

ENHANCEMENTS TO THE TIMING SIMULATOR
OF THE ENHANCED MODULAR
SIGNAL PROCESSOR

By

STEVEN CRAIG NELSON

Bachelor of Science in Arts and Sciences

Oklahoma State University

Stillwater, Oklahoma

1983

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
December, 1988

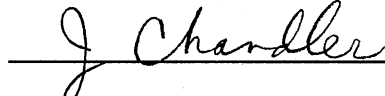
Thesis
1988
N4305e
cop. 2

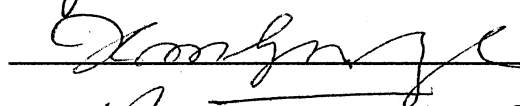
ENHANCEMENTS TO THE TIMING SIMULATOR
OF THE ENHANCED MODULAR
SIGNAL PROCESSOR

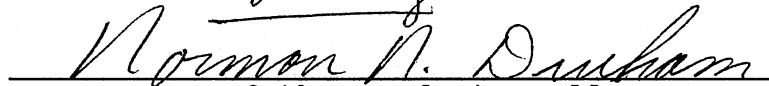
Thesis Approved:



Thesis Advisor







Dean of the Graduate College

ACKNOWLEDGMENTS

I wish to express my sincere appreciation and gratitude to Dr. G. E. Hedrick and the Department of Computing and Information Sciences for the many kindnesses and opportunities extended me during my graduate study at Oklahoma State University. These experiences have presented many unexpected opportunities.

I would also like to thank Dr. G. E. Hedrick, Dr. J. P. Chandler, and Dr. K. M. George for their support and flexibility in serving on my thesis committee.

Special thanks go to Mark Vasoll and Gregg Wonderly for their comraderie and enthusiasm throughout our time at Oklahoma State.

I owe a great debt to the Brothers of the Alpha Chapter of Kappa Kappa Psi, the staff of the National Office of Kappa Kappa Psi/Tau Beta Sigma, the OSU Bands, and in particular, Dr. Joseph P. Missal and Mr. William Ballenger of the Music Department for their incredible encouragement and trust. Without the artistry of music as a balancing force in my life, I would never have completed this degree.

In the end, I must thank my family, especially my parents for their support and many sacrifices, in spite of the tragedies.

TABLE OF CONTENTS

Chapter	Page
I. INTRODUCTION	1
Realtime Signal Processing.	1
Naval Signal Processors	2
Dataflow	3
Dataflow Architectures	5
Dataflow as a Technique for Signal Processing	11
Signal Processing Dataflow Architectures	11
II. THE ENHANCED MODULAR SIGNAL PROCESSOR.	17
ECOS.	17
EMSP.	21
III. THE EMSP TIMING SIMULATOR.	27
Node Execution Cycle.	29
Input	30
Output.	31
Implementation of Functional Elements	32
IV. ENHANCEMENTS TO THE TIMING SIMULATOR OF THE ENHANCED MODULAR SIGNAL PROCESSOR	35
Simulator Input Procedures.	35
Configuration Feasibility Determination	40
Dynamic Graph State Output Procedure.	42
V. Conclusions.	48
Future Work	49
A SELECTED BIBLIOGRAPHY	50
APPENDICES.	53
APPENDIX A - FIGURES	54
APPENDIX B - SAMPLE OUTPUT	64

LIST OF FIGURES

Figure	Page
1. Simple Dataflow Graph	55
2. Fundamental Functions of Dataflow Processing Elements	56
3. Basic Dataflow Machine Structures	57
4. Diagram of the DFSP Architecture.	58
5. Diagram of the DDSP Architecture.	59
6. ECOS Sample Graph	60
7. ECOS Node Diagram	61
8. Diagram of the EMSP Architecture.	62
9. Diagram of EMSP Node Execution Cycle.	63

NOMENCLATURE

AGV	Accept Graph Variable Instruction
AIS	Accept Instruction Stream Instruction
AP	Arithmetic Processor
AQ	Accept Queue Instruction
ASP	Advanced Signal Processor
CBUS	Control Bus
CPP	Command Program Processor
CQ	Consume Queue Instruction
DDC	Data Driven Communication
DDM1	Data-Driven Machine #1
DDP	Distributed Data Processor
DDPL	Data Driven Program Language
DDSP	Data Driven Signal Processor
DFSP	Data Flow Signal Processor
DTN	Data Transfer Network
ECOS	EMSP Common Operating System
EMSP	Enhanced Modular Signal Processor
FFT	Fast Fourier Transform
GIP	Graph Instantiation Parameter
GM	Global Memory
GV	Graph Variable
IOP	Input Output Processor

LAU	langage a assignation unique
NEP	Node Execution Parameter
NOC	Number of Conditions
PIP	Primitive Interface Procedure
QOC	Queue Over Capacity Instruction
QOT	Queue Over Threshold Instruction
QUC	Queue Under Capacity Instruction
QUT	Queue Under Threshold Instruction
RFIS	Request For Instruction Stream Instruction
RGV	Request Graph Variable Instruction
RQ	Request Queue Instruction
SIS	Send Instruction Stream
WGV	Write Graph Variable Instruction
WQ	Write Queue Instruction

CHAPTER I

INTRODUCTION

The purpose of this thesis is to design necessary enhancements to the Enhanced Modular Signal Processor Timing Simulator originally created by Marilyn Aiken [1].

This chapter is a review of the literature on the major topics concerning real-time signal processors, dataflow, and other signal processing architectures. The following chapters deal in detail with the EMSP methodology and architecture, the EMSP Timing Simulator and the enhancements made to it.

Realtime Signal Processing

Signal processing, in its broadest terms, is an activity of spectrum analysis with a wide variety of applications from the low-frequency spectrum of seismology through the acoustic spectrum of sonar, speech and music, to the video spectrum of radar systems.

In the period leading up to and following World War II, analog signal processing was at the forefront of the advancement. This was based not only on the level of technology, but also on the economy of the application.

The formal theory of the digital signal processing of today did not emerge until the mid-1960's. The prospect of complete digital signal processing systems began to be realized with the advent of integrated circuit technology. The first major impetus to the science began with the release of the Cooley-Tukey paper in 1965 on a new method of computing the discrete Fourier transform, the basis of many signal processing filtering techniques. The value of this new method lies in the reduction of the computing time of the transform by one to two orders of magnitude, from $O(n^2)$ to $O((n/2)^2)$.

The attraction of digital signal processing over analog signal processing lies in the capability of digital systems to achieve a guaranteed accuracy and essentially perfect reproducibility [18].

Naval Signal Processors

The United States Navy began development of its first-generation signal processing system in the 1960's. The system was to be designed using common equipment that could be configured for a variety of applications aboard surface ships, submarines, helicopters and airplanes. The Advanced Signal Processor (ASP) successfully fulfilled the Navy's signal processing needs with one software development system and a small number of hardware modules that were configured for each application.

By the late 1970's, the tremendous increase in technology created a serious increase in the signal processing needs of the Navy. A new second-generation signal processing system was necessary. This new system, the Enhanced Modular Signal Processor (EMSP), was designed to increase the processing density, and to correct one of the main drawbacks of the ASP, namely, the inability to process many parallel channels of data at the same time. These needs led to the development of the EMSP Common Operational Support (ECOS) dataflow methodology and its implementation on the EMSP [4].

Dataflow

Most attempts to define "dataflow", begin by describing what it is not; namely not "von Neumann." Introduced in 1946, by John von Neumann, this concept is the familiar serial method of organization that dominates computer languages and architectures to this day [12].

Characterized by a sequential one word-at-a-time instruction stream and an incremental instruction counter, this organization has some drawbacks that other methodologies have sought to overcome. A principal area of interest is in increasing the utilization of resources, thereby increasing the system throughput. Many times, in a von Neumann machine, resources not needed for the currently executing instruction are idle.

One way to reduce the amount of idleness in the various component parts of a system is to exploit the inherent parallelism of the an algorithm. The concept of parallelism relies on the functionality and asynchronization of the operations to be performed by the algorithm. An operation that is functional is one, that when properly executed, does not affect the operation of another directly. A simple example would be the relationship between the addition and subtraction operations. Both operations execute in a predictable manner, regardless of the operation of the other. In other words, an operation that exhibits functionality is one that, given the same data, will always yield the same results whenever it is called to be executed. Its position in the calling sequence of an algorithm does not affect its outcome. The asynchronization of an operation concerns the idea that a given operation is not time dependent. That is, other than the normal logic flow of the algorithm, a given operation may be performed at any time during the execution of the algorithm, and will not be affected by the execution of any other operation. [22]

Dataflow is a conceptual computer organization that inherently exploits parallelism. Its basic tenet is that an instruction should execute as soon as all of the necessary operands are available. This type of organization has also been described as "data-driven" and is intended to allow the fastest throughput. [16]

The easiest visualization of dataflow is through the use of a directed graph. Each node of the graph represents an operation to be performed. In the simplest case, a node may be a low level operation, such as addition or subtraction. At a higher level, a node may represent a complex function, such as a Fast Fourier Transform (FFT). The arcs connecting the nodes represent the paths by which the data "flows" from one node to the next. [20] Figure 1 (Appendix A) is an example of a simple dataflow graph.

Dataflow Architectures

The many dataflow architectures proposed and implemented to date, while widely varying, can be viewed as providing three basic elements. Each machine must provide a way of executing the nodal operations, provide a way of storing a description of the graph implementation, and provide a mechanism for collecting and matching the data "tokens" as they are made available.

A General Model

A general model of a dataflow machine, such as that given by Veen [25], begins with descriptions of the processing element and the activity template. The nodes of the dataflow program, in some machines, are implemented in the form of a data structure called a template. Each template contains a description of the node and information or

storage of the input tokens. The node description may either be an operand code, in the low-level sense, or a shorthand name for an instruction stream to be executed, in a higher level sense. A list of destination addresses (output arcs) would also be included.

