MIT/LCS/TR-339

GENERIC SOFTWARE FOR EMULATING MULTIPROCESSOR ARCHITECTURES

Richard Mark Soley

This blank page was inserted to preserve pagination.

Generic Software for Emulating Multiprocessor Architectures

by

Richard Mark Soley

May, 1985

This research was supported in part by the Advanced Research Projects Agency under contract N00014-75-0661 and in part by various grants from the International Business Machines Corporation.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Laboratory for Computer Science

Cambridge Massachusetts

Generic Software for Emulating Multiprocessor Architectures

by Richard Mark Soley

Arvind, Associate Professor of Computer Science and Engineering, Thesis Supervisor

Abstract

The expense of designing, prototyping, and testing a new computer architecture (particularly non-traditional supercomputer architectures, such as the dataflow machine) is enormous. The relative inflexibility of hardware to experimental changes increases the need to fully test a new architectural idea.

A software architecture for prototyping and testing single- and multiprocessor computer architectures is outlined. An overall design is discussed, noting the need for such a system, how it would be used to model and test various architectures, and possible implementation paths.

Various extensions and uses of such a generic prototyping system are also discussed, including extensions for modelling shared-resource systems, centrally synchronized systems, and distributed timing simulation systems. In addition, two uses of the system are presented, in particular the Tagged Token Dataflow Architecture, noting various methods in which such a machine may be simulated under a generic emulation package.

Keywords: computer architecture, emulation, simulation, dataflow

Acknowledgments

First and foremost, I must thank Professor Arvind, my thesis superviser, for his continued support and ideas for this project and thesis.

Many thanks go also to the other members of the Functional Languagues and Architectures group, past and present, who provided a stimulating environment in which to work and contributed greatly to the finished product this paper presents. Thanks particularly to Gregory Papadopoulos, Robert Iannucci, Paul Fuqua, and Poh Chuan Lim.

A million thanks to my (now much larger) family: Joseph, David, Tim, Jack, Zoltán, Maria Eugenia, Bernardo, and Clarita Inés.

Most important, I thank my wife and friend, Isabel Thérèse Szabó, for helping with the ideas and execution of this thesis, and for her infinite support and love.

to
Barbara Lee &
Micheline Miriam Monique Veronique

CONTENTS

FIGURES

Chapter 1

Introduction: Goals of a General Emulation Facility

1.1 The Multiprocessor Problem

The world is full of large, complex systems made up entirely of very small, simple components. Many of the interesting problems faced by scientists, engineers, and managers today are some of these large-scale computation-intensive problems, composed of highly interacting cellular parts. Unfortunately, the dimensions of these problems are generally such that no computer in existence today can begin to supply the needed computational power.

This does not eliminate the need for solutions to such problems, however. In a 1983 report,[1] the National Science Foundation noted a wide array of applications, both algorithmic and inherently heuristic (i.e., either NP-complete or with unknown solutions) which cannot be solved in a reasonable time using the fastest computational engines available today. Their summary included:

- Simulation of large-scale biological systems on the cellular molecular level.
- Simulation of chemical reactions and other materials and electrical research at the quantum mechanical level, where classical (and simpler) physical methods fail.
- Simulation of enormous fluid problems (i.e., global atmospheric modeling) in order to forecast fluid movement. Immediate applications would include accurate weather forecasting.
- Modeling of large-scale astrophysical events, such as supernovae and star births.

- Modeling of global economic systems.
- Simulation engines for large collections of logic-level or transistor-level electronic circuits of VLSI proportions.

The list can include applications which have much less algorithmic solutions, such as:

- Control of large and complex robotic systems.
- Diagnosis of machine and system malfunctions, and suggestions for corrective action.
- Diagnosis and treatment plans for human diseases.
- Knowledge-based expert artificial intelligence programs for more widely-based areas, including real-time machine understanding of connected human speech.

Each of these applications demands far more computational power and speed than the fastest and largest computers available today. The amazing speeds of today's supercomputers, such as the Cray-1S (cycle time: 12 nanoseconds, maximum sustained computation speed approximately 80 megaflops)[2] or the Cyber 205 (cycle time: 20 nanoseconds, about 50 megaflops)[3] pale in comparison to the combinatorial explosion of problem size promised by the problems above. In addition, the prohibitive cost of the supercomputers of today also limits their usefulness. Clearly some departure from the architectures of today is called for, as linear scaling of current computational power will not handle the problems of today, and even less those of tomorrow.

