ENHANCED MODULAR SIGNAL
PROCESSOR TIMING
SIMULATOR

By

MARILYN OPITZ AIKEN

Bachelor of Science

in Electrical Engineering

University of Oklahoma

Norman, Oklahoma

1982

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 1987

Thesis
1987
A29le
cop. 2

OKLAHOMA STATE UNIVERSITY LIBRARY

ENHANCED MODULAR SIGNAL

PROCESSOR TIMING

SIMULATOR

Thesis Approved:

Sharilyn A. Thoreson
Thesis Adviser

R. E. H[illegible]

D D Fisher

Norman N. Durham
Dean of the Graduate College

1275573

PREFACE

A timing simulator for a static Signal Processing Graph Notation graph was developed for the Enhanced Modular Signal Processor, a data flow computer developed by Bell Telephone Laboratories for the United States Navy. The user inputs the system configuration and the topology of the graph. To implement channels, a constant rate for each channel is read and the timing simulator uses this rate to detect input queues over threshold.

The output consists of the system configuration, queue data information, functional element utilization, node execution information, and an optional timing diagram. This allows the user to simulate graphs for comparison or to simulate modifications to the system and test the feasibility of the proposed modification.

I wish to thank the professors of the Computing and Information Sciences Department for their support and encouragement during my stay at Oklahoma State University. In particular, I wish to express my sincere thanks to Doctor Thoreson for her guidance and continued prodding. I want to thank her for letting me make my own mistakes and then encouraging me to try again and not give up.

I also wish to thank my other committee members, Doctor D.D. Fisher and Doctor G.H. Hedrick, for their advisement in the course of this work and my stay at Oklahoma State University.

Special thanks go to my family for their continued support and concern, but most of all I want to thank my husband and daughter, Calvin and Christina, for their constant support and encouragement.

TABLE OF CONTENTS

Chapter Page

I. INTRODUCTION 1

Introduction to Data Flow 2
Data Flow Architectures 4
Signal Processing and Data Flow 6
Objectives 8
Reasons for Simulator 9

II. ENHANCED MODULAR SIGNAL PROCESSOR COMMON OPERATIONAL SUPPORT SOFTWARE METHODOLOGY 10

Signal Processing Graph Notation 11
Command Program 12

III. ENHANCED MODULAR SIGNAL PROCESSOR 14

Arithmetic Processor 15
Command Program Processor 17
Control Bus 17
Data Transfer Network 18
Global Memory 20
Input/Output Processor 20
Scheduler 21

IV. ENHANCED MODULAR SIGNAL PROCESSOR TIMING SIMULATOR 24

Data Structures 24
Instructions 26
Initialization 28
Input 29
Output 31
Functional Element Conflict Resolution . . . 32
Main Program 33
Support Procedures 34
Arithmetic Processor 35
Control Bus/Data Transfer Network 36
Input Output Processor 36
Global Memory 37
Scheduler 38

V. TEST SIMULATION 41

VI. CONCLUSIONS 44

Future Work. 45

SELECTED BIBLIOGRAPHY. 46

APPENDIXES 49

APPENDIX A - FIGURES. 50

APPENDIX B - ARITHMETIC PROCESSOR INSTRUCTIONS. . . . 61

APPENDIX C - CONFIGURATION FOR TEST CASE. 66

APPENDIX D - TOPOLOGY INPUT 68

APPENDIX E - CONFIGURATION INPUT. 71

APPENDIX F - SIMULATION OUTPUT. 75

APPENDIX G - TIMING DIAGRAM SIMULATION OUTPUT 78

LIST OF FIGURES

Figure | Page

1. Simple Data Flow Graph. 51
2. Data Flow Activity Template for Figure 1. 51
3. An Enhanced Modular Signal Processor Common Operational Support Software Methodology Sample Graph. 52
4. Signal Processing Graph Notation of Figure 3. . . . 53
5. List of Command Program Instructions. 54
6. Enhanced Modular Signal Processor System Architecture 55
7. Control Bus Interface 56
8. Timing Simulator Hierarchy Chart. 57
9. Graph Execution Process 58
10. List of Primitives Not Implemented. 59
11. Test Case Topology. 60

CHAPTER I

INTRODUCTION

John von Neumann in 1946 introduced the concept of a sequential, centralized control executing instructions and a linear memory storing instructions, data, and results. The von Neumann concept has thrived in computer design since its introduction in 1946. Advances in semiconductor device technology and look-ahead instruction decoding have produced vast improvements in speed of execution. Future advances in semiconductor device technology are limited by heat dissipation and basic physical laws, thus a new approach for increasing the speed of execution was necessary. Organizational advances such as pipelining increase performance, but improvements are limited by the sequential control of the von Neumann concept.

Exploiting parallelism was seen as the solution to future improvements in execution speed. The von Neumann machine with its incremental program counter and with partial results being passed between instructions via a memory cell made the specifications of parallelism difficult. Methods to explore and extract parallelism have proved useful and significant.

Introduction to Data Flow

In 1966, Karp and Miller (19) introduced the concept of data flow. In the 1970's, Jack Dennis applied the data flow concept to the design of computer architectures. Data flow is based on two principles, asynchrony and functionality (16,17). When applied to data flow, asynchrony implies an instruction is executable when and only when all required inputs are available. Functionality implies all instructions are functions which, by definition, necessitates an instruction execute without side effects. The first principle implies an instruction is triggered at the earliest possible moment in the execution of a program, thus parallelism is implicitly denoted by the data flow method. The second principle implies the parallelism can be exploited since the order of execution of operations is without side effects. Consequently, two enabled nodes can execute concurrently or in either order without affecting the final results of the task.

To explain why these two principles are significant, an introduction and explanation of data flow is necessary. Data flow is based upon the flow of data through a program in contrast to the von Neumann flow of control concept. Data flow can best be explained by the use of data flow graphs. Figure 1 (Appendix A) gives an example of a simple data flow graph. To fire node 1 which means to execute the instruction at node 1, inputs A and B must both be present on the arcs to node 1. If only input A

is present or only input B, the node is not ready to fire. Upon the reception of a data token (data item or input) on both input A and input B, node 1 (and node 2 in the example) will execute if two processors are available, else one or both will await an available processor. Intermediate results will be matched with other inputs to the succeeding node until all tokens are available. The node will then fire. The presence of the data or tokens causes the node to fire, unlike the von Neumann concept where the existence of control, i.e. the program counter, causes the node (instruction) to fire. Upon execution of node 1 and node 2, node 3 will be fired by inputs C and D.

In the computer, data flow programs are denoted by activity templates. Figure 2 gives the activity templates for the data flow graph in Figure 1. In a data flow manner, activity templates are ready for execution upon reception of all operands and the result is empty. Classical data flow states that each input consists of a single token and that each output consists of a single token. Thus a node cannot fire if the output has a token present on the arc. Modifications to classical data flow allow multiple inputs and outputs, but require token labeling to distinguish different instances of the inputs and outputs.

Data flow languages and data flow architectures exploit parallelism. Computer data flow languages are in general designed to overcome three limitations to von Neumann languages. First is the complexity of resolving all

parallelism in current serial languages. Second, side effects from procedures, go to's, and multiple assignments (variables being reassigned more than once) make exploiting parallelism difficult. Third, serial languages are difficult to verify. Much research into structured programs and program verification has been done to serial languages. Data flow languages can use this research to incorporate structuring and ease of verification into the developing languages.

Data Flow Architectures

Data flow architectures are being designed to exploit parallelism, to utilize Large Scale Integration and Very Large Scale Integration technologies effectively, and to create an easier to program machine. To exploit parallelism is actually a method to obtain higher speeds which is the final goal. Effective utilization of Large Scale Integration and Very Large Scale Integration technologies will improve chip capacity and will capitalize on the cost effectiveness of large numbers of a few types of functional elements. The goal of the data flow architecture is a computer with high performance at an acceptable cost, that is also reliable.

Extensive research has been made into data flow languages and data flow architectures. Numerous data flow languages have been developed such as Val, ID, and LAU. These languages focus on implicitly expressing parallelism.

Similarly, extensive research has been made into data flow architecture. The Massachusetts Institute of Technology static data flow architecture proposed by Dennis and Misunas (9) in 1975 is an example of a ring-based data flow architecture. A prototype of the proposed architecture has not yet been built, but the basic ideas have been used in Texas Instruments Data-Driven Processor, the Toulouse LAU system, and the Manchester Data Flow Processor (22).

The Texas Instruments Data Driven Processor executes Fortran programs. Each operation or node has a maximum of thirteen inputs and thirteen outputs. Memory is local to a processor, thus each node is assigned a processor. Results from node executions are transferred over the interconnection network to the E-bus. The Data-Driven Processor with four processors has been built and tested, but was not commercially exploited.

The Toulouse LAU System, Texas Instruments Data-driven Processor, and the Manchester Data Flow Processor are all ring-based data flow architectures. The Toulouse LAU System has global memories, an execution unit of one to thirty-two processors, a control unit, and an interface. Each node has a maximum of two inputs and several outputs. A prototype of the LAU System with thirty-two processors has been built and tested. The Texas Instruments and Toulouse designs are static architectures which allow only one instance of a node, while the Manchester Data Flow Processor is a dynamic architecture with more than one instance of a node allowable.

Each node has a token associated with the node distinguishing the node from other instances of the node. Simulations have shown ring-based architectures have a bottleneck in the communication paths (15,22,25,28).

The Utah Data-Driven Machine attempted to overcome this communication path problem with a tree structure and the Irvine Data Flow machine uses a N * N communication network (where N is the number of processing elements) for token passing. The Utah Data-Driven Machine is an eight leaf tree structure with the superior elements at the root and inferior elements at the leaves. The superior processor schedules the work of inferior processors in the tree structure. A working prototype of the machine is operational and being evaluated. The Irvine data flow machine was designed to exploit Very Large Scale Integration and to provide a high-level, highly concurrent program organization (25). Packet communication is over a N * N communication network for token passing between processing elements.

Signal Processing and Data Flow

These data flow machines and others proved the feasibility of data flow architectures. Simulations of designs showed bottlenecks that impeded potential execution speed improvements. Using information gained by simulations and/or results from other data flow designs, American Telephone and Telegraph Bell Laboratories commenced research and design on a data flow architecture for signal or data pro-

cessing. In 1982, under contract to the United States Navy, American Telephone and Telegraph Bell Laboratories channeled this research into the underlying design for the Enhanced Modular Signal Processor, the United States Navy's next generation standard signal processor (3).

In research concurrent with research on the Enhanced Modular Signal Processor, the feasibility of executing signal processing applications to a data flow architecture was investigated by a research group at Helsinki University of Technology, Helsinki (18). Their simulation revealed digital signal processing algorithms are generally data value independent, i.e. the sequencing of operations in the algorithm is independent of the data values. Digital signal processing applications are represented by block diagrams of high-level signal processing operations, e.g., Fast Fourier Transforms and bandwidth filtering. These blocks or high-level operations typically are free of side effects, data value independent, and of high computational complexity in terms of elementary arithmetic operations. A continuous stream of source data being processed by a repeatedly executed fixed set of algorithms is the scenario in a real-time digital signal processing task. The architecture Hartimo, et al (18) propose, a Data Flow Signal Processor, is a dynamic token labeling architecture with packet communication. The results of the simulation showed the Data Flow Signal Processor architecture can efficiently handle real-time signal processing applications.

Objectives

The objective of this work is to design and implement a timing simulator for the Enhanced Modular Signal Processor (EMSP). A timing simulator implements the timing required to execute a series of instructions in contrast to a functional simulator which implements the actual output resulting from the execution. Thus the EMSP timing simulator simulates the timing to execute a Signal Processing Graph Notation (SPGN) static graph. The Output for the simulator is the utilization factors for each functional element, the number of firings (executions) for each node, the number of channel firings for each channel, and the state of the queues when the simulation ends. An optional output is a timing chart with opcode identifiers for instructions. Simulation timing depends on the execution time of primitives (algorithms) and the number of words in a data transfer on the Control Bus or the Data Transfer Network. The simulator handles these calculations by using formulas derived from the primitive manual (13) and by knowing the transfer rate and transfer protocols on the Control Bus and the Data Transfer Network. The user can answer all input queries by looking at the SPGN graph, the command program, and the system configuration for the EMSP.

The following chapters more fully explain the EMSP timing simulator. Chapter II discusses SPGN and command programs. Chapter III discusses the EMSP architecture. Chapters IV gives the details of the design and implementa-

tion of the EMSP timing simulator. Chapter V goes through a test simulation of a simple graph on a sparse EMSP configuration. Chapter VI is a summary and a discussion of future work.

Reasons for Simulator

Simulations of proposed computer architectures have become a valuable design tool. Simulators have evaluated the performance of many data flow architecture designs. These simulators reveal design flaws and unexpected complications. Simulations of data flow computers have proved the feasibility of the data flow methodology.

The timing simulator for the Enhanced Modular Signal Processor will be used to evaluate the Enhanced Modular Signal Processor and any future modifications to the Enhanced Modular Signal Processor. Research into the effects of different memory management schemes has been proposed. The timing simulator will be used as a tool in these evaluations. Research into the effects of different system configurations will provide useful information on the limits of the design. The effects of adding resources will reveal optimum configurations and functional element selections. Saturation points for the system will reveal feasibility information on the maximum utilization factors. The Enhanced Modular Signal Processor timing simulator will provide an important and useful tool in the evaluation of the Enhanced Modular Signal Processor and in the evaluation of proposed modifications.

CHAPTER II

ECOS METHODOLOGY

The Enhanced Modular Signal Processor Common Operational Support Software methodology was developed to buffer a signal processing engineer from the programming of a signal processing application and the architecture of the machine executing the program. After dealing with the problems associated with the Advanced Signal Processor, the Navy realized a new methodology was needed to reduce development and maintenance costs of application software.(2) The difficulty in programming applications led the Navy to propose A Common Operational Support Software methodology. A Common Operational Support Software methodology (ACOS) was written in a graph notation that paralleled the block diagram structure of a digital signal processing application. Operations, or primitives as they are more commonly called, are implementation dependent. Thus the Digital Signal Processing engineer could specify the programs in a graph notation easily translatable to A Common Operation Support Software methodology (2,11,29).

Signal Processing Graph Notation

An Enhanced Modular Signal Processor Common Operational Support Software methodology (ECOS) graph executes according to data flow principles. It does not adhere to classical data flow in two aspects (11). Classical data flow requires one token or data element per arc. In contrast, ECOS renames the arcs as queues and allows multiple instances of data elements per queue. Each node execution can require more than one data element per queue. A threshold value specifies how many elements must exist on a queue before the node can execute. Each node execution also specifies an offset (number of data elements to skip over) and a read amount (number of data elements to read) and a consume amount (number of data elements to consume). After node execution, the number of data elements written to the respective output queue(s) is determined by the primitive based on the input read amounts. A Common Operational Support Software methodology allows more than one instance of a node to execute at a time, but gives a warning of possible indeterminacy if this principle is practiced. Secondly, ECOS differs from classical data flow in the nonspecification of node execution. Each node represents a highly complex computation whose implementation is not specified in A Common Operational Support Software methodology. Thus, the execution of a single microprogrammed node instruction may execute in a traditional von Neumann method.

Figure 3 (Appendix A) shows a sample Enhanced Modular Signal Processor Common Operational Support Software methodology graph. Node 2 and node 3 can execute concurrently or in either order after the execution of node 1. The Enhanced Modular Signal Processor Common Operational Support Software methodology graph is translated to Signal Processing Graph Notation in Figure 4 (Appendix A). Signal Processing Graph Notation denotes a static graph or graph realization which is compiled into a load module.

The Enhanced Modular Signal Processor executes the ECOS methodology. The EMSP supports ECOS primitives of a high computational complexity to reflect one or more blocks of a digital signal processing graph. The operations are microprogrammed, machine independent, and executed by single-thread, control flow architectures (12), while ACOS is machine independent and follows a data flow methodology. The Enhanced Modular Signal Processor adheres to the principle of one instance of a graph node instance.

Command Program

To manage graphs, Command Programs were developed. Command Programs, which are application dependent, control graph execution and interaction. Command Programs are written in a High Order Language such as ADA and use a set of procedure calls (Command Program Signal Processing Graph Notation). A command program creates and controls a graph realization into an executing graph instance. More than one

instance of a realization is acceptable, with each instance being created and managed by a Command Program. Command Program instructions are listed in Figure 5 (Appendix A).

CHAPTER III

ENHANCED MODULAR SIGNAL PROCESSOR

The Enhanced Modular Signal Processor is a distributed control, multiprocessor architecture designed to implement Enhanced Modular Signal Processor Common Operational Support Software methodology (2). Under contract to the United States Navy, American Telephone and Telegraph Bell Laboratories is designing the Enhanced Modular Signal Processor as the Navy's next generation standard signal processor (3). Independent modules or functional elements partition the tasks of control, memory management, node scheduling, input/output, and node execution. This modular architecture was chosen to meet design criteria related to throughput, reliability, modularity, and programmability. Throughput is improved by the selection of the data flow methodology, the selection of a crossbar switch to handle multiple data paths in parallel, a token passing control bus, and a separated system control. The data flow methodology was chosen because it naturally exploits the inherent parallelism of signal processing applications and because of the asynchrony and functionality of signal processing graph nodes. Reliability is handled by constant self monitoring by functional elements, by backup critical functional elements, and an

error recovery mechanism. The modular structure of the Enhanced Modular Signal Processor allows the addition of Global Memories and processors to increase memory management or processing power. The criteria pertaining to programmability were met by implementing the Enhanced Modular Signal Processor Common Operational Support Software methodology.

Each module of the Enhanced Modular Signal Processor can execute concurrently with other modules, each module executing a different function of the graph instance(s). Each module type has its own operating system functions and each Enhanced Modular Signal Processor function executes asynchronously. Parallel processing among the Arithmetic Processors was selected to meet the requirement of a throughput rate of over a billion operations per second and the ability to upgrade the system by a factor of sixteen from the minimum configuration. The Enhanced Modular Signal Processor system architecture is shown in Figure 6 (Appendix A).