There are three fundamental functions that take place in the processing elements (Figure 2, Appendix A). The first is the enabling function. This function sequentially accepts the tokens as they are made available and places them into storage. This function also determines whether the node for which a token is destined has received the data required for it to execute. If a node is determined to be ready to fire, the enabling unit creates an executable packet containing the input data, information on the operation to be executed (either the operand or the instruction stream), and the output destinations. This packet is then presented to the functional unit of the processing element.

The functional unit executes the packet and computes the output values. These output values are combined with the destination addresses into new tokens that are sent back to the enabling unit in what completes a circular pipeline.

This describes the execution cycle of most dataflow architectures in a most general way. Many other factors influence just how a particular design is implemented. Some of the important factors that need consideration include the

number of input arcs that are allowed into a node, the number of tokens that are necessary on an arc, how reentrant code may be implemented, and how non-trivial data structures are maintained.

A simple but important variation of this model would include those machines that use tagged (or "colored") tokens in allowing reentrant node execution. In such machines, nodes are shared between different instances of a graph. Tokens with matching tags are considered part of the same node execution. This makes it impractical to store the tokens in the nodes themselves, due to finite storage for the tokens. A solution that is often employed (Figure 2, Appendix A), splits the enabling unit into two stages: a matching function, which checks the destined node to determine whether it is enabled; and a fetching function, which matches the tokens and the node description into an executable packet, as described above.

The literature suggests that every description of a dataflow machine presents a unique design. Veen [25] suggests that most designs will actually conform to one of three basic structures (Figure 3, Appendix A). A "one-level" machine matches the general model in that there is only pipeline concurrency within a processing element. Execution of instructions is only within a processing element with the output tokens being used in the same processing element or communicated to another processing element.

A "two-level" machine is one in which the functional unit of the processing element actually consists of many "functional elements" which are capable of executing packets concurrently. Any executable packet is allocated to any idle functional element.

A "two-stage" machine has split processing elements with an extra communication medium between the enabling and functional units. This can be especially advantageous if some of the functional elements have specialized usages.

Communication

A dataflow machine which utilizes direct communication has adjacent nodes of the graph allocated to the same processing element or to processing elements that are capable of communicating directly. The important aspect of this is that tokens are delivered through the communication medium in the same order that they were received. This implies that the determinancy of the graph is maintained. A design which utilizes packet communication offers a greater opportunity for load distribution and parallelism in executing the processing elements, since data is communicated in packet form. Its advantages though are tempered by concerns over contention in the data paths and the maintenance of the determinancy of the graph execution.

Architectural Examples

Veen [25] presents an excellent summary of these various architectural types. The most important example of the direct communication machines is the Data-Driven Machine #1 (DDM1), first described by Davis. This machine utilizes a tree arrangement for the configuration of the processing elements. While this is the oldest example of a working dataflow machine, it has a serious bottleneck problem at the root of the tree as communication proceeds between the processing elements.

The Distributed Data Processor (DDP), the language assignment unique (LAU), and the uPD7281 Dataflow Image Processor, are all examples of static packet communication machines. The DDP machine was constructed at Texas Instruments and uses a locking method to protect reentrant graphs. The prototype, built with four processing elements, uses a ring-structured communication unit with a direct feedback link for the tokens that stay within that processing element.

The LAU machine, constructed in Toulouse, France, was designed around strong processing elements with the higher level structure left unspecified. The #0 prototype, completed in 1980, was a single processing element of a conventional microprocessor with 32 functional elements. This machine differs from other designs in that the data and

instruction memories are separate and utilize a multiphase communication path between the functional elements.

The uPD7281 Dataflow Image Processor, developed by NEC Electronics, is capable of being used as a small processing element in a dataflow machine. It has a seven-stage circular pipeline in such a way that tokens that are addressed to the same processing element never leave the chip. The design is also capable of regulating the level of parallelism. If a processing element is underutilized, preference is given to tokens that will increase the amount of parallelism.

Another architecture that deserves note is of the tagged token type. The Manchester Dataflow Machine, described by Gurd and Watson, uses that tagged token idea to increase parallelism for reentrant graphs. It is a two-stage machine, as described earlier, and has a four unit pipeline. The token queue, matching unit, fetching unit, and the functional unit are each internally synchronous, but uses asynchronous protocols to communicate externally. With fixed packet sizes, more than 30 packets can be processed simultaneously. Several other designs, including one for digital signal processing (See DDSP) have been based on these ideas.

Dataflow as a Technique for Signal Processing

As an application for dataflow technology, real-time signal processing seems well suited. The main attributes of signal processing are: (1) a well-defined sequence of data value independent algorithms that are (2) repeatedly applied to signal values as they are received [13,17]. These algorithms are data value independent in the sense that the values received have no effect on the sequencing of the operations. With a continuous stream of input data processed by a repeatedly executing set of algorithms, there is a significant possibility to overlap executions for greater parallelism. In an effort to reduce the complexity of the dataflow program graphs created for these applications, the nodes of the graph utilize the higher design methodology and associate nodes with the complex operations to be performed rather than the individual instructions.

Signal Processing Dataflow Architectures

A number of other dataflow architectures designed specifically for real-time signal processing exist. Two described below, are of interest to the EMSP.

DFSP: A Data Flow Signal Processor

DFSP is an architecture presented by Hartimo, Kronlof, Simula, and Skytta [13], with the following design goals made to meet the special needs of signal processing.

(1) Reentrant code is provided by using colored tokens. This allows for less overhead in terms of both computations and the memory space needed for the high level operations.

(2) Every result packet declares an activity template as its destination. The activity templates, found in the activity store, are temporary storage for those operations that have received at least one operand but are not ready to execute. A new template is created if a matching one has not been found (i.e., this result packet represents the first operand to be received for this operation).

(3) The processing elements themselves have no access to any shared data structures, as may be found in other architectures. This lowers overhead by reducing the memory management complications. Instead, operand data are circulated as part of a double bus architecture. The thick pipes of Figure 4 (Appendix A) represent the data path of the operand data, and the thin pipes represent the paths for control flow.

The operation of the various functional units of the DFSP architecture are described below. (Figure 4, Appendix A)

The update unit is used to keep the status of the various activity templates as current as possible. When a result packet arrives from the processing elements, the packet is hashed with the current activity templates in search of a destination operation. A new activity template is created if necessary. Storage may also be allocated for operand values. The update unit then sends a transfer request to the result transfer unit.

The result transfer unit initiates a transfer of a result data block, upon command, and updates the trigger field of the destination activity template. The DFSP architecture does not count the number of operands needed for each operation; rather, it calculates the total size of the accumulated operands needed for that operation. The trigger field of an activity template is set to this value. As each operand arrives at its designated activity template, the trigger field is decremented by the size of the operand. An operation is deemed ready to execute when the trigger field reaches zero. At that point, the activity template address is placed into the queue.

The fetch unit pairs up ready to execute activity templates with an idle processing unit containing the necessary operation code. An operation packet is sent to the

processing element and a data transfer request is placed in the data queue.

The data transfer unit receives requests to transfer operand data blocks to specified processing elements. When completed, the unneeded activity template is marked as free.

The processing elements are the functional units that perform the actual computations. An idle processing unit informs the fetch unit about being idle and waits to receive an operation packet. Once the operation packet and the operand data, from the data transfer unit, are available, the operation is performed and result packets are sent to all the specified destinations, via the update unit.

Multiple instantiations of a given node are possible by the use of colored tokens, reentrant code and the reliability of the update unit to associatively manage the various token/operand matching operations needed.

DDSP: The Data Driven Signal Processor

The DDSP system [14] is a dataflow architecture that can be configured from 1 to 32 processors without software modification. Designed for signal processing, the designers state that large DDSP systems can exceed Cray-1 and CDC STAR-100 supercomputers in processing capability.

Designed as an alternative to array processors by providing the same low cost computing power with greater system flexibility, DDSP has some interesting characteristics.

(1) DDSP utilizes a skew algorithm for routing data among processors. Each data token generated has a label field appended to it to distinguish it from different instances of the same token (i.e., dynamic graph execution with colored tokens). These label fields are used by the skew algorithm to route tokens to specific processors. This offers the opportunity for a uniform distribution of processing and localizing communication to nearby processors.

(2) A special data structure for communicating data between procedures has been developed. Called a data driven communication (DDC) structure, it is passed as a pointer in procedure calls. The structure contains data, to be used in the computations, and control information such as array sizing and return pointers.

Each DDSP processor (Figure 5, Appendix A) contains an input queue, matching store and a processing element. The input queue is used for temporary storage and helps in load leveling. The matching store is an associative memory that is used to pair up tokens with identical keys. The keys contain an 11-bit node address and a 16-bit label field that defines token attributes. When a match is found the token pair and the key are sent to the processing element for execution of the node.

The processing element contains a microprogram sequencer that controls both an arithmetic processor and a

label processor which for the most part are independent of each other.

The interconnection network is a linear wraparound connection of the DDSP processors, overlaid by a three level tree. The linear connection of the processors is used for short distance communication between nearby processors; while the tree structure is for longer movement of data. Each token contains its own network destination and routes itself to the various processors or I/O ports in the system.

This appears to be a relatively simple architectural model for dataflow signal processing. The inherent complexity of this class of applications has been retained in the programming. A high level language called the Data Driven Program Language (DDPL) breaks a program into program blocks, procedures, actions and node definitions. Procedures contain node definitions which are the basic units used. Node definitions contain all the executable code and are designed to be independent of other node definitions. This will allow nodes to execute with maximum parallelism. The generalized labelling helps manage multiple activations of the same node.