1.2 The Multiprocessor Solution

Since the early days of computer technology, the possibility of interconnection of multiple computers in order to form a larger, faster unit has been considered. Many architectural experiments, such as Illinois' Illiac IV system[4] and Carnigie Mellon's C.mmp[5] attempted to address this possibility without resorting to a great departure from the basic computer system architecture envisioned in 1946 by John von Neumann and his colleagues.[6]

Since the inception of the von Neumann flow-of-control design computer, single-processor design has pervaded all commercial, and most experimental, computational structures. Even multiprocessor designs have suffered the ill effects of the *von Neumann bottleneck,* the problem of limited access between processor (or *processors*) and memory.[7, 8, 9] Some modern supercomputers attempt to overcome this problem by means of extensive pipelining within a single processor;[10] though this avoids any problem of memory contention among multiple processors, it introduces either the need to re-code applications in a specially tuned pipelined/vectorized language, or to use high-powered program compilation techniques.[11]

This "brute force" approach to multiprocessing, including contended memory, poorly or too closely synchronized processors, and a reliance on programmer-specified parallelism has not even approached the need for a *scalable* multiprocessor. In a scalable system, doubling the number of processors in the system should double the system's performance; no machine to date has yet come close to this clearly useful goal.

What we need, then, are protocols for connecting multiple processors into a scalable system, and methods of programming which limit hardware idle time and minimize the need for programmer specification of parallelism. Research into exactly these issues is well under way; possible architectures range from the dataflow systems of Dennis[12] and Arvind[13] to Intel's Ada-based iAPX-432[14] to the Connection Machine[15] to Xerox' *Enterprise*[16] distributed computing environment, to M.I.T.'s Apiary system.[17] However, this increasing body of literature almost never reaches any consensus, except in one area; construction of an experimental supercomputer, they all agree, is costly.

Amassing the hardware and software necessary for implementing an entire multiprocessor machine can be expensive and time consuming, particularly for a one-shot statistics gathering evaluation of a proposed architecture. Projects generally spend well over one million dollars to get going,[1] and often need to completely overhaul their designs halfway through the work of implementation.

An anonymous corollary to Murphy's law goes: *"You will only be able to do it properly if you have to do it more than once."* Given the prohibitive construction costs for supercomputers, the need to redesign midstream, and the difficulty of performing mathematic analyses on proposed arcitectures before construction, we need an easy-to-use, high performance, low cost, and highly accessible tool for constructing, running, and testing

simulations (or emulations) of proposed architectures before a researcher commits to specific hardware or silicon (VLSI chips).

The remainder of this thesis outlines the goals, implementation, and uses of the *Multiprocessor Emulation Facility,* or M.E.F., under construction at the M.I.T. Laboratory for Computer Science.[18] The author designed the software for the M.E.F.; specific design decisions from that work will be described here.

1.3 Why Emulation?

Upon recognition of the complexities involved in attempting construction of a new computer architecture, the step usually taken is *simulation* of the architecture. In order to garner the important aspects of the target architecture, carefully written (and well timed) code is constructed to simulate the running new machine. The effects (under simulation) of different values for parameters such as interconnection protocols, buffer sizes, and the like are studied in order to ascertain the best structure for the machine to be built.

The approaches to system prototyping that will be discussed in this thesis are called *interpretation, emulation,* and *simulation.*

- *Interpretation,* the most general of the three, specifies that the prototype mirrors the real architecture at some specified level, such as instruction set or high-level language semantics.

- *Simulation* specifies the general scheme of executing some functional definition of a prototype (*functional* simulation) while gathering timing statistics expected while running the real machine (*timing* simulation).

- *Emulations* mimic the actions of a system functionally without any explicit gathering of timing information. Instead, the timing of the modeled architecture is implied by the scheduling of the functional units themselves.

Lacking the overhead of timing and synchronization to support timing, the emulation route shows the greatest promise of getting simulated results in the smallest amount of time; in addition, it is easy to map the architecture of a multiprocessor system onto a multiprocessor emulation system. The new path along which architecture prototypers would follow is

outlined in figure 1; instead of a simple path of design, simulation, and finally implementation, we add a new path, *emulation*, which can proceed in parallel with any simulation effort or *in lieu* of a simulation effort, depending on time constraints.