Arithmetic Processor

The Arithmetic Processors execute the node instruction (primitive), a microprogrammed realization of a highly computationally complex signal processing algorithm, e.g., Fast Fourier Transform, Finite Impulse Response/Infinite Impulse Response filters, beamformers. The Arithmetic Processors are a single-thread, control flow architecture designed to efficiently execute vector multiply and add operations, which

are characteristic of signal processing algorithms. Each Arithmetic Processor operates asynchronously and independently of other Arithmetic Processors. A scheduled node arrives at the Arithmetic Processor in the form of an instruction stream and node setup begins. After accepting the instruction stream and decoding the operands, the Arithmetic Processor requests all input queues and input graph variables from Global Memories. The Arithmetic Processor accepts the queues and graph variables from the Global Memories and stores the data along with the instruction stream in the node execution and control local memory. Upon reception of all inputs, the node is ready to enter the execution phase where the node primitive is executed in the Arithmetic Processor. Upon completion of the execution phase, the node enters the breakdown phase. Breakdown includes the sending of Write Queue and Write Graph Variable instructions for all output queues and graph variables and the sending of a Consume Queue instruction to the Global Memory for all input queues. Queues are not consumed until the breakdown phase because of fault tolerance. If a fault occurs during the execution of a node, the Scheduler reschedules the node on a different processor and the queue will not have been corrupted in Global Memory.

The Arithmetic Processor has separate arithmetic and control units and can support three nodes, one in each of the three phases: setup, execution, and breakdown. This ability increases throughput and allows overlapping or pipelining of node executions. When a node completes the setup

phase, the Arithmetic Processor issues a Request for Instruction Stream to notify the Scheduler that the processor is ready for another instruction.

Command Program Processor

The Command Program Processor executes command programs, handles error detection and recovery, initialization, communication with external devices, and testing and debugging. The Command Program Processor executes command programs which start and stop graphs, start and stop input and output, link and unlink input and output queues, and create queues and graph instances. The Command Program Processor does not participate in graph execution. The Command Program Processor can respond to the failure of functional elements by reconfiguring the Enhanced Modular Signal Processor to remove the element in a graceful system degradation. System initialization is handled by the Command Program Processor and the Command Program Processor communicates with each functional element at system initialization to verify its ability to respond. Testing and debugging instructions are provided by the Command Program Processor during the verification stages.

Control Bus

The Control Bus provides a communication path between functional elements for control messages. The Control Bus uses a token passing technique of arbitration. Messages are transferred on the Control Bus according to the following

procedure. Each functional element upon completing a transmission on the Control Bus begins asserting the Count Clock line (14). See Figure 7 (Appendix A). Each port (functional elements are located on ports) increments its internal counter value upon receiving the Count Clock line clock pulse. If the internal counter value matches the functional element's unique count value and has a message to transmit, the functional element asserts Stop Clock and places the destination port identification word on the Control Bus. The destination port responds with a Transmit Successful (or Transmit Failure if a parity error occurs) and the transmission of the control message commences. To provide synchronization, when the functional element asserting the Count Clock reaches the end of the scan cycle, it asserts Reset Clock and all functional elements reset their internal count value.

The Control Bus provides bidirectional communication between all functional elements. Messages passed on the Control Bus are short compared to messages passed on the Data Transfer Network and are mainly control and status information. The Control Bus transfers messages over an 8-byte data path asynchronously at a maximum data rate of 4.61 megabytes per second (14).

Data Transfer Network

To avoid the bottleneck caused by data path contention cited in many previous data flow computers, a Data Transfer

Network was designed with one or two two by two, four by four, eight by eight, and/or sixteen by sixteen crossbar switches. Up to N (N by N switch) unidirectional communication paths may be connected in parallel, provided each switch input and switch output port is unique with a maximum seven megabytes per second data transfer rate per path. To increase the number of functional elements that can be configured, each switch input has a four to one multiplexer called a concentrator and each switch output has a one to four demultiplexer called a distributor. Enhanced Modular Signal Processor specifications do not allow processor to processor communication or memory to memory communication over the Data Transfer Network. Consequently, a maximum of sixty-four processors is allowable and a maximum of sixty-four Schedulers and Global Memories is possible for a dual Data Transfer Network configuration with two sixteen by sixteen switches.

Two levels of Data Transfer Network arbitration occur concurrently. Arbitration on each concentrator follows a first come first served scheme if the concentrator is not busy. If the concentrator is servicing a transfer request, upon completion of the transfer, the concentrator passes control to the next element requesting a transfer. Connected data transfers on the Data Transfer Network cannot be interrupted, thus if a requested destination is busy with another data transfer, the concentrator will lose its turn

in the arbitration scheme. The concentrator then attempts to service any other pending requests. Each Data Transfer Network follows a token passing method of arbitration. Each concentrator has a time slot and during its allotted time, it will connect a data path, if a data transfer request is pending on the concentrator and if the destination distributor is free.

Global Memory

The Global Memory stores, implements, and manages instruction streams, queues, and graph variables. The Global Memory implements the creation of nodes, queues, and graph variables when graph instances are created. The Global Memory provides dynamic memory management, for allocating and deallocating memory as queues are written and consumed. The Global Memory maintains queue information including threshold, dynamic number of data elements, and the size of data elements to perform dynamic memory management and automatic threshold detection. Every time a queue is consumed or written, the Global Memory automatically checks queue threshold and capacity information and reports any queue events to the Scheduler.

Input/Output Processor

The Input/Output Processor performs, as its name implies, all tasks supporting the input and output of data. The Principles of Operation for the Enhanced Modular Signal

Processor manual (14) lists the following tasks for the Input/Output Processor.

1. To perform signal input/output, the Input/Output Processor implements the process link between the external world and the Enhanced Modular Signal Processor executing graph.
2. To handle many signal data channels with different characteristics, the Input/Output Processor supports concurrent execution of multiple input/output processes with multi-tasking.
3. To achieve data bandwidth reduction, the Input/Output Processor handles front-end signal processing.
4. To support data transfer and control functions, the Input/Output Processor handles inter-functional element communication.
5. To perform Input/Output Processor tasks, the Input/Output Processor manages all its internal resources.

Scheduler

The Scheduler is the functional element in the Enhanced Modular Signal Processor responsible for implementing the data flow methodology. To schedule nodes in a data flow methodology, the Scheduler maintains four data bases storing queue, node and processor information. The Queue to Node Map stores the node identification number for the input and

output nodes for each queue. The Node Characteristic Table contains the number of input queues, type of processor required for execution (e.g., Arithmetic Processor, Input/Output Processor), the identification of the Global Memory containing the node's instruction stream, and the number of conditions, a dynamic count of the input queues yet to go over threshold (2). The Free Functional Element list maintains a list of free processors available for processing nodes. The Ready Node List consists of all nodes that are eligible for execution, but for which no appropriate processor is available.

When the Scheduler receives a Queue Over Threshold, Queue over Capacity, etc., message from a Global Memory, the Scheduler searches the Queue to Node Map for the output node's node identification number. Using the output node's node identification number, the Scheduler decrements the node's Number of Conditions in the Node Characteristic Table for a Queue Over Threshold message. When the Number of Conditions reaches zero, the node is scheduled by sending a Send Instruction Stream to the Global Memory containing the instruction stream (listed in the Node Characteristic Table) if an appropriate processor is free, otherwise the node is placed on the Ready Node List to await a free processor. If the node is placed on the Ready Node List, it will be scheduled when an appropriate processor sends a Ready For Instruction Stream to the Scheduler. Otherwise, the Scheduler will respond to the Ready For Instruction Stream

by placing the processor's identification number on the Free Element List.

CHAPTER IV

ENHANCED MODULAR SIGNAL PROCESSOR TIMING SIMULATOR

In chapter I, the motivation behind designing the Enhanced Modular Signal Processor (EMSP) timing simulator was stated: to provide an important and useful tool in the evaluation of the EMSP and in the evaluation of proposed modifications. Simulation is an accepted practice in the evaluation of the performance of proposed designs. Simulation can estimate performance as well as test proposed modifications. This chapter describes the implementation of the EMSP timing simulator.

The EMSP timing simulator was written in a modular procedural style in C language on a Perkin Elmer 3230. Figure 8 (Appendix A) gives the hierarchy chart for the EMSP timing simulator. The modular procedural style was selected to improve readability, maintainability, and modifiability.

Data Structures

Data structures except for static data structures local to procedures are centrally defined in a header file. Structures were declared for graph execution instructions, functional elements, nodes, channels, and queues. Graph

execution instructions are placed on the event list, ready list, Control Bus request table, and the Data Transfer request table. A graph execution instruction is placed on the event list if the functional element receiving the instruction is busy or if the instruction has an event occurrence time greater than the simulated clock time. Graph execution instructions are placed on the Control Bus request table or the data transfer request table, if the sender functional element is requesting action from the receiver functional element. The functional element structure records the EMSP system configuration for each element, utilization, and request activity.

The node data structure, queue data structure, and channel data structure record graph topology, static setup information, and dynamic information pertaining to number of data elements on a queue. For each input and output to a node, the node structure records whether it is a graph instantiation parameter, a graph variable, or a queue. The value of each graph instantiation parameter and graph variable is stored and the global memory functional element identification number and element size is stored for each graph variable and queue in the node data structure. The queue data structure maintains all node execution parameters and capacity information for each queue. Channel data rate information is kept in the channel data structure. Consequently, between all the structures, the topology of the graph and the parameters of the static graph are defined.

Dynamic capacity of the queues and the dynamic state of the EMSP are maintained by the data structure elements of the graph execution instructions, functional elements, and queues.

Instructions

Two types of instructions are defined for the EMSP. Graph execution instructions are used to implement the data flow methodology. Graph execution instructions are transferred between functional elements on the Control Bus or the Data Transfer Network. These instructions pertain to node scheduling and include requesting queues or graph variables by the processor, writing or consuming queues by a processor, sending instruction streams, etc. Figure 9 (Appendix A) shows the graph execution process of a scheduled node. Each instruction listed is a graph execution instruction, but this is not an inclusive list. Numerous graph execution parameters deal with initialization, booting, error detection and handling, or dynamic graph modifications.

A step-by-step explanation of Figure 9 (Appendix A) will clarify how the EMSP implements the data flow methodology. Although Figure 9 (Appendix A) is the graph execution process of a single scheduled node, each node follows this execution process. Processing for one node begins with an Input/Output Processor or an Arithmetic Processor writing to a queue over the Data Transfer Network to a Global Memory. If the queue goes over threshold (a threshold specified in

the input graph), a Queue Over Threshold message is sent to the Scheduler over the Control Bus. If all input queues for the node are over threshold and an appropriate processor is free, the Scheduler sends a Send Instruction Stream over the Control Bus to the Global Memory storing the Instruction Stream. The Global Memory locates the Instruction Stream and sends the Instruction Stream to the processor specified in the Send Instruction Stream. After Accepting the Instruction Stream, the Arithmetic Processor requests all input queues and graph variables by sending a Request Queue for each input queue or a Request Graph Variable for each input graph variable to the Global Memory over the Control Bus. After all queues and graph variables are accepted by the Arithmetic Processor with Accept Queue and Accept Graph Variable instructions, the processor sends a Ready For Instruction Stream to the Scheduler over the Control Bus and commences primitive or node execution. Upon completion of the primitive execution, the processor writes all output queues and graph variables by sending the Write Queue and Write Graph Variable instructions over the Data Transfer Network. If the Global Memory detects a Queue Over Threshold or Capacity as the queue is written, the Global Memory sends a Queue Over Threshold or Capacity instruction to the Scheduler and a new node may be scheduled. After all outputs are written, the processor commences consuming all input queues by issuing the Consume Queue instruction over the Control Bus to the Global Memory. If a queue is over threshold after consumption, a Queue Over Threshold message

is sent to the Global Memory or if the queue goes under threshold, a Queue Under Threshold message is sent to the Global Memory.

In Figure 9 (Appendix A), one step of the graph execution has requests for graph variables and queues as input and writing graph variables and writing and consuming queues as output to an Arithmetic Processor. This is the primitive execution phase where the signal processing application algorithms specified by the Signal Processing Graph Notation graph nodes are executed. These primitives or arithmetic processor instructions are arithmetic and/or logical calculations that may be highly computationally complex signal processing algorithms such as Fast Fourier Transforms or Infinite Impulse Responses, or simple vector logical functions. The primitives are executed by a von Neumann sequential, centralized control Arithmetic Processor. A complete list of the primitives are given in Appendix B.

Initialization

Initialization consists of defining all variables in the functional element structures to null conditions, to nullifying all node pointers, to setting the time to zero, nullifying all channel information, and initializing all request lists to empty. The instruction list, the Control Bus list, and the Data Transfer Network list must be initialized to empty lists before input to the system creates initial instructions. Initialization of the lists and the TIME variable are necessary for proper program execution,

but other initialization was done as a precautionary measure to insure a valid known initial state of the simulator.

Input

After initialization, inputs are read from two input files whose names are interactively entered. The first file contains the static graph's topology. For each node, the inputs must be entered in the following order. First, the node's unique identification number must be read and the node's primitive mnemonic (Appendix B). The simulator uses the mnemonic to locate the primitive in a table and to get information about the primitive's inputs, outputs, and timing requirements. The simulator next reads the type (graph variable, graph instantiation parameter, or queue) for each input. If the input is a graph variable, the graph variable identification number is read. If the input is a queue, the queue identification number, the threshold, consume, and read node execution parameters are read. If one of the primitive inputs is a family of queues, a -1 must be entered to signal the end of the family. After the inputs for the node are read, the outputs are read. The simulator reads the type (graph variable or queue) of output and the output's identification number. If the variable is a queue, the valve amount is read. If one of the primitive outputs is a family of queues, a -1 must be entered to signal the end of the family. The information in this input file completely specifies the static graph or graph realization.

To identify the EMSP system and a more dynamic graph, a second file of information is necessary. This file contains output parameters, the system configuration, channel information, and variable information. The simulator first reads the maximum number of time units the simulator is to simulate and whether the user prefers a timing chart. Next the simulator defines the system configuration by reading the number of Data Transfer Networks and the switch size of each Data Transfer Network, and the functional element information. Each functional element has the following ordered input: functional element identification number, type of functional element (Arithmetic Processor, Scheduler, etc.), Data Transfer Network of the concentrator, concentrator number, element of concentrator, Data Transfer Network of distributor, distributor number, and element of distributor. A -1 as the functional element identification number is used to signal the end of functional element information.

Next channel information is read. Channel information is used to fully define the queues that are input or output to the graph. The simulator reads the following ordered channel information: channel identification number, channel priority, channel rate, the queue attached to the channel, the functional element identification number of the Input/Output Processor, and whether the channel is an input or output channel. A -1 as the channel identification number is used to end the channel information.

Finally, the simulator reads the node's Global memory identification number, variable values, and queue capacity information. For each node, the simulator reads the identification number of the Global Memory storing the instruction stream. Sequentially for each input except for the queue the variable value is read, and except for the graph instantiation parameter the Global Memory identification number is read, and for each input queue, the queue capacity is read. Sequentially for each output, the Global Memory is read. If the output is a queue attached to an output channel (sink), the capacity of the queue and the threshold is read after the Global Memory identification number since the this information is normally read when the node at the head of the queue is read. If the output is of type queue and not a sink queue, the capacity is read after the Global Memory identification number.

Output

Output for the EMSP timing simulator consists of a title, a system configuration chart, a utilization factor for each functional element, a total time to execute a graph (or a maximum time), and an optional step-by-step runtime utilization graph, node execution information, channel execution information, and queue information. The configuration chart echoes the input configuration and will be useful to verify the input configuration and as a reference when doing comparison studies. After the configuration chart, the total time to execute the static graph is

stated or, if the static graph is a continuous loop, the maximum execution time specified. Utilization factors will be used to evaluate the Enhanced Modular Signal Processor and any proposed modifications. The number of firings per node is in a table giving the node identification number, the node opcode, and the number of times the node was fired. The optional step-by-step runtime utilization graph is a timing chart for the Enhanced Modular Signal Processor and is optional because of the large quantity of the output and the severe degradation to the performance of the simulator. Node, channel, and queue information gives the number of the times the node or channel was scheduled and the number of elements on the queue at the time the simulation stops.

Functional Element Conflict Resolution

The EMSP Principles of Operation Manual (14) does not fully define the hardware implementation of the Control Bus or the Data Transfer Network. The manual states the Control Bus allocates timing slots based on an internal count value which is a function of the functional element's identification number. The function relating the identification number to the timing slot and the method of arbitration resolution are unpublished. Consequently, a decision was made to allocate timing slots from 0 to the maximum possible number of functional elements in a linear order on the simulator.

The manual (14) states the Data Transfer Network simultaneously creates data paths between functional elements, resolves concentrator element conflicts, and resolves concentrator conflicts. Since the method of conflict resolution was not fully defined with start up conditions, it was decided to select arbitrarily the lowest numbered element of a concentrator and the lowest numbered concentrator. Conflict resolution and Data Transfer Network operations follow the method stated in Chapter III.

Main Program

The main program either directly or indirectly calls all procedures and handles the actual time scheduling. Before starting the timing simulation, the main program calls the initialization procedure, and the input procedures. Then, it calls the output procedures to print a title to describe the simulation and the simulated system configuration for future reference. To start the graph, the main program initializes the time to zero and calculates the firing times of all source nodes. The main program then enters a loop that executes the event processing of the functional elements, the Control Bus, and the Data Transfer Network. Instructions are scheduled according to an event list. Events or instructions are placed on a singly-linked list in ascending order of the execution time. The Data Transfer Network and Control Bus are not event scheduled because the Enhanced Modular Signal Processor per-

forms arbitration resolution every time unit concurrently with instruction execution. After the loop maximum time has exceeded or no further instructions are executable, the main program prints utilization factors and a complete list of the number of times each node or channel was scheduled, and queue execution information.