CHAPTER II

THE ENHANCED MODULAR SIGNAL PROCESSOR

The Enhanced Modular Signal Processor (EMSP) has been called a hybrid dataflow machine. This comes from the fact that not only does the EMSP use graphs to describe and execute signal processing algorithms in the dataflow manner, it also uses additional command programs and a separate command program processor to control and manipulate the program graphs in a control flow manner.

The first part of this chapter deals with the Enhanced Modular Signal Processor Common Operational Support (ECOS) methodology. This methodology is the design basis around which the Enhanced Modular Signal Processor was developed. The remainder of this chapter will then discuss in detail the architecture of the EMSP itself.

EMSP Common Operational Support

Methodology

The Enhanced Modular Signal Processor Common Operational Support (ECOS) methodology is the design environment developed for creating signal processing applications. ECOS is not a classical dataflow methodology. While still based on the concept of a directed graph representation of

the signal processing algorithm, the arcs of the graph have been redefined as queues and thus allow more than one data element on an arc. The various components of ECOS operate on this basis.

Graphs

A signal processing graph is comprised of nodes, which represent the specific signal processing operation (primitive) to be executed; and, a set of queues that represent the flow of data through the graph. A node may also represent a subgraph, which at instantiation time would be expanded into nodes, queues, and graph variables as if it were a macro definition. Figure 6 (Appendix A) is an example of a simple ECOS Graph.

Graph Variables

Graph variables represent memory elements that contain one data element that can be used as control variables or algorithm coefficients to the signal processing algorithm. Graph variables are not dataflow entities and do not affect the execution of the node, but are available to the node once it has started executing.

Nodes

Each node of the graph represents a specific processing operation, called a primitive and a primitive interface

procedure (PIP) that provides the interface between the node and the primitive.

Each node also requires one or more input ports and zero or more output ports. Associated with each port is a set of node execution parameters (NEP) that define how the queue attached to each port is utilized. The diagram of Figure 7 (Appendix A) illustrates the ECOS node construction.

Node Execution Parameters

The node execution parameters (NEP) associated with input ports include the threshold, offset, read, and consume amounts. The threshold amount is the number of data elements that must exist in a particular queue before the associated node is ready to execute. The offset amount is the number of data elements to skip over at the head of the queue before reading data from the queue. The read amount is the number of data elements to read from that point in the queue. The consume amount is the number of data elements to remove from the queue once the node has finished executing.

The node execution parameter associated with output ports is the valve amount. The valve amount specifies how much of the output data is placed on the output queues or whether it be discarded.

While the threshold NEP must be specified at start-time and remains fixed throughout the graph execution, the other NEPs may be fixed or may be calculated by the node's primitive interface procedure (PIP) prior to the node execution.

Primitives

The basic building block of the methodology is the primitive and it is definitely machine dependent. It contains the code necessary to perform the defined node function.

Primitive Interface Procedure

Primitive Interface Procedures (PIP) exist to provide the logical connections between the various input and output queues attached to the node ports and the inputs and outputs of the primitive. The PIP also has a nodal intelligence with the ability to calculate values of NEPs. This, in essence, allows for run-time alterations of the node execution.

Queues

Queues provide the logical connections between the nodes of the graph. As first-in first-out data structures they have the ability to expand and contract as needed. The

same node may be both at the head and at the tail of a queue to provide a feedback capability.

Command Programs

Command programs are written by application programmers in a standard high order language with special signal processing statements embedded. These command programs are not dataflow and obey standard control flow rules. The capabilities of command programs typically include starting and stopping graphs and performing exception handling when errors are detected.

The Enhanced Modular Signal

Processor

The architecture of the Enhanced Modular Signal Processor was designed to implement the dataflow methodology of ECOS. Bloch [4] describes it as a distributed control, multiprocessor architecture which provides runtime support of data movement through the graph, node execution management, queue and graph variable management, and graph reconfiguration. Various types of functional elements comprise the EMSP architecture, and are connected together by a control bus and a data transfer network. Data and control information is transferred between functional units by the transmission of system level instructions over these communication paths. Each functional element type has a

unique instruction set designed for its particular activity. An illustration of the EMSP architecture can be found in Figure 8 (Appendix A).

Data Transfer Network

All system level instruction containing large data blocks are transferred over the data transfer network (DTN). An example of an instruction which is passed over the DTN is the Accept Queue (AQ) instruction, which is used to transfer data from a memory to a processor. This instruction consists of an operation code, a request identifier, a queue identifier, the number of data words, and the actual queue data. This is the defined path for the AQ instruction as large amounts of data may transferred this way. Each DTN can be described as a multiple path, unidirectional source directed switching network, configurable with a varying number of ports, depending on the application. A transfer may be active for each port pair (source and sink) providing there is no contention for a destination element. As new transfers are initiated, existing transfers are not interrupted. Sufficient handshaking and buffering is provided so the asynchronous communication between functional elements with different transfer rate capabilities can be accomplished [4]. Communication occurs in one direction in the DTN with each functional unit connected to both a sink and a source port.

Command Program Processor

The functional element in which command programs execute is the Command Program Processor (CPP). The CPP also is responsible for acting on requests for graph control, providing runtime system services and process control. The CPP is implemented using a standard Navy computer, the AN/UYK-44, embedded within the EMSP [4].

Global Memory

The main storage units of the EMSP are its Global Memories (GM). An EMSP may be configured with one or more independent GMs, each on a pair of DTN ports. Directly implemented within the GMs are the queues, graph variables and instruction streams. For this reason, the GMs are responsible for providing dynamic memory management, allocation of storage as queue data are written and deallocation of storage as queue data are consumed. As stated above, requests for data instruction are answered with Accept instructions containing data. Global memories also maintain information, such as the number of words in a queue and the value of the threshold. These are used to determine if a queue has gone over threshold. Each time a queue is written or consumed, the relationship of the resulting number of data elements to the threshold is checked. If the

appropriate conditions are met, the queue is reported to the Scheduler as Queue Over Threshold (QOT).

Input Output Processor

Input and output procedures from the EMSP are initialized by command programs and are executed by an Input/Output Processor (IOP). Once started, an IOP performing input will receive data from an external channel, process the data, and place it on one or more queues which serve as inputs to a graph. Similarly, an IOP performing output removes data from an output queue and transmits it over an external channel. Interface protocols associated with communication with external devices are handled by the IOP. Internal memories are used to help assure efficient data transfer with synchronization of external devices and buffering of data.

Arithmetic Processor

The functional element that implements the primitives is the arithmetic processor (AP). As with GMs and IOPs, there may be many APs in an EMSP configuration, each operating independently. Primitive algorithms are implemented as microcode programs, which execute in an AP; and all APs in an EMSP configuration are loaded with the microcode for all primitives to be used in the application graph. Thus, any AP which is provided with the unique information describing a particular node instance can be the processing resource

used to execute the primitive [4]. An Instruction stream (IS), stored in a GM at start time, is used to implement the ECOS Primitive Interface Procedure. The IS is provided to an AP by the GM at the direction of the Scheduler, when the node is eligible for execution.

Three phases make up a node execution. The setup phase occurs while the IS is executing in the AP's control unit to calculate NEP amounts (if necessary), and to read control and signal data into the local operand memories. With the completion of the setup phase, the node execution begins with the execution of the primitive in the node's arithmetic unit. The last phase, the breakdown phase, begins with the completion of the primitive execution. Here, the IS executes to write to output queues and consume from input queues. The internal structure of the AP, with independent control and arithmetic units and split local memories, allows data transfers between the AP and GMs to be concurrent with primitive execution. As a result, an AP may be servicing three nodes simultaneously [4].

Scheduler

As a graph executes, the heart of the EMSP operation is the Scheduler. Its main responsibility is to schedule the execution of the nodes on the processing resources. Several databases used to support the dataflow operation of the architecture are also maintained by the Scheduler. The first

of these is the Queue to Node Map, which stores the topology of all executing graphs by maintaining a list of nodes connected to the head and tail of each queue. A Node Characteristics Table is used to store necessary information on each node, including the number of input queues, the type of processor required for execution (AP or IOP), the identification of the GM containing the node's IS, and a dynamic count of the number of input queues which are yet to be over threshold. A third database is the Free Functional Element List which maintains a record of the functional elements which are currently available. A Ready Node List is also used to maintain a list of the nodes that are eligible for execution, but for which no processing resource is available.

CHAPTER III

THE EMSP TIMING SIMULATOR

The EMSP Timing Simulator, created by Marilyn Aiken [1], is designed to provide a tool to evaluate the Enhanced Modular Signal Processor architecture and any proposed modifications. The basic premise of the simulator is to simulate the operation of the various functional elements of the EMSP architecture based on their individual timing restraints. This enables the user to experiment with various hardware configurations for a given signal processing graph.

The timing simulator basically is an event driven program. Each activity; i.e., graph instruction, needed to simulate the operation of the EMSP architecture is created with a start-time relative to a simulated clock. The timing of each activity is calculated based on published time restraints that are part of the architectural design of the EMSP. For the most part, these are static times, but for some activities, especially those related to data transfer, these times must be calculated.

Graph execution instructions are instructions passed between functional elements. A Queue Over Threshold (QOT) instruction serves as an example. When a Global Memory detects that a queue has gone over threshold, it sends a QOT

instruction out over the Control Bus to the Scheduler. The Scheduler then takes an appropriate action. Such an action may simply be updating the dynamic queue information for the nodes involved, or the scheduling of a node to execute if the conditions for that node to execute have been met. If the functional element for which a graph instruction is destined is busy, or the instruction has an occurrence time greater than the simulator clock, the instruction is placed on the central event list. The Control Bus request table and the Data Transfer request table contain graph instructions that are requesting action from a particular functional element. Each functional element scans these tables to determine whether another functional element is requesting action. The ready list contains a list of nodes reported by the Scheduler to be ready to execute.

The node data structure records the types and identifiers for each of the node inputs and outputs. The value of each graph instantiation parameter is stored as well as the Global Memory identification number and element size for each graph variable and queue. 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. Taken as a whole, the graph topology is defined across these various structures.