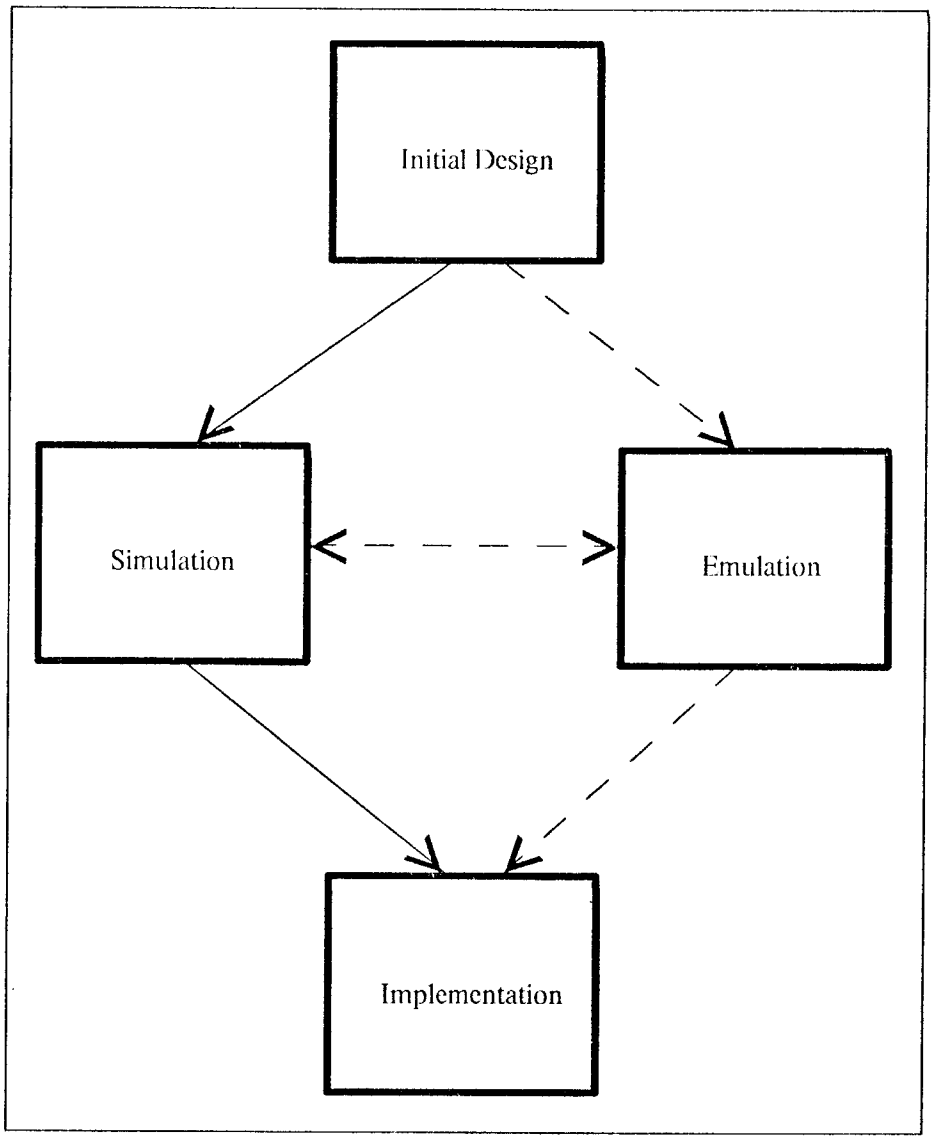

Figure 1: Architecture Design Path with Emulation

1.4 A General Emulation Facility

The point of the Multiprocessor Emulation Facility[18] (M.E.F.) is to allow multiprocessor architecture proposals that are still in the initial states of design to be evaluated *cheaply and quickly,* by emulation and simulation.

The basic abstraction of the M.E.F. must include a basic configurable multiprocessor structure, with general purpose processors at each node of a group of interconnection networks. Each *target element* of the proposed architecture is logically represented by software, with all functional aspects represented. The user of the M.E.F. will write a functional description of each different type of target element in his or her multiprocessor system, which will be used to detail the logical structure of the general machine.

We stress the phrase *target processing element,* or **T.E.,** in order to differentiate between T.E.'s (the computation elements of a proposed architecture to be emulated) and *physical processors,* or P.P.'s, which denote the processing elements of the multiprocessor system (M.E.F.) performing the emulation.

Software tools to provide usage of this abstract structure with minimal overhead must also be included, with primitives for the functional description of T.E. execution and interconnection. This includes all of the structure necessary to allow the M.E.F. user to ignore the physical interconnection structure of the M.E.F., and instead concentrate on the interconnection structure intended by his or her own research. Thus, a researcher using the M.E.F. could simulate widely variant coupling strategies, from a loosely-coupled packet-switched network to a tightly-coupled central memory multiprocessor, simply by using the interconnection primitives which interface directly with one of the available interconnection networks of the facility.