Support Procedures

A number of procedures are necessary for sorting, searching, calculating execution information, and creating dynamic memory allocation. These support procedures are invisible to the user and are mentioned only for the purpose of fully defining the simulator. Because of the frequency of accessing elements of the channel, node, and queue arrays, it is feasible to order these arrays. Quicksorts are used to sort the nodes and channels. A linear insertion method is used to order the queues since before each insertion, the queue array is searched for the queue. Binary searches were chosen to take advantage of the sorted arrays. Procedures to calculate the timing for node execution, to calculate the size of variables, and to calculate the produce amount for queues were placed in separate procedures to improve readability and modifiability.

Arithmetic Processor

Since the Arithmetic Processor can handle three nodes simultaneously, each Arithmetic Processor required more than the timing variable in the functional element data structure. The timing variable in the functional element data structure is used for the node or nodes in setup or breakdown mode. A separate timing variable is required for the node in each Arithmetic Processor in the execution mode. To handle the requirement of only one node in setup mode, the Arithmetic Processor sends a Ready for Instruction Stream when a node has completed setup mode. To handle only one node in execution mode, the execution node timing is checked for the specified Arithmetic Processor and if a node is in execution mode, the execution time is increased by the execution time of the node completing setup mode.

Arithmetic Processors implement all instructions necessary to execute a primitive. The primitive execution time is calculated when the dynamic node information is read and is stored in the node information. The Arithmetic Processor handles Accept Instruction Stream instructions and then generates the appropriate Request Queue and Request Graph Variable instructions. After all inputs for a primitive have been accepted with Accept Queue and Accept Graph Variable instructions, the primitive enters the execution mode. During breakdown mode, the Arithmetic Processor generates Write Graph Variable or Write Queue instructions to all output variables and Consume Queue instructions to all input queues.

Control Bus/Data Transfer Network

The procedures to handle the simulation of the Control Bus and the simulation of the Data Transfer Network are very similar. The procedures begin with a loop that traverses the linked list of waiting requests and places the requests in the appropriate element of an array ordered by functional element identification number (Control Bus) or concentrator configuration (Data Transfer Network). This loop places all requests on the request array for future scheduling.

Next, another loop handles the timing for attempting the data transfer or making the data transfer. It checks the request array and if an instruction is waiting to be scheduled it attempts to schedule or schedules the request. The Data Transfer Network procedure checks the status of the destination distributor. If the distributor is busy, the procedure implements the timing for an attempted transfer, otherwise the procedure implements the timing for the data transfer. Scheduling follows the methodology discussed in Chapter II.

Input/Output Processor

To allow for flexibility with input and output, the Input/Output Processor design specifications were kept at a minimum. The Input/Output Processor handles a minimal number of instructions. The EMSP design specifications allow for Input/Output Processors to execute a variable number of channel instructions simultaneously. EMSP imple-

mentations to-date have restricted the Input/Output Processor to executing only one channel instruction at a time and this is the approach of the EMSP timing simulator.

The Input/Output Processor issues itself an Execute Node instruction when an input channel goes over threshold. The Input/Output Processor sends a Write Queue to the Global Memory storing the queue. An Execute Instruction Stream instruction writes to an output channel and has a set time associated with the instruction. The other two instructions the Input/Output Processor implements of significance to a static graph are Continue Node Data Transfer and Suspend Node Data Transfer. Suspend Node Data Transfer suspends a channel until a Continue Node Data Transfer reverses the suspension.

Global Memory

To implement the timing for Global Memory operations, a Global Memory procedure was designed in the timing simulator. A Global Memory executing an instruction cannot be interrupted. Therefore, upon receiving an instruction, the Global Memory procedure first checks if the Global Memory is free and if not, replaces the instruction on the event list with a future event time.

The Global Memory handles requests for instructions, queues, and graph variables. The timing to execute these requests is a function of the number of words in the node instruction stream, queue, or graph variable. The Global Memory procedure after calculating the timing places the

Accept Instruction, Accept Queue, or Accept Graph Variable instruction on the Data Transfer Network. The Global Memory procedure handles Write Queue and Consume Queue instructions by updating the queue number of data items and sending any appropriate Queue Over Capacity, Queue Under Capacity, Queue Over Threshold, and/or Queue Under Threshold instructions over the Control Bus to the Scheduler.

Scheduler

The Scheduler procedure handles the timing simulation for the Scheduler functional element. The Scheduler can execute only one instruction at a time. If the Scheduler is busy when an instruction on the event list is ready to be scheduled, the instruction must be returned to the event list with a time equivalent to the Scheduler's next free time.

A brief explanation of a few of the Scheduler's instructions are included as an overview. To handle Queue Over Capacity instructions, the Scheduler procedure updates the queue's status and if the queue is attached to an input channel, sends a Suspend Node Data Transfer instruction to the channel. For an internal or source queue, a Queue Over Threshold instruction causes the Scheduler to decrement the number of conditions variable (if the number of conditions was not previously decremented for the queue) for the node at the queue's head. If the number of conditions is zero and the node or channel is not suspended, the Scheduler attempts to schedule the node. If the free processor list

has an available processor, the node is scheduled by sending a Send Instruction Stream instruction to the Global Memory storing the instruction. If a processor is not available, the instruction is placed on a ready list to await a free processor. If the queue is a sink queue, the Queue Over Threshold instruction is handled by verifying the channel is not suspended and sending an Execute Instruction Stream instruction to the Input/Output Processor. A Queue Under Capacity instruction changes the status of the queue and node, and, if the node's number of conditions is zero, the node is scheduled or placed on the ready list. A Ready for Instruction Stream instruction is implemented by scheduling a waiting node if available or by placing the processor's functional element identification number on the free processor list.

CHAPTER V

TEST SIMULATION

A sample graph consisting of three nodes executing on a five functional element system (two Global Memories) was selected as an example for further discussion. Appendix C gives the complete details of the EMSP configuration for the simulation and the graph to be simulated. After initialization, the simulator requests the maximum number of microseconds the user desires to simulate, whether the user desires to enter debug mode where an optional timing chart is printed, and the names of the two input files. The first input file contains the graph topology and can be easily translated from a Signal Processing Graph Notation program. When queried about threshold, read, consume, and offset amounts, a negative number implies the node execution parameter is dependent on a previous input parameter. For example, if the threshold for parameter two, a queue, is equivalent to N, parameter one, a negative one would be input for the threshold for parameter two. This interdependency is very common and this method was chosen as a quick and easy method for the user to remember and follow. Appendix D gives the query session for the input of the graph in

Appendix C. Comments are contained within delimiters /*COMMENT*/ and user inputs are contained within delimiters *input* for ease of reading.

Input file two contains the system configuration and all graph information besides topology (file one). The system configuration procedure reads input file two for functional element identification numbers, functional element type (Arithmetic Processor, Global Memory, etc.), and Data Transfer Network configuration of the element's concentrator and distributor (Appendix C). The channel identification number, channel priority, channel rate, Input/Output Processor, and channel type (Input or Output) for each channel is read by the channel procedure. Next, the read values procedure, after ordering the nodes by node identification number, reads the identification numbers of the Global Memory storing the nodes, graph variables, and queues. All values for graph instantiation parameters and graph variables and all other pertinent information are read. Appendix E gives the complete query session for the graph in Appendix C. The interested reader will find the session well documented and self-explanatory.

Output for the simulation consists of a table of the EMSP system configuration, functional element utilization information, node execution information, channel execution information, and queue execution information. The system configuration table echoes the system configuration input by

the user. The function element utilization information gives the utilization factors for each functional element. Utilization is calculated as the number of time units the functional element was busy divided by the number of time units the simulation simulated. This method of utilization calculation was modified for the Arithmetic Processor to handle the Arithmetic Processor's ability to execute a node and set up a node simultaneously. The utilization for the Arithmetic Processor calculates busy time as the time the element was executing setup or breakdown instructions plus the time executing the instruction. Thus an Arithmetic Processor could have a possible two hundred percent utilization. The node execution information gives the number of node firings for each node, the opcode for the node, and the node identification number. Channel execution information gives the channel identification number and the number of channel firings. Queue execution information gives the queue identification number, the number of data items on the queue at the end of the simulation, and the nodes or channels at the head and tail of each queue. Appendix F gives the test simulation for the inputs specified in Appendix B, Appendix C, and Appendix D. Appendix G gives a complete listing of the test simulation with the optional timing chart. The timing chart is documented and instructions give the exact time units to the nearest one hundredth of a time unit (divide the time in the instruction comment by one hundred).

CHAPTER VI

CONCLUSIONS

The Enhanced Modular Signal Processor (EMSP) timing simulator simulates the timing for a static graph during the graph execution process. Nodes are scheduled using a modified data flow methodology which allows multiple elements on a data input. A scheduled node follows the graph execution process shown in Figure 9 (Appendix A). Inputs to the system consist of interactively entering the names of two input files and the specified input files. Output consists of the system configuration, an optional timing chart, functional element utilization, node and channel execution information, and queue information.

Figure 10 (Appendix A) gives a complete list of the primitives that could not be executed. Six of the primitives (the ones with a star in Figure 10 (Appendix A)) were unexecutable because they had more than seven inputs. A fixed length opcode data table was chosen for quick searches and ease of modification, a design decision was made to allow only seven inputs to save storage. If the opcode data table had allowed twelve inputs, the above six primitives would have been supported, but at a cost of five extra bytes per primitive.

would have been supported, but at a cost of five extra bytes per primitive.

The other nine unexecutable primitives had input or output queues whose length were dependent on a node's execution. Since the primitives were not executed, the length of the queues was unknown and the timing for Read Queue and Write Queue instructions could not be calculated.

Future Work

The EMSP does not simulate a dynamic graph where input variables and channels can be dynamically changed during graph execution. One reason for not simulating a dynamic graph is the huge amount of overhead and bookkeeping necessary to handle a dynamic graph. A second reason is the difficulty in specifying the timing for dynamic instructions. The third reason is the timing associated with dynamic actions to the graph execution processing instructions are generally in ratios exceeding 1000 to 1. But the major reason for simulating static graphs is the lack of a thorough understanding of the effects on the graph of a dynamic system. Using the static EMSP timing simulator and information gathered from test simulations, a more thorough understanding of the EMSP can be obtained. Using this understanding and by obtaining more information from the EMSP designers, i.e. Bell Telephone Laboratories, a dynamic EMSP simulator could be designed to perform more thorough analysis of the EMSP data flow computer.

SELECTED BIBLIOGRAPHY

(1) Arvind, and V. Kathail, "A Multiple Processor Data Flow Machine that Supports Generalized Procedures", Conference Proceedings the 8th Annual Symposium on Computer Architecture, Minneapolis, Minnesota, May 12-14, 1981, pp291-302.

(2) Bloch, Fedrick H., "The Enhanced Modular Signal Processor", Proceedings of the Seventeenth Annual Pittsburgh Conference on Modeling and Simulation, Vol. 16, Part 3, 1986, pp. 829-836.

(3) Brown, N.H., "The EMSP Dataflow Computer", Proceedings 17th Hawaii International Conference System Sciences, Honolulu, Hawaii, January, 1984, pp39-48.

(4) Burkowski, F.J., "A Multi-user Data Flow Architecture", Conference Proceedings the 8th Annual Symposium on Computer Architecture, Minneapolis, Minnesota, May 12-14, 1981, pp327-340.

(5) Damodaran, Meledath and Amitava Hazra, "Methods for System Simulation on a Restricted Data Flow Architecture", ACM National Conference Proceedings 1981, November 9-11, pp60-66.

(6) Davis, Alan L. and Robert M. Keller, "Data Flow Program Graphs", Computer, Vol. 15, No. 2 (February, 1982), pp26-41.

(7) Dennis, Jack B., "Data Flow Supercomputers", Computer, Vol. 13, No. 11 (November 1980), pp48-56.

(8) Dennis, Jack B., Willie Y.P. Lim, and William B. Ackerman, "The MIT Data Flow Engineering Model", Proceedings of the IFIP World Computer Congress 1983, pp553-560.

(9) Dennis, J.B. and D.P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor", Proceedings 2nd Annual Symposium on Computer Architecture, January 20-22, 1975, pp126-132.

(10) Dennis, Jack B., Joseph Stoy, and Bhaskar Guharoy, "VIM: An Experimental Multi-user System Supporting Functional Programming", Proceedings International Workshop on High Level Computer Architecture 1984, pp1.1-1.9.

(11) ECOS Tutorial: Preliminary, April 26, 1985, pp1-29.

(12) EMSP/ASP Common Operational Support Software Methodology Specification Version 3.0, Prepared by Analytic Disciplines, Inc. (now Evaluation Research Corporation) under contract to the Naval Research Laboratory, May 31, 1984.

(13) Enhanced Modular Signal Processor (EMSP) Primitive Analysis Specification, CDRL C130, dated February 22, 1985, prepared for the Naval Sea Systems Command, PMS412 by AT&T Bell Laboratories on behalf of AT&T Technologies under contract N00024-81-C-7318.

(14) Enhanced Modular Signal Processor (EMSP) Principles of Operation, Prepared by AT&T Bell Laboratories for the Naval Sea Systems Command (PMS412), March 15, 1985.

(15) Farrell, Edward F., Noondin Ghani, and Philip Treleaven, " A Concurrent Computer Architecture and A Ring Based Implementation", Conference Proceedings of the 6th Annual Symposium on Computer Architecture, April 23-25, 1979, pp1-11.

(16) Gajski, D.D., D.J. Padua, D.J. Kuck, and R.H. Kuln, "A Second Opinion on Data Flow Machines and Languages", Computer, Vol. 15, No. 2 (February 1982), pp58-69.

(17) Gostelow, Kim P., and Robert E. Thomas, "A View of Dataflow", Proceedings of the AFIPS National Computer Conference 1979, June 4-7, Vol. 48, pp. 629-635.

(18) Hartima, Iiro, Klaus Kronlof, Olli Simula, and Jorma Skytta, "DFSP: A Data Flow Signal Processor", IEEE Transactions on Computers, Vol. C-35, No. 1, January 1986, pp23-33.

(19) Karp, R.M. and R.E. Miller, "Properties of a Mode for Parallel Computation: Determinancy, Termination, and Queuing", SIAM Journal of Applied Mathematics, Vol. 14, No. 6 (November, 1966), pp 1390-1411.

(20) Patnaik, L.M., R. Govindarajan, and N.S. Ramodoss, "Design and Performance Evaluation of EXMAN: An EXtended MANchester Data Flow Computer", IEEE Transactions on Computers, Vol. C-35, No. 3, March 1986, pp229-244.

(21) Srini, Vason P., "A Fault-Tolerant Dataflow System", Computer, Vol. 18, No. 3 (March 1985), pp54-68.

(22) Srini, Vason P., "An Architectural Comparison of Dataflow Systems", Computer Vol. 19, No. 3 (March 1986), pp68-88.

(23) Srini, V.P., "An Architecture for Extended Abstract Data Flow", Conference Proceedings the 8th Annual Symposium on Computer Architecture, Minneapolis, Minnesota, May 12-14, 1981, pp303-325.

(24) Treleaven, Philip C., "Exploiting Program Concurrency in Computing Systems", Computer, Vol. 12, No. 1 (January 1979), pp42-49.

(25) Treleaven, Philip C., David Brownbridge, and Richard P. Hopkins, "Data Driven and Demand Driven Computer Architecture", Computing Surveys, Vol. 14, No. 1, March, 1982, pp93-143.

(26) Treleaven, Philip C., Richard P. Hopkins, and Paul W. Rautenbach, "Combining Data Flow and Control Flow Computing", The Computer Journal, Vol. 25, No. 2, February, 1982, pp207-217.

(27) Watson, Ian, and John Gurd, "A Practical Data Flow Computer", Computer, Vol. 15, No. 2 (February 1982), pp 51-57.

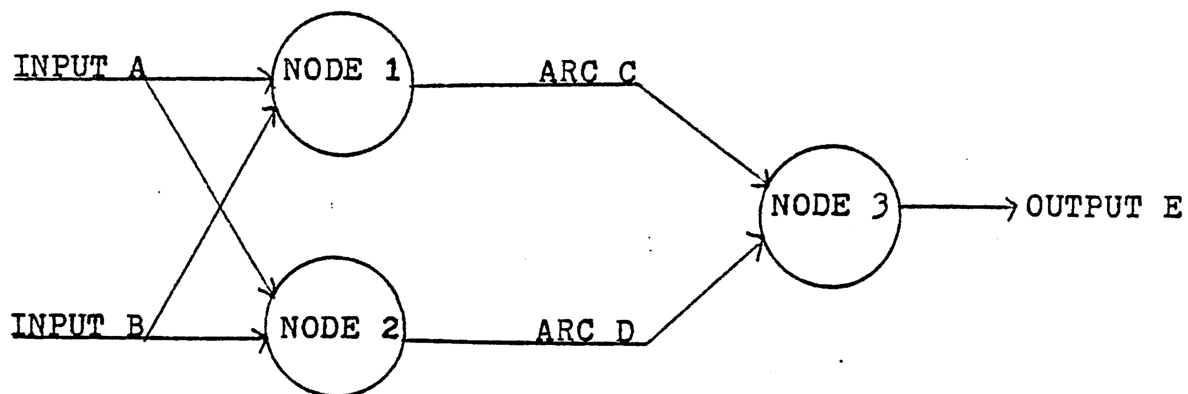
(28) Watson, Ian, and John Gurd, "A Prototype Data Flow Computer with Token Labelling", Proceedings of the AFIPS National Computer Conference 1979, Vol. 48, pp623-628.

(29) Wu, Y.S., "A Common Operational Software (ACOS) Approach to a Signal Processing Development System:, U.S. Naval Research Laboratory, Washington, D.C. 20375, ICASSP83, Boston, pp1172-1175.