Node Execution Cycle

The graph execution process follows a defined deterministic method that implements the dataflow methodology. The following explanation shows the steps that must occur for the execution of one node. This is an ongoing asynchronous methodology, and once commenced the operation of each functional element is based on the exchange of instructions and the availability of data. Figure 9 (Appendix A) assists in the understanding of this process.

First, as sensor data is input to the graph, via the Input/Output Processor (IOP), it is written to the input queues. Write Queue instructions are generated by the IOP as needed. When the Global Memory detects that the queue has gone over threshold, a Queue Over Threshold (QOT) instruction is sent to the Scheduler.

When the Scheduler has received a QOT instruction, the number of conditions necessary for the node involved is decremented. When all of the conditions have been satisfied, then the node is considered ready to execute. A Send Instruction Stream (SIS) instruction is sent to the Global Memory containing the instruction stream for the ready node's primitive.

The Global Memory builds an Accept Instruction Stream (AIS) which contains the primitive instruction stream and

sends it across the Data Transfer Network (DTN) to the Arithmetic Processor (AP) designated by the Scheduler.

Based on the particular primitive, the AP enters the setup phase of its operation and sends appropriate Request Queue (RQ) or Request Graph Variable (RGV) instructions to the Global Memory. The Global Memory responds with Accept Queue (AQ) or Accept Graph Variable (AGV) instructions containing the requested data.

Once all the required data have been received, the AP enters the execution phase. A Request For Instruction Stream (RFIS) instruction is sent to the Scheduler notifying it that it has completed the setup phase and is free to begin the setup on another node.

Finally, when the execution phase is completed, the results are written to the proper Global Memories with Write Queue (WQ) or Write Graph Variable (WGV) instructions. Once completed, Consume Queue (CQ) instructions are generated to remove data from the previous input queues. It should be noted that Write Queue instructions generated by the node completion may make subsequent node execution possible as queues go over threshold.

Input

Input to the simulator currently is provided either by user entry or by two configuration files. The first file contains the static graph topology. Each node is defined by

an identification number, an opcode mnemonic of the signal processing primitive to be executed, and a number of graph variables and queues which serve as inputs and outputs to the node. Queues also are defined as part of the node definition and include the threshold, read, consume, and valve amounts.

The second file contains the information needed to define the EMSP hardware and system configuration. This information contains information to define the size and number of data transfer networks. Each functional element is defined by type (Arithmetic Processor, Scheduler, etc.) and its connection to the data transfer network. This file also contains information to define the channels that serve as inputs and outputs to the simulator. This information includes the channel rates and the queue identifiers to which it is connected. This file also contains the memory configuration of the graph topology. That is, which node instruction streams, queues, and graph variables are stored in which Global Memories. Graph instantiation values are also contained in this file.

Output

Output from the simulator consists of a system configuration chart, a list of utilization figures for each functional element, node execution information, channel ex-

ecution information and queue information. An optional runtime execution graph can be provided.

Implementation of Functional Elements

A discussion of the implementation of each of the functional elements will be helpful in understanding the functionality of the simulator.

Arithmetic Processor

The procedure for implementing the Arithmetic Processor (AP) takes into account each of the three states of node execution: setup, execution, and breakdown. Separate timing variables effect the three timing states. The primitive execution time for the execution phase is calculated based on the opcode mnemonic. Graph instructions generated by the operation of the AP include Accept Instruction Stream (AIS), Send Instruction Stream (SIS), Request Queue (RQ), as well as others. The timings for these instructions will ultimately be accounted for as each instruction is processed. Instructions generated by the breakdown phase, such as Write Queue (WQ) and Consume Queue (CQ) are handled as is appropriate for that primitive.

Control Bus/Data Transfer Network

The procedures that implement the Control Bus (CBUS) and Data Transfer Network (DTN) are quite similar. The ba-

sis purpose of these procedures is to order the list of requests and to handle conflict for destination functional elements. The important point to remember is that scheduling of data transfers is based on the current clock time with rescheduling to a future time for conflict resolution. This applies either when concurrent instructions are seeking the same functional element, or when a functional element is already busy.

Global Memory

The Global Memory (GM) procedure simulates the noninterruptible nature of a true Global Memory. As an instruction is received by a GM, the central event list is checked to determine whether that GM is currently active. If it is active, then the instruction is rescheduled with a future event time.

The timings to handle GM instructions involving data transfer are functions of the number of words to be transferred. Once the timing has been calculated then the appropriate Accept Instruction Stream (AIS), Accept Queue (AQ), or Accept Graph Variable (AGV) instruction is placed on the Data Transfer Network. Updating queues is achieved by Write Queue (WQ) or Consume Queue (CQ) instructions with the sending of the appropriate Queue Over Threshold (QOT), Queue

Over Capacity (QOC), Queue Under Capacity (QUC), or Queue Under Threshold (QUC) instructions to the Scheduler over the Control Bus.

Scheduler

Like the Global Memory, the Scheduler procedure simulates the noninterruptible nature of the true Scheduler. An instruction reaching the Scheduler while it is busy is be rescheduled for a future event time. The Scheduler monitors the graph status by updating the status of each node, the number of conditions needed to execute each node, and the connections of the external channels to the graphs. In addition, the Scheduler maintains the free processor list and schedules nodes to execute based on the status of each node and the arithmetic processors. The event list is maintained as necessary.

CHAPTER IV

ENHANCEMENTS TO THE TIMING SIMULATOR OF THE ENHANCED MODULAR SIGNAL PROCESSOR

Simulator Input Procedures

The EMSP Timing Simulator, as it was designed originally, had only adequate input procedures with which to configure it. Input consisted of a mixed combination of user supplied input together with two relatively unformatted configuration files. In the first case, the user could enter all necessary input from the keyboard after which the simulator would execute. The disadvantage of this method is that no configuration files were created or used by the simulator. The second method utilizes only the two user created configuration files. These files were to be created by an editor with no interaction or error checking by the simulator.

The first file contained the information necessary to create the program graph topology for execution by the timing simulator. It contained such information as the node identification number, the primitive mnemonic indicative of the signal processing algorithm to be executed by that node, and other information describing the inputs and output to

each node. Examination of this file by a user yielded no useful information.

The second file contained hardware configuration information necessary to define each of the functional elements of the EMSP architecture. Other information in the file included graph instantiation parameters, memory placement information, queue capacities and primary values for graph variables.

In the end, this two file arrangement, while adequate to immediate needs, did not make the timing simulator particularly easy to operate and made creation of the configuration files extremely error prone.

The first part of this thesis project was to design and implement new procedures to create the configuration files for the simulator. This was undertaken with three goals in mind. First, to create a more acceptable method for creating the simulator configuration files. Second, to provide a method of interpreting the configuration files. And, finally, to implement a method to verify the connectivity of the graph topology input to the simulator.

The first two goals were realized by the creation of several new procedures which, while mirroring the sequences of the original input procedures, provide a much better environment with which to work.

The UNIX System V curses library routines were used throughout the simulator to provide split screen input. Split in half horizontally, the top half of the screen is used to prompt the user for the required keyboard input. The bottom half of the screen is used to echo the input in an interpretive way, as described above.

Nineteen new procedures in four files were created to make these changes. The first file, `newmain.c`, contains a new `main()` procedure for the simulator. In C programming, the function `main()` is always the first procedure called. This procedure replaces the `main()` procedure previously found in the file `main.c`. This new `main()` procedure is responsible for defining the curses environment and creating the split screen display. This procedure also is responsible for prompting the user for the names of the configuration files. This implementation alters the original design by creating three configuration files. The first file contains the graph topology information as it was originally designed. The second file contains only the EMSP hardware configuration information, as described above, with the third file holding the memory configuration and instantiation information. This change was made to make each file more specific to the information that it contains.

If any of the three file names supplied by the user does not exist, the user is given the option of reentering

that name, in case a typographical error occurred, or, entering the routines necessary to create that file. Once the three files exist, then the user is given the choice of executing the simulator with these three configuration files, or aborting the execution of the simulator. This allows the user the option of creating zero or more of the configuration files and executing the simulator at one sitting, or using the simulator only as a tool to create the configuration files for later use.

The second file, `config.c`, contains three procedures: `create_top()`, `create_config()`, and `create_mem()`. These procedures assist in the creation of the various configuration files. If a user wishes to create a new configuration file, then one of these procedures is called with the new file name as a parameter. The `create_top()` procedure is called for the creation of a new graph topology file; the `create_config()` procedure is called for the creation of a new hardware configuration file; and, the `create_mem()` procedure is likewise called for the creation of a new memory configuration file. Each of these procedures makes the appropriate system calls to create and open the new file with the appropriate error checking. Once the file has been successfully created, the appropriate querying procedure is called to prompt the user for the required keyboard input.

`Query.c`, the third file, is an adapted version of the original `read.c` file. This file contains the same

procedures as `read.c`, renamed and adapted for use in the `curses` environment. All of the output necessary to update the split screen display, including echoing the user input, generating the interpretive lower screen, and performing the file output to the configuration files, is done in these procedures. Some preliminary initialization of the simulator data structures is also done to insure that all of the data necessary to operate the simulator is asked for.

The fourth file is `emsp.c`. This file contains a routine called `emsp()` which is another new version of the original `main()`. This routine is used to start the execution of the timing simulator, taking into account the new `curses` environment, and the use of three configuration files, instead of two. Routines in the file `readm.c`, are called to read the configuration files and initialize the timing simulator.

Other changes were made throughout the simulator to standardize its use in the `curses` environment. This is particularly true in the `support.c` file which contains all of the output procedures. No formatting changes were made from the original, apart from the use of appropriate function calls for use with `curses`.