Another important goal of an M.E.F. is ease of use, via local- or wide-area networks. This goal establishes the intent of using an M.E.F. in a "batch mode" as a processor of various multiprocessor architecture proposals.

Reconfigurability and *partitionability,* or the ability to execute multiprocessor emulations using varying physical structures, are also necessary. Since we might wish to process several emulation experiments concurrently, or at least to continue experimentation while some of the facility is down for repair or regular maintenance, it is of prime importance for an M.E.F. to be insensitive to particular P.P. or interconnection availability. Thus, the

constructed M.E.F. should be capable of executing with any number of available P.P.'s using any number of available interconnection networks of the same or different hardware and protocol type. This ability also aids in a form of "fault tolerance," wherein experimentation can continue through a loss of some or all of the facility's equipment.

An M.E.F. design must itself be a multiprocessor in order to force systems architects to think in a distributed, multiprocessing way when he or she begins work on a new architecture. Without the multiprocessing frame of mind, time can be wasted in fruitless research into dead ends. Anecdotes about major systems that worked in simulation on single processors and then failed while running in a multiprocessor environment are many. A good example is a Multics experimental system which executed without flaw on a (single processor) development machine, but failed quickly while running on a (multiprocessor) user machine.[19]

We therefore add confusion to the differentiation of a multiprocessor system under simulation or emulation, and the M.E.F. multiprocessor architecture itself. When we speak of simulating or emulating a particular architecture using an M.E.F. as the construction tool,* there is a constant confusion between the (1) *architecture being tested* and the *architecture of the particular M.E.F. implementation* and (2) the *elements* of the architecture being tested and the elements of the real M.E.F. host system. Throughout this thesis we will use the terms "target machine" (or architecture) and "M.E.F." (or "host" or "physical" machine) to differentiate between the emulated and real multiprocessors, and "target element" and "physical processor" to differentiate between the computational elements of the two. Figure 2 displays this differentiation. Note that the phrase "target element" does not apply only to the purely *computational* elements of a target architecture, but to all elements (including perhaps memory, network nodes, etc.).

---

* Or, as the prototype MIT M.E.F. has been called, a "Multiprocessor Sandbox" or "Multiprocessor Playground."