APPENDIXES

APPENDIX A

FIGURES



FORMULA: E = (A + B) * (A - B)

Figure 1. Simple Data Flow Graph.

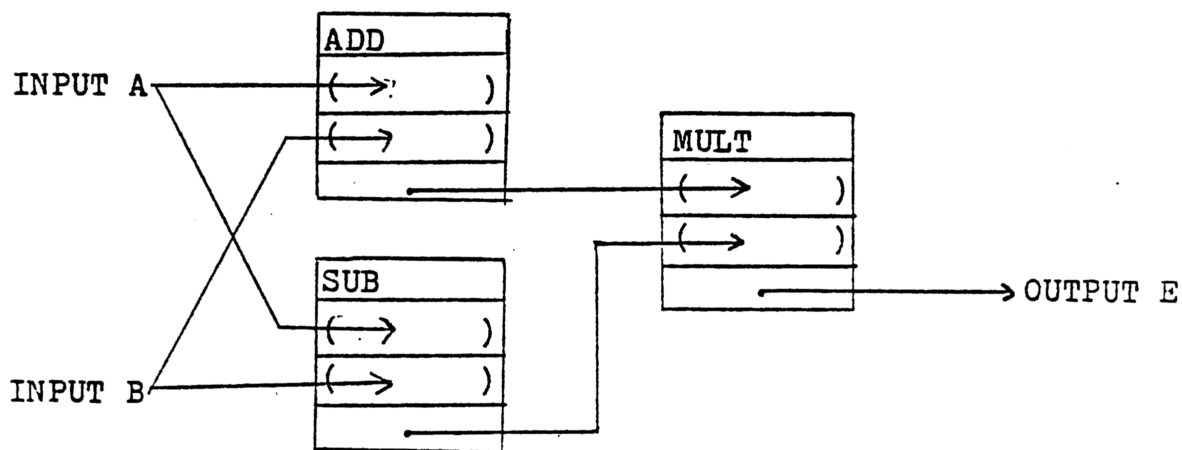


Figure 2. Data Flow Activity Template for Figure 1.

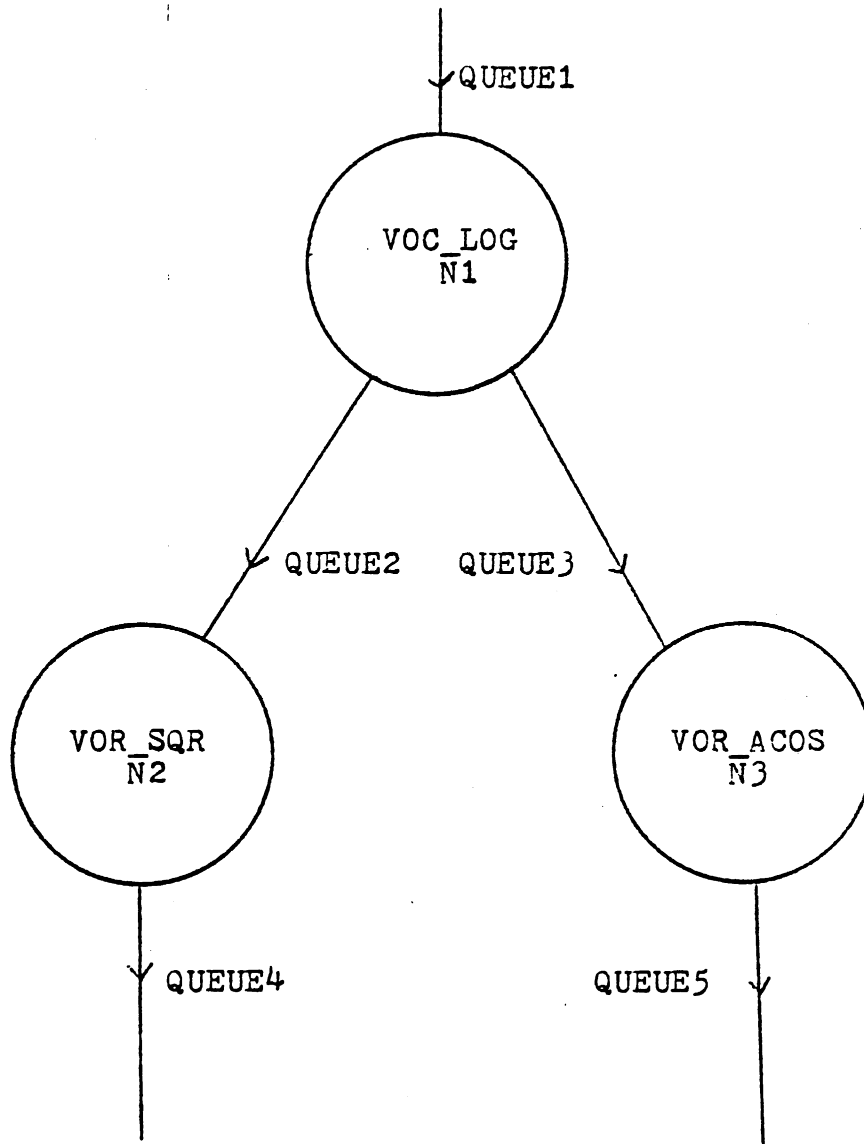


Figure 3. An Enhanced Modular Signal Processor Common Operational Support Software Methodology Sample Graph.

```
%GRAPH(FIGURE3
 GIP = N:INT
 INPUTQ = QUEUE1:CFIXED
 OUTPUTQ = QUEUE4,
           QUEUE5:FIXED)
%%
%QUEUE(QUEUE2,QUEUE3:FIXED)
%%
%NODE(N1
  PRIMITIVE = VOC_LOG
  PRIM_IN = QUEUE1 THRESHOLD = N
                   READ = N
                   OFFSET = 0
                   CONSUME = N
  PRIM_OUT = QUEUE2,QUEUE3)
%%
%NODE(N2
  PRIMITIVE = VOR_SQR
  PRIM_IN = QUEUE2 THRESHOLD = N
                   READ = N
                   OFFSET = 0
                   CONSUME = N
  PRIM_OUT = QUEUE4)
%%
%NODE(N3
  PRIMITIVE = VOR_ACOS
  PRIM_IN = QUEUE3 THRESHOLD = N
                   READ = N
                   OFFSET = 0
                   CONSUME = N
  PRIM_OUT = QUEUE5)
%%
%ENDGRAPH
```

Figure 4. Signal Processing Graph Notation of Figure 3.

COMMAND	DESCRIPTION
SPAWN	Create a process
ABORT	Abort a process
START	Start a process
STOP	Stop a process
INITIO	Link queue to channel
STARTIO	Start or resume a channel
STOPIO	Stop a channel
CREATEQ	Create a queue
DESTROYQ	Destroy a queue
INITQ	Initialize a queue
FLUSHQ	Remove all elements from a queue
CONNECTQ	Connect a queue to Command Program
DISCONNECTQ	Disconnect a queue from a Command Program
ADDDATA	Add data elements to a queue
WAITDATA	Wait for data on a queue
CREATEGV	Create a graph variable
DESTROYGV	Destroy a graph variable
READGV	Read a graph variable
WRITEGV	Write a graph variable
UNLINK	Unlink a queue
LINK	Link a queue
REINIT	Reinitialize all queues and graph variables
RESUME	Resume a process

Figure 5. List of Command Program Instructions

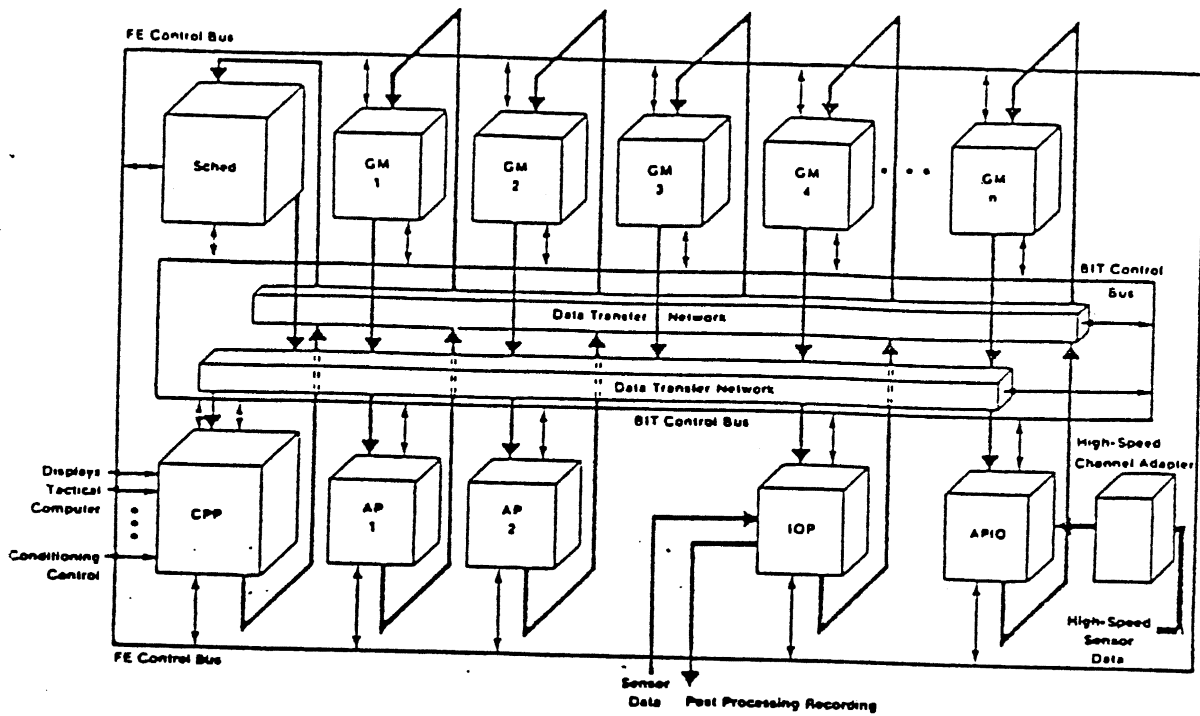


Figure 6. Enhanced Modular Signal Processor System Architecture.

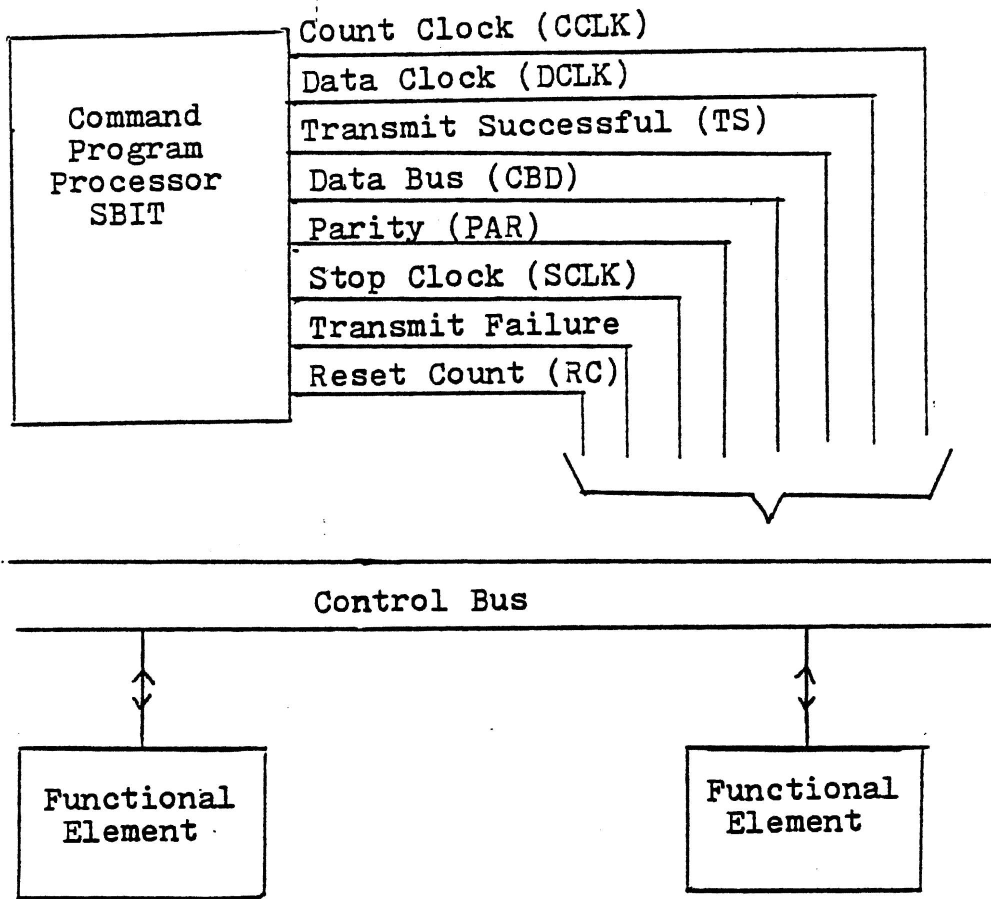


Figure 7. Control Bus Interface.

```
main()
  init()
  read_nodes()
  read_config()
  read_iop()
  read_values()
  output_title()
  output_config()
  start_iop()
  CBUS()
  DTN()
  AP()
  GM()
  IOP()
  SCH()
  output_util()
  output_chan()
  output_queues()

read_nodes()
  create_node()
  ap_opcode()
  sort()

read_config()
  schinit()

read_iop()
  set_queue()
  create_queue()

read_values()
  cal_size()
  cal_produce()
  cal_time()

CBUS()
  delete_list()
  insert_list()

DTN()
  delete_list()
  insert_list()

AP()
  create_instruct()
  set_channel()
  set_node()
  delete_list()
  insert_list()

GM()
  set_channel()
  create_instruct()
  set_node()
  insert_list()
  delete_list()

IOP()
  create_instruct()
  set_channel()
  set_node()
  insert_list()
  delete_list()

SCH()
  create_instruct()
  set_channel()
  set_node()
  insert_list()
  delete_list()

ap_opcode()
  search()
  read_queue()
  oread_queue()

read_queue()
  set_queue()
  create_queue()

oread_queue()
  set_queue()
  create_queue()
```

Figure 8. Timing Simulator Hierarchy Chart.

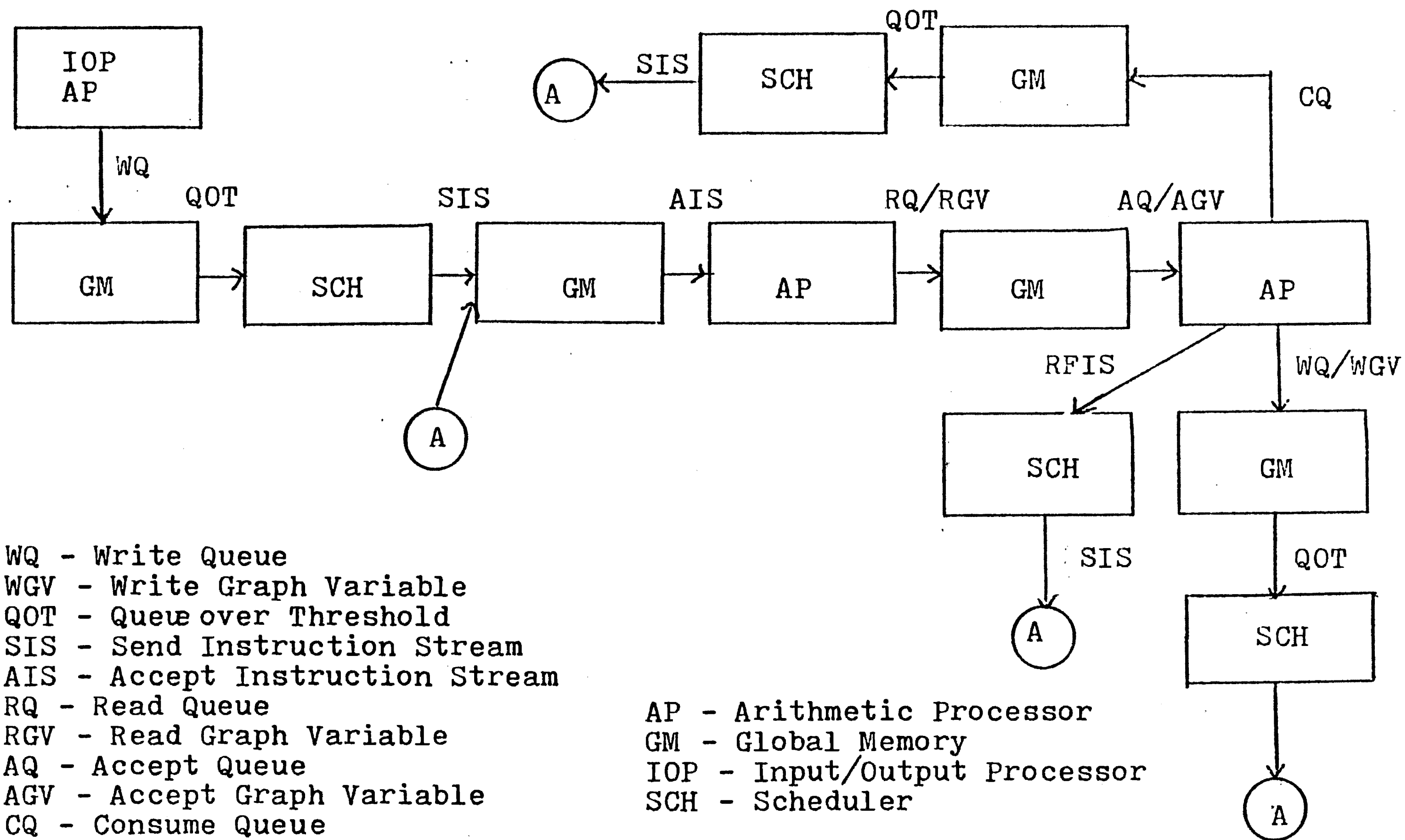


Figure 9. Graph Execution Process.