Two more procedures were created to determine the connectivity of the graph topology. It is assumed that the original design did not include sufficient error checking in this area because it was not intended for the configuration

files to be user generated; rather, they would be provided by some other facility. To overcome this, the procedures `matrix()` and `dfs()` were created in the file `matrix.c`. These procedures utilize simple data structure techniques to verify the connectivity of directed graphs, such as dataflow graphs. The procedure `matrix()` utilizes the `ptr_queue` structure of the simulator to build an adjacency matrix. Each queue of the simulator is examined and an entry in the matrix is made for each node connected tail and head of each queue. To understand this, it is important to remember that each queue in a dataflow graph represents a directed arc. Queues are first-in-first-out data structures with data placed on the tail and data removed from the head of the queue. Determining whether each node is connected to the graph is the responsibility of the `dfs()` procedure. This routine is a simple recursive depth-first-search algorithm. Using the adjacency matrix created by the `matrix()` routine, each node is visited based on the entries in the matrix. If any nodes have not been visited when the `dfs()` procedure finishes, then those nodes are considered disconnected from the graph. The simulator will abort under this condition as this is not considered a safe condition.

Configuration Feasibility Determination

The second part of this project was to develop a means for determining whether a particular hardware and memory

configuration is appropriate to execute a given graph topology. An appropriate configuration is one in which the graph could execute without any serious bottlenecks or loss of data. These problems are indicated by the overflow of any of the graph queues. If there is an insufficient number of the appropriate functional elements, namely Global Memories and Arithmetic Processors, then the timing dependencies inherent in the node primitives and data transfers reduce the number of node executions in a given period of time. If the processor cannot keep up with the data input data rates, then the queues begin to go over capacity with data arrival.

The detection of this Queue Over Capacity condition is an important process and one of the purposes for the original design of the timing simulator. Unfortunately, this concept was overlooked in the final version of the simulator. This oversight is corrected easily. A procedure, called `unfit()`, was created to be called whenever a Queue Over Capacity (QOC) situation is detected. Appropriate calls are in the Global Memory procedure to detect QOC in the internal queues, and in the Scheduler procedure to handle QOC in the channel queues. This procedure is found in the file `unfit.c` and, when called, generates output noting that state of each node and queue in the simulator at the detection time of the Queue Over Capacity. The normal output procedures are also called to generate the

utilization figures for the execution of the graph up to that point. This procedure also handles the ending of the curses environment and the closing of the configuration files prior to aborting the simulator.

Dynamic Graph State Table

The final part of this project involves the design of a graph state table at each simulated clock time interval during the execution of the timing simulator. In other words, the state of each node and its queues are shown throughout the execution of the graph. This output provides the user more information about the nature of the signal processing graph executing on the timing simulator. Such information is helpful in judging the parallelism of the signal processing algorithm and the efficiency of the hardware configuration.

The information presented includes the current simulated time, and the following information about each node of the graph:

- a. Node Identification Number.
- b. Primitive mnemonic.
- c. Current state of each node. (i.e. Busy, Idle)
- d. Current state of each input queue.
(i.e. QOT, QOC, QUT, QUC, etc.)
- e. Percentage of capacity and threshold for each queue.

In terms of the execution of dataflow graphs, a node of the graph is considered to be idle until all of the

conditions necessary for the node to execute have been met. This generally means that all of the input queues have gone over threshold. At that point the node is considered to be executing.

In terms of the EMSP architecture, a node can be busy for a number of reasons, each related to the status of the various functional elements. When a node has satisfied each of the conditions needed to fire, the Scheduler attempts to schedule the node for immediate execution. If no Arithmetic Processor (AP) is available, the node is placed on a list of ready nodes and then is scheduled on a first-come-first-served basis on the next available AP. This occurs provided that there are no other instances of that node currently executing. This is a necessary condition as this simulator was not designed to be a tagged token architecture and does not have the capability to handle multiple instances of a given node.

Once the node reaches a particular AP, it may have other periods of waiting related to the data transfers necessary during the setup and breakdown phases of the node execution. These waiting periods are time spent on the servicing lists for the Control Bus (CBUS) and the Data Transfer Network (DTN).

To avoid problems with multiple node instances, the simulator marks a node as busy from the point at which all

of the execution conditions are met, to the completion of the breakdown of the node execution.

For the purposes of the dynamic state output, any node appearing on the ready node list, the Scheduler's instruction list, the DTN waiting list, or the CBUS waiting list, is considered to be executing. Any node not on one of these lists is considered idle.

The design of the dynamic state output of the graph execution would be as follows:

(1) During each simulated clock cycle, traverse the instruction list, the ready list, the DTN waiting list, and the CBUS waiting list, marking in the state table each node appearing on one of these lists. Each of these lists is maintained as a singly linked list data structure of the following construction:

```

struct instruct {
    int time;
    int opcode;
    struct FE *sender;
    struct FE *receiver;
    int l_message;
    int nodeid;
    int message;
    struct queue *queue;
    struct instruct *next;
};

```

(2) For each node of the graph, determine the state of each of the input queues. It is not necessary to do the same for the output queues as they will be considered as inputs to other queues or as final output from the graph.

There are two different design approaches that can be taken, based on the preexisting data structures of the simulator. The first method involves four levels of indi-

rection (pointers) and fewer sequential array traversals. The second is much simpler in its use of pointers, but would require more searching for the same information. The advantage of one over the other depends on the relative sizes of the graph topologies used on the simulator.

The first method would utilize four interlinked C structures. The first structure is an array of pointers where each element points to a node of the graph. The structure is defined as follows:

```
struct ptr_node {
    int nodeid;
    struct node *n_ptr;
} ptr_node[MAX_NODES];
```

The configuration of each node, as well as some static and dynamic information concerning the node is given in the node structure:

```
struct node {
    short nodeid;
    short opcode;
    short num_inputs;
    short NOC;
    short prior;
    short GM;
    short type;
    short suspended;
    short firings;
    short exec_time;
    short nodesize;
    struct var i_var[I_MAX];
    struct var o_var[O_MAX];
};
```

For this discussion, the important part of this structure is the "struct var i_var[I_MAX]" definition. This is

an array of structures used to store information on each of the node's inputs. Each of the inputs, whether it is a Graph Variable (GV), a Graph Instantiation Parameter (GIP), or a Queue, is recorded in this structure:

```

struct var {
    short type;
    short GM;
    short size;
    short value;
    struct queue *q_ptr;
};

```

To reach the information about each queue needed by the dynamic state output, a traversal of the `i_var` array is necessary. For each input of type QUEUE, the queue structure pointed to by `q_ptr` would be accessed for the necessary information. The queue structure is defined as follows:

```

struct queue {
    int queueid;
    int head_node;
    int tail_node;
    int threshold;
    int consume;
    int read;
    int capacity;
    int sizeof_data;
    int GM;
    int data_items;
    int status;
    int produce;
};

```

An example of the indirection needed to reference any of this information would therefore require something of the form:

```
ptr_node[i].n_ptr->i_var[j].q_ptr->capacity
```

with sequential traversals needed for the `ptr_node` and the `i_var` arrays.

The second method requires simpler data structures to end up with the same queue data arrived at above. Like the `ptr_node` structure above, a `ptr_queue` structure exists to give immediate access to the queue information:

```
struct ptr_queue {
    int queueid;
    struct queue *q_ptr;
} ptr_queue[MAX_NODES];
```

The "`struct queue *q_ptr`" definition gives access to the same queue structure described above. The procedure for getting the information needed for the dynamic state output requires that for each node in the graph, a traversal of the `ptr_queue` array is necessary. Each queue is checked to find those queues defined with the current node as the head node for that queue. Data is removed from the heads of queues, therefore a queue is an input queue to the node at its head.

Because the `q_ptr` array would have to be sequentially traversed for each node in the graph, this method may be more time consuming than the first method. However, the in-direction needed is much simpler:

```
ptr_queue.q_ptr->capacity
```

As information for each node is collected, the dynamic state output would be constructed to generate the desired output. Appendix B contains sample output of the dynamic state of the graph during execution.

CHAPTER V

CONCLUSIONS

The purpose of this project is to study the dataflow architecture called the Enhanced Modular Signal Processor, and to make enhancements to EMSP Timing Simulator created by Marilyn Aiken [1].

The EMSP Timing Simulator is a tool for testing various configurations of the EMSP hardware to determine its suitability for particular signal processing graphs. Suitability is determined not by any computed results of the signal processing graph, but by the ability of the hardware configuration to execute the graph within the timing restraints of the graph operations.

The enhancement part of this project develops the procedures necessary to correct some oversights in the original timing simulator. The first part of this three part project created new input procedures to create the needed configuration files and to test the connectivity of the input graph. The second part enhanced the simulator by detecting the internal conditions needed to determine whether a particular hardware and memory configuration is insufficient of the executing graph. The third part of the project created the procedure needed to generate a dynamic graph state table.

Such a table shows the state of the executing graph at each tick of the simulated clock. The state of each node and its input queues are shown. This information would be useful to the application programmer to determine bottlenecks and levels of parallelism in the signal processing graph.

Suggestions for Future Work

As a tool for the study of the Enhanced Modular Signal Processor, the Timing Simulator is a good beginning. Possible future work on the simulator must address the limitation on the size of the graph topologies that can be simulated. Conclusive data on the operation of this architecture is impossible without adequate test topologies.

The configuration file concept is still not adequate for easy use by the user. Redesigning the formats of each of the files with a new parsing method would greatly improve the usability of the simulator.

