the IBM PC/386) and the input queue (a double buffer) in the 1553B is not full, the input from the input buffer is transferred into the input queue. After that the input flag (T) is toggled to indicate to the IBM PC/386 that the input has been received. Similarly, the output and FDTs are transferred from their respective queues to buffers for transmitting to the IBM PC/386 and then their respective flags are toggled in the transmit buffer. In case of a new control AMOS from the IBM PC/386, PI bus is grabbed, control AMOS is copied onto status AMOS in 1553B. Then status AMOS is broadcasted to all 1750As followed by release of the PI bus and toggling of control flag (T) in the 1553B. Both Source and Sink are considered as node '0' by AMOS. If an input data packet is available in the 1553B, TBI is more than or equal to the injection interval (T), and the algorithm is ready to accept new input (indicated by the absence of tokens on all output edges of node '0' in the algorithm graph), subroutine B is executed (Figure 38(c)). The 1553B is instructed to grab the PI bus semaphore, update graph data structure to indicate injection of a new input, and broadcast a D event which include input data and instruction for updating the graph data structure. After that the FDT for the input injection is time tagged and put into FDT queue and queue length is updated. If the FDT queue is full, an error flag and an error code are set. Following this, the 1553B releases the PI bus and returns to the main routine.

If an algorithm output is generated (indicated by the presence of tokens on all the input edges of node '0'), subroutine C is executed as shown in Figure 38(c). If

the mode of operation is TMR or duplex, a vote is taken among data on all input edges of node '0' to generate the final output data. A header for F event is generated for the output. Then the PI bus is grabbed, tokens are removed from all input edges of the node '0', and an instruction is sent to all processors to do the same. The F header is time tagged and put into the FDT queue. The output is stored in the double buffer of the output queue. In case the FDT or the output queue are full, an error flag is set and the error code is reset. Then the PI bus is released and execution is returned to the main loop.

While in this continuous checking for communication with the IBM PC/386, input injection, and arrival of output, the code of 1553B can jump out to a DMA routine following a direct memory access data transfer from a 1750A. In this routine, the header (FDT) from the graph structure is time tagged as shown in Figure 38(d). If it is an output node (any node feeding node '0'), the header also is stored in memory location reserved for its color. Then the FDT is stored in the FDT queue. In case, FDT queue is full, the error flag and the error code are set. After that the execution control is returned to an instruction in the main routine from where it jumped to the DMA routine.

4.4 IBM PC/386 Software

All inputs for the application algorithm are initially stored in an input file in the IBM PC/386. All outputs and FDTS are also accumulated in output and FDT files, respectively in the IBM PC/386. In addition, a 32 word control instruction

(control AMOS) is sent to the 1553B and a 32 word status information (status AMOS) is received from the 1553B. A 32 word buffer (control block) is maintained in the IBM PC/386 for accumulating all desired changes in the system which is finally copied onto the control AMOS. The control block is shown in Figure 39 which includes commands for fault injection and for algorithm modification by dummy nodes and control edges. The features of the IBM PC/386 software are described below before a detailed description of the software organization.

Major tasks of the IBM PC/386 are as follows. The code of IBM PC/386 has to set up input and the control block for 1553B and has to collect outputs, the FDT, and the status AMOS from the 1553B. The code also is used to check on conditions for errors and for any changes in the number of 1750As. Depending on the number of 1750As in the system, a minimum injection interval (T) and an algorithm modification table are selected. The modify table is used to specify how the original algorithm is to be changed with dummy nodes and control edges to match the number of functional units (NFU). The modify table is written onto words 6 to 31 of the control block as commands for changes as shown in Figure 39. There is a two word instruction in the control block or the modify table for specifying dummy nodes on an single edge or a single control edge. For example, words 6 and 7 will specify the first algorithm modification from the original graph. The high order byte of word 6 indicates a command to specify whether the change is insertion/deletion of dummy nodes or a control edge. The low order byte of word 6 and high order byte of word 7 specifies initial (predecessor) and terminal (successor) node of the control edge or the algorithm

edge on which dummy nodes are to be inserted or deleted. The low order byte of word 7 is an indicator of the total number of dummy nodes.

Another feature of the IBM PC/386 code is the ability of fault injection which is needed for testing purposes. The user can instruct to inject faults based on the number of input data being processed. The fault conditions are stored as error injection tables which can be accessed by a pointer EPTR (Figure 40(b)). At proper time, an error injection table is selected and written into words 3 to 5 of the control block as shown in Figure 39. The high and low byte of word 3 of the control block is used to instruct AMOS to remove and insert a functional unit of specified identification (ID), respectively from the pool of functional units. The word 4 and the high order byte of 5 of the control block is used to instruct to remove a specified 1750A while in Execute, Self Test or Idle state (with this instruction, the specified functional unit stops communicating and processing, abruptly). The low order byte of word 5 of the control block is used to instruct a 1750A to commit a computation error.

Also, the IBM PC/386 has a timer which runs out if there is no new data exchange with the 1553B in a specified time period. This timer is initialized by an user specified value and is updated by its initial value each time a new data is transmitted or received from the IBM PC/386. It can initiate a recovery process if a functional unit does not respond in the Self Test state. A counter is maintained to keep track of number of inputs being processed. This counter is used to insert faults after a specific number of inputs are executed. After all the inputs are processed, an end of file marker (EOF) is set. The software of IBM PC/386 is used to analyze

FDTs for detecting errors or changing number of functional units before storing them into the FDT file. There is an error counter for each functional unit which is compared against a maximum tolerable value (Error_Limit). A detailed flow chart for the IBM PC/386 software is described in Figure 40 (a) through (f). The code of IBM PC/386 can be executed while data are read from or written to IBM PC/386 buffers by the 1553B.

After initialization in the main routine, a loop is executed for checking conditions of termination, recovery, updating error counters, injection of the input and the control block, and arrival of the output or FDTs. If the Stop signal is set (an interrupt routine sets the Stop signal following an user input as shown in Figure 40(c)), the code execution is terminated. The execution is also terminated if the timer is out and EOF is set, the error flag is set, or the fatal error flag is set. If the timer specified time has passed but all inputs have not been processed, this is an indicator of a fault and the recovery process is initiated. If the error flag is set, it means either overflow in the output or the FDT queue in the 1553B and the program is therefore terminated. The fatal error flag is the indicator that all three inputs in a TMR voting were different and the program is terminated.

If the input flag (T) and (R) are the same in the IBM PC/386, a subroutine called input is executed as shown in Figure 40(b). First it is checked whether there is a new input for processing in the input file. If there is no such input, end of file marker (EOF) is set and the execution of subroutine input is complete. Otherwise, a new input is transferred to the input buffer. When the counter is at a pre-specified

number, an error injection table is selected based on the value of EPTR and is placed on words 3 to 5 in the control block. After the fault injection, EPTR is incremented by one and is checked against a constant Max_Tables (10 for a maximum of ten error injection tables). When the EPTR is more than Max_Tables, it is reset to 1 so that no more faults are injected. Whether a fault is injected or not, the input counter is increased by 1 and the input flag is toggled before returning from the subroutine.

If the output flag in the transmit and the receive buffer are not the same (indicating arrival of a new input), a subroutine called output is executed as shown in Figure 40(c). Output is transferred to an output file and the output flag is toggled.

If FDT flags (T) and (R) are not the same, it is an indicator of arrival of new FDTs in the FDT buffer. Then subroutine FDT is executed as described in Figure 40(d). For each FDT, the type of event is checked. If event is a D, the fatal error bit is examined. If it is set, the fatal error flag is set. The next task of the code is to check for voting error in the D event. If any, the corresponding error counters for the respective functional units (the one which produced wrong results) are increased. If event is a R, the check bit is examined. If the check bit is set, action bit is checked. If the action bit is 1 (0) , the number of functional units (NFU) is increased (decreased) by the number of changes. A new T is picked from the injection table and a modify table is selected for the integer lower bound of the ratio (NFU/mode). The selected modify table is transferred to parts of the control block and the change bit is set. After the above processing of all FDTs for on-line control, FDTs are put into a FDT file, FDT flag (T) is toggled, and the execution is returned to the main routine.

After returning from subroutine FDT, subroutine check FUN error counters is executed (Figure 40(e)) to check if a functional unit error counter is more than a pre-specified value Error_Limit. If so, change is set and the corresponding functional unit ID is put into the remove ID of the control block. Then error counter for the functional unit which is removed is reset to zero. When this control block will reach 1750As, the corresponding functional unit will be removed from the pool of functional units. After that execution sequence is returned to the main routine.

The next decision block in the main routine is used to check whether change bit is set which indicate that a new control instruction is in the control block. If so and also if control flags (T) and (R) are the same, subroutine control as described in Figure 40(f) is executed. The control block is copied onto the control AMOS and the control flag (T) is toggled. Then, status AMOS from receive buffer is transferred onto the control block and the change is reset. This is done to check for any diagnostic warning from 1750As. If Diagnostic Warning byte is nonzero, it is an indicator that a functional unit (specified by an ID) is spending too many cycles in the Self Test state. In that case Diagnostic Warning byte is copied into Recover in Self Test byte in the control block and change is set for removing the functional unit. Then execution is returned to the main routine.

The last decision block in the main routine is to reinitialize the timer in case there was a data transfer between PC and 1553B. After that execution sequence is returned to the beginning of the main routine loop.

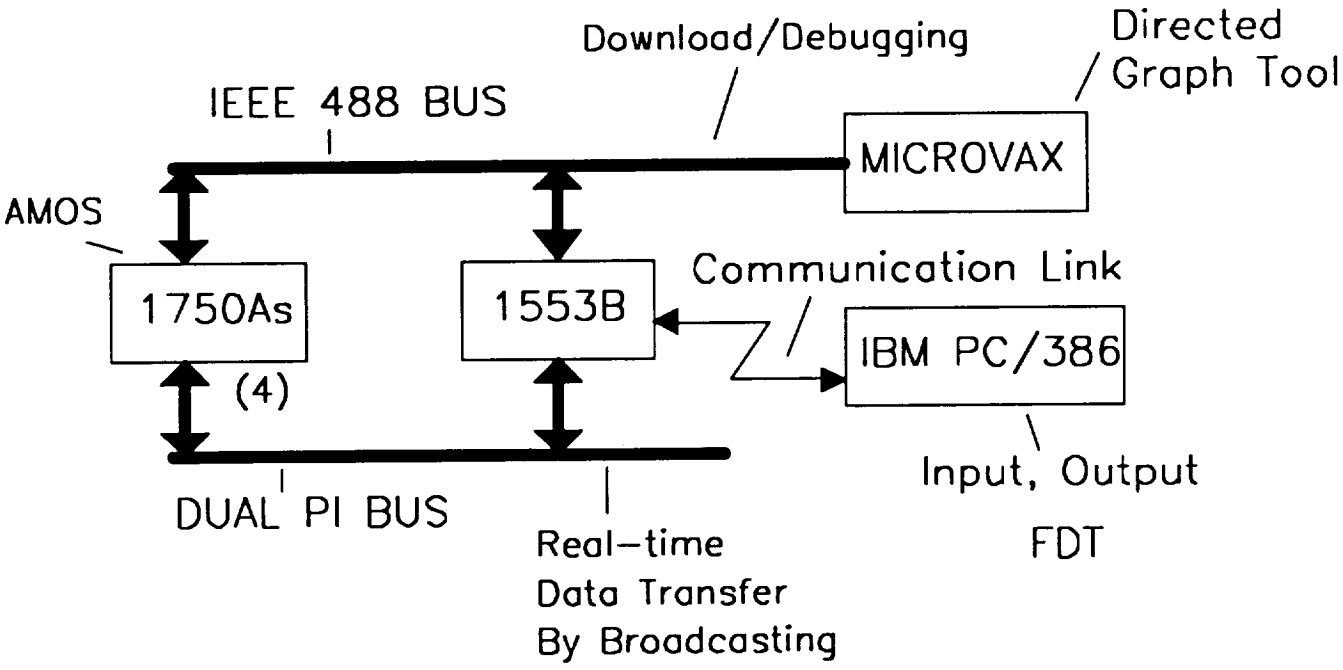


Figure 18. ADM testbed.

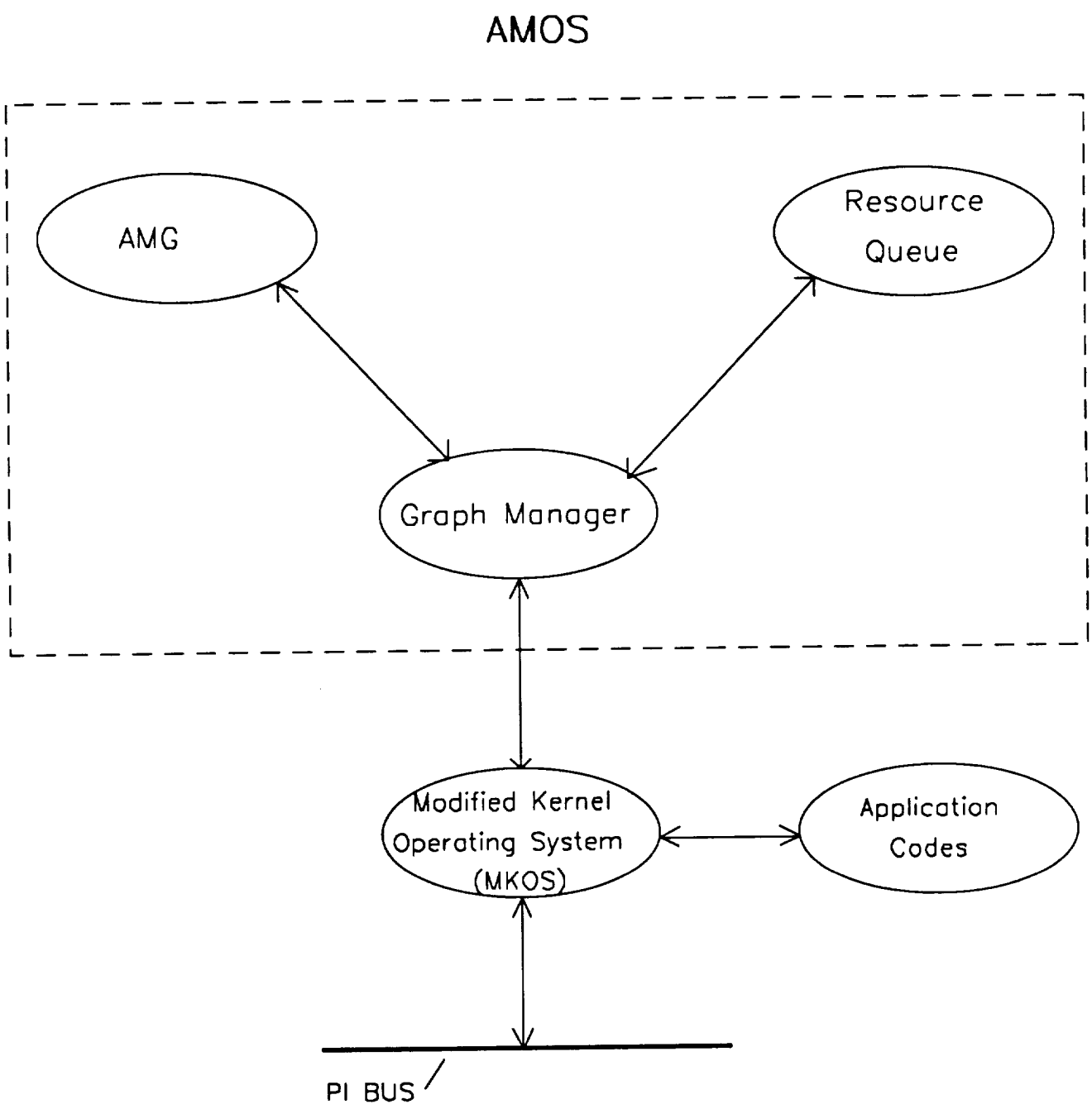


Figure 19. Logical components of AMOS.