MNEMONIC	PRIMITIVE
FIR_CNS	Finite Impulse Response Filter(Complex, N Stage)
FIR_RNS	Finite Impulse Response Filter(Real, N Stage)
CDM_RFIR	Complex Demodulation and FIR Filter (Real, Fixed Freq.)
BFR_FREQ	Frequency Domain Beamformer
BFR_FREQB	Frequency Domain Beamform (B)
BFR_GEA	Prester and Adaptive Beamform
BFR_GEAB	Prester and Adaptive Beamform
BFR_TIME	Time Domain Beamform
SSP_BEPR	Bearing Estimate Pre-process Real
SSP_CONV	Conversion
SSP_FROD	Frequency Determination
SSP_INDX	Peak Index
SSP_PKDT	Peak Detect
SSP_PKPK	Peak Pick
SSP_STAT	Period Statistics

Figure 10. List of Primitives Not Implemented.

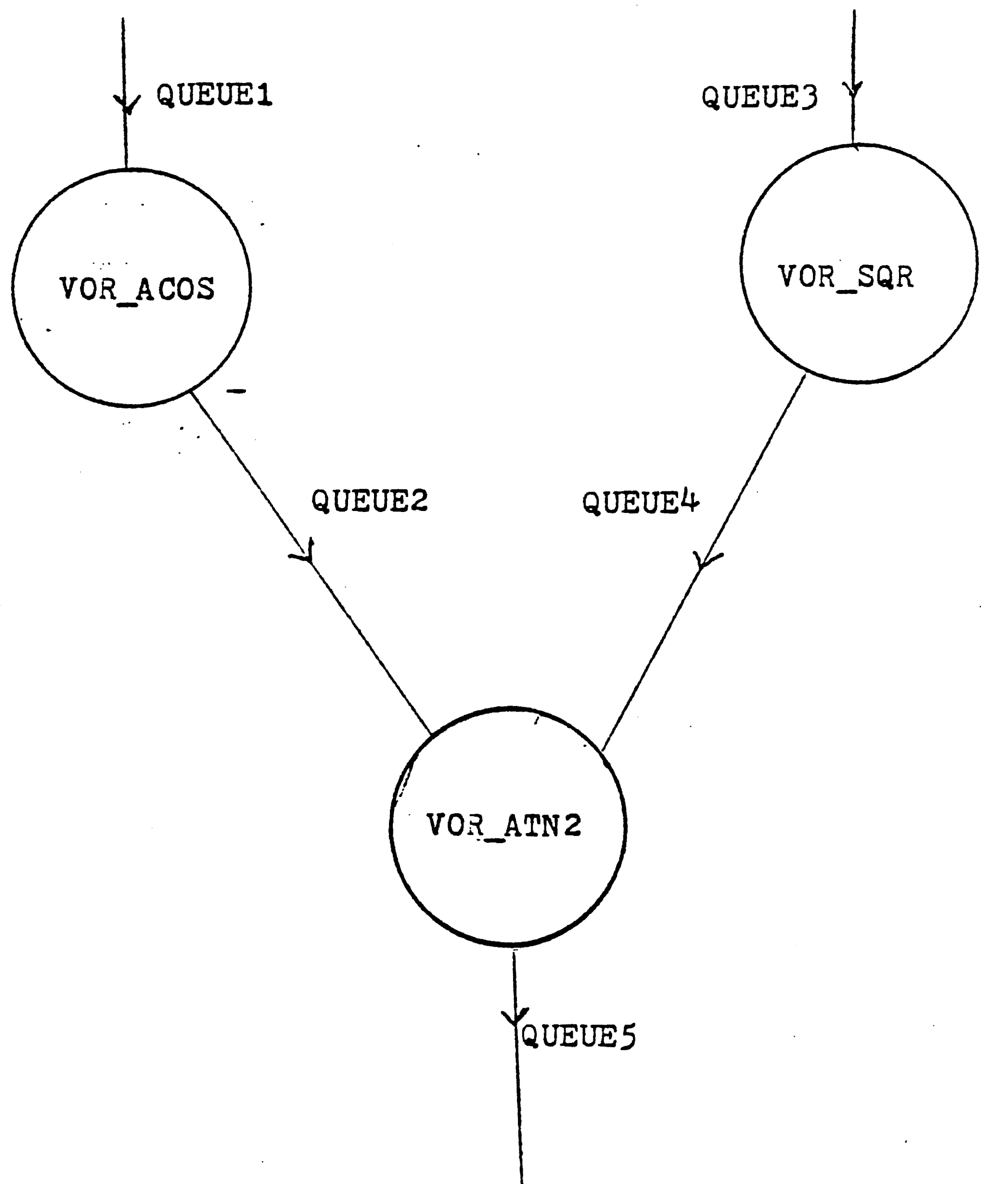


Figure 11. Test Case Topology.

APPENDIX B

ARITHMETIC PROCESSOR INSTRUCTIONS

MNEMONIC	PRIMITIVE
VOR_ACOS	Vector Arc-cosine
VOR_ASIN	Vector Arc-sine
VOR_ATAN	Vector Arc-tangent
VOR_ATN2	Vector Arc-tan(Y,X)
VOR_CMP1	Vector One's Complement
VOR_COS	Vector Cosine
VOR_EXP	Vector Exponential
VOR_INDX	Vector Indexing
VOR_LOG	Vector Logarithm
VOR_MAG	Real Vector Magnitude
VOR_MOD	Vector Modulus
VOR_NEG	Vector Negate
VOR_SHF	Vector Arithmetic Shift by S Bits
VOR_SIN	Vector Sine
VOR_SQR	Vector Square
VOR_SQRT	Vector Square Root
VOC_ARG	Vector Argument
VOC_ARGB	Vector Argument (B)
VOC_CONJ	Complex Vector Conjugate
VOC_EXP	Complex Vector Exponential
VOC_LOG	Complex Vector Logarithm
VOC_MAG	Complex Vector Magnitude
VOC_POL	Rectangular to Polar Conversion
VOC_RECT	Polar to Rectangular Conversion
VOL_AND	Vector Logical 'AND' Mask
VOL_LSHF	Vector Logical Shift by S Bits
VOL_OR	Vector Logical 'OR'
VOL_XOR	Vector Logical 'Exclusive OR'
VRR_INP	Real Vector Inner Product
VRR_SADD	Vector-Scalar Add
VRR_SDIV	Vector-Scalar Divide
VRR_SDIVB	Vector-Scalar Divide (B)
VRR_SMUL	Vector-Scalar Multiply
VRR_VADD	Real Vector-Vector Add
VRR_VDIV	Real Vector-Vector Divide
VRR_VMUL	Real Vector-Vector Multiply
VRR_VMULB	Real Vector-Vector Multiply (B)
VRR_VSUB	Real Vector-Vector Subtract
VRC_ADD	Real-Complex Vector Add
VRC_DIV	Real-Complex Vector Divide
VRC_MUL	Real-Complex Vector Multiply
VCC_INP	Complex Vector Inner Product
VCC_SADD	Complex Vector-Scalar Add
VCC_SDIV	Complex Vector-Scalar Divide
VCC_SMUL	Complex Vector-Scalar Multiply
VCC_VADD	Complex Vector-Vector Add
VCC_VDIV	Complex Vector-Vector Divide
VCC_VMUL	Complex Vector-Vector Multiply
VCC_VSUB	Complex Vector-Vector Subtract
VLL_AND	Vector Logical 'AND'
VLL_OR	Vector Logical 'OR'

VLL_XOR	Vector Logical 'XOR'
MOR_3X3I	Real Matrix Inverse (3X3)
MOR_TPSE	Real Matrix Transpose
MOR_TRCE	Real Matrix Trace
MOC-3X3I	Complex Matrix Inverse (3X3)
MOC_TPSE	Complex Matrix Transpose
MOC_TRCE	Complex Matrix Trace
MRR_MUL	Real Matrix Multiply
MCC_MUL	Complex Matrix Multiply
VCM_CTHS	Vector Compare and Threshold
VCM_MMX	Vector Maximum/Minimum
VCM_SORT	Vector Straight Selection Sort
VCM_THRS	Vector Threshold
VCM_WDWC	Vector Window Containment
DMC_BF1FX	BFP to Fixed Conversion (Fixed Blocks)
DMC_CTOR	Complex to Real Conversion
DMC_FAFX	Fixed Array to Fixed Mode Conversion
DMC_FXBF	Fixed to Block Floating Point Conversion
DMC_FXFA	Fixed to Fixed Array Conversion
DMC_RTOC	Real to Complex Conversion
DFC_CCAT	Vector Concatenate
DFC_CCATB	Block Floating Point Concatenate (B)
DFC_CNRMB	Complex Block Normalization (B)
DFC_CREPB	Complex Replicate (B)
DFC_SEP	Complex Separate
DFC_SEPB	Complex Separate (B)
DFC_DSD	Data Scaling and Display
DFC_DYNR	Dynamic Range Check
DFC_MTC	Multiply T C
DFC_MTCB	Multiply T C (B)
DFC_MTR	Multiply T R
DFC_MTRB	Multiply T R (B)
DFC_PACK	Data Bit Pack
DFC_RCAT	Vector Concatenate
DFC_RDMUX	Vector Demultiplex
DFC_REQ	Requantization
DFC_RMUX	Vector Multiplex
DFC_RNRMB	Real Block Normalization
DFC_RREP	Real Replicate
DFC_RSEP	Real Separate
DFC_SRP	Selectable Replicate
DFC_SRPB	Selectable Replicate (B)
DFC_UPAK	Data Bit Unpack
DCP_AVG1	Linear Averaging Filter, Single Average
DCP_AVGN	Linear Averaging Filer, Multiple Averages
DCP_CPWR	Power (Complex Input)
DCP_CSMG	Complex Spectral Magnitude
DCP_CSMGB	Complex Spectral Magnitude (B)
DCP_CYTG	Cycle Trigger
DCP_EAV1	Exponential Averaging Filter, Single Average

DCP_EAVN	Exponential Averaging Filter, Multiple Averages
DCP_FRQW	Frequency Weighting
DCP_FRQWB	Frequency Weighting (B)
DCP_LINT	Linear Interpolation
DCP_LMR	Local Mean Removal
DCP_NME	Noise Mean Estimation
DCP_QINT	Quadratic Interpolation
DCP_RINT	Running Integration
DCP_STI	Short Term Integration
DCP_ZFIL	Zero Fill
DCP_ZFILB	Zero Fill (B)
DCP_ZFILC	Zero Fill Complex
DCP_ZFILCB	Zero Fill Complex (B)
FFT_CBCB	Complex Block to Complex Block FFT
FFT_CBRB	Complex Block to Real Block FFT
FFT_CCB	Complex to Complex Block FFT
FFT_CRB	Complex to Real Block FFT
FFT_R2CB	Two Real to Complex Block FFT
FFT_RB2CB	Two Real Block to Complex Block FFT
FFT_RBCB	Real to Complex Block FFT
FFT_RCB	Real to Complex Block FFT
IIR_C10	Infinite Impulse Filter- Complex, 1 Pole
IIR_C11	Infinite Impulse Filter- Complex, 1 Pole, 1 Zero
IIR_C21	Infinite Impulse Filter- Complex, 2 Poles, 1 Zero
IIR_C22	Infinite Impulse Filter- Complex, 2 Poles, 2 Zeroes
IIR_R10	Infinite Impulse Filter- Real, 1 Pole
IIR_R11	Infinite Impulse Filter- Real, 1 Pole, 1 Zero
IIR_R21	Infinite Impulse Filter- Real, 2 Poles, 1 Zero
IIR_R22	Infinite Impulse Filter- Real, 2 Poles, 2 Zeroes
FIR_C1S	Finite Impulse Response Filter(Complex, One Stage)
FIR_C2S	Finite Impulse Response Filter(Complex, Two Stage)
FIR_CNS	Finite Impulse Response Filter(Complex, N Stage)
FIR_R1S	Finite Impulse Response Filter(Real, One Stage)
FIR_R2S	Finite Impulse Response Filter(Real, Two Stage)
FIR_RNS	Finite Impulse Response Filter(Real, N Stage)

CDM_CFF	Complex Demodulation (Complex, Fixed Freq.)
CDM_CVF	Complex Demodulation (Complex, Variable Freq.)
CDM_RFF	Complex Demodulation (Real, Fixed Freq.)
CDM_RFIR	Complex Demodulation and FIR Filter (Real, Fixed Freq.)
CDM_RVF	Complex Demodulation (Real, Variable Freq.)
BFR_FREQ	Frequency Domain Beamformer
BFR_FREQB	Frequency Domain Beamform (B)
BFR_GEA	Prester and Adaptive Beamform
BFR_GEAB	Prester and Adaptive Beamform
BFR_TIME	Time Domain Beamform
SSP_ABTK	A-B Tracker
SSP_BEPC	Bearing Estimate Pre-process Complex
SSP_BEPCB	Bearing Estimate Pre-process Complex (B)
SSP_BEPR	Bearing Estimate Pre-process Real
SSP_BGS	Bearing Smooth
SSP_BIN	Bin Detection
SSP_CONV	Conversion
SSP_DCD	Difar Coherent Detection
SSP_DNS	Difar Null Steer
SSP_DOP	Doppler Compensate
SSP_FROD	Frequency Determination
SSP_INDX	Peak Index
SSP_LPP	Linear Peak Pick
SSP_ONOF	On/Off Switch
SSP_PKDT	Peak Detect
SSP_PKPK	Peak Pick
SSP_PUL1	Pulse Detection
SSP_SELF	Self Noise Removal
SSP_SPKE	Spoke Suppress
SSP_STAT	Period Statistics
SSP_ZCRS	Zero Crossing Detect

APPENDIX C

CONFIGURATION FOR TEST CASE

EMSP CONFIGURATION FOR SIMULATION

FEID	TYPE	CONCENTRATOR			DISTRIBUTOR		
		DTN	CON	ELEMENT	DTN	DIS	ELEMENT
1	SCH	0	5	0	1	3	3
2	GM	1	6	1	0	6	2
3	AP	0	7	2	1	5	1
4	IOP	1	3	3	0	4	0
5	GM	0	15	3	1	7	2

queueid	head_node	tail_node	threshold	consume	read	size
1	1	1	5	5	5	1
2	3	1	5	5	5	1
3	2	2	10	10	10	1
4	3	2	9	9	9	1
5	3	3	2	2	2	1

queueid	GM	type	data_items	produce	capacity
1	2	2	0	5	100
2	2	0	0	5	30
3	5	2	0	10	20
4	2	0	0	10	40
5	2	1	0	2	15

nodeid	opcode	num_inputs	NOC	GM	firings	exec_time
1	14	1	1	2	0	285
2	28	1	1	5	0	1000
3	25	2	2	2	0	600

```
nodeid     type   size   sm value
     1     GV       1     2     5  /* INPUT  */
     1     QUEUE    5     2     0  /* INPUT  */
     1     QUEUE    5     2     0  /* OUTPUT */
nodeid     type   size   sm value
     2     GV       1     5    10 /* INPUT  */
     2     QUEUE   10     5     0 /* INPUT  */
     2     QUEUE   10     2     0 /* OUTPUT */
nodeid     type   size   sm value
     3     GV       1     2     2 /* INPUT  */
     3     QUEUE    2     2     0 /* INPUT  */
     3     QUEUE    2     2     0 /* INPUT  */
     3     QUEUE    2     2     0 /* OUTPUT */
```

APPENDIX D

TOPOLOGY INPUT

```
INPUT the node identification number    /* NODE 1 */
and answer all queries about the node
Input -1 to quit when asked the node id          * 1*
INPUT opcode mnemonic for AP instruction  *VOR_SQR*
GIP -4, GV -3, OR QUEUE -5                       *-3*
INPUT Graph Variable Identification Number       * 1*
GIP -4, GV -3, OR QUEUE -5                       *-5*
INPUT queue id.  Answer all questions
about the queue.    /*INPUT:  QUEUE 1*/          * 1*
INPUT threshold.    /* Dependent on GV value*/ *-1*
INPUT consume amount.                            *-1*
INPUT read amount.                               *-1*
GV -3 or QUEUE -5  /*OUTPUT:  QUEUE 2*/          *-5*
INPUT queue id.  Answer all questions
about the queue.                                 * 2*
INPUT valve amount for output queue.             *-1*
                                        /* NODE 2 */
INPUT the node id and answer all queries about the node.
Input -1 to quit when asked the node id          * 2*
INPUT opcode mnemonic for AP instruction *VOR_ACOS*
GIP -4, GV -3, OR QUEUE -5                       *-3*
INPUT Graph Variable Identification Number       * 2*
GIP -4, GV -3, OR QUEUE -5                       *-5*
INPUT queue id.  Answer all questions
about the queue.  /*INPUT: QUEUE 3*/             * 3*
INPUT threshold.  /* Dependent of GV value*/     *-1*
INPUT consume amount.                            *-1*
INPUT read amount.                               *-1*
GV -3 or QUEUE -5 /*OUTPUT:  QUEUE 4*/           *-5*
INPUT queue id.  Answer all questions
about the queue.                                 * 4*
INPUT valve amount for output queue.             *-1*
                                        /* NODE 3 */
INPUT the node id and answer all queries about thenode.
Input -1 to quit when asked the node id          * 3*
INPUT opcode mnemonic for AP instruction *VOR_ATN2*
GIP -4, GV -3, OR QUEUE -5                       *-3*
INPUT Graph Variable Identification Number       * 3*
GIP -4, GV -3, OR QUEUE -5 /*INPUT:  QUEUE 2*/ *-5*
INPUT queue id.  Answer all questions
about the queue.                                 * 2*
INPUT threshold. /* Threshold constant value*/ * 5*
INPUT consume amount.                            * 5*
INPUT read amount.                               * 5*
GIP -4, GV -3, OR QUEUE -5 /*INPUT: QUEUE 4*/  *-5*
INPUT queue id.  Answer all questions
about the queue.                                 * 4*
INPUT threshold.                                 * 9*
INPUT consume amount.                            * 9*
INPUT read amount.                               * 9*
GV -3 or QUEUE -5 /*OUTPUT: QUEUE 5*/            *-5*
INPUT queue id.  Answer all questions
about the queue.                                 * 5*
INPUT valve amount for output queue.             *-1*
```

```
                      /* NO MORE NODES SO INPUT -1*/
INPUT the node id and answer all queries about thenode.
Input -1 to quit when asked the node id          *-1*
```