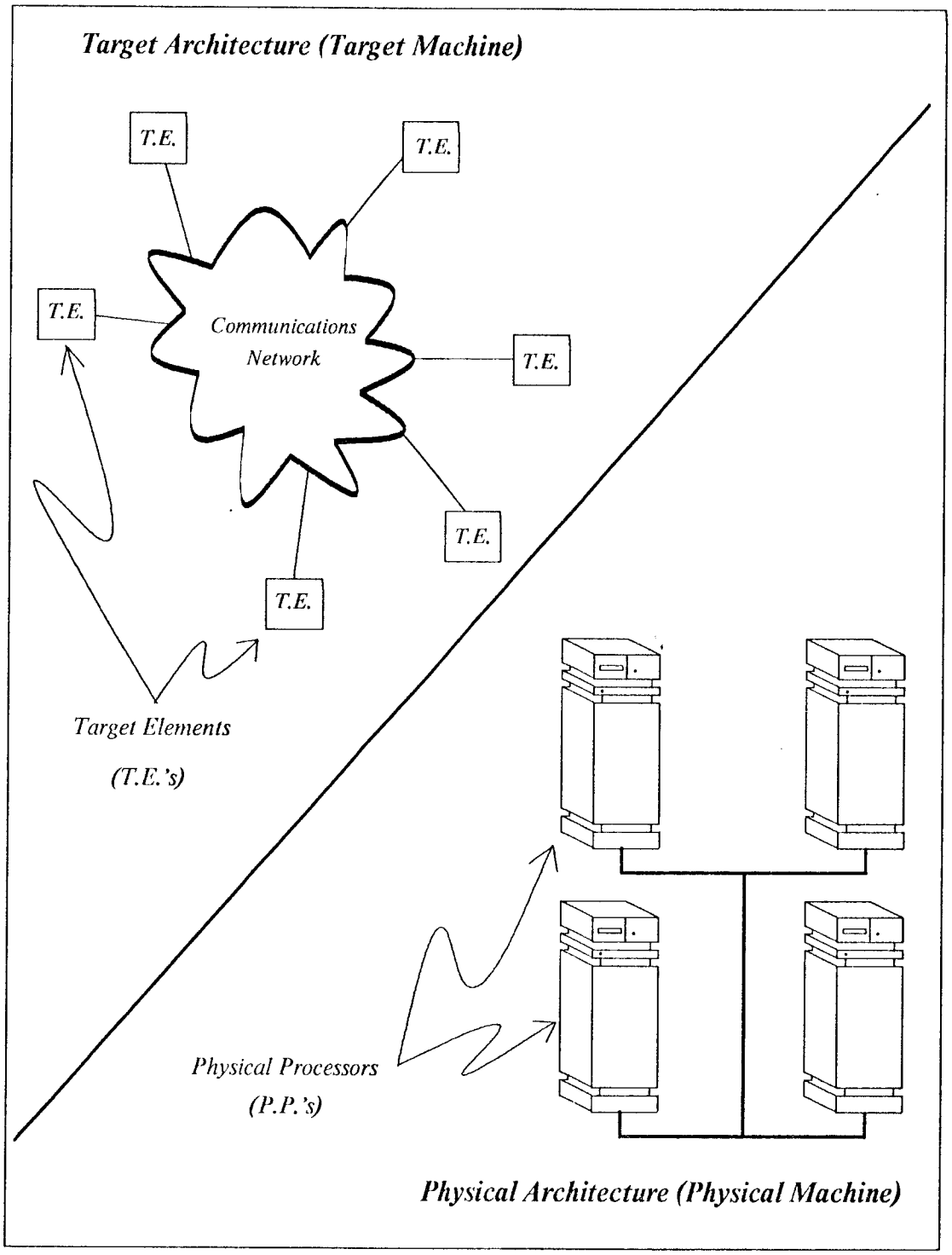

Figure 2: Target (Application) versus Physical (M.E.F.) Architectures

In order to gain more emulation speed without astronomical prices, and to force systems architects to begin thinking of computers in a distributed, multiprocessing way, an emulation facility should be designed to execute on many parallel general purpose processors, linked via a variety of communications media. To enable the construction of a parallel processor such as the M.E.F., there are several abilities required of the physical processing nodes which make up the computational power of the system. Since the M.E.F. must be able to implement multiple general purpose architectures, it must itself be implemented on relatively general purpose hardware. In addition, ease of programming (particularly rapid prototyping) in a well-integrated environment and support for multiple interprocessor communications architectures simultaneously are important, since they allow fast development of emulation experiments as well as room for experimentation with various communications media and protocols.

All the node power in the world in an M.E.F. would be useless without some interconnection scheme; a *multiprocessor* facility must have some linking mechanism. Specific needs for a sufficiently general purpose facility to support many different experiments are outlined in this section.

The first requirement for a useful facility interprocessor communications interface is that it be *multi-master.* A network that supports only a single master (such as a simple single-master bus structure) is useless in a facility that needs to support several simultaneous computations with free exchange of information initiated by any processor. Overhead to support a single-master structure, with bus master interrupts from processor wishing to initiate transfers, would probably render an M.E.F. unusable.

In addition, an M.E.F. should have a *multi-path* interface, to support multiple simultaneous communications between processors. Without this ability, processors that are relatively matched in performance could contend too much for the communications medium, causing a thrashing effect not unlike paging performance degradation on an overloaded timesharing computer.

Of course, an M.E.F. connection medium should be very high speed, with a communications bandwidth on the order of the available processor bandwidth. A mismatch in this regard could make the von Neumann bottleneck return in its usual form — communications overhead between (in this case) processing elements. Such an imposing

bottleneck could make it difficult to diagnose bottlenecks in the *prototype architecture* under evaluation on a facility.

We also would like to have some flexibility in network configuration to support an M.E.F. Flexibility in connection topology not only supports fault tolerance (via compartmentalization of a facility), but also supports the above-mentioned goal of a reconfigurable overall computation structure, in which an M.E.F. could be broken up into simultaneously functioning separate parts working on different problems.

The last wish on our communications wish-list is traceability, i.e., the need to support diagnosis of failed communications nodes or media during execution of an M.E.F. Support in this area could also add to the understanding of an emulation via statistics of network traffic and usage during execution of a real experiment.