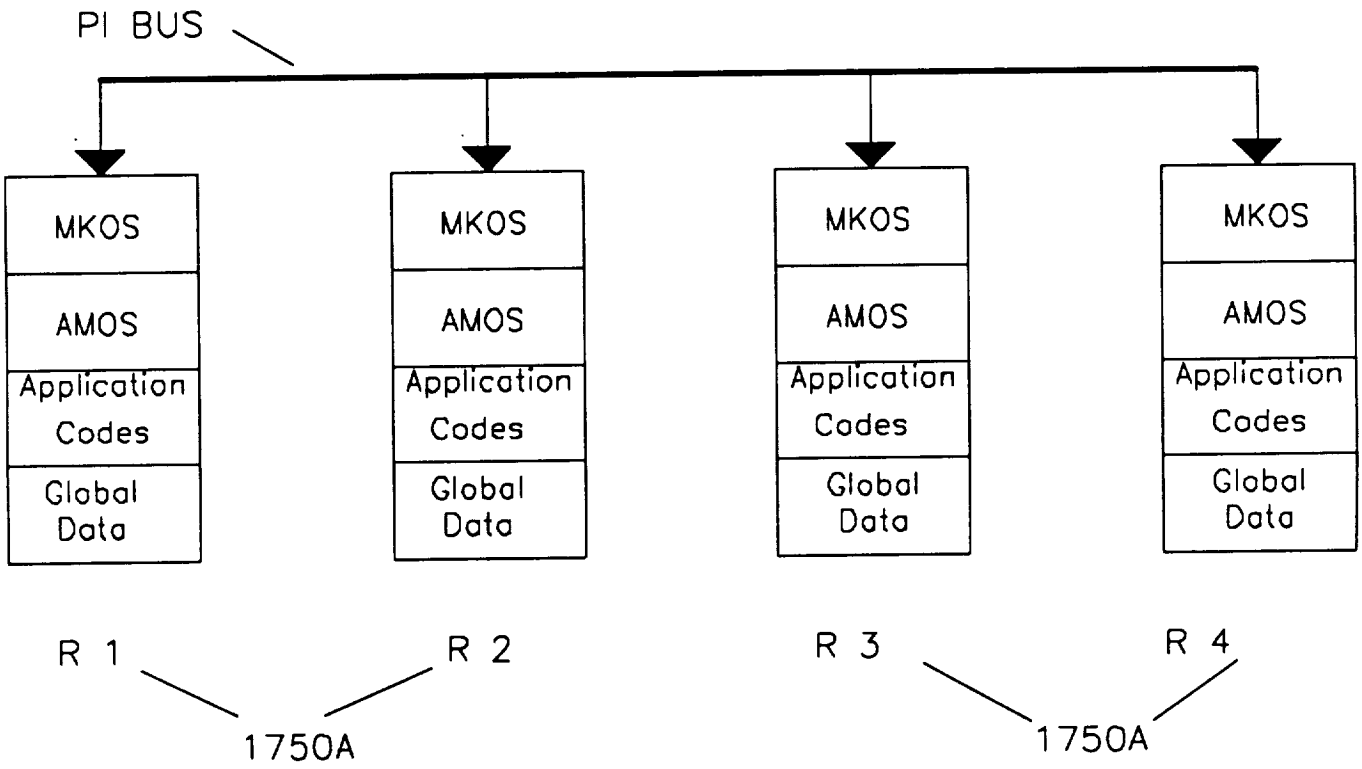


Figure 20. ATAMM implementation in ADM.

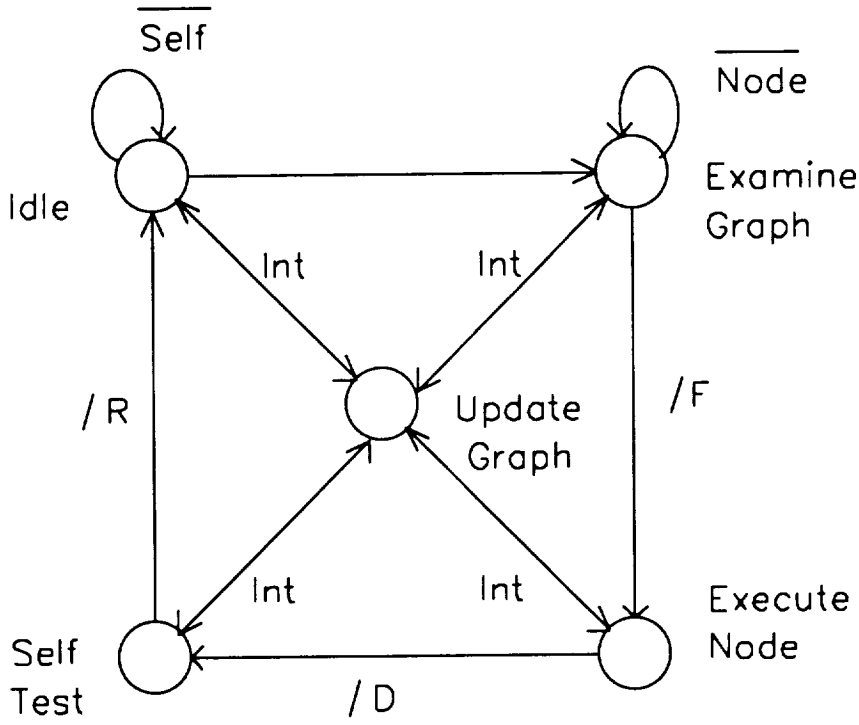


Figure 21. AMOS resource state diagram.

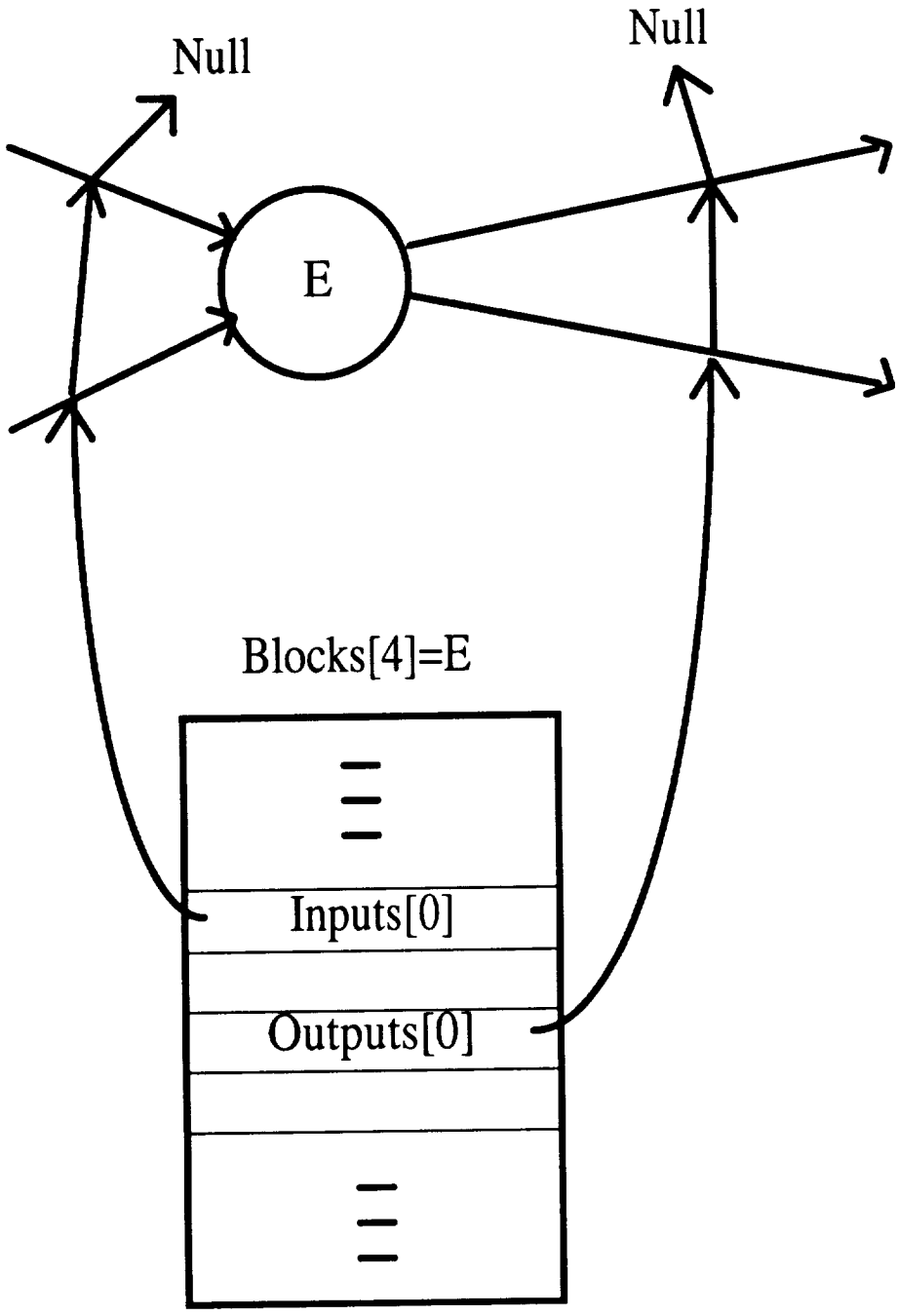


Figure 22. A pictorial representation of inputs and outputs linked lists.

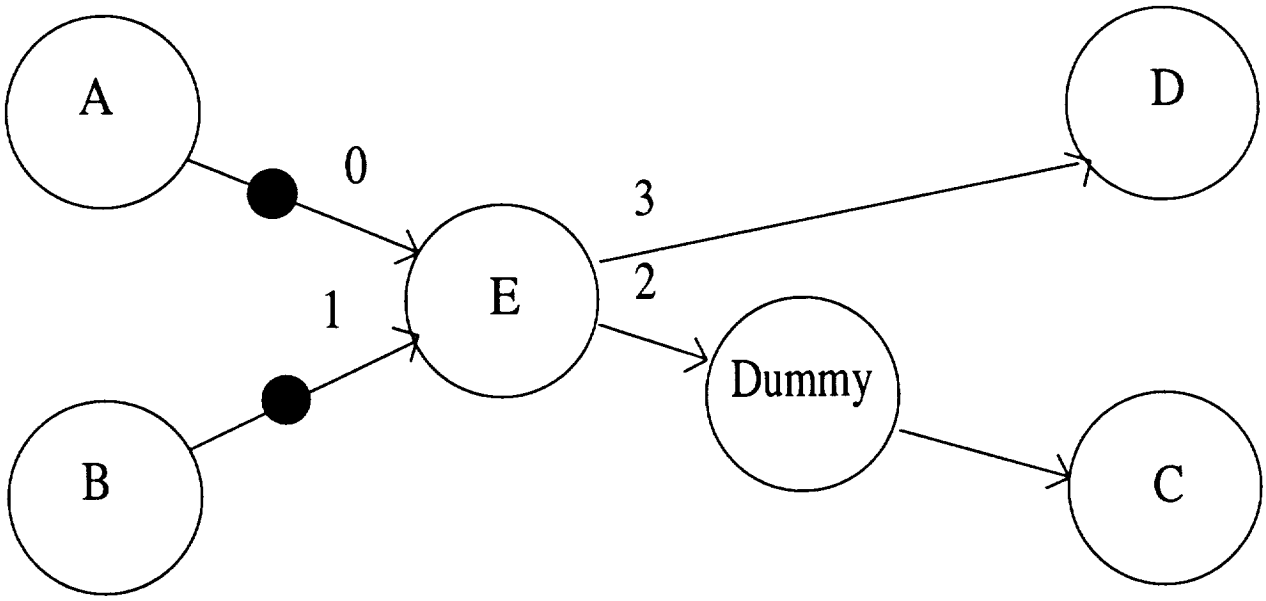


Figure 23. AMG graph.

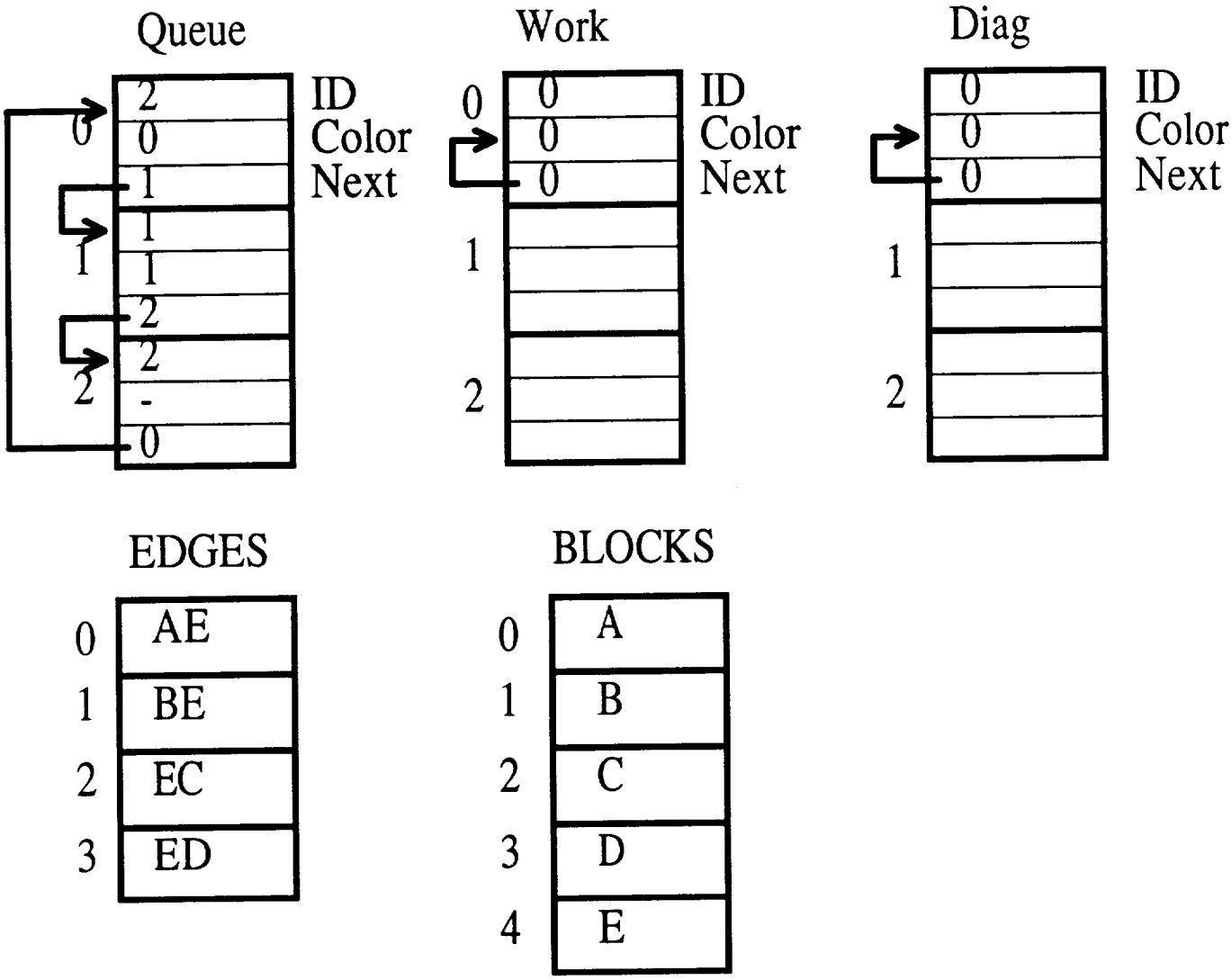


Figure 24. Data structures.