APPENDIX E

CONFIGURATION INPUT

```
            /* INPUT SYSTEM CONFIGURATION */
HOW MANY DTN'S, DATA TRANSFER NETWORKS, 1 or 2    * 2*
INPUT switch size for DTN[0]                      *16*
INPUT switch size for DTN[1]                      * 8*
            /* Scheduler with FEID of 1 */
Input Functional Element ID for each element.
 INPUT -1 TO QUIT.........                        * 1*
INPUT type of Functional element, AP = 0
, CPP = 4, GM = 1, IOP = 3, SCH = 2               * 2*
INPUT which DTN concentrator is on, 0 or 1        * 0*
INPUT which concentrator on DTN                   * 5*
INPUT which element on concentrator               * 0*
INPUT which DTN distributor is on, 0 or 1         * 1*
INPUT which distributor on DTN                    * 3*
INPUT which element on distributor                * 3*
           /* GLOBAL MEMORY with FEID 2 */
Input Functional Element ID for each element.
 INPUT -1 TO QUIT.........                        * 2*
INPUT type of Functional element, AP = 0
, CPP = 4, GM = 1, IOP = 3, SCH = 2               * 1*
INPUT which DTN concentrator is on, 0 or 1        * 1*
INPUT which concentrator on DTN                   * 6*
INPUT which element on concentrator               * 1*
INPUT which DTN distributor is on, 0 or 1         * 0*
INPUT which distributor on DTN                    * 6*
INPUT which element on distributor                * 2*
           /* ARITHMETIC PROCESSOR with FEID 3*/
Input Functional Element ID for each element.
 INPUT -1 TO QUIT.........                        * 3*
INPUT type of Functional element, AP = 0
, CPP = 4, GM = 1, IOP = 3, SCH = 2               * 0*
INPUT which DTN concentrator is on, 0 or 1        * 0*
INPUT which concentrator on DTN                   * 7*
INPUT which element on concentrator               * 2*
INPUT which DTN distributor is on, 0 or 1         * 1*
INPUT which distributor on DTN                    * 5*
INPUT which element on distributor                * 1*
          /* INPUT/OUTPUT PROCESSOR with FEID 4*/
Input Functional Element ID for each element.
 INPUT -1 TO QUIT.........                        * 4*
INPUT type of Functional element, AP = 0
, CPP = 4, GM = 1, IOP = 3, SCH = 2               * 3*
INPUT which DTN concentrator is on, 0 or 1        * 1*
INPUT which concentrator on DTN                   * 3*
INPUT which element on concentrator               * 3*
INPUT which DTN distributor is on, 0 or 1         * 0*
INPUT which distributor on DTN                    * 4*
INPUT which element on distributor                * 0*
        /* GLOBAL MEMORY with FEID 5 */
Input Functional Element ID for each element.
 INPUT -1 TO QUIT.........                        * 5*
INPUT type of Functional element, AP = 0
, CPP = 4, GM = 1, IOP = 3, SCH = 2               * 1*
INPUT which DTN concentrator is on, 0 or 1        * 0*
```

```
INPUT which concentrator on DTN                          *15*
INPUT which element on concentrator                      * 3*
INPUT which DTN distributor is on, 0 or 1                * 1*
INPUT which distributor on DTN                           * 7*
INPUT which element on distributor                       * 2*
        /* NO MORE FUNCTIONAL ELEMENTS SO -1*/
Input Functional Element ID for each element.
 INPUT -1 TO QUIT.........                               *-1*

        /* INPUT CHANNEL INFORMATION */
        /* CHANNEL 1 ATTACHED TO QUEUE 1*/
INPUT channel id.  Input -1 to quit.  Answer all questions
about the channel.                                       * 1*
INPUT priority of channel.                               * 1*
INPUT channel rate of input                          * 50000*
INPUT id of queue channel is attached.                   * 1*
INPUT id of FEID of IOP, Functional element id
 of Input Output processor.                              * 4*
INPUT 2 or OUTPUT 1 Channel                              * 2*
        /* CHANNEL 2 ATTACHED TO QUEUE 3*/
INPUT channel id.  Input -1 to quit.  Answer all
questions about the channel.                             * 2*
INPUT priority of channel.                               * 2*
INPUT channel rate of input                         * 100000*
INPUT id of queue channel is attached.                   * 3*
INPUT id of FEID of IOP, Functional element id
 of Input Output processor.                              * 4*
INPUT 2 or OUTPUT 1 Channel                              * 2*
        /* CHANNEL 3 ATTACHED TO QUEUE 5*/
INPUT channel id.  Input -1 to quit.  Answer all
questions about the channel.                             * 3*
INPUT priority of channel.                               * 3*
INPUT channel rate of input                         * 300000*
INPUT id of queue channel is attached.                   * 5*
INPUT id of FEID of IOP, Functional element id           * 4*
 of Input Output processor.
INPUT 2 or OUTPUT 1 Channel                              * 1*
        /* NO MORE CHANNELS SO INPUT -1*/
INPUT channel id.  Input -1 to quit.  Answer all
questions about the channel.                             * -1*

        /* INPUT NODE INFORMATION THAT IS DYNAMIC*/
                        /* NODE 1*/
NOTE:  Most input queues and graph variables are
located in same Global Memory as Node InstructionStream

INPUT GM of GRAPH VARIABLE                               * 2*
INPUT value of GV                                        * 5*
INPUT GM of QUEUE                                        * 2*
INPUT capacity.                                         *100*
                        /* NODE 2*/
NOTE:  Most input queues and graph variables are
located in same Global Memory as Node InstructionStream
```

```
INPUT GM of GRAPH VARIABLE                        * 5*
INPUT value of GV                                 *10*
INPUT GM of QUEUE                                 * 5*
INPUT capacity.                                   *20*
            /* NODE 3*/
NOTE:  Most input queues and graph variables are
located in same Global Memory as Node InstructionStream

INPUT GM of GRAPH VARIABLE                        * 2*
INPUT value of GV                                 * 2*
INPUT GM of QUEUE                                 * 2*
INPUT capacity.                                   *30*
INPUT GM of QUEUE                                 * 2*
INPUT capacity.                                   *40*
 /* Since output queue 5 is attached to a channel */
 /* extra information must be gathered about the  */
 /* queue at this time, i.e. the capacity and     */
 /* threshold.                                    */
INPUT GM of QUEUE                                 * 2*
INPUT capacity.                                   *15*
INPUT threshold.                                  * 2*
```

APPENDIX F

SIMULATION OUTPUT

TIMING SIMULATOR FOR THE ENHANCED MODULAR SIGNAL PROCESSOR

EMSP CONFIGURATION FOR SIMULATION

FEID	TYPE	CONCENTRATOR			DISTRIBUTOR		
		DTN	CON	ELEMENT	DTN	DIS	ELEMENT
1	SCH	0	5	0	1	3	3
2	GM	1	6	1	0	6	2
3	AP	0	7	2	1	5	1
4	IOP	1	3	3	0	4	0
5	GM	0	15	3	1	7	2

FUNCTIONAL ELEMENT UTILIZATION

FEID	TYPE	UTILIZATION
1	SCH	43
2	GM	40
3	AP	30
4	IOP	2
5	GM	14

TOTAL TIME = 1000.