1.5 A Real M.E.F. Implementation

An M.E.F. attempting to meet the above goals is in use and under further construction at the M.I.T. Laboratory for Computer Science. We will refer to this implementation of "an M.E.F." as the "MIT MEF," or just "MEF."* In addition to support software (which is described in the balance of this thesis), the M.E.F. is composed of the following elements:

A mix of Texas Instruments' *Explorer* and Symbolics' *3600* family Lisp machine symbolic processors were used in the prototype M.E.F. implementation. These processors were chosen for their general purpose high-performance architectures, designed for multiprogramming symbolic processing (i.e., they are fully tagged architectures with automated garbage collection). These machines also include productivity tools such as bit-mapped graphic output and mouse-based graphic input, high-level networking support. In addition, the open (non-proprietary) architecture and availability of microprogramming for the Texas Instruments machine offered more support for the hardware work necessary to interconnect the P.P.'s of the prototype M.E.F. in a high-bandwidth manner.

---

* In order, of course, to follow the universe's standard policy of confusion and tendency toward randomness.

The first available local interconnection network, for both machines, was ten megabit/second nominal standard Ethernet, using the M.I.T. Chaos network protocols. This simple bus-topology network is still in use in the prototype, but severely limits communications between the P.P.'s of the system.

At this writing, a high-bandwidth circuit switch based on the Bolt, Baranek, & Newman Butterfly machine is nearing completion. The development of this hyper-cube topology switch included an implementation of a general network channel adapter (NuCA) for the Texas Instruments Explorer[20], in order that this processor could utilize the circuit switch and future interconnection strategies.

Planning is also well under way for a high-speed packet-switched network to replace this circuit switched network. The proposed packet switch will also utilize a hyper-cube topology, though with bidirectional communication along each link. Development of the packet switch is proceeding using custom and semi-custom VLSI design techniques. Current target specifications for both the circuit and packet switches, and a comparison with the available Ethernet technology, are noted in figure 3, below.

Ethernet Specifications

Raw Link Bandwidth ..... 1.25 Mbytes/sec
Switch Node Connections ..... 1 x 1
Useful portion of bandwidth 10%

Maximum total aggregate bandwidth
10% (1 processor x 1 connection x 1.25) ..... .125 Mbytes/second
= 1 Mbit/second

Circuit Switch Specifications

Raw Link Bandwidth ..... 3 Mbytes/sec
Switch Node Connections ..... 4 x 4
Useful portion of bandwidth 20%

Maximum total aggregate bandwidth
20% (64 processors x 3 connections x 3) ..... 115 Mbytes/second
= 920 Mbits/second

Packet Switch Specifications

Raw Link Bandwidth ..... 4 Mbytes/sec
Switch Node Connections ..... 8 x 8
Useful portion of bandwidth 80%

Maximum total aggregate bandwidth
80% (64 processors x 7 connections x 4) ..... 1,433 Mbytes/second
= 11.5 Gbits/second

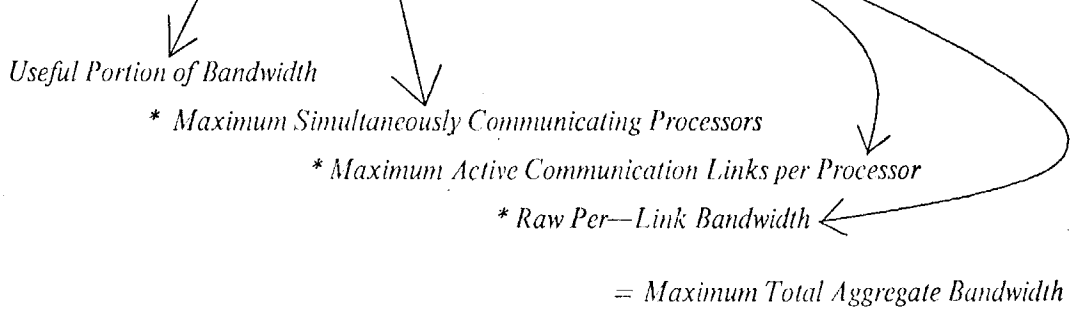

Figure 3: Switch Specifications

Chapter 2

Structure of a Multiprocessor Emulation Facility

For experimental prototyping of a proposed multiprocessor machine architecture, an M.E.F. software must provide a basic system abstraction, composed of lower-level abstractions closely coupled to the designs of systems architects, but far enough from actual design decisions to support many different multiprocessor designs. A good general abstract structure may be found in figure 4.