Edges[0]=AE

1	Segments
1	Items
0	Initial
4	Terminal
1	Edge_Color
1	Output_Width
[AE-Queue[0]]	Terminal_Ptr
[AE-Queue[1]]	Initial_Ptr
[Edges[1]]	Next_Input
Nil	Next_Output

Edges[1]=BE

1	Segments
1	Items
1	Initial
4	Terminal
1	Edge_Color
1	Output_Width
[BE-Queue[0]]	Terminal_Ptr
[BE-Queue[1]]	Initial_Ptr
Nil	Next_Input
Nil	Next_Output

Edges[2]=EC

2	Segments
0	Items
4	Initial
2	Terminal
1	Edge_Color
0	Output_Width
[EC-Queue[0]]	Terminal_Ptr
[EC-Queue[0]]	Initial_Ptr
Nil	Next_Input
Nil	Next_Output

Edges[3]=ED

1	Segments
0	Items
4	Initial
3	Terminal
1	Edge_Color
0	Output_Width
[ED-Queue[0]]	Terminal_Ptr
[ED-Queue[0]]	Initial_Ptr
Nil	Next_Input
[Edges[2]]	Next_Output

Figure 25. Structure of edges.

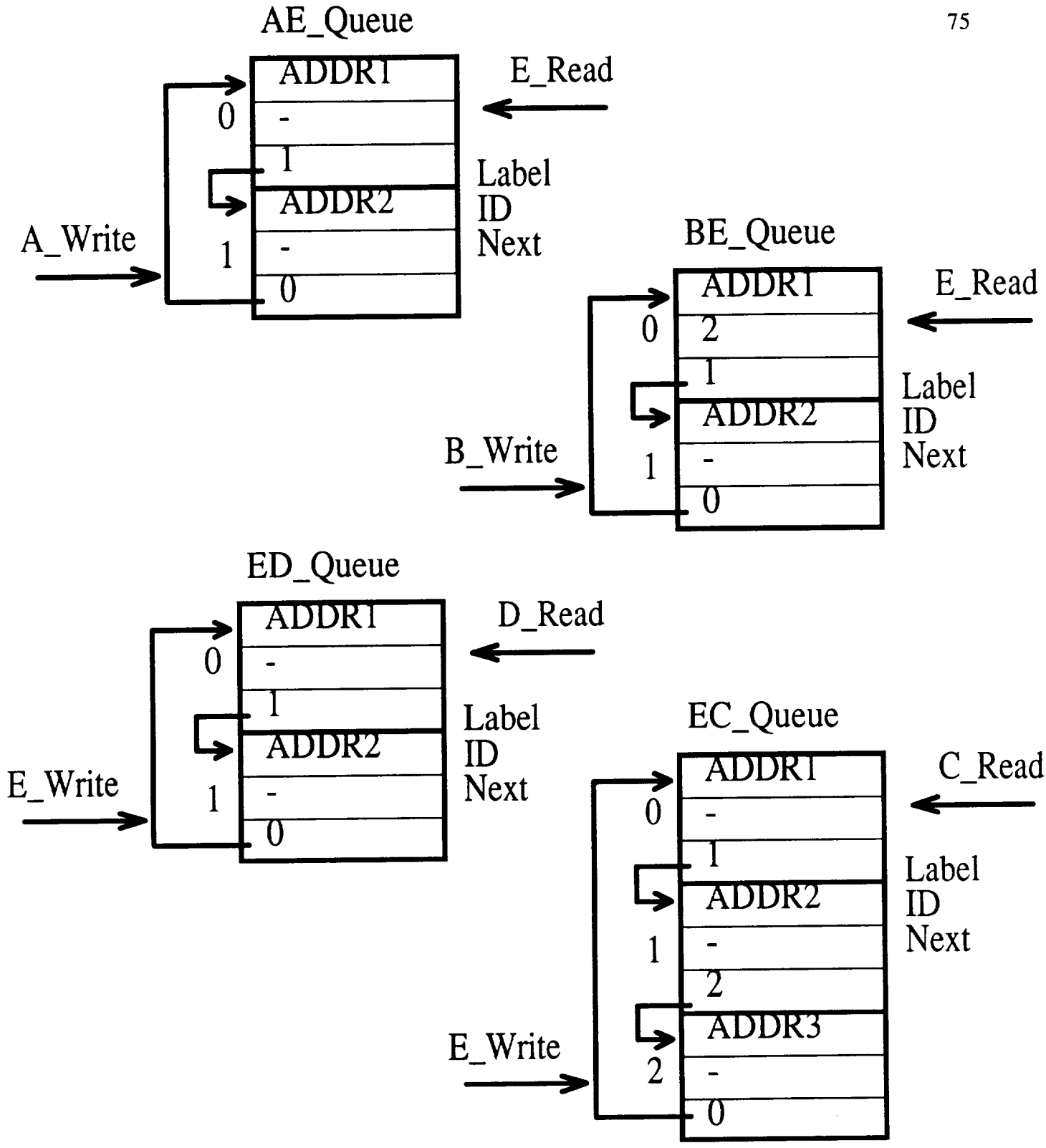


Figure 26. Data structure of Edge_Queues.

Blocks[4]=E

Module E	Modulem ID
-	ID[0]
-	ID[1]
-	ID[2]
0	Busy_Ctr
1	Done_Ctr
0	Enable_Ctr
2	In_Summary[0]
-	In_Summary[1]
-	In_Summary[2]
2	Out_Summary[0]
-	Out_Summary[1]
-	Out_Summary[2]
[Edges[0]]	Inputs[0]
-	Inputs[1]
-	Inputs[2]
[Edges[3]]	Outputs[0]
-	Outputs[1]
-	Outputs[2]

Figure 27. Structure of Blocks.

Blocks[4]=E

Module E	Modulem ID
-	ID[0]
-	ID[1]
-	ID[2]
0	Busy_Ctr
0	Done_Ctr
1	Enable_Ctr
0	In_Summary[0]
-	In_Summary[1]
-	In_Summary[2]
2	Out_Summary[0]
-	Out_Summary[1]
-	Out_Summary[2]
[Edges[0]]	Inputs[0]
-	Inputs[1]
-	Inputs[2]
[Edges[3]]	Outputs[0]
-	Outputs[1]
-	Outputs[2]

Figure 28. Blocks E enabled.

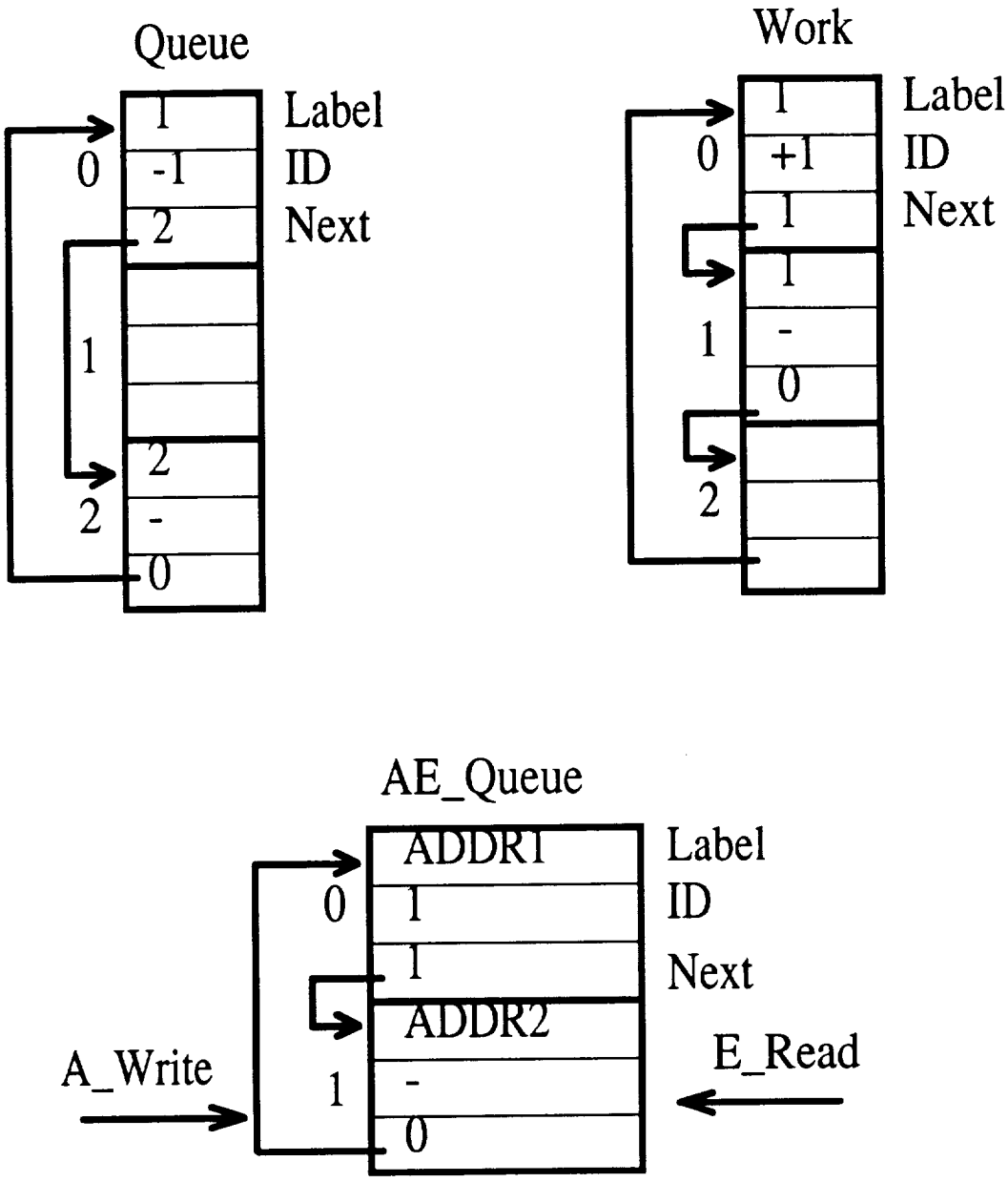


Figure 29. Busy reading A.

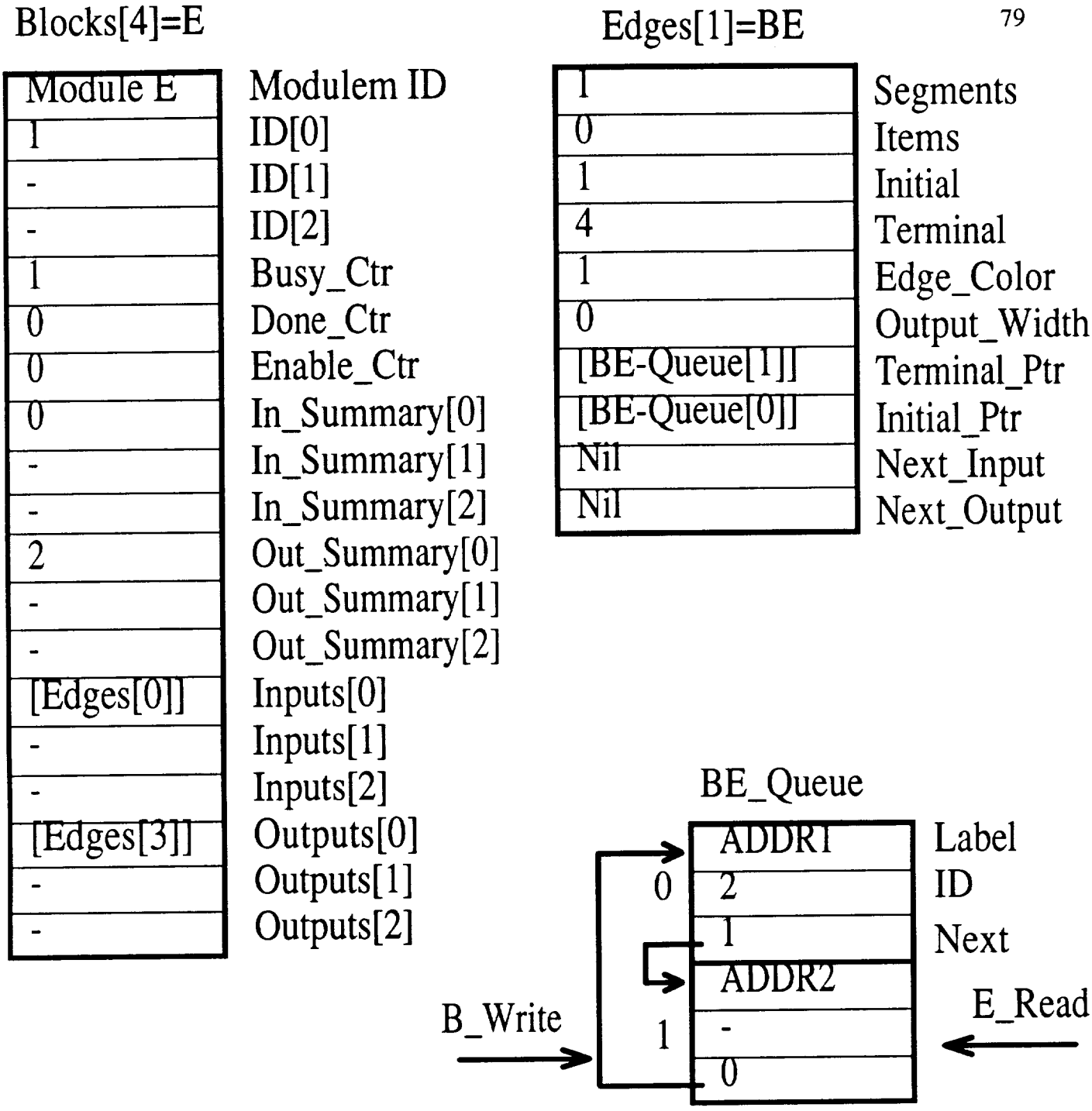


Figure 30. Busy reading B.

Blocks[4]=E

Module E	Modulem ID
1	ID[0]
-	ID[1]
-	ID[2]
1	Busy_Ctr
0	Done_Ctr
0	Enable_Ctr
0	In_Summary[0]
-	In_Summary[1]
-	In_Summary[2]
2	Out_Summary[0]
-	Out_Summary[1]
-	Out_Summary[2]
[Edges[0]]	Inputs[0]
-	Inputs[1]
-	Inputs[2]
[Edges[3]]	Outputs[0]
-	Outputs[1]
-	Outputs[2]

Figure 31. Busy processing.