NODE EXECUTION INFORMATION

NODE ID	OPCODE	NODE FIRINGS
1	14	2
2	28	2
3	25	1

CHANNEL EXECUTION INFORMATION

CHANNEL ID	CHANNEL FIRINGS
1	10

2	3
3	1

QUEUE EXECUTION INFORMATION

QUEUE ID	DATA ITEMS	HEAD NODE	TAIL NODE
1	35	1	1
2	5	3	1
3	10	2	2
4	11	3	2
5	0	3	3

APPENDIX G

TIMING DIAGRAM SIMULATION OUTPUT

TIMING SIMULATOR FOR THE
ENHANCED MODULAR SIGNAL PROCESSOR

EMSP CONFIGURATION FOR SIMULATION

FEID	TYPE	CONCENTRATOR			DISTRIBUTOR		
		DTN	CON	ELEMENT	DTN	DIS	ELEMENT
1	SCH	0	5	0	1	3	3
2	GM	1	6	1	0	6	2
3	AP	0	7	2	1	5	1
4	IOP	1	3	3	0	4	0
5	GM	0	15	3	1	7	2

```
TIMING CHART FOR EMSP GRAPH

  TIME   1   2   3   4   5
     0
     .
     .
     .
    99
/* TWO GRAPH PROCESS INSTRUCTS FIRE AT SAME TIME ONE IS */
/* RESCHEDULED FOR TIME 103.  INSTRUCT 30 IS EXN SO     */
/* A CHANNEL HAS GONE OVER THRESHOLD AND FIRED.         */
INSTRUCT opcode 30, time 10000, receiver 4, sender 4, node 1
CALLING IOPP
INSTRUCT opcode 30, time 10000, receiver 4, sender 4, node 2
CALLING IOPP
   100               4
   101               4
   102               4
/* RESCHEDULED INSTRUCT EXN IS NOW EXECUTED AS IOP IS FREE*/
INSTRUCT opcode 30, time 10300, receiver 4, sender 4, node 2
CALLING IOPP
   103               4
   104               4
   105               4
/* EXN INSTRUCT TRIGGERED A WRITE QUEUE TO QUEUE 1 */
INSTRUCT opcode 64, time 10584, receiver 2, sender 4, node 1
CALLING GM
   106       2
   107       2
   108       2
/* EXN INSTUCTION TRIGGERED A WRITE QUEUE TO QUEUE 3 */
INSTRUCT opcode 64, time 10892, receiver 5, sender 4, node 2
CALLING GM
   109       2           5
   110       2           5
   111       2           5
   112       2           5
```

```
   113       2          5
   114       2          5
   115       2          5
   116       2          5
   117       2          5
   118       2          5
   119                  5
   120                  5
   121                  5
   122                  5
   123
   124
   125
   126
   127
   128
   129
   130
/* QUEUE 1 HAS GONE OVER THRESHOLD. WRITE QUEUE INSTRUCT */
/* TRIGGERED A QUEUE OVER THRESHOLD INSTRUCT.            */
INSTRUCT opcode 42, time 13040, receiver 1, sender 2, node 1
CALLING SCH
   131   1
   132   1
   133   1
   134   1
   135   1
/* QUEUE 3 HAS GONE OVER THRESHOLD.  WRITE QUEUE INSTRUCT*/
/* TRIGGERED A QUEUE OVER THRESHOLD INSTRUCT BUT         */
/* SCHEDULER IS BUSY SO RESCHEDULE>                      */
INSTRUCT opcode 42, time 13536, receiver 1, sender 5, node 2
CALLING SCH
   136   1
   137   1
   138   1
   139   1
   140   1
   141   1
   142   1
   143   1
   144   1
   145   1
   146   1
   147   1
   148   1
   149   1
   150   1
/* RESCHEDULED QUEUE OVER THRESHOLD INSTRUCT FOR QUEUE 3*/
INSTRUCT opcode 42, time 15040, receiver 1, sender 5, node 2
CALLING SCH
   151   1
   152   1
   153   1
   154   1
```

```
   155   1
   156   1
   157   1
/* SEND INSTRUCT STREAM INSTRUCTION TRIGGERED BY QUEUE 1*/
/* GOING OVER THRESHOLD.  NODE 1 IS FIRING.            */
INSTRUCT opcode 53, time 15747, receiver 2, sender 1, node 1
CALLING GM
   158   1   2
   159   1   2
   160   1   2
   161   1   2
   162   1   2
   163   1   2
   164   1   2
   165   1   2
   166   1   2
   167   1   2
   168   1   2
   169   1   2
   170   1   2
   171
   172
   173
   174
/* ACCEPT INSTRUCT STREAM SENT BY GLOBAL MEMEORY TO    */
/* ARITHMETIC PROCESSOR.                              */
INSTRUCT opcode 3, time 17430, receiver 3, sender 2, node 1
CALLING AP
   175           3
   176           3
   177           3
   178           3
   179           3
   180
   181
   182
   183
   184
   185
   186
   187
   188
   189
   190
   191
   192
   193
   194
   195
   196
   197
   198
   199
/* REQUEST GRAPH VARIABLE FROM GLOBAL MEMORY FOR NODE 1  */
```

```
/* SENT BY ARITMETIC PROCESSOR.                                  */
INSTRUCT opcode 47, time 19918, receiver 2, sender 3, node 1
CALLING GM
/* CHANNEL 1 HAS FIRED AGAIN AND CHANNEL 2.  CHANNEL 2   */
/* BLOCKED AGAIN AND RESCHEDULED.  OPCODE 30 IS EXN.      */
INSTRUCT opcode 30, time 20000, receiver 4, sender 4, node 1
CALLING IOPP
INSTRUCT opcode 30, time 20000, receiver 4, sender 4, node 2
CALLING IOPP
   200        2       4
   201        2       4
   202        2       4
   /* RESCHEDULED CHANNEL 2 EXN or EXECUTE NODE INSTRUCT */
INSTRUCT opcode 30, time 20300, receiver 4, sender 4, node 2
CALLING IOPP
   203        2       4
   204        2       4
/* WRITE QUEUE TRIGGERED ON QUEUE 1 BY CHANNEL FIRING.   */
/* MUST BE RESCHEDULED BECAUSE OF PREVIOUS REQUEST GV.   */
INSTRUCT opcode 64, time 20482, receiver 2, sender 4, node 1
CALLING GM
   205        2       4
   206        2
   207        2
   208        2
   209        2
/* WRITE QUEUE TRIGGERED ON QUEUE 3 BY CHANNEL FIRING.   */
INSTRUCT opcode 64, time 20902, receiver 5, sender 4, node 2
CALLING GM
   210        2           5
   211        2           5
/* RESCHEDULED WRITE QUEUE ON QUEUE 1                    */
INSTRUCT opcode 64, time 21129, receiver 2, sender 4, node 1
CALLING GM
   212        2           5
   213        2           5
/* GLOBAL MEMORY SENDING ACCEPT GRAPH VARIABLE INSTRUCT */
/* TO ARITHMETIC PROCESSOR FOR NODE 1.                  */
INSTRUCT opcode 2, time 21392, receiver 3, sender 2, node 1
CALLING AP
   214        2   3       5
   215        2   3       5
   216        2   3       5
   217        2   3       5
   218        2   3       5
   219        2   3       5
   220        2   3       5
   221        2   3       5
   222        2   3       5
/* REQUEST QUEUE SENT TO GLOBAL MEMEORY FROM ARITHMETIC   */
/* PROCESSOR FOR NODE 1.  RESCHEDULED.                    */
INSTRUCT opcode 50, time 22269, receiver 2, sender 3, node 1
CALLING GM
   223        2   3
```

```
/* RESCHEDULE REQUEST QUEUE.                                   */
INSTRUCT opcode 50, time 22384, receiver 2, sender 3, node 1
CALLING GM
   224         2   3
   225         2   3
   226         2   3
   227         2   3
/* QUEUE OVER THRESHOLD ONE QUEUE 3 IN RESPONSE TO WRITE */
/* QUEUE BECAUSE OF CHANNEL FIRING.                            */
INSTRUCT opcode 42, time 22751, receiver 1, sender 5, node 2
CALLING SCH
   228    1    2   3
   229    1    2   3
   230    1    2   3
   231    1    2   3
   232    1    2   3
   233    1    2   3
   234    1    2   3
   235    1    2
   236    1    2
   237    1
   238
   239
/* GLOBAL MEMORY SENDING ACCEPT QUEUE INSTRUCT TO     */
/* ARITHMETIC PROCESSSOR.                             */
INSTRUCT opcode 4, time 23926, receiver 3, sender 2, node 1
CALLING AP
   240             3
   241             3
   242             3
   243             3
   244             3
   245             3
   246             3
   247             3
   248             3
   249             3
/* QUEUE OVER THRESHOLD.  QUEUE 1 OVER BECAUSE OF      */
/* WRITE QUEUE                                         */
INSTRUCT opcode 42, time 24955, receiver 1, sender 2, node 1
CALLING SCH
   250    1        3
   251    1        3
   252    1        3
   253    1        3
   254    1        3
   255    1        3
   256    1        3
   257    1        3
   258    1        3
   259    1        3
   260
   261
   262
   263
```

```
   264
   265
   266
   267
   268
/* NODE 1 HAS COMPLETED EXECUTION AND IS WRITING QUEUE 2*/
/* NOTE ALL WRITE QUEUES EXCEPT TO OUTPUT QUEUES GIVE   */
/* THE NODE IDENTIFICATION NUMBER OF THE HEAD NODE TO   */
/* THE QUEUE.                                           */
INSTRUCT opcode 64, time 26824, receiver 2, sender 3, node 3
CALLING GM
   269       2
   270       2
   271       2
   272       2
/* NODE 1 HAS COMPLETED SO ARITHMETIC PROCESSOR IS READY*/
/* FOR NEXT INSTRUCT SO READY FOR INSTRUCTION STREAM */
INSTRUCT opcode 45, time 27215, receiver 1, sender 3, node 1
CALLING SCH
   273    1    2
   274    1    2
   275    1    2
   276    1    2
   277    1    2
   278    1    2
   279    1    2
   280    1    2
   281    1
   282    1
   283    1
   284    1
   285    1
   286    1
   287    1
   288    1
   289    1
   290
   291
   292
   293
   294
/* UPON RECEPTION OF THE READY FOR INSTRUCT STREAM, NODE */
/* 2 WHICH WAS WAITING ON THE READY LIST IN THE SCHEDULER*/
/* IS SCHEDULED WITH A SEND INSTRUCT STREAM.             */
INSTRUCT opcode 53, time 29454, receiver 5, sender 1, node 2
CALLING GM
   295                  5
   296                  5
   297                  5
   298                  5
   299                  5
/* QUEUE 2 HAS GONE AVER THRESHOLD WITH THE WRITE QUEUE  */
/* TRIGGERED BY THE EXECUTION OF NODE 1.                 */
INSTRUCT opcode 42, time 29922, receiver 1, sender 2, node 3
```

```
CALLING SCH
/* CHANNEL 1 AND 2 HAVE FIRED AGAIN.  CONFLICT AGAIN AND  */
/* RESCHEDULED.                                           */
INSTRUCT opcode 30, time 30000, receiver 4, sender 4, node 1
CALLING IOPP
INSTRUCT opcode 30, time 30000, receiver 4, sender 4, node 2
CALLING IOPP
   300   1         4   5
   301   1         4   5
   302   1         4   5
INSTRUCT opcode 30, time 30300, receiver 4, sender 4, node 2
CALLING IOPP
   303   1         4   5
   304   1         4   5
/* NODE 1's FIRING HAS TRIGGERED A CONSUME QUEUE INSTRUCT*/
/* ONE QUEUE 1 ATTACHED TO NODE 1.                       */
INSTRUCT opcode 72, time 30432, receiver 2, sender 3, node 1
CALLING GM
/***** NOTE THE PARALLELISM ON THE FUNCTIONAL ELEMTENTS*****/
   305   1   2     4   5
/* WRITE QUEUE FOR QUEUE 1 TRIGGERED BY CHANNEL.RESCHEDULED*/
INSTRUCT opcode 64, time 30506, receiver 2, sender 4, node 1
CALLING GM
   306   1   2         5
   307   1   2         5
   308   1   2
   309   1   2
/* WRITE QUEUE FOR QUEUE 3 TRIGGERED BY CHANNEL 2.     */
INSTRUCT opcode 64, time 30926, receiver 5, sender 4, node 2
CALLING GM
   310       2         5
   311       2         5
   312       2         5
/* GLOBAL MEMORY SENDING ACCEPT INSTRUCT STREAM TO     */
/* ARITHMETIC PROCESSOR FOR NODE 2.                      */
INSTRUCT opcode 3, time 31206, receiver 3, sender 5, node 2
CALLING AP
   313       2   3     5
   314       2   3     5
   315       2   3     5
   316       2   3     5
/* REPORT NODE DONE WAS ADDED TO NOTIFY THE SCHEDULER A   */
/* NODE HAD FINISHED.  EMSP DOES NOT ALLOW TWO INSTANCES  */
/* OF THE SAME NODE.  OPCODE IS ONLY FOR SIMULATOR AND NOT*/
/* A INSTRUCT ON EMSP SO IT TAKES NO TIME.             */
INSTRUCT opcode -4, time 31632, receiver 1, sender 2, node 1
CALLING SCH
/* RESCHEDULED WRITE QUEUE FOR NODE 1                     */
INSTRUCT opcode 64, time 31632, receiver 2, sender 4, node 1
CALLING GM
   317       2   3     5
   318       2         5
   319       2         5
   320       2         5
```

```
   321       2          5
   322       2          5
   323       2
   324       2
   325       2
   326       2
/* QUEUE OVER THRESHOLD FOR QUEUE 1                            */
INSTRUCT opcode 42, time 32664, receiver 1, sender 2, node 1
CALLING SCH
   327    1   2
   328    1   2
   329    1
   330    1
   331    1
   332    1
/* REQUEST GRAPH VARIABLE.  NODE 2.                           */
INSTRUCT opcode 47, time 33237, receiver 5, sender 3, node 2
CALLING GM
   333    1             5
   334    1             5
   335    1             5
   336    1             5
   337    1             5
/* QUEUE 3 HAS GONE OVER CAPACITY.  GLOBAL MEMORY SENT */
/* QUEUE OVER CAPACITY MESSAGE TO SCHEDULER.            */
/* RESCHEDULED.                                         */
INSTRUCT opcode 41, time 33719, receiver 1, sender 5, node 2
CALLING SCH
   338    1             5
   339    1             5
   340    1             5
   341    1             5
   342    1             5
   343    1             5
   344    1             5
   345    1
   346    1
/* RESCHEDULED QUEUE OVER CAPACITY INSTRUCT.          */
INSTRUCT opcode 41, time 34664, receiver 1, sender 5, node 2
CALLING SCH
/* ACCEPT INSTRUCT STREAM TO ARITHMETIC PROCESSOR FOR*/
/* NODE 2 FROM GLOBAL MEMORY.                          */
INSTRUCT opcode 2, time 34692, receiver 3, sender 5, node 2
CALLING AP
   347    1       3
   348    1       3
   349    1       3
   350    1       3
   351    1       3
   352    1       3
   353    1       3
   354    1       3
   355    1       3
   356    1       3
```

```
   357   1        3
   358   1        3
   359   1        3
/* QUEUE OVER THRESHOLD INSTRUCT FOR NODE 1.          */
/* RESCHEDULED.                                          */
INSTRUCT opcode 42, time 35923, receiver 1, sender 2, node 1
CALLING SCH
   360   1        3
   361   1        3
/* RESCHEDULED QUEUE OVER THRESHOLD.                    */
INSTRUCT opcode 42, time 36164, receiver 1, sender 2, node 1
CALLING SCH
   362   1        3
   363   1        3
   364   1        3
/* REQUEST QUEUE FOR NODE 2 FOR QUEUE 3.                */
INSTRUCT opcode 50, time 36496, receiver 5, sender 3, node 2
CALLING GM
   365   1        3       5
   366   1        3       5
   367   1        3       5
   368   1                5
   369   1                5
/*  QUEUE OVER THRESHOLD FOR QUEUE 3.  RESCHEDULED.    */
INSTRUCT opcode 42, time 36978, receiver 1, sender 5, node 2
CALLING SCH
   370   1                5
   371   1                5
/* RESCHEDULED QUEUE OVER THRESHOLD FOR QUEUE 3.       */
INSTRUCT opcode 42, time 37164, receiver 1, sender 5, node 2
CALLING SCH
   372   1                5
   373   1                5
   374   1                5
   375   1                5
   376   1                5
   377   1                5
   378   1                5
   379   1
   380   1
   381   1
   382
   383
   384
/* ACCEPT QUEUE FOR QUEUE 3 AND NODE 2.                 */
INSTRUCT opcode 4, time 38416, receiver 3, sender 5, node 2
CALLING AP
   385            3
   386            3
   387            3
   388            3
   389            3
   390            3
   391            3
```

```
/* QUEUE OVER CAPACITY INSTRUCT ON A CHANNEL REQUIRES*/
/* A SNDT OR STOP NODE DATA TRANSFER MESSAGE TO BE SENT */
/* TO THE INPUT/OUTPUT PROCESSOR.                       */
INSTRUCT opcode 71, time 39168, receiver 4, sender 1, node 2
CALLING IOPP
ERROR CHANNEL 2 HAS OVERRAN QUEUE 3
   392           3   4
   393           3   4
   394           3   4
   395           3
   396           3
   397           3
   398           3
   399           3
/* SAME OLD CHANNEL FIRINGS.                            */
INSTRUCT opcode 30, time 40000, receiver 4, sender 4, node 1
CALLING IOPP
INSTRUCT opcode 30, time 40000, receiver 4, sender 4, node 2
CALLING IOPP
   400           3   4
   401           3   4
   402           3   4
/* CHANNEL 2 HAS BEEN STOPPED SO NO TIME NECESSARY FOR EXN*/
INSTRUCT opcode 30, time 40300, receiver 4, sender 4, node 2
CALLING IOPP
   403           3
   404           3
   405           3
/* CHANNEL FIRING TRIGGERED A WRITE QUEUE ON QUEUE 1   */
INSTRUCT opcode 64, time 40502, receiver 2, sender 4, node 1
CALLING GM
   406       2
   407       2
   408       2
   409       2
   410       2
   411       2
   412       2
   413       2
   414       2
/* NODE 2 HAS COMPLETED SET UP MODE IN ARITHMETIC PROCESSOR*/
/* SO SENDS READY FOR INSTRUCT STREAM (RFIS) TO         */
/* SCHEDULER.                                             */
INSTRUCT opcode 45, time 41442, receiver 1, sender 3, node 2
CALLING SCH
   415   1   2
   416   1   2
   417   1   2
   418   1
   419   1
/* RFIS FROM INPUT/OUTPUT PROCESSOR WHEN CHANNEL STOPPED.  */
INSTRUCT opcode 45, time 41910, receiver 1, sender 4, node 2
CALLING SCH
   420   1
```

```
    421   1
    422   1
    423   1
    424   1
    425   1
    426   1
    427   1
    428   1
    429   1
    430   1
    431   1
/* RFIS RESCHEDULED AT TIME 420.                               */
INSTRUCT opcode 45, time 43142, receiver 1, sender 4, node 2
CALLING SCH
/* WRITE QUEUE TRIGGERED BY EXECUTION OF NODE 3.               */
INSTRUCT opcode 64, time 43162, receiver 2, sender 3, node 3
CALLING GM
    432   1   2
    433   1   2
    434   1   2
    435   1   2
    436   1   2
    437   1   2
    438   1   2
    439   1   2
    440   1   2
    441   1   2
/* SEND INSTRUCT STREAM TRIGGERED BY RFIS                   */
INSTRUCT opcode 53, time 44135, receiver 2, sender 1, node 1
CALLING GM
    442   1   2
    443   1   2
    444   1   2
INSTRUCT opcode 53, time 44472, receiver 2, sender 1, node 1
CALLING GM
    445   1   2
    446   1   2
/* QUEUE OVER THRESHOLD TRIGGERED BY CHANNEL FIRING AND        */
/* WRITE QUEUE.                                                */
INSTRUCT opcode 42, time 44603, receiver 1, sender 2, node 1
CALLING SCH
    447   1   2
    448   1   2
INSTRUCT opcode 42, time 44842, receiver 1, sender 2, node 1
CALLING SCH
    449   1   2
    450   1   2
    451   1   2
/* CONSUME QUEUE TRIGGERED BY EXECUTION OF NODE 2.          */
INSTRUCT opcode 72, time 45113, receiver 5, sender 3, node 2
CALLING GM
    452   1   2           5
    453   1   2           5
    454   1   2           5
```

```
   455   1   2           5
   456   1   2           5
   457   1   2           5
   458   1               5
   459                   5
   460                   5
   461                   5
/* ACCEPT INSTRUCT STREAM FOR NODE 1.                  */
INSTRUCT opcode 3, time 46102, receiver 3, sender 2, node 1
CALLING AP
   462           3       5
   463           3       5
/* SUPERFICIAL REPORT NODE DONE FOR NODE 2.              */
INSTRUCT opcode -4, time 46313, receiver 1, sender 5, node 2
CALLING SCH
   464           3
   465           3
   466           3
   467
   468
   469
   470
   471
   472
   473
/* QUEUE OVER THRESHOLD FOR NODE 3 AND QUEUE 2 or 4.    */
INSTRUCT opcode 42, time 47345, receiver 1, sender 2, node 3
CALLING SCH
   474   1
   475   1
   476   1
   477   1
   478   1
   479   1
/* REQUEST GRAPH VARIABLE FOR NODE 1.                  */
INSTRUCT opcode 47, time 47918, receiver 2, sender 3, node 1
CALLING GM
   480   1   2
   481   1   2
   482   1   2
   483   1   2
/* QUEUE OVER THRESHOLD FOR NODE 2 AND QUEUE 3.        */
INSTRUCT opcode 42, time 48400, receiver 1, sender 5, node 2
CALLING SCH
   484   1   2
   485   1   2
   486   1   2
   487   1   2
   488   1   2
   489   1   2
   490   1   2
   491   1   2
   492   1
   493   1
```

```
/* RESCHEDULED FROM TIME 484.                                    */
INSTRUCT opcode 42, time 49345, receiver 1, sender 5, node 2
CALLING SCH
   494   1
/* ACCEPT GRAPH VARIABLE.                                        */
INSTRUCT opcode 2, time 49476, receiver 3, sender 2, node 1
CALLING AP
   495   1        3
   496   1        3
   497   1        3
   498   1        3
   499   1        3
/* CHANNEL 1 FIRED.  EXECUTE NODE (EXN).                         */
INSTRUCT opcode 30, time 50000, receiver 4, sender 4, node 1
CALLING IOPP
   500   1        3   4
   501   1        3   4
   502   1        3   4
   503   1        3
   504   1        3
   505   1        3
/* WRITE QUEUE FOR QUEUE 1                                       */
INSTRUCT opcode 64, time 50512, receiver 2, sender 4, node 1
CALLING GM
   506   1   2    3
   507   1   2    3
/* REQUEST QUEUE.                                                */
INSTRUCT opcode 50, time 50723, receiver 2, sender 3, node 1
CALLING GM
   508   1   2    3
   509   1   2    3
   510   1   2    3
   511   1   2    3
   512   1   2    3
   513   1   2    3
   514       2    3
   515       2
   516       2
   517       2
/* RESCHEDULED FROM TIME 508.                                    */
INSTRUCT opcode 50, time 51767, receiver 2, sender 3, node 1
CALLING GM
   518       2
   519       2
   520       2
   521       2
   522       2
   523       2
   524       2
   525       2
   526       2
   527       2
   528       2
   529       2
```

```
/* QUEUE OVER THRESHOLD FOR QUEUE 1.                          */
INSTRUCT opcode 42, time 52955, receiver 1, sender 2, node 1
CALLING SCH
   530   1   2
   531   1
   532   1
   533   1
/* ACCEPT QUEUE.                                              */
INSTRUCT opcode 4, time 53368, receiver 3, sender 2, node 1
CALLING AP
   534   1       3
   535   1       3
   536   1       3
   537   1       3
   538   1       3
   539   1       3
   540           3
   541           3
   542           3
   543           3
   544           3
   545           3
   546           3
   547           3
   548           3
   549           3
   550           3
   551           3
   552           3
   553           3
   554           3
   555
   556
   557
   558
   559
   560
   561
   562
/* WRITE QUEUE TRIGGERED BY EXECUTION OF NODE 1               */
INSTRUCT opcode 64, time 56238, receiver 2, sender 3, node 3
CALLING GM
   563       2
   564       2
   565       2
   566       2
   567       2
   568       2
   569       2
   570       2
/* RFIS TRIGGERED BY COMPLETION OF THE EXECUTION OF NODE 1*/
INSTRUCT opcode 45, time 57007, receiver 1, sender 3, node 1
CALLING SCH
   571   1   2
```

```
    572   1   2
    573   1   2
    574   1   2
    575   1
    576   1
    577   1
    578   1
    579   1
    580   1
    581   1
    582   1
    583   1
    584   1
    585   1
    586   1
    587   1
    588
    589
    590
    591
    592
/* SEND INSTRUCT STREAM FOR NODE 3 TRIGGERED BY RFIS.*/
INSTRUCT opcode 53, time 59246, receiver 2, sender 1, node 3
CALLING GM
    593       2
    594       2
    595       2
    596       2
    597       2
/* QOT ON QUEUE 2 OR 4 TRIGGERED BY NODE 1 EXECUTION */
INSTRUCT opcode 42, time 59714, receiver 1, sender 2, node 3
CALLING SCH
    598   1   2
    599   1   2
/* EXN */
INSTRUCT opcode 30, time 60000, receiver 4, sender 4, node 1
CALLING IOPP
    600   1   2       4
    601   1   2       4
    602   1   2       4
/* CONSUME QUEUE 1 TRIGGERED BY NODE 1 EXECUTION */
INSTRUCT opcode 72, time 60224, receiver 2, sender 3, node 1
CALLING GM
    603   1   2
    604   1   2
    605   1   2
/* WRITE QUEUE 1 TRIGGERED BY EXN. */
INSTRUCT opcode 64, time 60508, receiver 2, sender 4, node 1
CALLING GM
/* CONSUME QUEUE WAS RESCHEDULED FROM TIME 603 */
INSTRUCT opcode 72, time 60589, receiver 2, sender 3, node 1
CALLING GM
/* WRITE QUEUE TIME RESCHEDULED AGAIN */
INSTRUCT opcode 64, time 60589, receiver 2, sender 4, node 1
```

```
CALLING GM
   606   1   2
   607   1   2
   608       2
   609       2
/* ACCEPT INSTRUCT STREAM SENT TO AP FROM GM FOR */
/* NODE 3*/
INSTRUCT opcode 3, time 60998, receiver 3, sender 2, node 3
CALLING AP
   610       2   3
   611       2   3
   612       2   3
   613       2   3
   614       2   3
   615       2   3
   616       2
   617       2
/* SUPERFICIAL REPORT NODE */
INSTRUCT opcode -4, time 61789, receiver 1, sender 2, node 1
CALLING SCH
/* WRITE QUEUE FOR EXN FINALLY NOT RESCHEDULED BUT */
/* EXECUTED.                                       */
INSTRUCT opcode 64, time 61789, receiver 2, sender 4, node 1
CALLING GM
   618       2
   619       2
   620       2
   621       2
   622       2
   623       2
   624       2
/* QUEUE OVER THRESHOLD FOR CONSUME QUEUE */
INSTRUCT opcode 42, time 62456, receiver 1, sender 2, node 1
CALLING SCH
   625   1   2
   626   1   2
   627   1   2
   628   1   2
   629   1   2
   630   1   2
/* REQUEST GRAPH VARIABLE.  SAME INSTRUCT BUT IT */
/* RGV GAVE IN ON PART OF A TIME UNIT AND WAS     */
/* RESCHEDULED LATER IN THE TIME UNIT.            */
INSTRUCT opcode 47, time 63029, receiver 2, sender 3, node 3
CALLING GM
INSTRUCT opcode 47, time 63044, receiver 2, sender 3, node 3
CALLING GM
   631   1   2
   632   1   2
   633   1   2
   634   1   2
   635   1   2
   636   1   2
   637   1   2
```

```
   638   1   2
   639   1   2
   640   1   2
   641   1   2
   642   1   2
   643   1
   644   1
   645
/* ACCEPT GRAPH VARIABLE.  NODE 3 TRYING TO EXECUTE */
INSTRUCT opcode 2, time 64554, receiver 3, sender 2, node 3
CALLING AP
   646           3
   647           3
   648           3
   649           3
   650           3
   651           3
   652           3
/* QOT FOR WRITE QUEUE */
INSTRUCT opcode 42, time 65261, receiver 1, sender 2, node 1
CALLING SCH
   653   1       3
   654   1       3
   655   1       3
   656   1       3
   657   1       3
   658   1       3
/* REQUEST QUEUE FROM AP TO GM FOR NODE 3*/
INSTRUCT opcode 50, time 65834, receiver 2, sender 3, node 3
CALLING GM
   659   1   2   3
   660   1   2   3
   661   1   2   3
   662   1   2   3
   663       2   3
   664       2   3
   665       2   3
   666       2
   667       2
   668       2
   669       2
   670       2
   671
   672
   673
   674
/* ACCEPT QUEUE FROM GM TO AP FOR NODE 3*/
INSTRUCT opcode 4, time 67424, receiver 3, sender 2, node 3
CALLING AP
   675           3
   676           3
   677           3
   678           3
   679           3
```

```
   680          3
   681          3
/* REQUEST QUEUE FROM AP TO GM FOR NODE 3    */
/* NOTE NODE 3 HAS TWO INPUT QUEUES (2 & 4) */
INSTRUCT opcode 50, time 68185, receiver 2, sender 3, node 3
CALLING GM
   682      2   3
   683      2   3
   684      2   3
   685      2   3
   686      2   3
   687      2   3
   688      2   3
   689      2   3
   690      2   3
   691      2   3
   692      2   3
   693      2   3
   694      2   3
   695
   696
   697
   698
/* ACCEPT QUEUE 4 FROM GM TO AP FOR NODE 3*/
INSTRUCT opcode 4, time 69846, receiver 3, sender 2, node 3
CALLING AP
   699          3
/* CHANNEL 1 FIRED AGAIN                   */
INSTRUCT opcode 30, time 70000, receiver 4, sender 4, node 1
CALLING IOPP
   700          3   4
   701          3   4
   702          3   4
   703          3
   704          3
   705          3
/* CHANNEL 1 FIRED AND IS WRITING TO QUEUE 1 */
INSTRUCT opcode 64, time 70546, receiver 2, sender 4, node 1
CALLING GM
   706      2   3
   707      2   3
   708      2   3
   709      2   3
   710      2   3
   711      2   3
   712      2   3
   713      2   3
   714      2   3
   715      2   3
   716      2   3
   717      2   3
   718      2   3
   719          3
   720
```

```
   721
   722
   723
   724
   725
   726
   727
/* NODE 3 COMPLETED EXECUTION AND IS WRITING QUEUE 5*/
INSTRUCT opcode 64, time 72758, receiver 2, sender 3, node 3
CALLING GM
   728       2
   729       2
   730       2
   731       2
   732       2
   733       2
   734       2
   735       2
   736       2
   737       2
   738       2
   739       2
   740
/* QOT FOR EXN AND WRITE QUEUE*/
INSTRUCT opcode 42, time 74001, receiver 1, sender 2, node 1
CALLING SCH
   741   1
   742   1
   743   1
   744   1
/* READY FOR INSTRUCT STREAM SINCE AP FINISHED */
/* SET UP MODE                                 */
INSTRUCT opcode 45, time 74469, receiver 1, sender 3, node 3
CALLING SCH
   745   1
   746   1
   747   1
   748   1
   749   1
   750   1
/* WAS RESCHEDULED FROM TIME 745 */
INSTRUCT opcode 45, time 75001, receiver 1, sender 3, node 3
CALLING SCH
   751   1
   752   1
   753   1
   754   1
   755   1
   756   1
   757   1
   758   1
   759   1
   760   1
   761   1
```

```
   762   1
   763   1
   764   1
   765   1
   766   1
   767   1
/* SEND INSTRUCT STREAM. NODE 2 JUST FIRED */
INSTRUCT opcode 53, time 76708, receiver 5, sender 1, node 2
CALLING GM
   768                   5
   769                   5
   770                   5
   771                   5
/* GOT BUT CHANNEL IS BUSY -- QUEUE 5 and EXECUTE NODE 3 */
INSTRUCT opcode 42, time 77176, receiver 1, sender 2, node 3
CALLING SCH
   772                   5
   773                   5
   774                   5
   775                   5
   776                   5
/* CONSUME QUEUE  EXECUTE NODE 3*/
INSTRUCT opcode 72, time 77686, receiver 2, sender 3, node 3
CALLING GM
   777         2         5
   778         2         5
   779         2         5
   780         2         5
   781         2
   782         2
   783         2
   784         2
   785         2
/* ACCEPT INSTRUCT STREAM FOR NODE 2 FROM GM5 TO AP */
INSTRUCT opcode 3, time 78554, receiver 3, sender 5, node 2
CALLING AP
   786         2   3
   787         2   3
   788         2   3
   789             3
   790             3
   791
   792
   793
   794
   795
   796
   797
   798
   799
/* EXECUTE INSTRUCT STREAM.  QUEUE 5 OR CHANNEL 3 */
/* IS BEING WRITTEN                                 */
INSTRUCT opcode 24, time 79904, receiver 4, sender 1, node 3
CALLING IOPP
```

```
/* EXN FOR NODE 1 AND QUEUE 1*/
INSTRUCT opcode 30, time 80000, receiver 4, sender 4, node 1
CALLING IOPP
   800                4
   801                4
   802                4
/* REQUEST QUEUE FOR WRITING TO CHANNEL 3 */
INSTRUCT opcode 50, time 80204, receiver 2, sender 4, node 3
CALLING GM
/* PREEMPTED BY EIS AND NOW EXECUTINT EXN */
INSTRUCT opcode 30, time 80204, receiver 4, sender 4, node 1
CALLING IOPP
   803        2       4
/* QOT */
INSTRUCT opcode 42, time 80372, receiver 1, sender 2, node 3
CALLING SCH
   804    1   2       4
   805    1   2       4
   806    1   2
   807    1   2
   808    1   2
/* WRITE QUEUE FOR QUEUE 1 TRIGGERED BY EXN. RESCHEDULED */
INSTRUCT opcode 64, time 80864, receiver 2, sender 4, node 1
CALLING GM
/* RESCHEDULED  */
INSTRUCT opcode 72, time 80882, receiver 2, sender 3, node 3
CALLING GM
   809    1   2
   810    1   2
   811    1   2
   812    1   2
   813    1   2
   814        2
/* RESCHEDULED WRITE QUEUE FOR EXN AND QUEUE 1*/
INSTRUCT opcode 64, time 81426, receiver 2, sender 4, node 1
CALLING GM
/* RESCHEDULED AGAIN */
INSTRUCT opcode 72, time 81426, receiver 2, sender 3, node 3
CALLING GM
   815        2
   816        2
   817        2
/* ACCEPT QUEUE FROM GM TO IOP.  OUTPUT CHANNEL 3 */
/* TRIGGERED.                                     */
INSTRUCT opcode 4, time 81774, receiver 4, sender 2, node 3
CALLING IOPP
   818        2       4
   819        2       4
   820        2       4
   821        2       4
   822        2       4
   823        2       4
   824        2       4
   825        2       4
```

```
   826        2        4
/* CONSUME QUEUE 4.  NODE 2 COMPLETED EXECUTION */
INSTRUCT opcode 72, time 82681, receiver 2, sender 3, node 3
CALLING GM
   827        2        4
   828        2        4
   829        2        4
   830        2        4
   831        2        4
/* QOT FOR QUEUE 1 FOR EXN */
INSTRUCT opcode 42, time 83114, receiver 1, sender 2, node 1
CALLING SCH
   832    1   2        4
   833    1   2        4
   834    1   2        4
   835    1   2        4
   836    1   2        4
/* REQUEST GRAPH VARIABLE FOR NODE 2 */
INSTRUCT opcode 47, time 83687, receiver 5, sender 3, node 2
CALLING GM
   837    1   2        4   5
   838    1   2        4   5
INSTRUCT opcode -4, time 83881, receiver 1, sender 2, node 3
CALLING SCH
   839    1                5
   840    1                5
   841    1                5
/* SUPERFICIAL REPORT NODE DONE */
INSTRUCT opcode -4, time 84114, receiver 1, sender 2, node 3
CALLING SCH
   842                     5
   843                     5
   844                     5
   845                     5
   846                     5
   847                     5
   848                     5
   849
   850
   851
/* ACCEPT GRAPH VARIABLE FOR NODE 2      */
INSTRUCT opcode 2, time 85176, receiver 3, sender 5, node 2
CALLING AP
   852           3
   853           3
   854           3
   855           3
   856           3
   857           3
   858           3
   859           3
/*QUEUE UNDER THRESHOLD */
INSTRUCT opcode 44, time 85919, receiver 1, sender 2, node 3
CALLING SCH
```

```
   860   1      3
   861   1      3
   862   1      3
   863   1      3
   864   1      3
/* REQUEST QUEUE 3 FOR NODE 2 */
INSTRUCT opcode 50, time 86492, receiver 5, sender 3, node 2
CALLING GM
   865   1      3      5
   866   1      3      5
   867   1      3      5
   868   1      3      5
   869   1      3      5
/* INPUT OUTPUT PROCESSOR COMPLETED OUTPUT TO CHANNEL */
/* READY FOR INSTRUCT STREAM.                        */
INSTRUCT opcode 45, time 86960, receiver 1, sender 4, node 3
CALLING SCH
   870   1      3      5
   871   1      3      5
   872   1             5
   873   1             5
   874   1             5
   875   1             5
   876   1             5
   877   1             5
   878   1             5
   879   1
   880   1
   881   1
   882   1
/* ACCEPT QUEUE FOR NODE 2.  NODE 2 WILL COMMENCE */
/* EXECUTION IN THE ARITHMETIC PROCESSOR.          */
INSTRUCT opcode 4, time 88228, receiver 3, sender 5, node 2
CALLING AP
   883   1      3
   884   1      3
   885   1      3
   886   1      3
   887          3
   888          3
   889          3
   890          3
   891          3
   892          3
/* CONSUME QUEUE.  IOP IS CONSUMING QUEUE AFTER WRITING*/
/* IT TO AN OUTPUT CHANNEL 3.                          */
INSTRUCT opcode 72, time 89248, receiver 2, sender 4, node 3
CALLING GM
   893       2  3
   894       2  3
   895       2  3
   896       2  3
   897       2  3
   898       2  3
```

```
   899        2   3
/* EXN AGAIN FOR CHANNEL 1 */
INSTRUCT opcode 30, time 90000, receiver 4, sender 4, node 1
CALLING IOPP
   900        2   3   4
   901        2   3   4
   902        2   3   4
   903        2   3
   904        2
   905
/* WRITING QUEUE FOR CHANNEL 1 FOR EXN */
INSTRUCT opcode 64, time 90524, receiver 2, sender 4, node 1
CALLING GM
   906        2
   907        2
   908        2
   909        2
   910        2
   911        2
   912        2
   913        2
   914        2
/* QUEUE UNDER THRESHOLD */
INSTRUCT opcode 44, time 91466, receiver 1, sender 2, node 3
CALLING SCH
   915   1   2
   916   1   2
   917   1   2
   918   1
   919   1
/* NODE 2 COMPLETED SET UP MODE.  READY FOR INSTRUCT */
/* STREAM SENT TO SCHEDULER                            */
INSTRUCT opcode 45, time 91934, receiver 1, sender 3, node 2
CALLING SCH
   920   1
   921   1
   922   1
   923   1
   924   1
/* READY FOR INSTRUCT STREAM RECEIVED BY SCHEDULER */
/* BUT BLOCKED BY QUEUE UNDER THRESHOLD.  RESCHEDULED */
INSTRUCT opcode 45, time 92466, receiver 1, sender 3, node 2
CALLING SCH
   925   1
   926   1
   927   1
   928   1
   929   1
/* WRITE QUEUE 4 FOR NODE 2 EXECUTION                  */
/*  WRITE QUEUE INSTRUCTS ALWAYS GIVE QUEUE NUMBER*/
/*  OF NODE AT HEAD OF THE QUEUE.                      */
INSTRUCT opcode 64, time 92974, receiver 2, sender 3, node 3
CALLING GM 930   1   2
   931   1   2
```

```
   932    1    2
   933    1    2
   934    1    2
   935    1    2
   936    1    2
   937    1    2
   938    1    2
   939    1    2
   940    1    2
   941    1    2
/* READY FOR INSTRUCT STREAM TRIGGERED NODE 1 ON READY*/
/* LIST IN SCHEDULER. SEND INSTRUCT STREAM TO GM2     */
/* TO START NODE 1 EXECUTING.                          */
INSTRUCT opcode 53, time 94173, receiver 2, sender 1, node 1
CALLING GM
   942         2
/* RESCHEDULED FROM TIME 942                           */
INSTRUCT opcode 53, time 94284, receiver 2, sender 1, node 1
CALLING GM
   943         2
   944         2
   945         2
   946         2
/* QUEUE OVER THRESHOLD FOR QUEUE 1                    */
INSTRUCT opcode 42, time 94641, receiver 1, sender 2, node 1
CALLING SCH
   947    1    2
   948    1    2
   949    1    2
   950    1    2
   951    1    2
/* CONSUME QUEUE 3 FOR NODE 2 EXECUTION.                */
INSTRUCT opcode 72, time 95151, receiver 5, sender 3, node 2
CALLING GM
   952    1    2            5
   953    1    2            5
   954    1    2            5
   955    1    2            5
   956    1                 5
   957                      5
   958                      5
/* ACCEPT INSTRUCT STREAM FROM GM TO AP FOR NODE 1 */
INSTRUCT opcode 3, time 95900, receiver 3, sender 2, node 1
CALLING AP
   959            3         5
   960            3         5
   961            3         5
   962            3         5
   963            3         5
/* SUPERFICIAL REPORT NODE DONE.                       */
INSTRUCT opcode -4, time 96351, receiver 1, sender 5, node 2
CALLING SCH
   964            3
   965
```

```
   966
   967
   968
   969
   970
   971
   972
   973
/* QUEUE OVER THRESHOLD FOR WRITE QUEUE FOR NODE 2   */
/* EXECUTION.                                        */
INSTRUCT opcode 42, time 97383, receiver 1, sender 2, node 3
CALLING SCH
   974   1
   975   1
   976   1
   977   1
   978   1
   979   1
/* REQUEST GRAPH VARIABLE FOR NODE 1 EXECUTION.     */
INSTRUCT opcode 47, time 97956, receiver 2, sender 3, node 1
CALLING GM
   980   1   2
   981   1   2
   982   1   2
   983   1   2
   984   1   2
/* QUEUE UNDER CAPACITY SENT TO SCHEDULER FROM GM.  */
/* QUEUE 3 ON CHANNEL 2 HAS GONE UNDER CAPACITY.    */
/* CHANNEL WILL BE TURNED BACK ON WITH A CNDT       */
/* INSTRUCT (CONTINUE NODE DATA TRANSFER)        */
INSTRUCT opcode 43, time 98438, receiver 1, sender 5, node 2
CALLING SCH
   985   1   2
   986   1   2
   987   1   2
   988   1   2
   989   1   2
   990   1   2
   991   1   2
   992   1
   993   1
/* RESCHEDULED FROM TIME 985                     */
INSTRUCT opcode 43, time 99383, receiver 1, sender 5, node 2
CALLING SCH
/* ACCEPT GRAPH VARIABLE FROM GM TO AP FOR NODE 1 */
INSTRUCT opcode 2, time 99386, receiver 3, sender 2, node 1
CALLING AP
   994   1       3
   995   1       3
   996   1       3
   997   1       3
   998   1       3
   999   1       3
 /* CHANNEL 1 HAS FIRED                           */
```

```
INSTRUCT opcode 30, time 100000, receiver 4, sender 4, node 1
CALLING IOPP
  1000   1       3   4
```