SELECTED BIBLIOGRAPHY

- (1) Aiken, Marilyn O., Enhanced Modular Signal Processor Timing Simulator, M.S. Thesis, Oklahoma State University, May 1987.
- (2) Baer, J., Computer Systems Architecture, Computer Science Press, Inc., 1980.
- (3) Brown, N. H., "The EMSP Dataflow Computer", Proceedings of the 17th Hawaii International Conference System Sciences, Honolulu, Hawaii, January, 1984.
- (4) Bloch, F. H., "The Enhanced Modular Signal Processor," Proceedings of the Seventeenth Hawaii International Conference on System Sciences, Vol. 17, January, 1984.
- (5) Davis, Alan L., and Robert M. Keller. "Data Flow Program Graphs," Computer, February, 1982.
- (6) Dennis, Jack B. "Data Flow Supercomputers," Computer, November, 1980.
- (7) Dennis, Jack B., and David P. Misunas. "A Preliminary Architecture for a Basic Data-Flow Processor," The 2nd Annual Symposium on Computer Architecture, January 20-22, 1975.
- (8) Dennis, Jack B., Joseph Stoy, and Bhaskar Guharoy. "VIM: An Experimental Multi-User System Supporting Functional Programming," Proceedings of the International Workshop on High Level Computer Architectures, 1984.
- (9) ECOS Tutorial: Preliminary, April 26, 1985.
- (10) 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, 1984.

- (11) Enhanced Modular Signal Processor (EMSP) Primitive Analysis Specification, CDRL C130, February 1985, prepared for the Naval Sea Systems Command, PMS412 by AT&T Bell Laboratories on behalf of AT&T Technologies N00024-81-C-7318.
- (12) Gostelow, Kim P., and Robert E. Thomas. "A view of dataflow", Proceedings of the AFIPS Conference, 1979. Vol. 48.
- (13) Hartimo, 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.
- (14) Hogenauer, Eugene B., Richard F. Newbold, and Yul J. Inn. "DDSP--A Data Flow Computer for Signal Processing," 1982 International Conference on Parallel Processing.
- (15) Hwang, K. and F. A. Briggs, Computer Architecture and Parallel Processing, McGraw-Hill Book Company, 1984.
- (16) Karp, R. M. and R. E. Miller, "Properties of a Model for Parallel Conventions: Determinancy, Termination, Queuing," SIAM Journal of Applied Mathematics, Vol. 14, No. 11, November, 1966.
- (17) Long, A. N., and S. A. Thoreson, "Operation of the Enhanced Modular Signal Processor," Oklahoma State University Technical Report, OSU-CIS-TR-87-05.
- (18) Rabiner, Lawrence R. and Bernard Gold. Theory and Application of Digital Signal Processing. Prentice-Hall, Inc., 1975.
- (19) Requa, Joseph E., and James R. McGraw, "The Piecewise Data Flow Architecture: Architectural Concepts", IEEE Transactions on Computers, Vol. C-32, No. 5, May 1983.
- (20) Srini, Vason P., "A Fault-Tolerant Dataflow System," Computer, March 1985.
- (21) Thoreson, S. A., and A. N. Long, "A Feasibility Study of a Memory Hierarchy in a Data Flow Environment," Oklahoma State University Technical Report, OSU-CIS-TR-85-01.
- (22) Treleaven, Philip C., "Exploiting Program Concurrency in Computing Systems," Computer, Vol. 12, January 1975.

- (23) Treleaven, Philip C., David R. Brownbridge, and Richard P. Hopkins, "Data-Driven and Demand-Driven Computer Architecture", Computing Surveys, Vol. 14, No. 1, March 1982.
- (24) Treleaven Philip C., Richard P. Hopkins, and Paul W. Rautenback, "Combining Data Flow and Control Flow Computing," The Computer Journal, Vol. 25, No. 2, February 1982.
- (25) Veen, Arthur H. "Dataflow Machine Architectures", ACM Computing Surveys, Vol. 18, No. 4, December 1986.
- (26) Watson, Ian, and John Gurd, "A Practical Data Flow Computer," Computer, January 1982.
- (27) Wong, F. S., and M. R. Ito, "A Large-Scale Data-Flow Computer for Parallel Signal Processing," Circuits and Computers, 1982.
- (28) Wu, Y. S., "A Common Operational Software (ACOS) Approach to a Signal Processing Development System: U.S. Naval Research Laboratory, ICASSP83.

APPENDICES

APPENDIX A
FIGURES

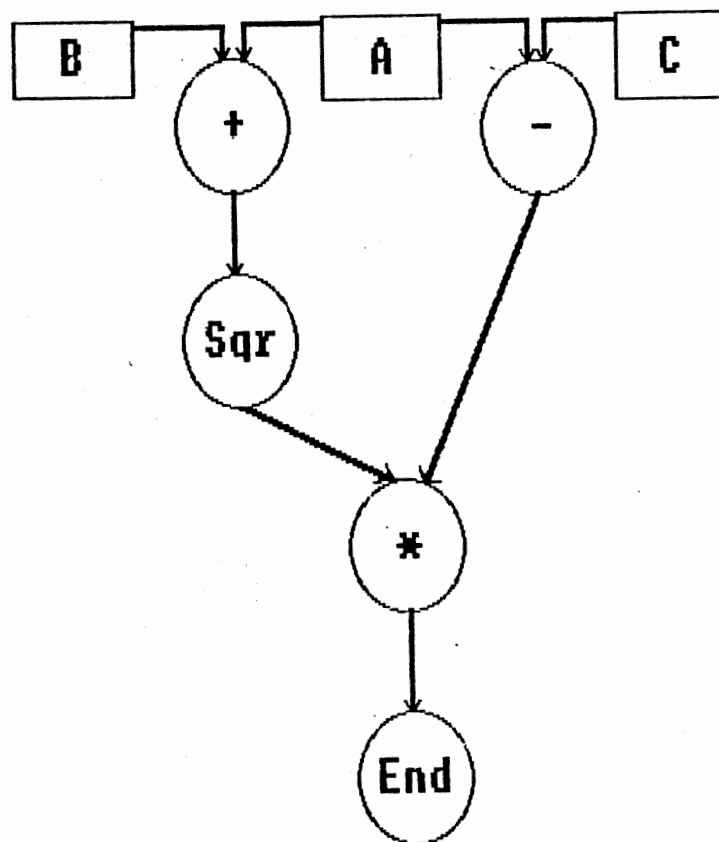
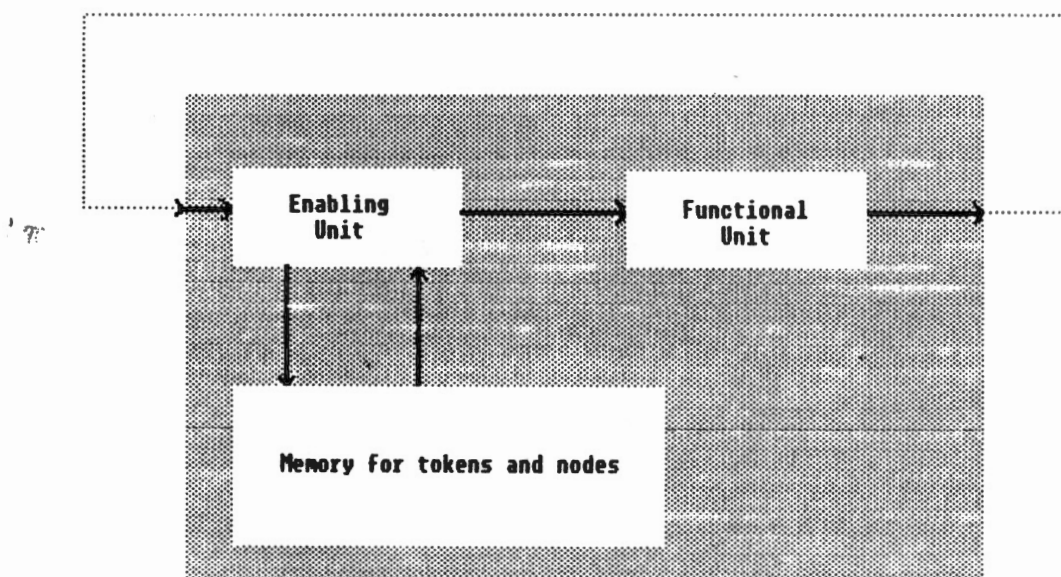
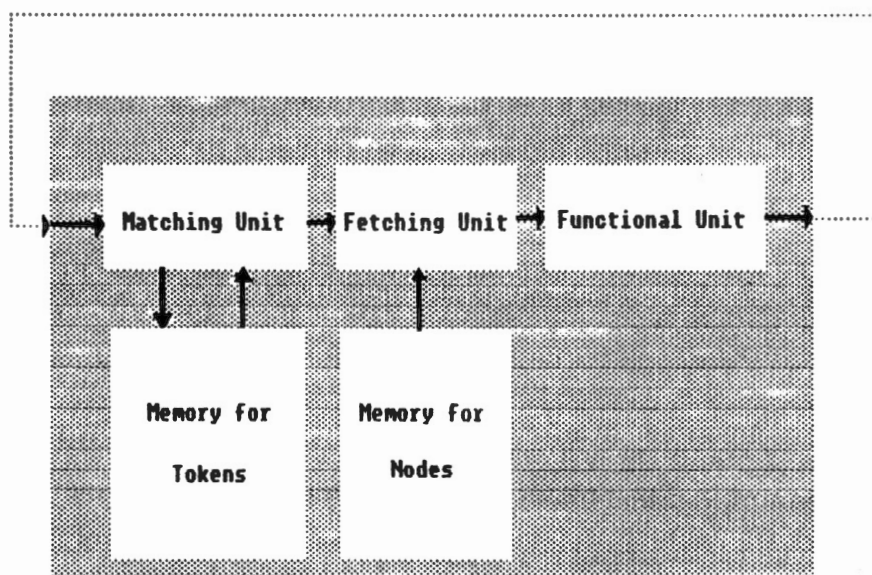