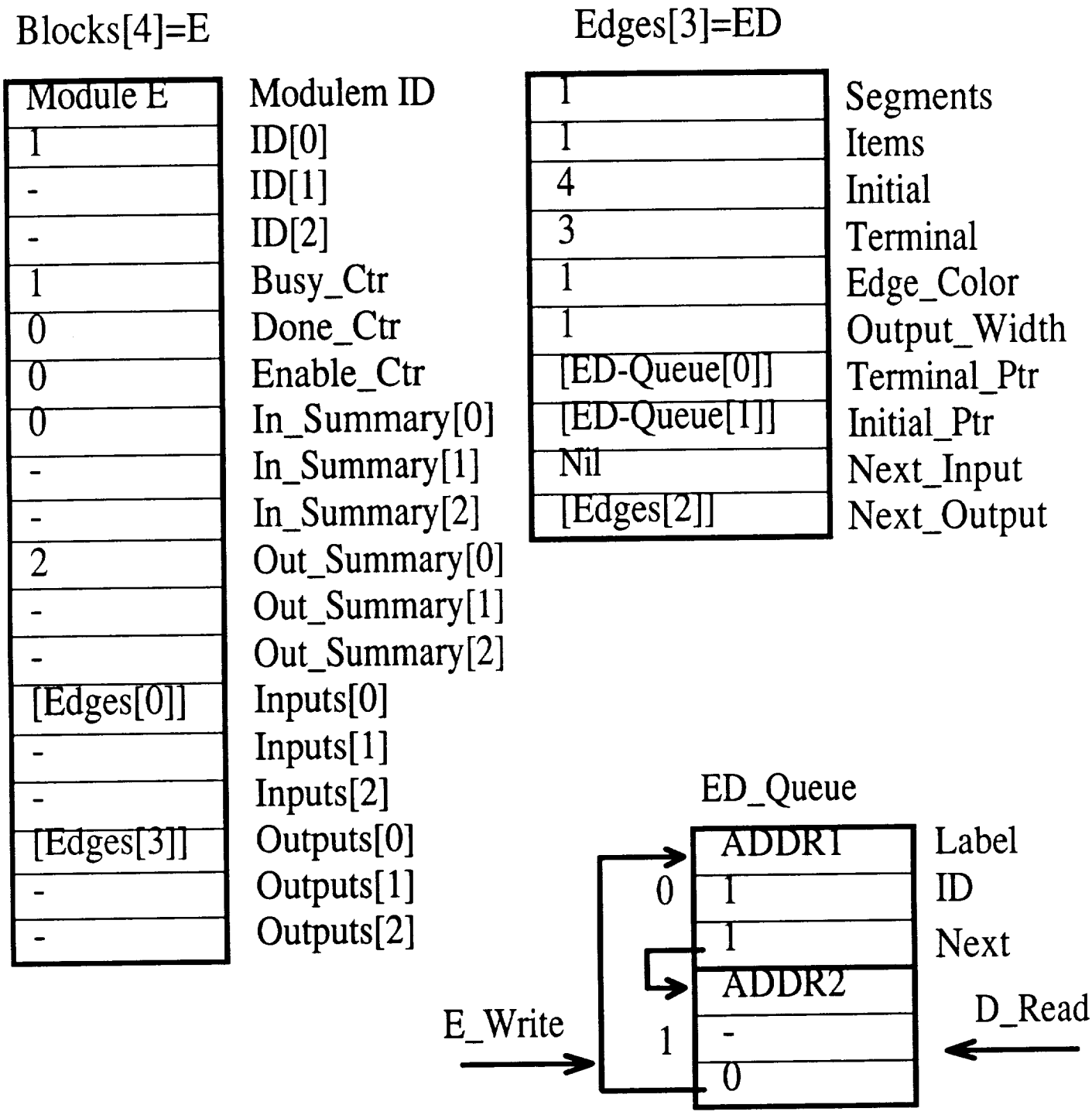


Figure 32. Busy writing to D.

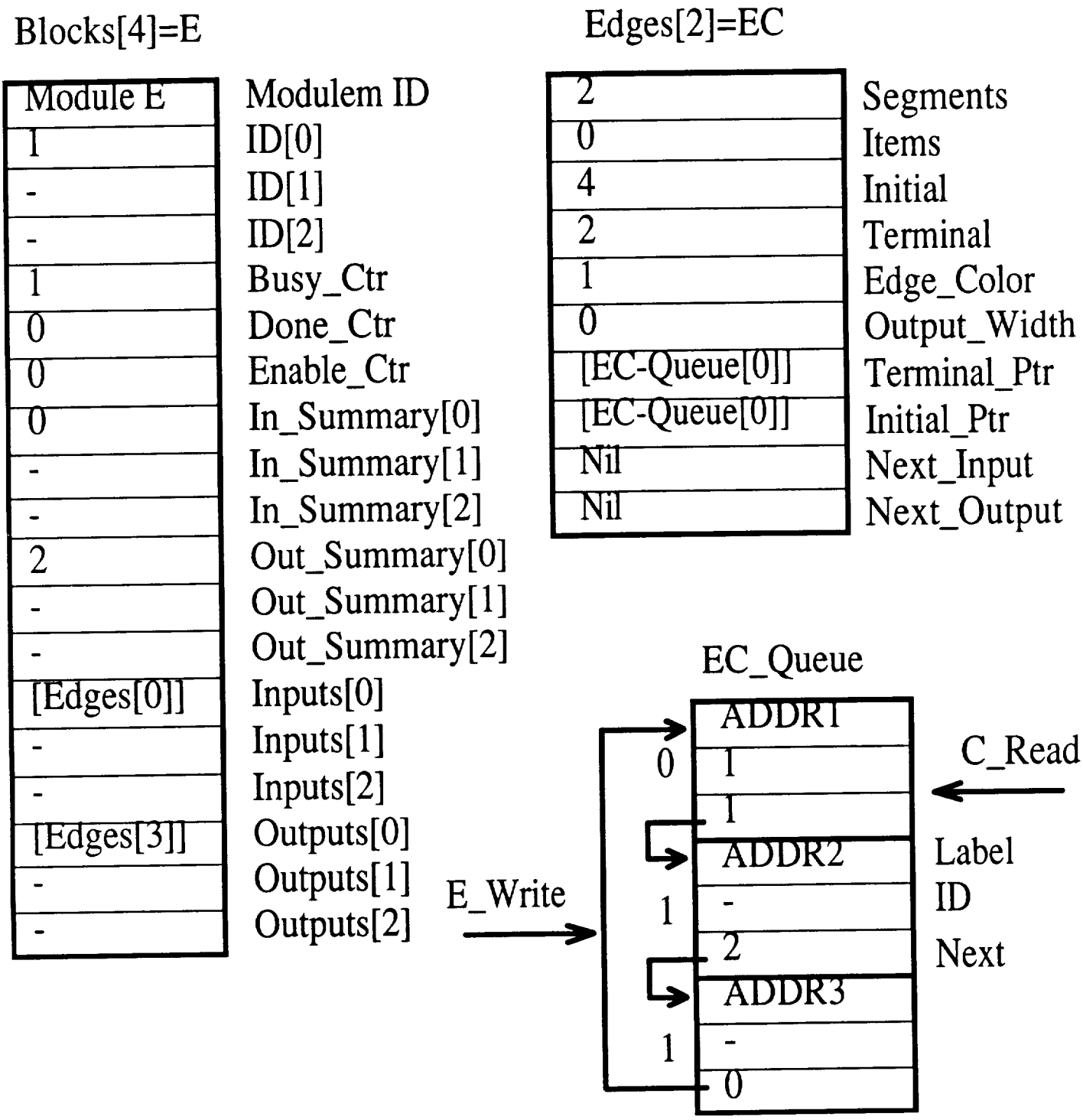


Figure 33. Busy writing to C and updating.

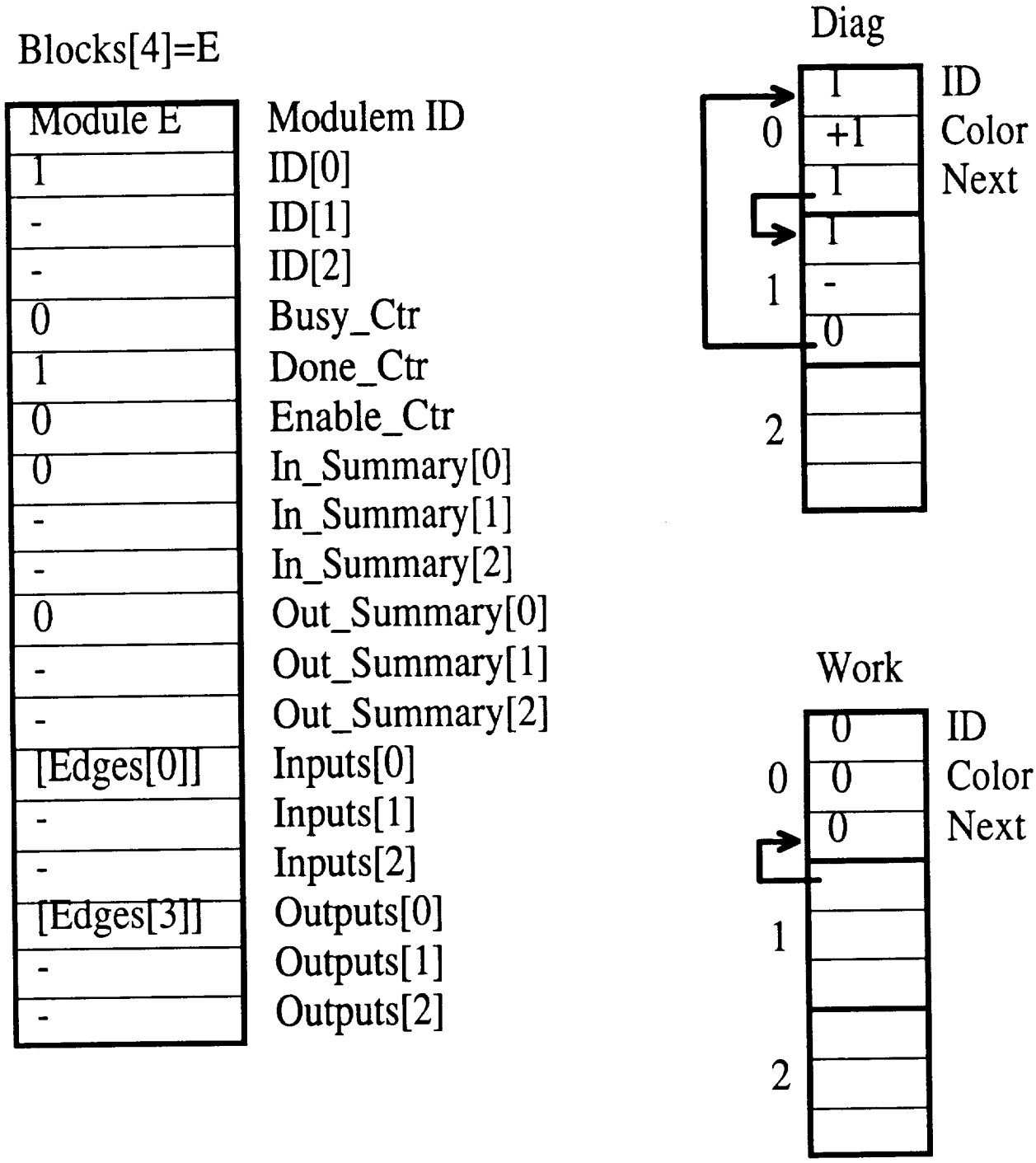


Figure 34. Done.

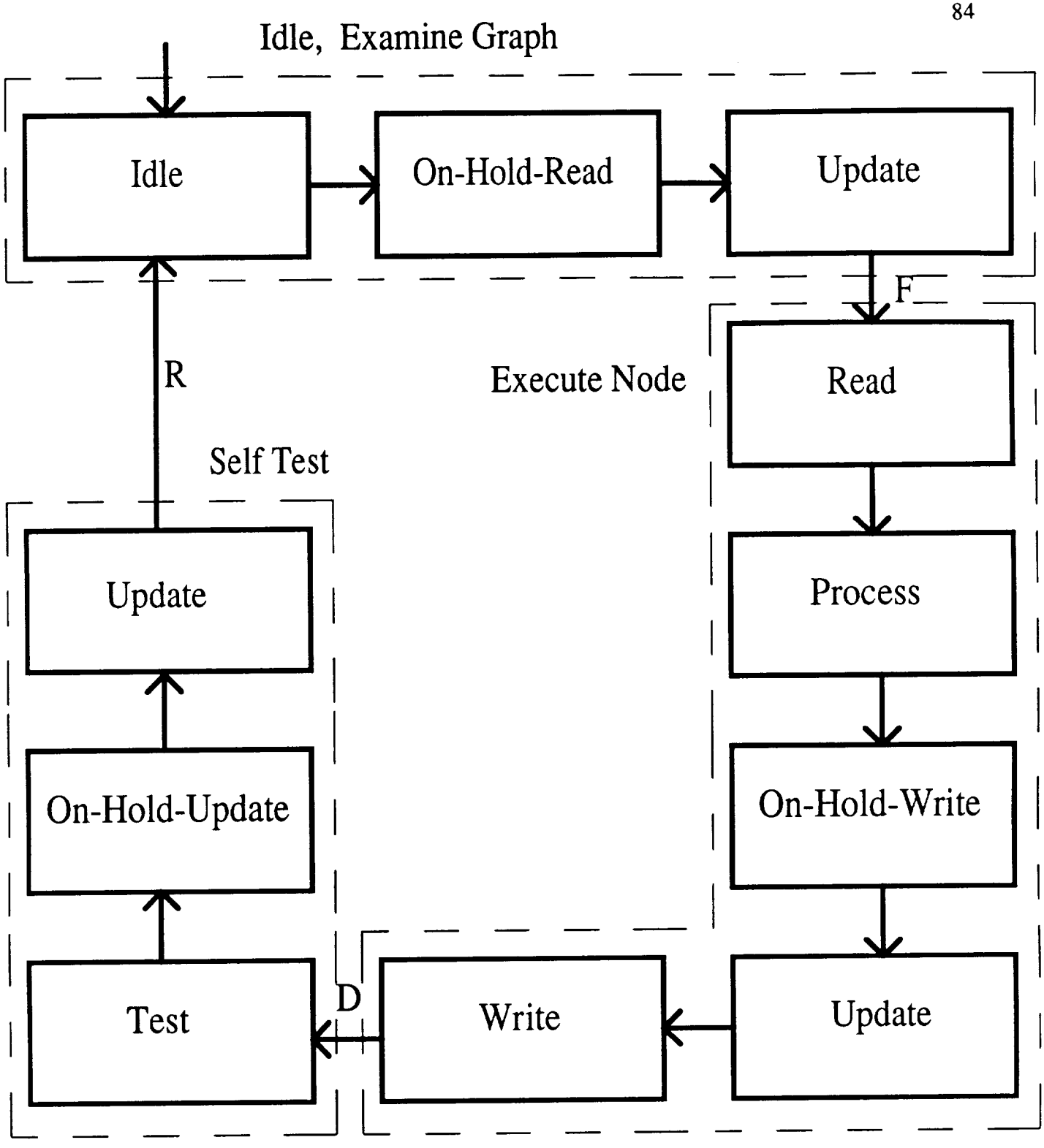
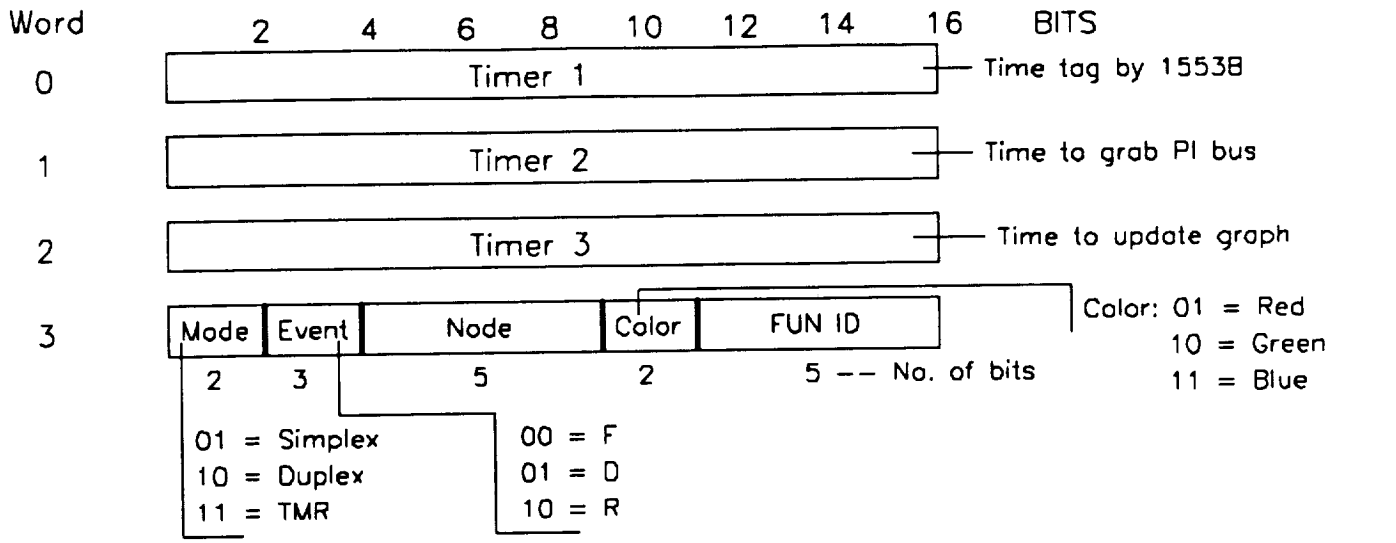


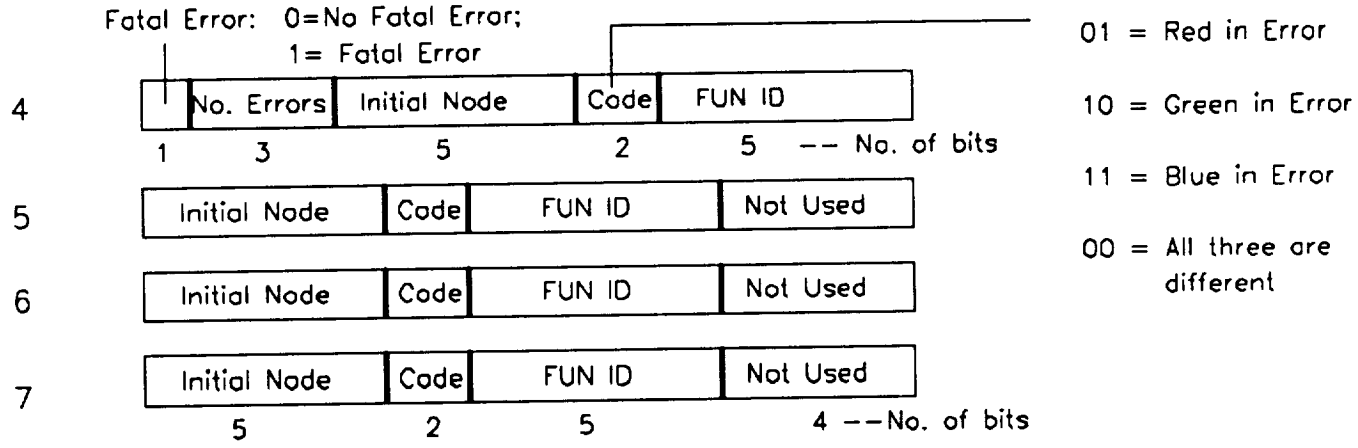
Figure 35. Functional unit operations.