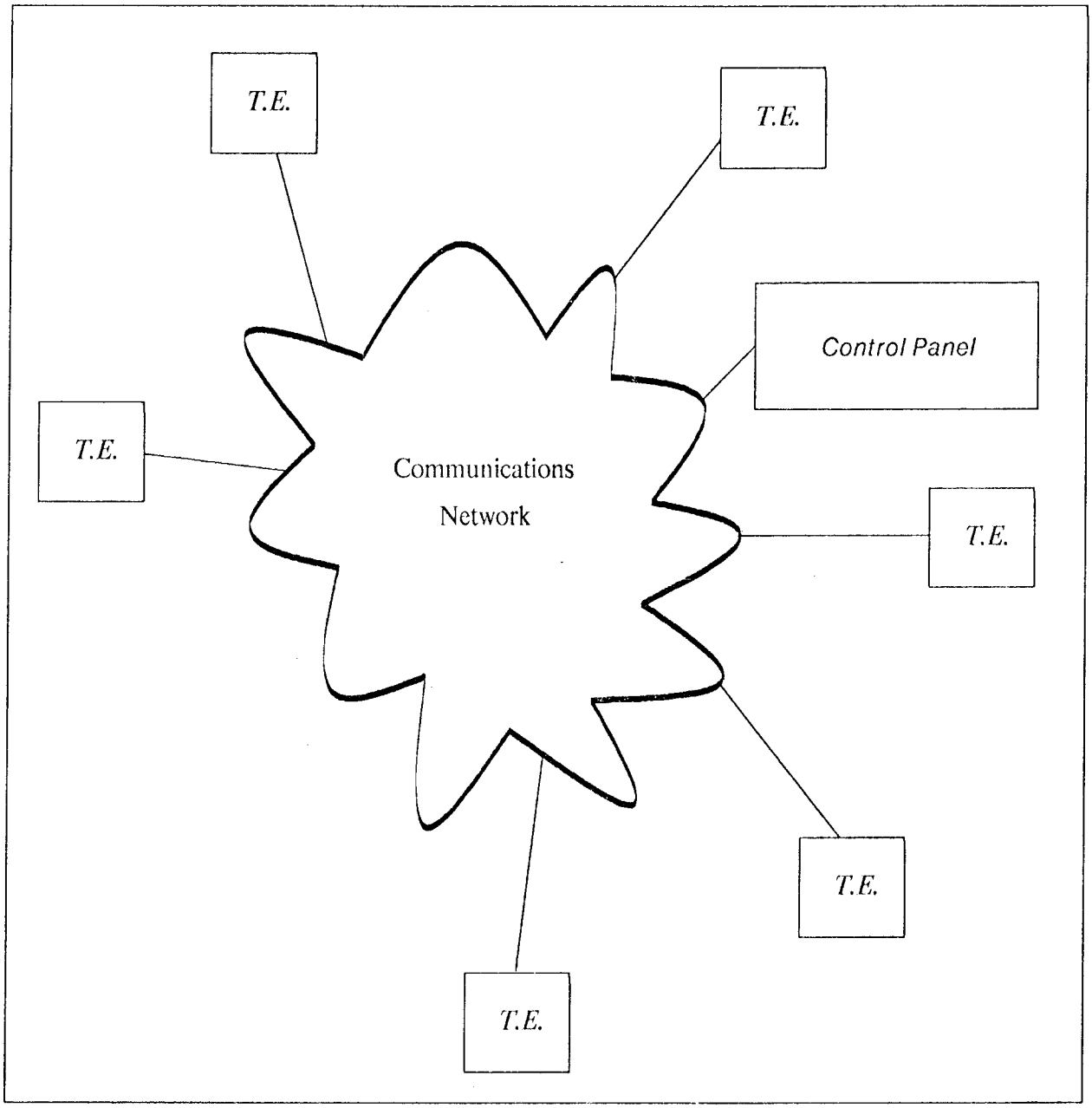

Figure 4: General Abstract Structure to be Emulated by an M.E.F.

The basic abstraction of a multiprocessor which the M.E.F. enforces is a single amorphous packet-switched communications medium surrounded by independently functioning, asynchronous processing elements (*T.E.'s*) with no shared state between the T.E.'s. This seems to be the most general case for a multiprocessor configuration. An M.E.F., in addition to this overall system abstraction, supports lower-level objects which correspond to the parts of a multiprocessor system. These include the amorphous interconnection network supplied by the MEF and the simulated topology enforced on it; more important, however, is the abstraction of a T.E. This is the basic unit of work in a multiprocessor emulation, a state-containing procedural definition of a single unit of a parallel processor. In software terminology, this T.E. abstraction is an abstract datatype akin to a flavor object in the Zetalisp language,[21] or a cluster in CLU.[22] These procedural definitions, taken with an interconnection network and a communication protocol over that network, comprise a complete definition of a multiprocessor computer architecture.