Figure 1. Simple Dataflow Graph

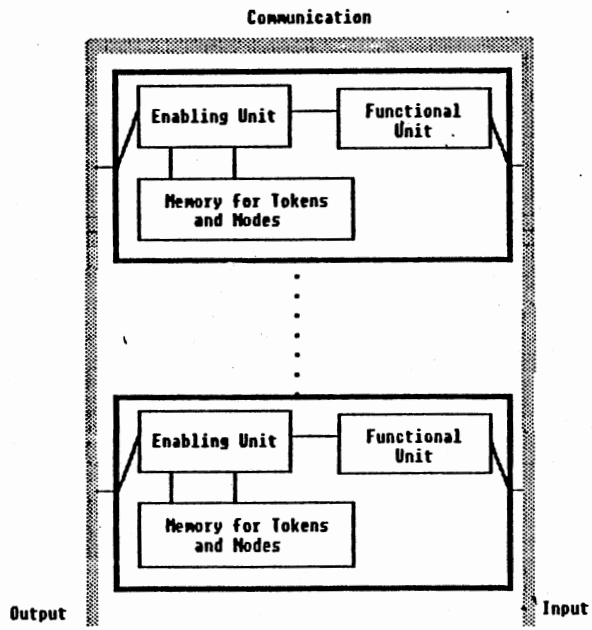


(a) Simple Dataflow Processing Element

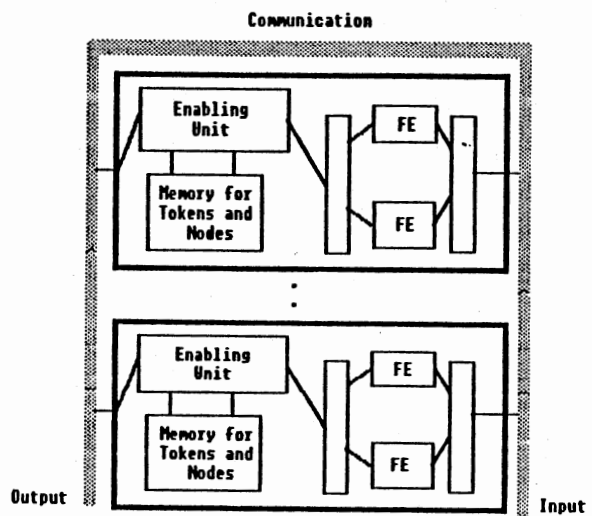


(b) Tagged-token Dataflow Processing Element

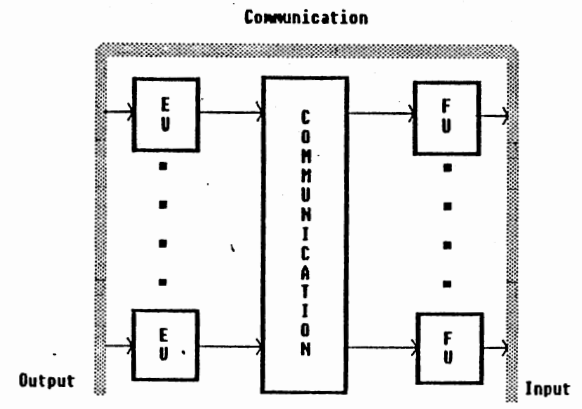
Figure 2. Fundamental Functions of Dataflow Processing Elements



(a) One-level Dataflow Machine



(b) Two-level Dataflow Machine



(c) Two-stage Dataflow Machine

Figure 3. Basic Dataflow Machine Structures

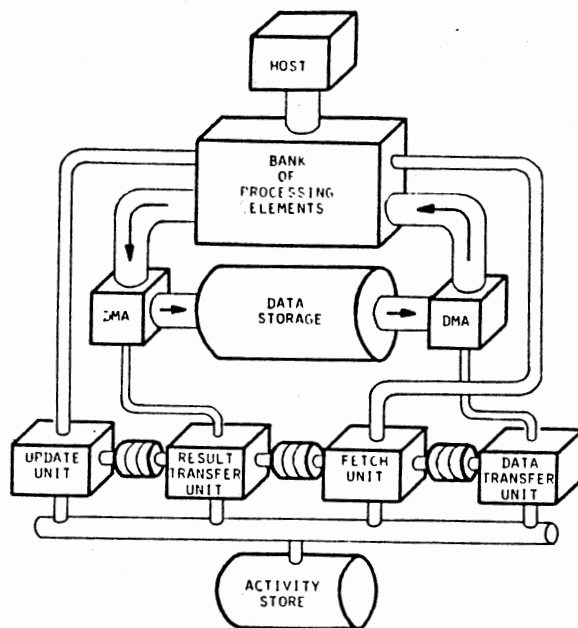


Figure 4. Diagram of the DFSP Architecture
(reproduced from [13])

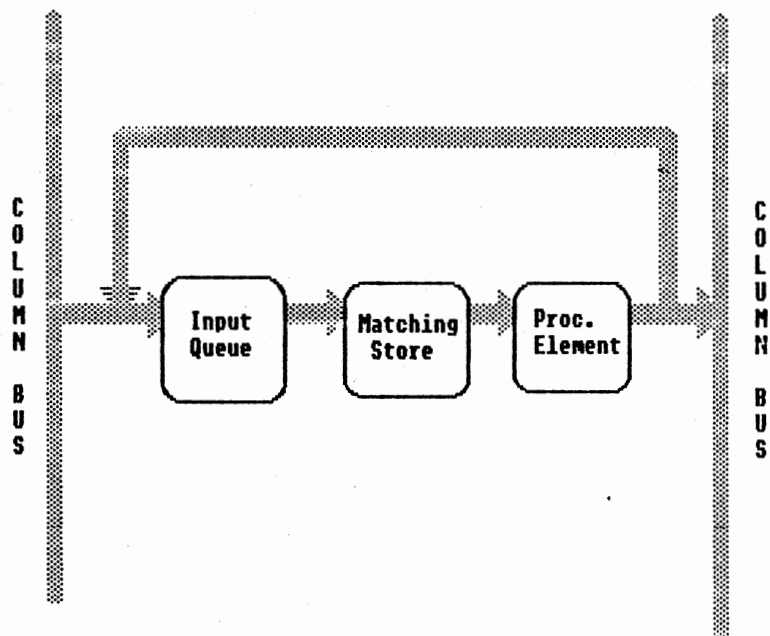


Figure 5. Diagram of the DDSP Architecture

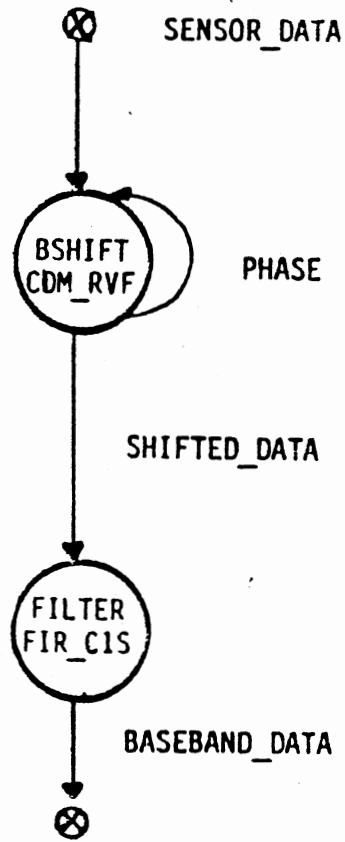
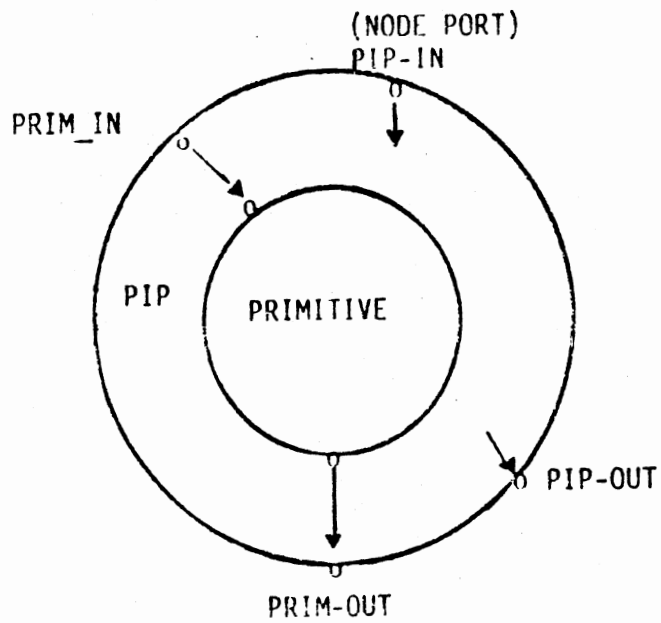


Figure 6. ECOS Sample Graph Topology
(reproduced from [9])



PORTS
 PIP-IN
 PIP-OUT
 PRIM-IN
 PRIM-OUT

Figure 7. ECOS Node Construction Diagram
 (reproduced from [9])