4–7: These words (64 bits) vary depending on whether it is a F, D, or R event.

F event: All bits for words 4–7 are zero (0).

D Event:



R Event:

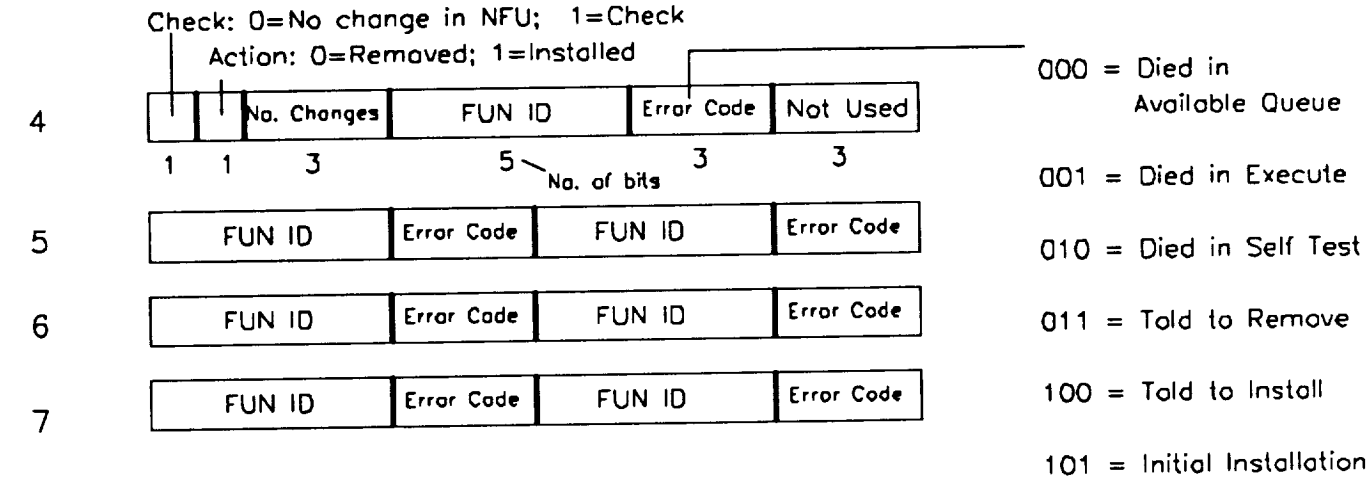


Figure 36. FDT (Fire, Data, Time) format.

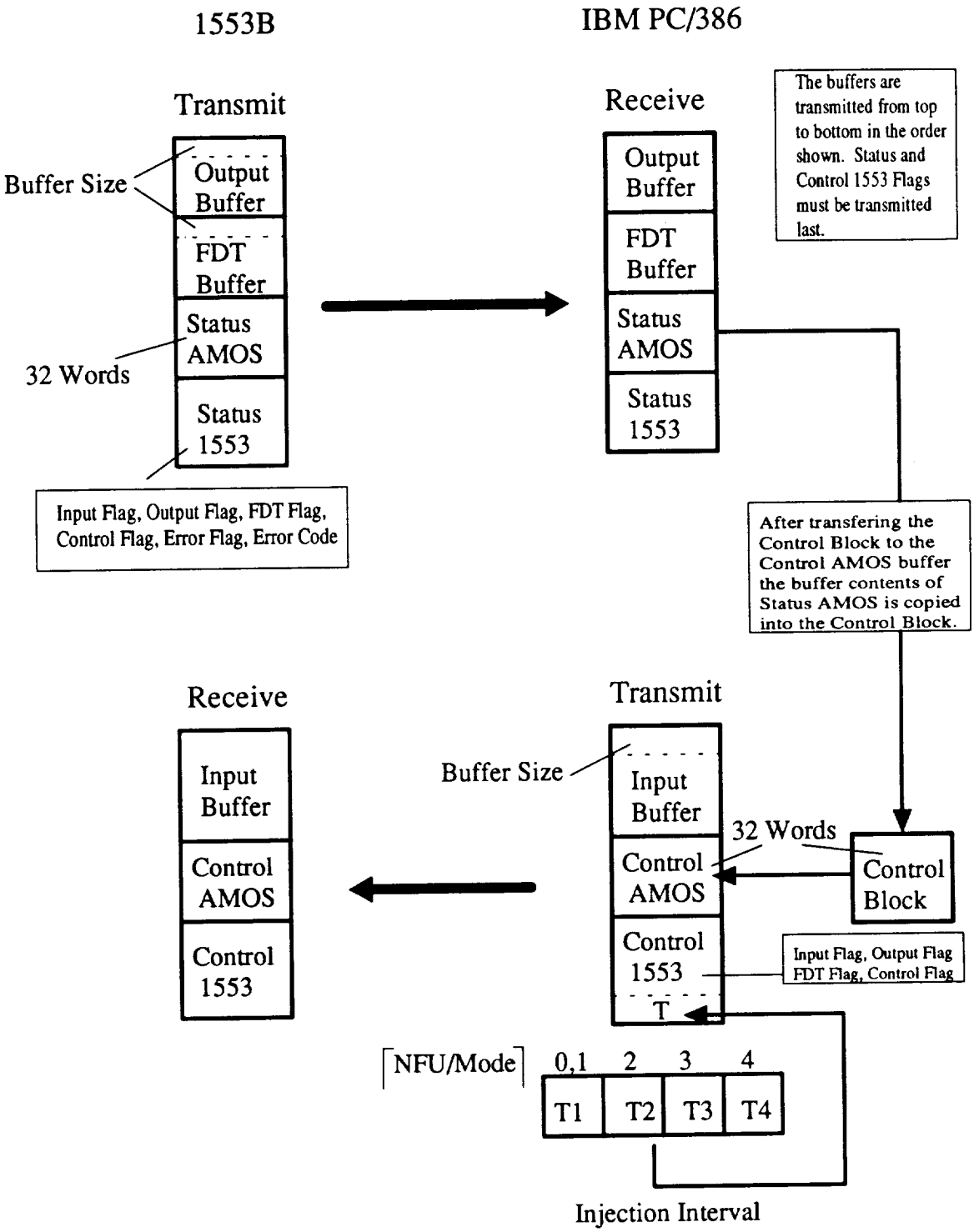


Figure 37. Communication memory map between 1553B and IBM PC/386.

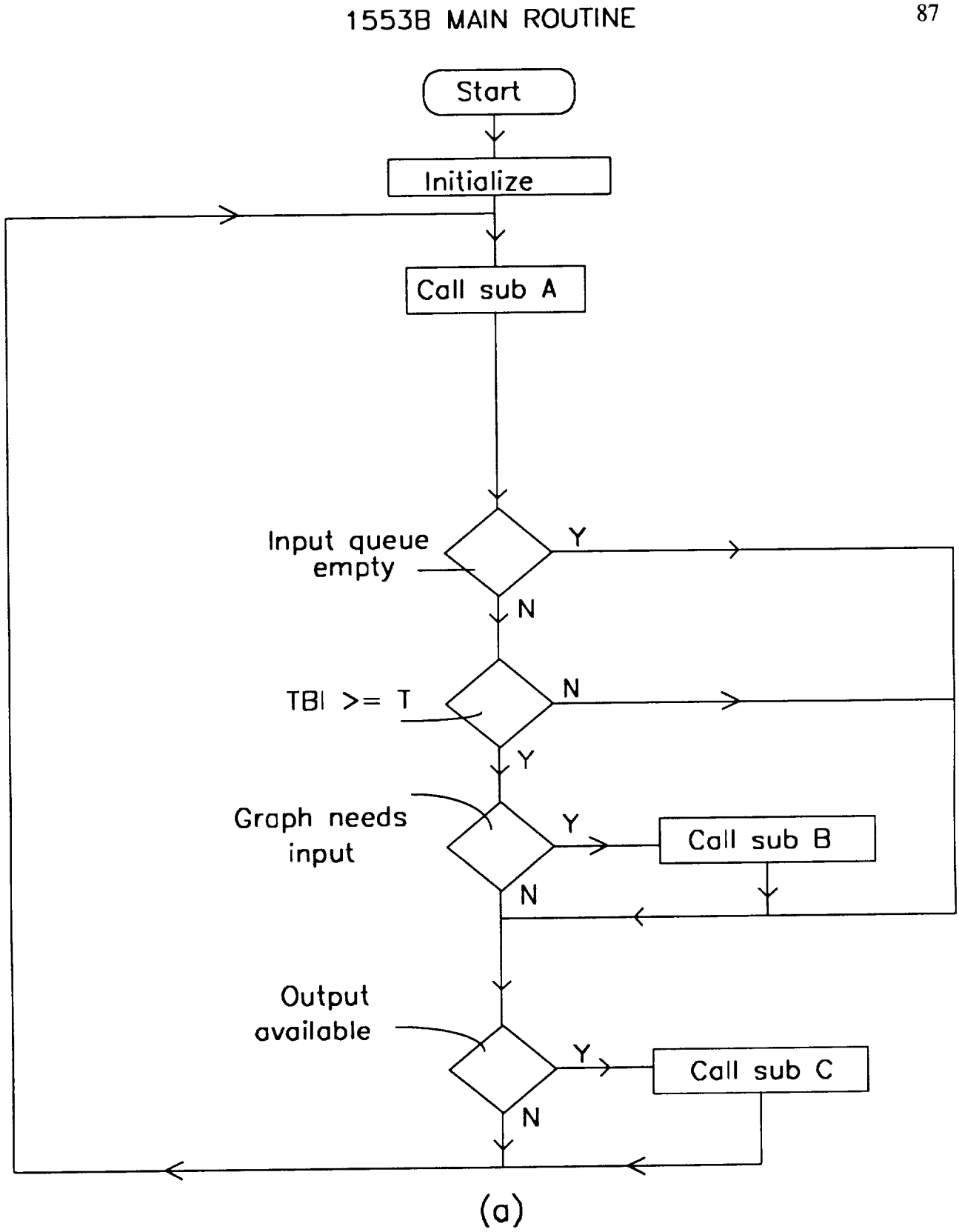


Figure 38. 1553B flow chart, (a) Main routine.

SUBROUTINE A OF THE 1553B MAIN ROUTINE

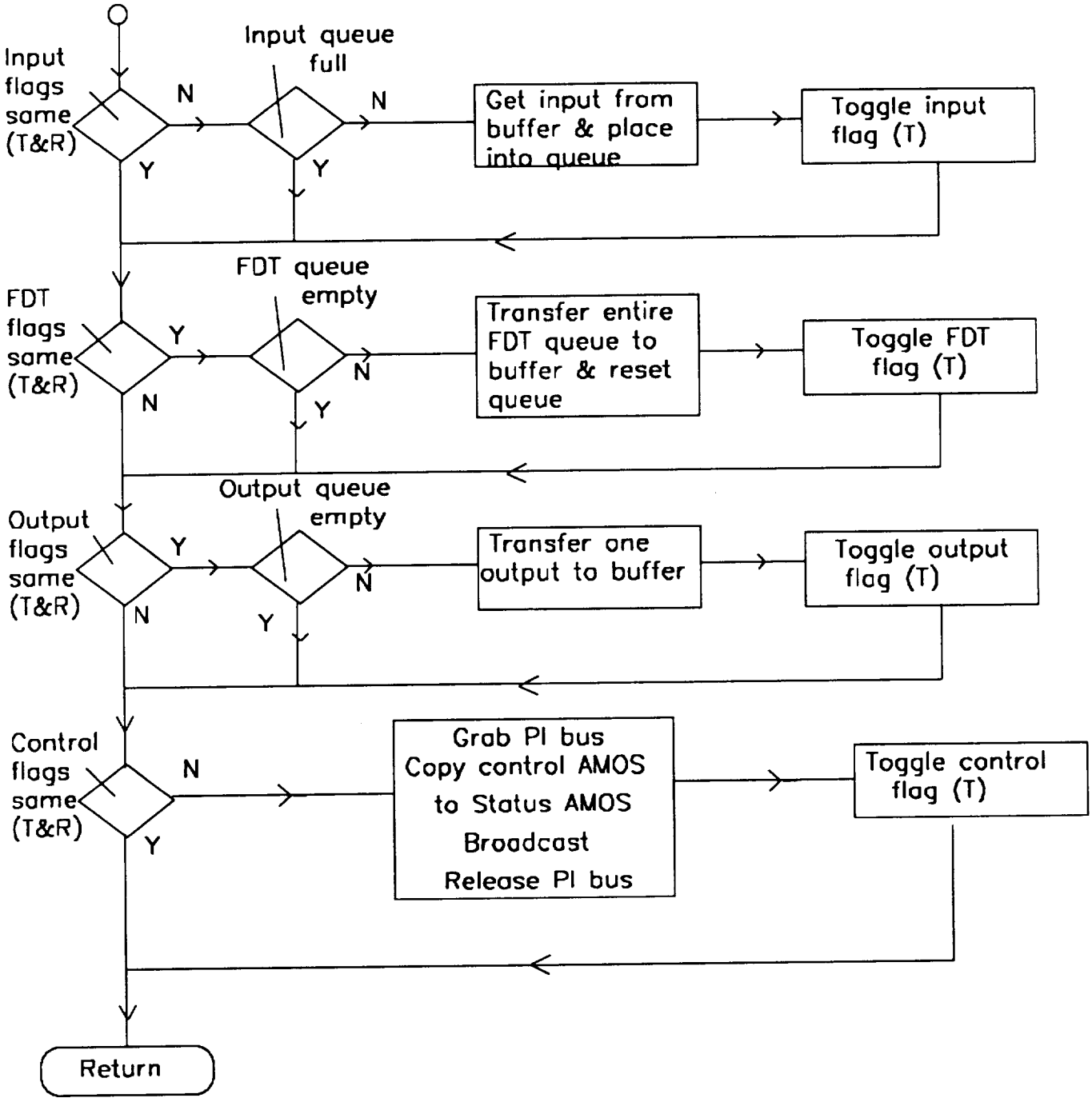


Figure 38(b). 1553B subroutine A.