FUNCTIONAL ELEMENT UTILIZATION

FEID	TYPE	UTILIZATION
1	SCH	43
2	GM	40
3	AP	30
4	IOP	2
5	GM	14

TOTAL TIME = 1000.

NODE EXECUTION INFORMATION

NODE ID	OPCODE	NODE FIRINGS
1	14	2
2	28	2
3	25	1

CHANNEL EXECUTION INFORMATION

CHANNEL ID	CHANNEL FIRINGS
1	10
2	3
3	1

QUEUE EXECUTION INFORMATION

QUEUE ID	DATA ITEMS	HEAD NODE	TAIL NODE
1	35	1	1
2	5	3	1
3	10	2	2
4	11	3	2
5	0	3	3

VITA

Marilyn Opitz Aiken

Candidate for the Degree of

Master of Science

Thesis: ENHANCED MODULAR SIGNAL PROCESSOR TIMING SIMULATOR

Major Field: Computing and Information Sciences

Biographical:

Personal Data: Born in Chickasha, Oklahoma, October 6, 1960, the daughter of G. W. and Thelma G. Opitz. Married to Calvin E. Aiken on June 12, 1982. Daughter Christina D. Aiken born September 9, 1984.

Education: Graduated from Chickasha High School, Chickasha, Oklahoma, in May, 1978; received Bachelor of Science Degree in Electrical Engineering from the University of Oklahoma in May, 1982; completed requirements for the Master of Science degree at Oklahoma State University in May, 1987.

Professional Experience: Engineer, Halliburton Services, Duncan, Oklahoma, June, 1982, to August, 1984; Graduate Assistant, Department of Computer Science, Oklahoma State University, November, 1984, to January, 1985.