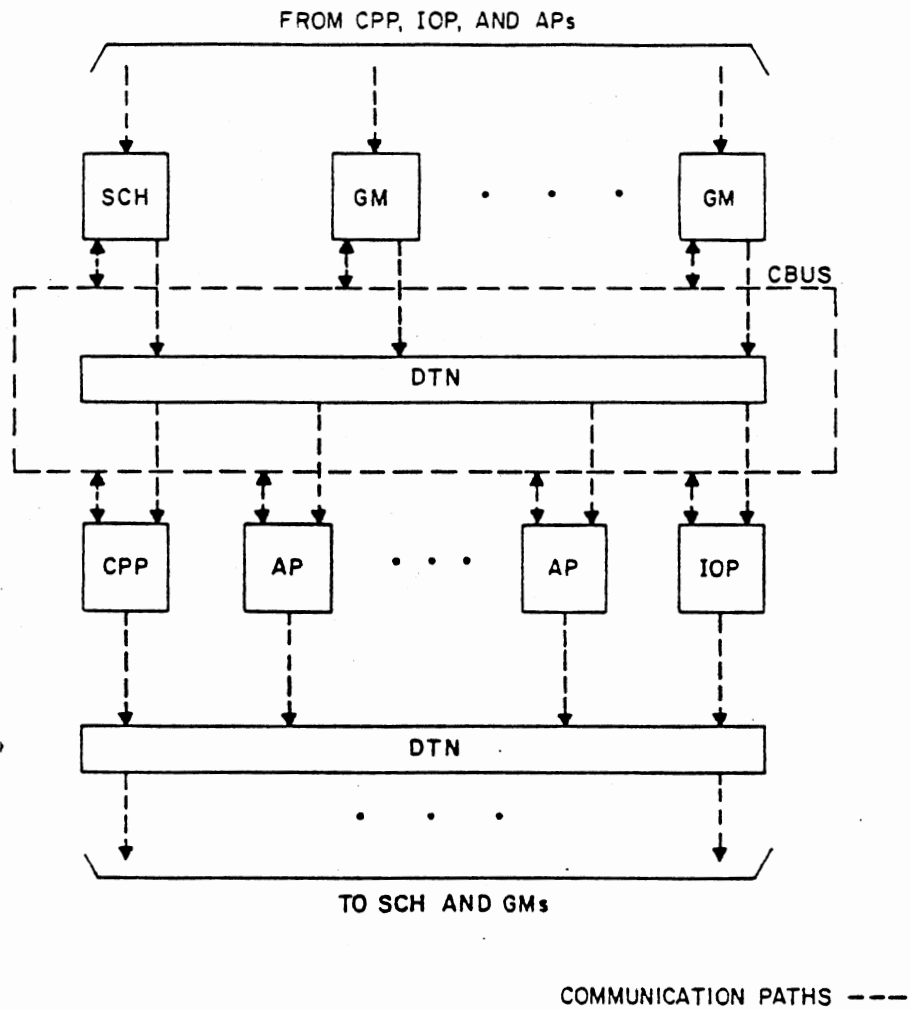


Figure 8. Diagram of the EMSP Architecture
(reproduced from [17])

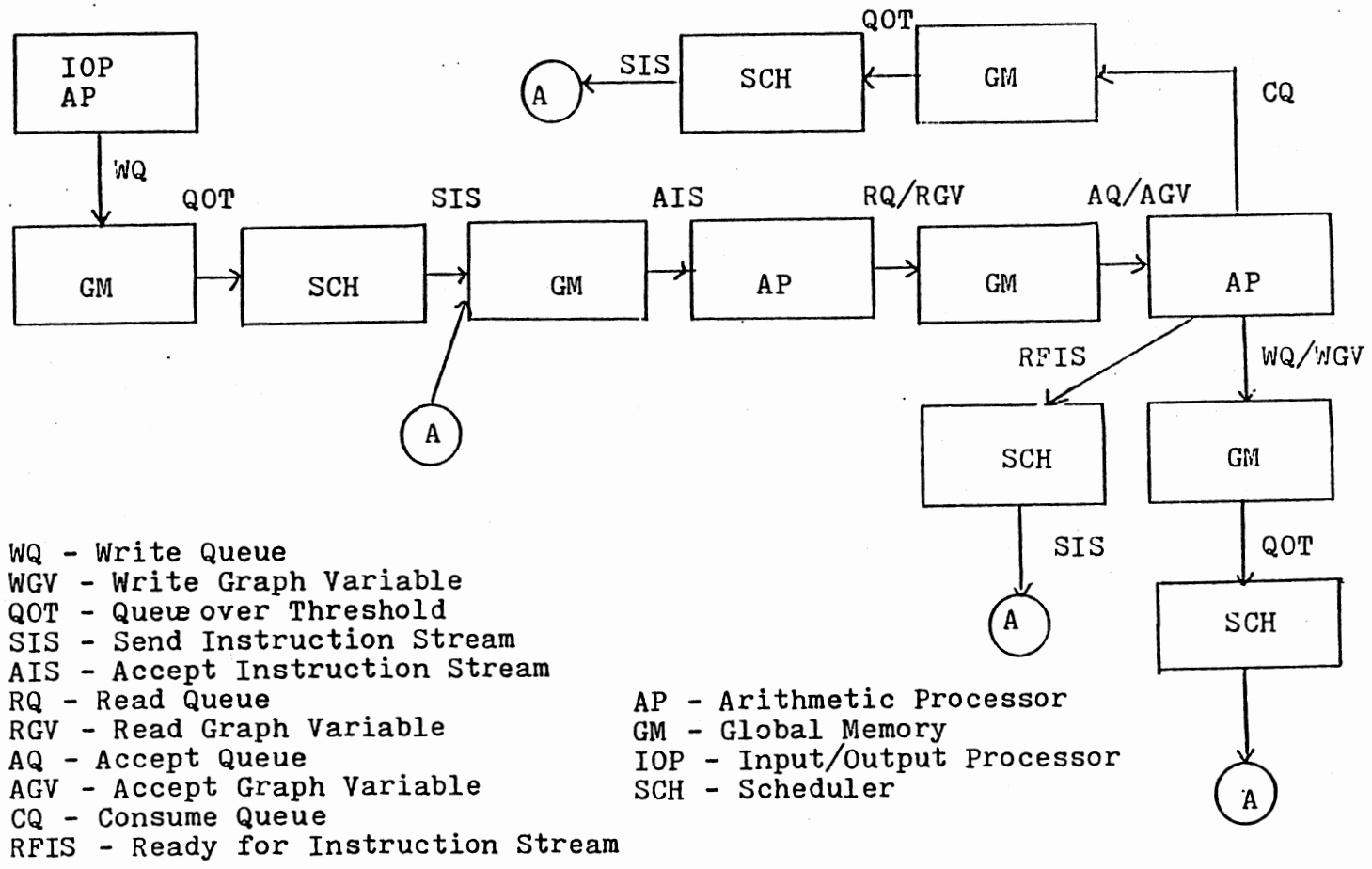


Figure 9. Diagram of the EMSP Node Execution Cycle (reproduced from [1])

APPENDIX B
SAMPLE COMPUTER OUTPUT

EMSP Timing Simulator

Enter the name of the graph topology file
to use or create: T1

The file: T1, does not exist.
Do you wish to:

1. Create this file.
 2. Reenter
-

How many nodes in this graph topology? 3

Enter the node identification number for node 0: 1

Enter the opcode mnemonic for this node: VOR_SQR

This node requires two inputs.

Enter the input type for input #0:
(GIP=-4, GV=-3, QUEUE=-5): -3

Enter the Graph Variable Identification Number: 1

Enter the input type for input #1:
(GIP=-4, GV=-3, QUEUE=-5): -5

Enter the queue id: 1
Enter the threshold: -1
Enter the consume amount: -1
Enter the read amount: -1

This node requires one outputs.
Enter output type for output #0:
(GV=-3, QUEUE=-5): -5

Enter the queue id: 2
Enter the valve amount for output queue: -1

.
.
.

Sample Output for Graph Topology
Input Procedure

Enter the name of the EMSP configuration file
to use or create: T2

The file: T1, does not exist.
Do you wish to: 1

1. Create this file.
2. Reenter

Enter the number of data transfer networkds [1/2]: 2

Enter the switch size for DTN[0]: 16

Enter the switch size for DTN[1]: 8

Number of functional elements to create: 5

Enter the Functional Element ID: 1

Enter the type of Functional Element,
(APP = 0, CPP = 4, GM = 1, IOP = 3, SCH = 2): 2

Enter which DTN the concentrator is on:

Enter which concentrator on DTN:

Enter which element on concentrator:

Enter which DTN the distributor is on [0/1]:

Enter which distributor on DTN:

Enter which element on distributor:

.
. .
. . .

Sample Output for Hardware Configuration
Input Procedure

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	55
2	GM	45
3	AP	32
4	IOP	1
5	GM	15

NODE EXECUTION INFORMATION

NODEID	OPCODE	NODE FIRINGS
1	14	6
2	28	5
3	25	4

CHANNEL EXECUTION INFORMATION

CHANNEL ID	CHANNEL FIRINGS
1	26
2	11
3	4

QUEUE EXECUTION INFORMATION

QUEUE ID	DATA ITEMS	HEAD NODE	TAIL NODE
1	100	1	1
2	10	3	1
3	60	2	2
4	14	3	2
5	0	3	3

Sample Simulator Output

Simulator time: 12300

Node: 1
Status: Busy

Input Queues

Queue:	1	Number of items on queue:	27
Capacity:	100	Percentage of capacity:	27
Threshold:	50	Percentage of threshold:	54
Read:	14	Consume:	14
Valve:	-1		

Queue:	3	Number of items on queue:	49
Capacity:	100	Percentage of capacity:	49
Threshold:	50	Percentage of threshold:	98
Read:	50	Consume:	50
Valve:	1		

.
. .
. .

Sample Dynamic Graph Output

VITA ²

Steven C. Nelson

Candidate for the Degree of

Master of Science

Thesis: ENHANCEMENTS TO THE TIMING SIMULATOR OF THE
ENHANCED MODULAR SIGNAL PROCESSOR

Major Field: Computing and Information Sciences

Biographical:

Personal Data: Born in Hobbs, New Mexico, February 25,
1961 the son of Philip E. and Mary M. Nelson

Education: Graduated from Hobbs High School, Hobbs, New
Mexico, in May 1979; received Bachelor of Science
Degree in Geology from the Oklahoma State
University in December, 1983; completed
requirements for the Master of Science degree at
the Oklahoma State University in December, 1988.

Professional Experience: Lecturer, System Administrator,
and Teaching Assistant, Department of Computing and
Information Science, Oklahoma State University,
August 1984 to August 1988. Consultant, The
National Office of Kappa Kappa Psi/Tau Beta Sigma,
Stillwater, Oklahoma, December 1984 to August 1988.