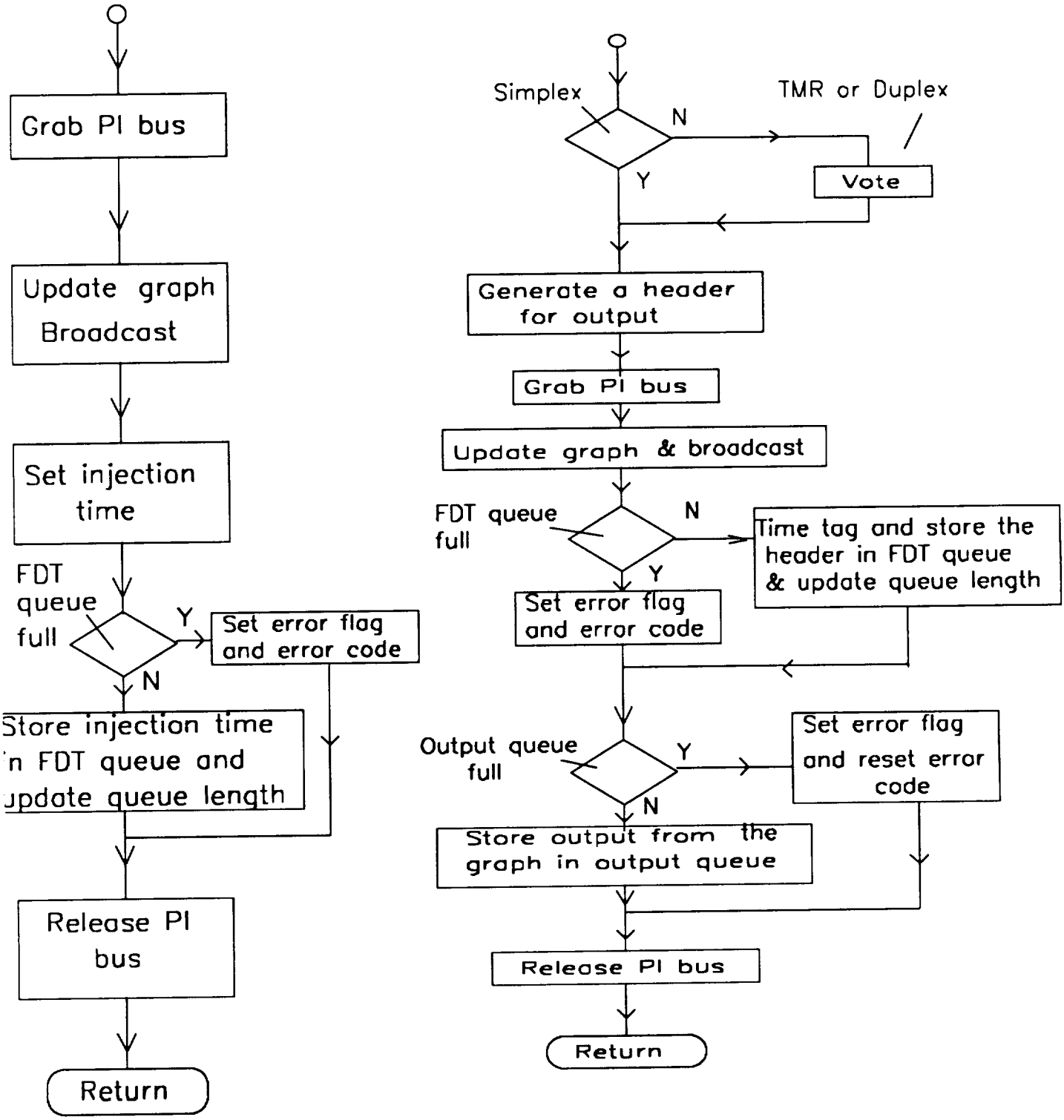


Figure 38(c). 1553B subroutines B and C.

1553B

DMA ROUTINE

1553 jumps to this routine after receiving a DMA from the PI bus

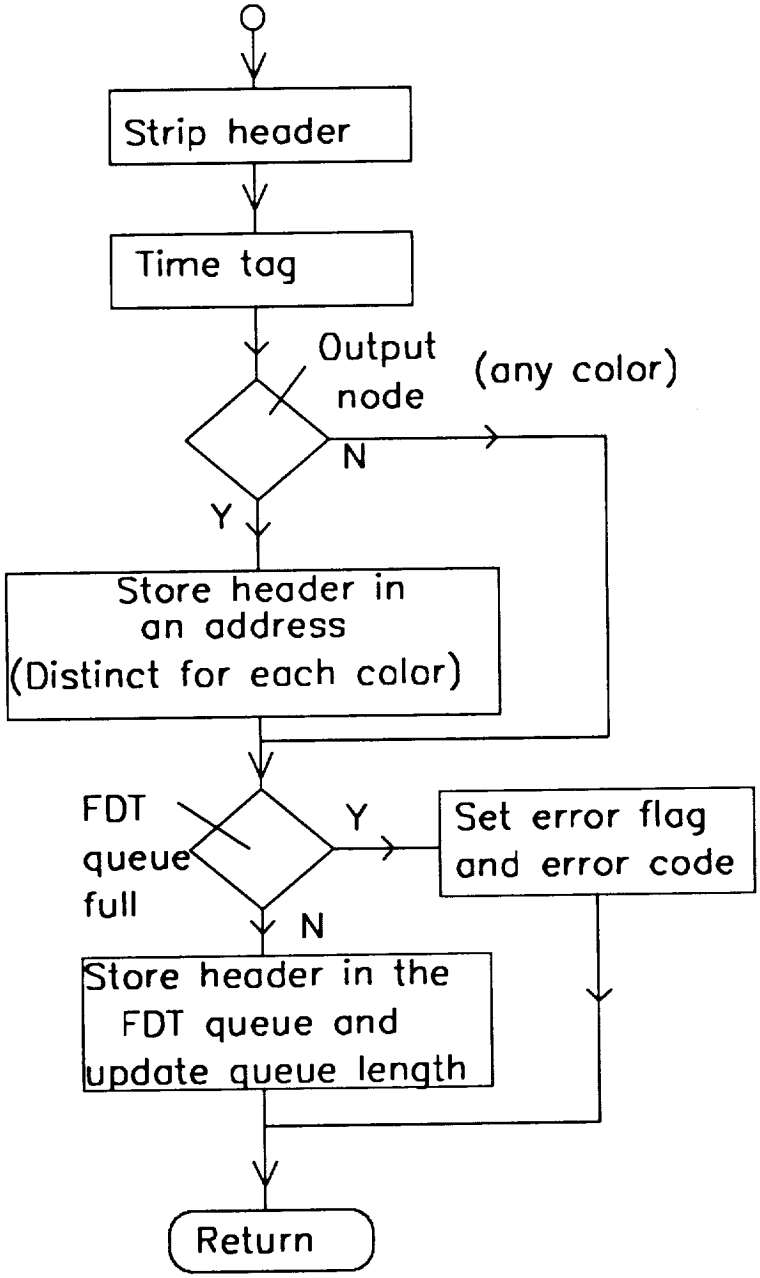


Figure 38(d). 1553B DMA routine.

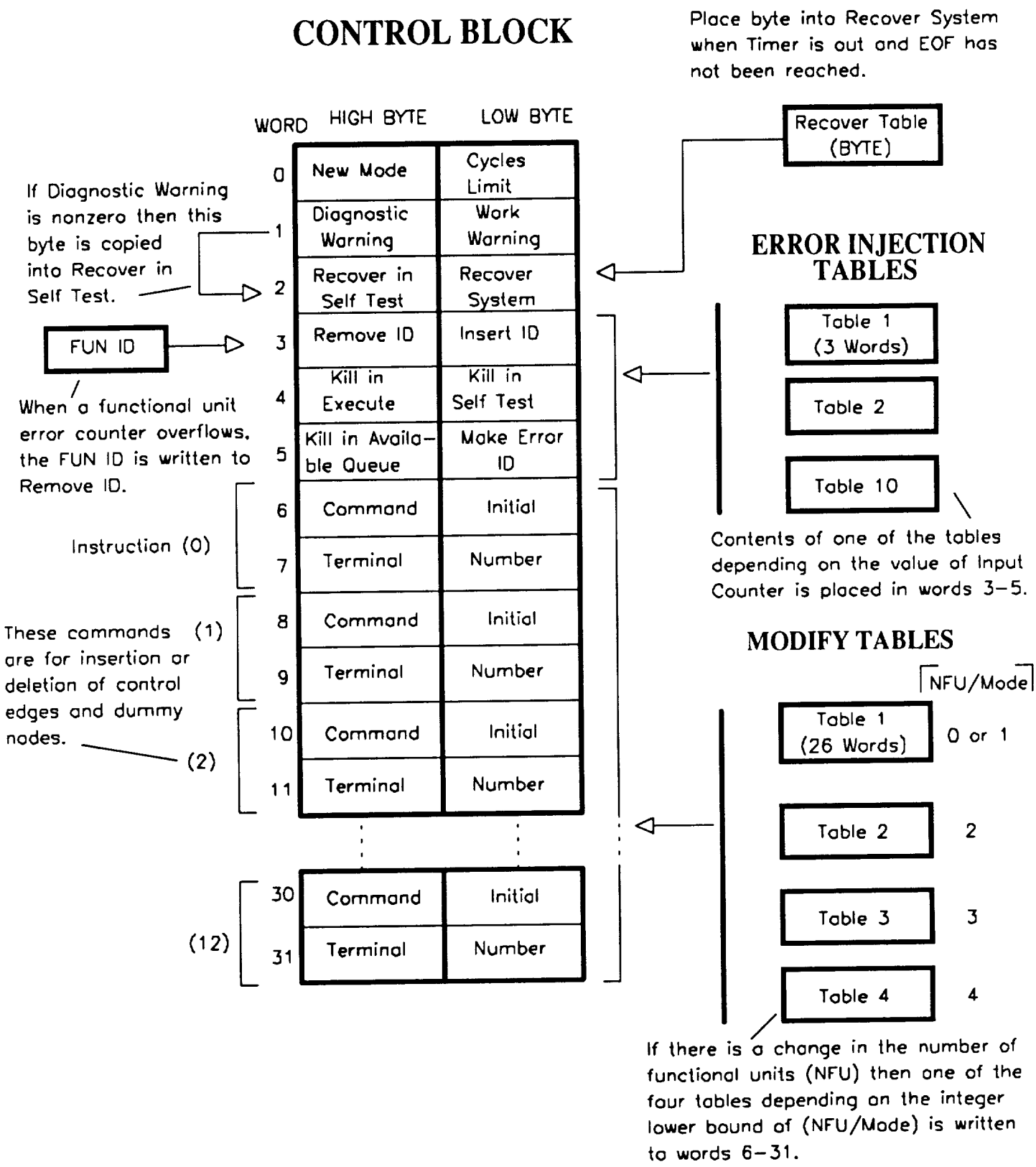


Figure 39. Format of Control Block, Modify Tables, and Error Injection Tables.

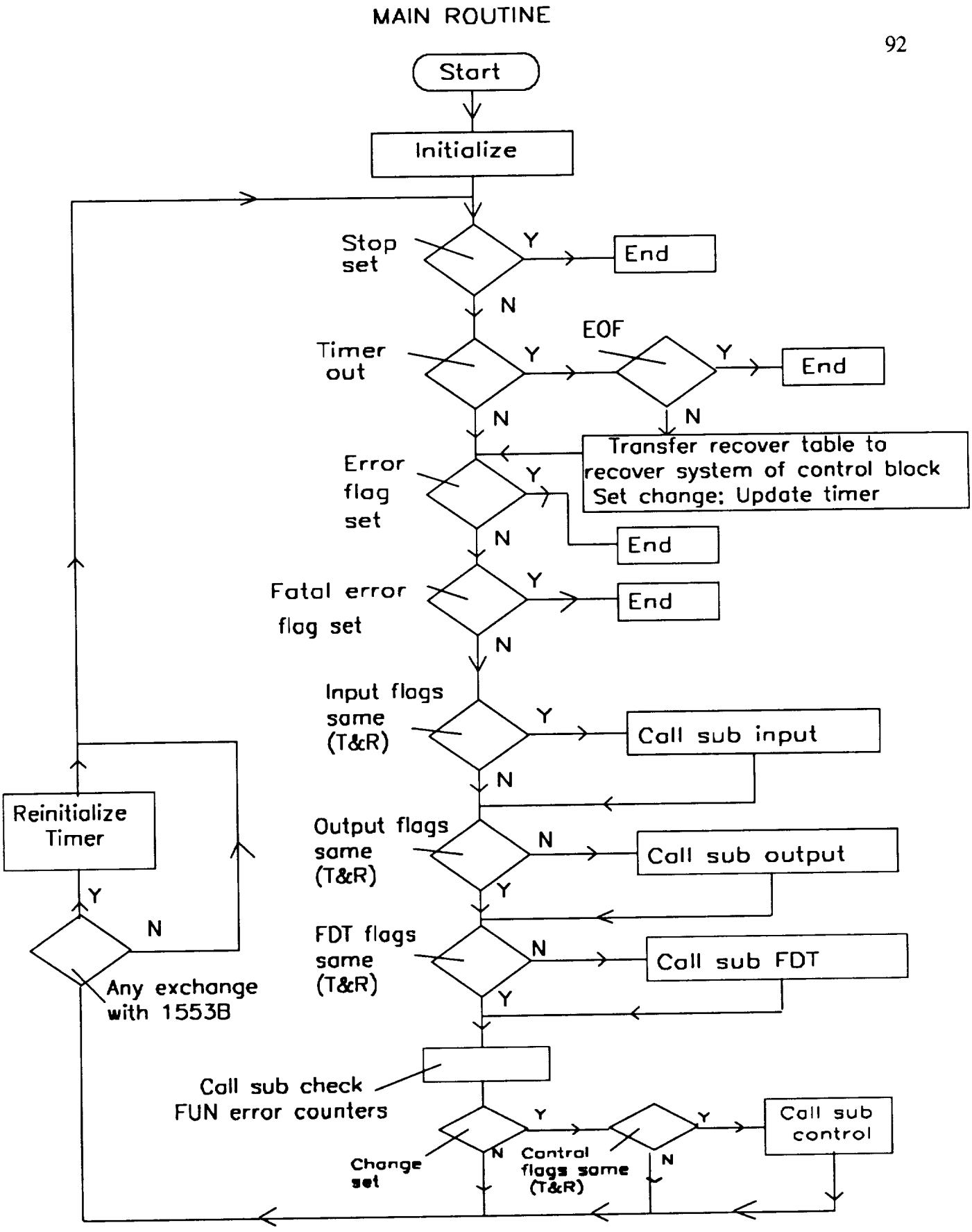


Figure 40. IBM PC/386 flow chart, (a) Main routine.

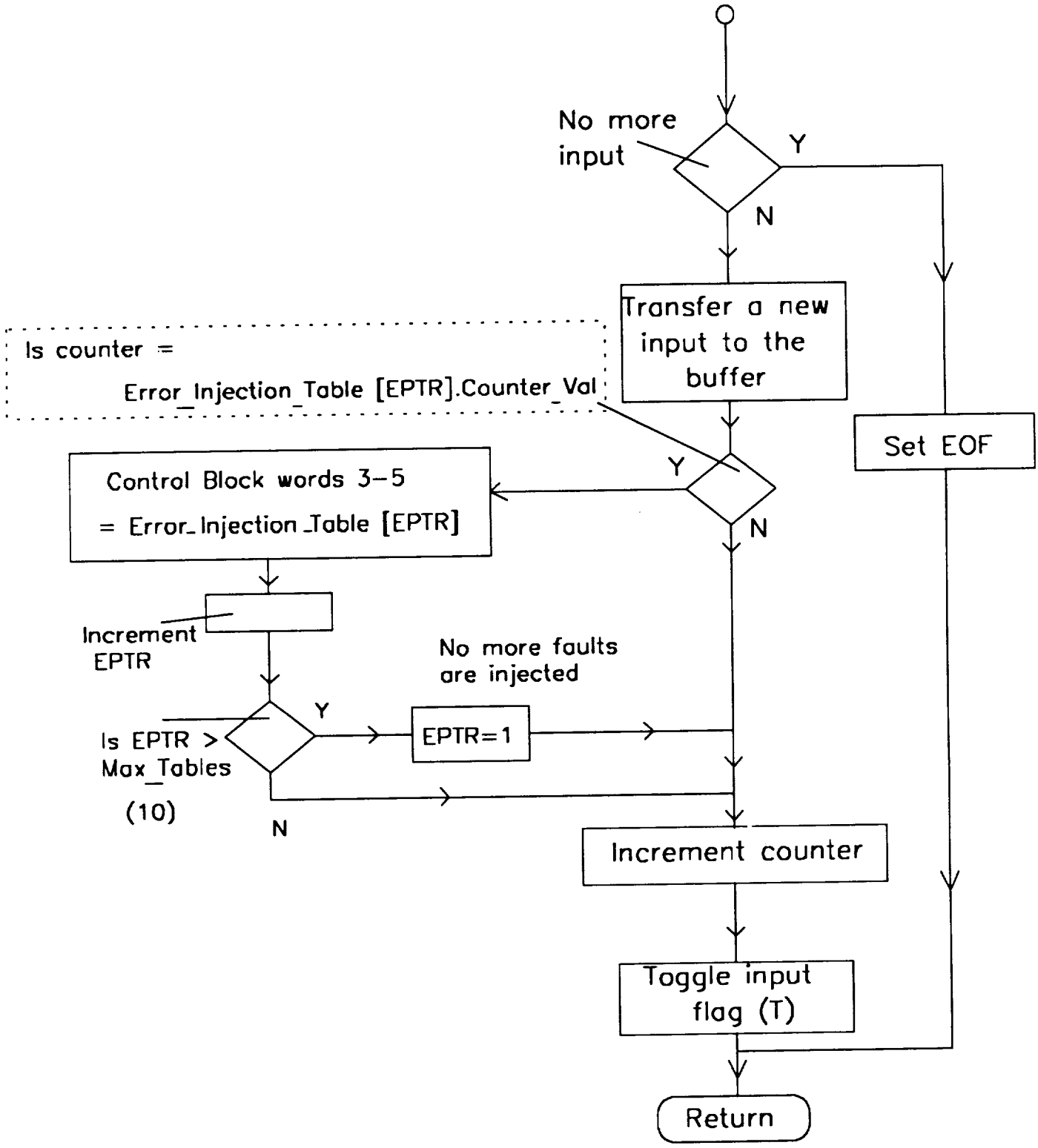


Figure 40(b). Subroutine input of IBM PC/386.

IBM PC/386

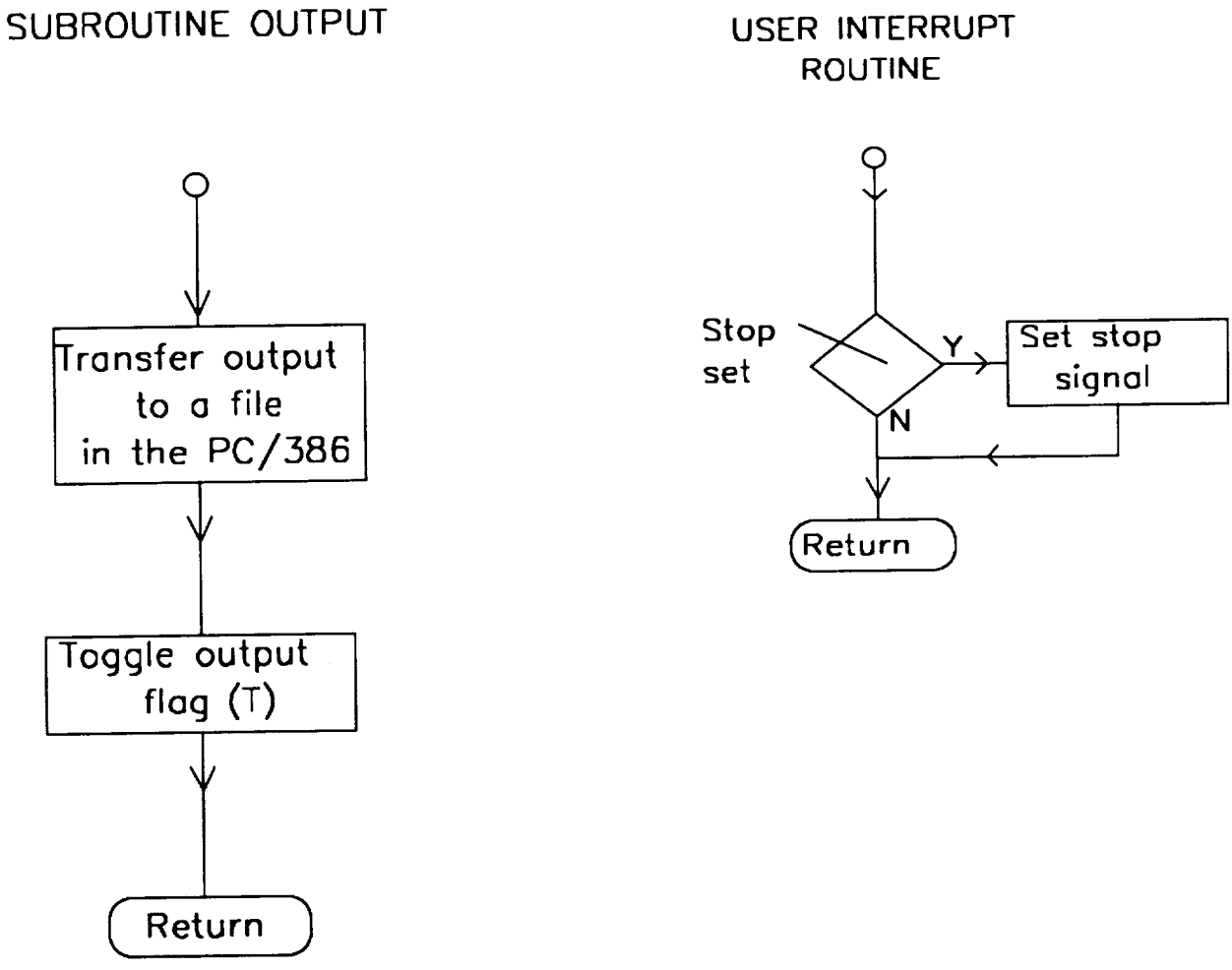


Figure 40(c). Subroutine output and user interrupt routine.

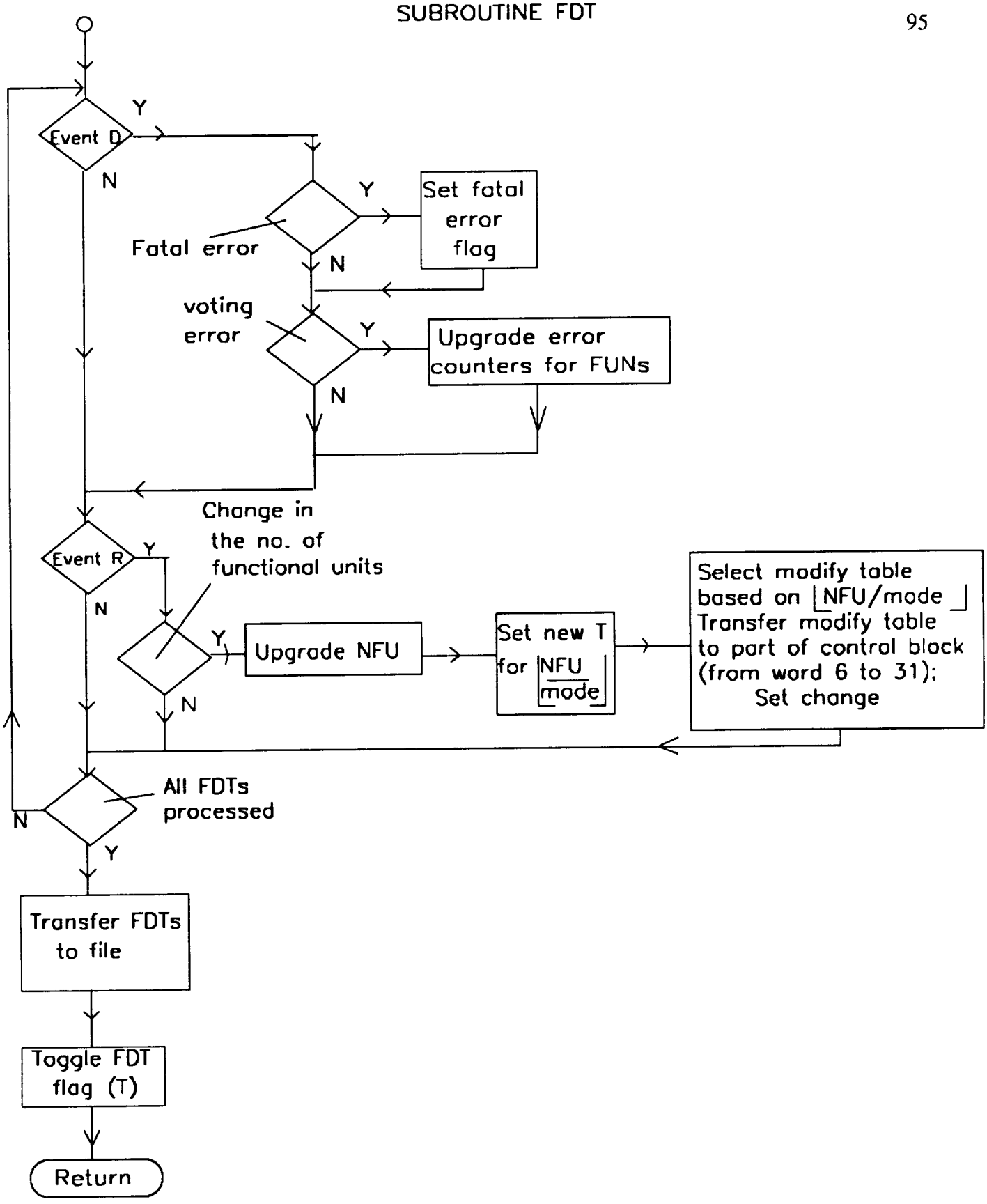


Figure 40(d). Subroutine FDT of IBM PC/386.

SUBROUTINE CHECK FUN ERROR COUNTERS

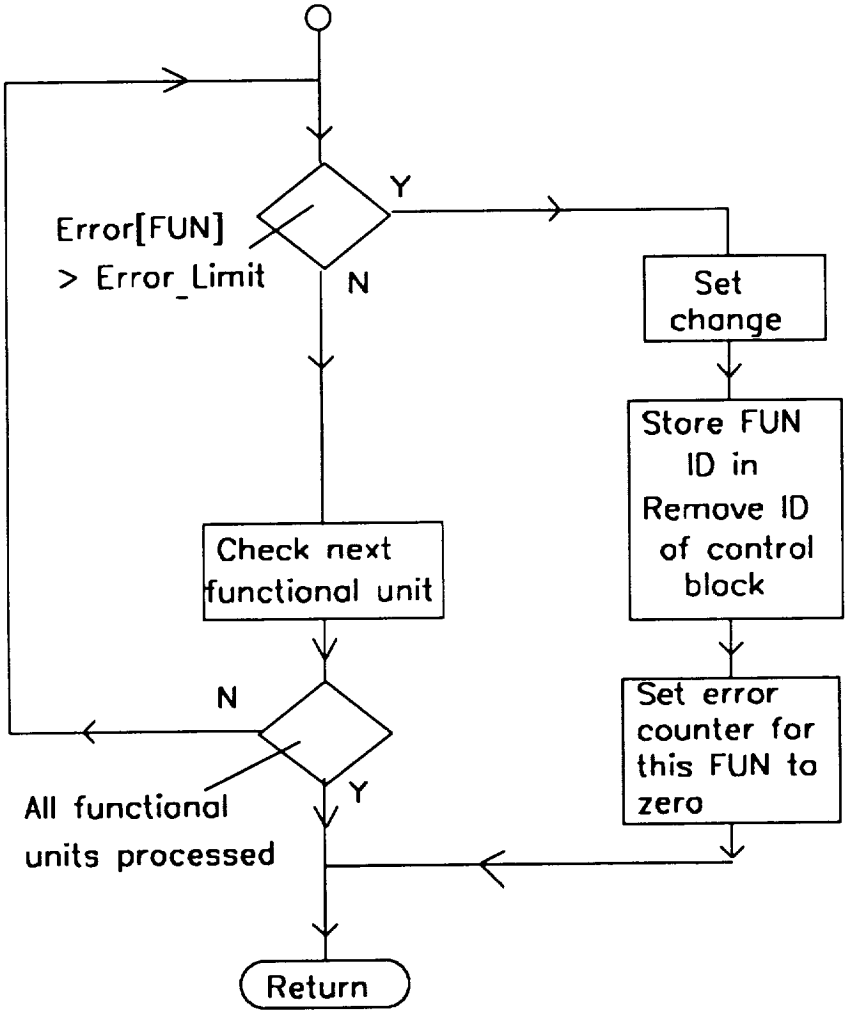


Figure 40(e). Subroutine check FUN error counters of IBM PC/386.

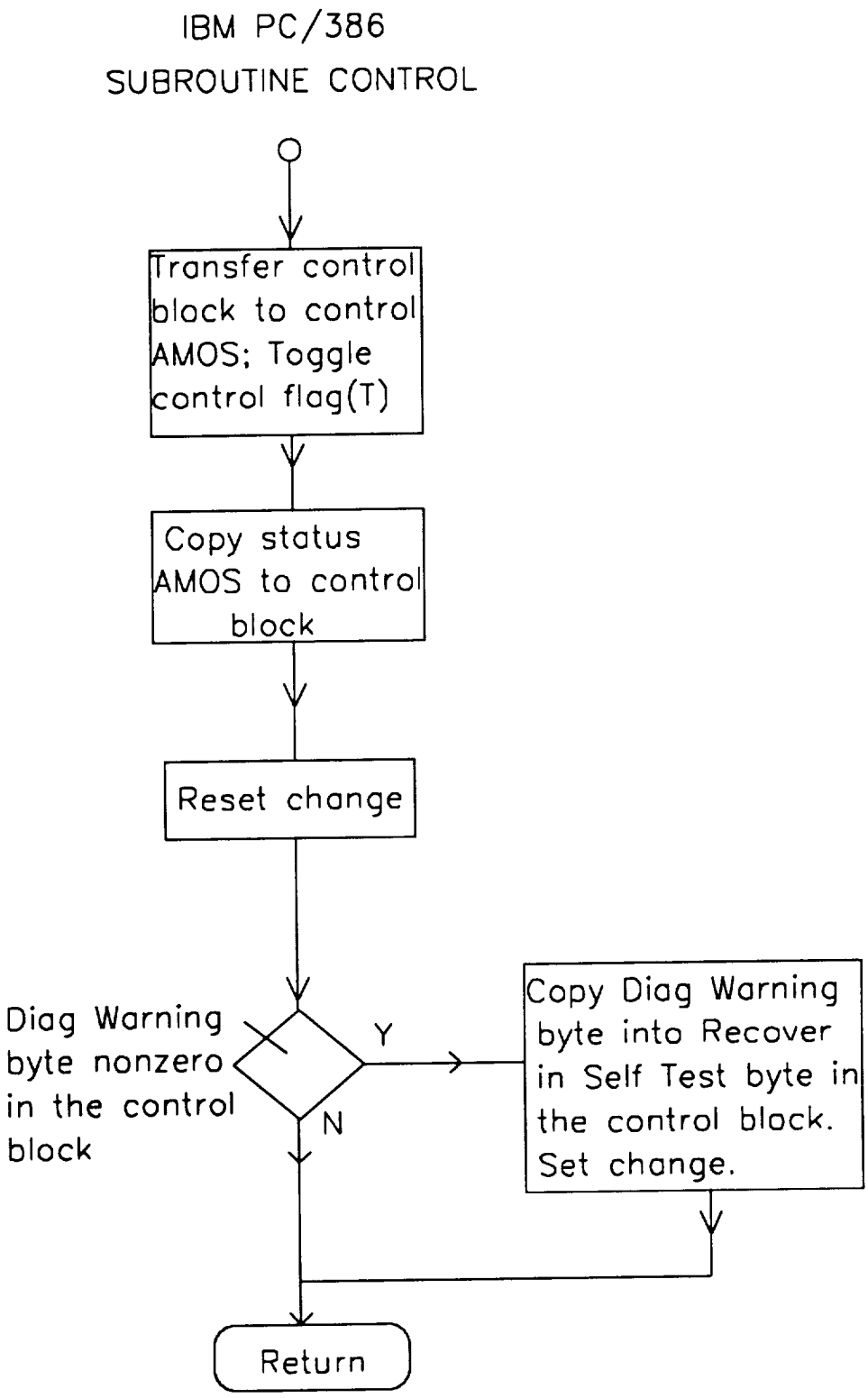


Figure 40(f). Subroutine control.

CHAPTER FIVE

ATAMM SUPPORT TOOLS

In order to facilitate the prediction of performance and resource requirements in an ATAMM defined testbed like ADM, a number of software support tools have been developed [9, 10, 11]. The ATAMM support tools include software tools named as Design Tool, Simulator, and Analyzer. The Design Tool is used to select an operating point for each possible value of available resource. The Simulator is used to simulate algorithm execution by AMOS on the ADM. Finally, the Analyzer is used to determine algorithm performance from FDT file outputs produced either by the Simulator or the ADM hardware. The order in which tools are used is described in Figure 41.

The software tools are developed in Windows 2.1 running under MS-DOS on IBM PC/386 or compatible. The programming language used for coding is C. Each tool consists of several component pieces called windows. Each window is constructed to solve a particular portion of the total problem. The displays of the ATAMM support tools are divided in these windows. Several dialogue boxes and message boxes are provided for the convenience of the user when running the tools. All options are menu driven and minimum keyboard interaction is required. Each display of the support tools can be selected from a main window. Any number of

displays can appear on the screen at the same time and a display can be removed from the screen at anytime.

The Design Tool is required for making off-line scheduling decisions and for performance prediction of algorithms. The Design Tool obtains the required input information from the AMG drawn using a graph editor. This window allows the user to create, modify, and store the AMG. The most important output of the Design Tool is the performance plane window which shows all operating points for a particular algorithm marked graph and enables the user to choose a particular set of operating points. Based on the user selection, the Design Tool generates a modify table which specifies control edges, buffer sizes, performance, and resource requirements for each operating point selected from the performance plane. A Simulator has been coded to simulate and test AMOS. The input parameters for the Simulator are the algorithm marked graph including all NMG transition times, the number of resources, and system overhead times such as bus access time, bus broadcast time, etc.. The input injection interval actually used in the Simulator and hardware systems is determined by adding a small overhead time to the Design Tool interval to account for bus contention time, functional unit test time, and increases in algorithm node times due to interrupts from other functional units. The Simulator reports all events associated with the execution of algorithm nodes for each data packet on a graph diagnostic file, called the FDT (Fire, Data, Time) file. The Analyzer has been developed to determine algorithm performance using the FDT file produced by the Simulator or actual hardware. It provides the means to examine the overall system behavior to obtain

performance measurements. The performance measurements indicate computing time, throughput, concurrency, and resource utilization attained by the system. This tool also provides measurements associated with system overhead.

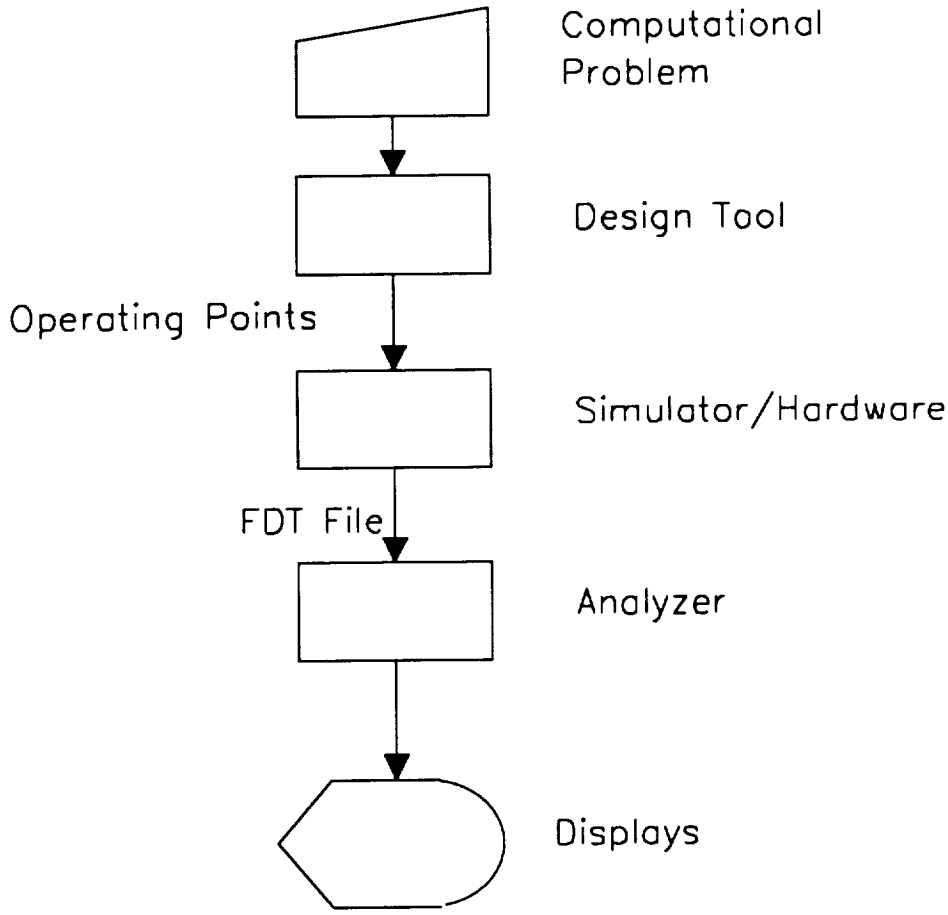


Figure 41. ATAMM support tools.

CHAPTER SIX

RESEARCH STATUS

There are several topics which are the subject of continuing and future research. Significant progress has been made towards the completion of the ADM system and it is currently partially operational (three out of four 1750As are functioning). The ATAMM model is being validated by experiments on the ADM testbed. A three node algorithm marked graph, shown in Figure 42, has been successfully executed on the ADM hardware by the NASA Langley Research Center. A number of algorithms are being analyzed by the ATAMM support tools. A space surveillance algorithm, described in Figure 43, is to be implemented on the ADM testbed. The results of design, simulations, experiments, and analysis of this algorithm will be described in a later report.

The ATAMM model is being extended to include multiple concurrent instantiations of node operations. Simultaneous execution of multiple algorithm graphs is being investigated and an overhead model to better account for architecture dependent parameters such as interprocessor communication and contention in communication is being developed. Although rare in control and signal processing algorithms, the model should be extended to include data dependent branching for completeness. Finally, studies are needed to determine the impact of node time

variations, where node latency is data dependent. Work is now in progress to incorporate these features in an enhanced ATAMM model which is to be realized in the Generic VHSIC Spaceborne Computer (GVSC) [3].

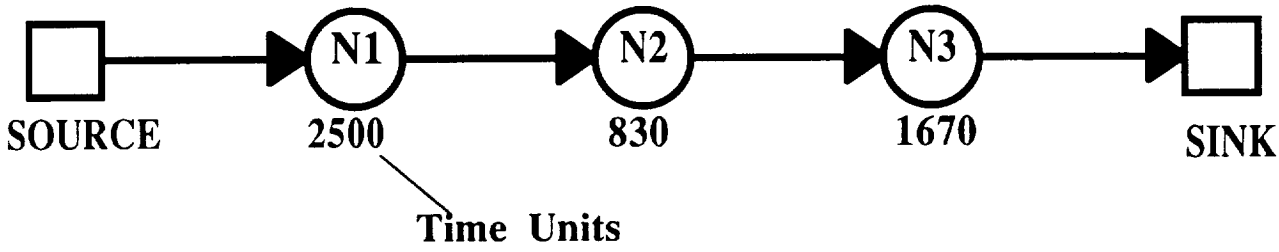
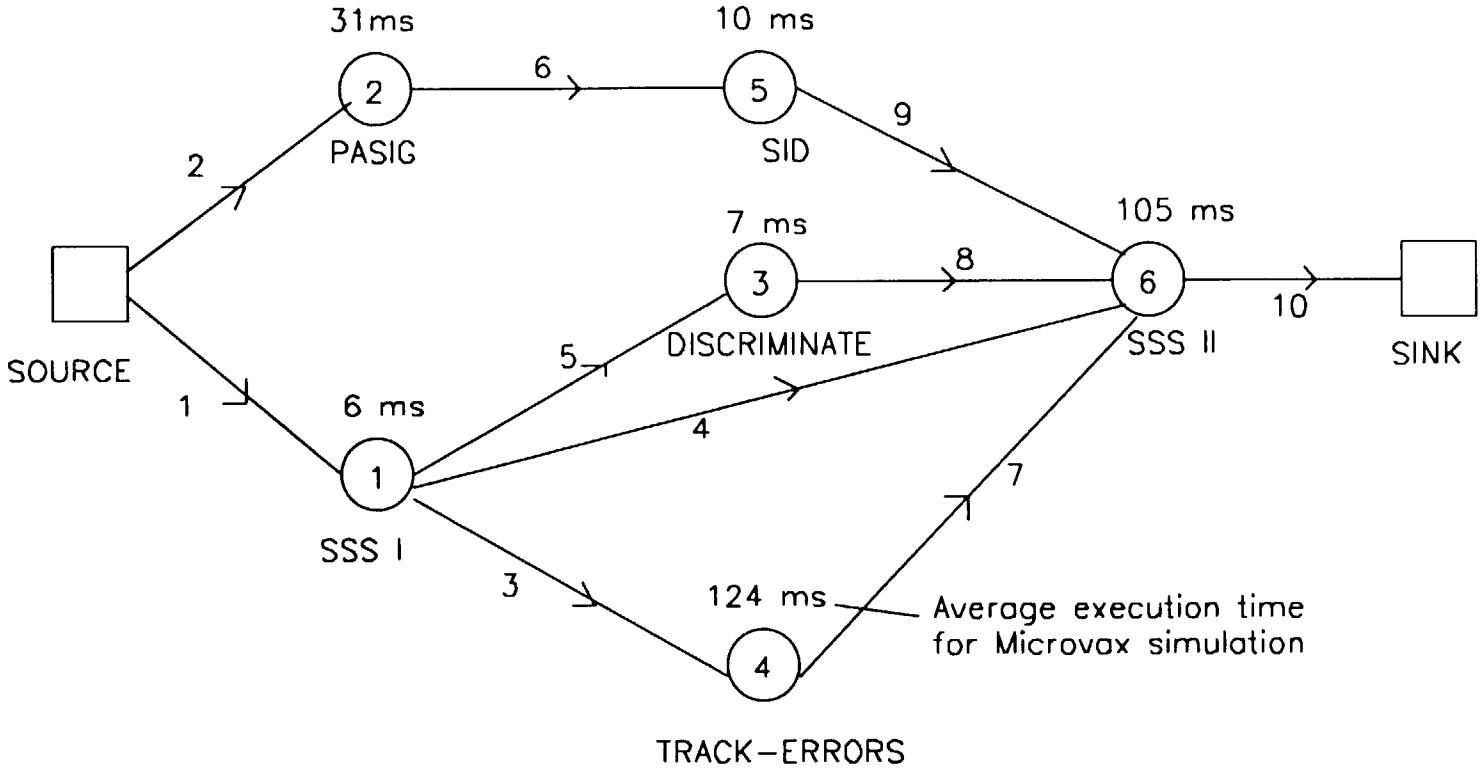


Figure 42. An algorithm marked graph executed on the ADM.



EDGE COMPOSITIONS :

1, 2 : Call-Number

3 : Call-Number, N-Hits

4 : Object-List-Index,
Num-In-Track

5 : Call-Number, T-In-Track

6 : Call-Number, PASIG-Obj-Disc
PASIG-Color

7 : Cov-Bad-Err, Errored-SV
ECI-Cov

8 : D-Conf, Temp-Class-Prob

9 : Call-Number, SID-Obj-Disc,
SID-Color

10 : Temp-Rec

Figure 43. A space surveillance algorithm.

REFERENCES

[1] J. W. Stoughton and R. R. Mielke, "Petri-Net Model for Concurrent Processing of Complex Algorithms,"Proceedings of Government Microcircuit Applications Conference, San Diego, CA, November 1986.

[2] R. R. Mielke, John W. Stoughton, and Sukhamoy Som, "Modeling and Performance Bounds for Concurrent Processing," Proceedings of the 8th International Conference on Distributed Computing Systems, San Jose, CA, June 1988.

[3] R. Mielke, J. Stoughton, S. Som, R. Obando, M. Malekpur, and B. Mandala "ATAMM Multicomputer Operating System Functional Specification," NASA Contractor Report, Grant NCC1-136, August 1990.

[4] J. L. Peterson, Petri Net Theory and the Modeling of Systems, Englewood Cliffs, NJ, Prentice Hall, 1981.

[5] R. R. Mielke, John W. Stoughton, and Sukhamoy Som, "Modeling and Optimum Time Performance for Concurrent Processing," NASA Technical Paper 4167, Grant NAG-1-683, August 1988.

[6] Sukhamoy Som, "Performance Modeling and Enhancement for the ATAMM Data Flow Architecture," Ph.D. Dissertation, Old Dominion University, Norfolk, VA, May 1989.

[7] S. Som, J. W. Stoughton, and R. R. Mielke, "Performance Modeling in the ATAMM Data Flow Architecture," Proceedings of the Ninth IEEE International Phoenix Conference on Computers and Communications, pp. 163-169, Scottsdale, Arizona, March 21-23, 1990.

[8] S. Som, R. R. Mielke, and J. W. Stoughton, "Strategies for Predictability in Real-Time Data-Flow Architectures," Proceedings of the 11th IEEE Real-Time System Symposium, Orlando, Florida, pp. 226-235, December 5-7, 1990.

[9] Robert L. Jones, III, "Diagnostic Software for Concurrent Processing Computer Systems," Master's Thesis, Old Dominion University, April 1990.

[10] Brij Mohan V. Mandala, "A software Design Tool for Predictable Performance in Real-Time Data Flow Architectures," Master's Thesis, Old Dominion University, December 1990.

[11] M. Malekpour, R. Obando, R. R. Mielke, and J. W. Stoughton, "ATAMM Simulation Tool for Data Flow Architectures," Proceedings of the 21st Annual Pittsburgh Conference on Modeling and Simulation, pp. 953-957, May 3-4, 1990.

[12] Vason P. Srini, "An Architectural Comparison of Dataflow Systems," Computer, pp. 68-88, March 1986.

[13] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," IEEE Transactions on Computers, vol. 36, pp. 24-35, January 1987.