In support of these basic abstractions are a set of primitive functions, including

- A method of simulating the functionality of the *target elements* to be modeled in the user's design.
- A method of recording, updating, and sensing *local state* inside each target element.
- Some way of transferring state between two target elements, along the lines of a packet-switched (or message-passing) or circuit-switched network, or via local physical processing element channels.
- A method of recording, statistics while programs are executed on an emulated architecture.
- A simple control interface for starting up, stopping, and examining the running and stopped states of the target machine.
- Tools for developing and debugging the emulation experiment code itself.
- Tools for setting M.E.F. (*physical processor*) hardware and software characteristics and parameters (e.g., network topology, error recovery schema, etc.).

2.1 The M.E.F.'s Abstract Model of a Target Architecture

In order to emulate a proposed machine architecture, the emulation facility requires a general model of such a system. With such a model, it can mirror the overall structure of a system symbolically. The emulation facility's machine model consists of several atomic objects, which are outlined in this chapter.

2.1.1 Processors and Processor State

The "hearts" of any multiprocessor system are, of course, the processors themselves. The M.E.F. supports an abstraction, called a *target element,* which maps to a real multiprocessor's single processors. The representation of an instance of this abstraction is a body of code that mirrors the functionality of the proposed "black box," depending on communication primitives proved by the M.E.F. The implemented M.E.F. system, as it resides in the Lisp Machine environment, expects *target element* definitions to be expressed in groups of Lisp statements (called *"forms"* or *"S-expressions"* by Lisp programmers). These *"processors"* may hold local state, much as a CLU cluster definition has local state.[22]

In order to correctly imitate real machinery, the M.E.F. must supply a related abstraction to the *processor,* called *processor state.* This abstraction allows the M.E.F. user to model registers, real memory, and all other hardware state via the natural programming language analog, the variable. The M.E.F. enforces inter-processor separation of variables, so that the experimenting user may use global state within all processors of the same type without any naming conflicts. In other words, two processors may each have a *register* abstraction with the same name (for instance, **R0**) without any accidental (or intentional) overlap.

In practice, we have found that most writers of T.E. implementations include in their T.E. model "state variables" that are not really part of the state of the processor being modeled (i.e., do not represent a latched value in the final hardware of the T.E.), but are included only to simplify coding of an implementation. These include temporaries used in computation of a T.E.'s final state, or any other artifact of the functional simulation of the T.E. hardware. These "false state variables" are represented in the same way as other state variables by a M.E.F. system.

Debugging tools provide a good example of these "false state variables." For example, programmer-invisible machine registers, or other "spy" devices for tracing the execution of the emulated machine, fall into this category.

2.1.2 Communications

The basic communication abstraction provided by the M.E.F. is quite simple; at the lowest level, it provides two interfaces. The first, *"receive,"* simply guarantees to pick up and return an message bound for the current T.E. No guarantee as to ordering of messages is made or implied; this simple function only guarantees to wait for any available input and return any one of the pending inputs. (There are non-blocking interfaces to check for pending input as well).

The more interesting primitive is *"send."* Given an outgoing message and a destination T.E. identifier, the *send* function must forward the message to the proper destination over the fastest available path (since there are generally multiple paths available), choosing the path based on mesage length and destination. In addition, this primitive also guarantees that by the time it returns to its caller, *no further interaction with M.E.F. primitives is necessary to forward the particular message with which* **send** *was called.* In other words, this primitive must provide for any queueing, backout and retry, or other paradigms for sending messages over multiple-access channels, and must not block the calling process except when absolutely necessary (i.e., if some pending output queue is full).

In addition to this simple abstraction, the M.E.F. provides a methodology for metering network delays and forwarding node usage, based on a static or dynamic simulated network map, which allows run-time calculation of the list of T.E.'s through which a message must pass to travel from some T.E. *i* to some other T.E. *j*.

2.1.3 Resource Management

One of the most important services provided by the M.E.F. is as an *"abstraction wall"* between the architectural experimenter and the actual M.E.F. implementation structure. This "wall" implies automatic management of hardware and software resources within the facility without the intervention of the experimenter. For example, though a user knows that a certain *target implementation* contains certain state and related functionality, he or she does not need to be aware of:

- How the M.E.F. represents the processor's state.
- On which physical processing element of the M.E.F. configuration the target element is executing (unless the user wishes to enforce some locality constraints for efficiency's sake).
- How inter-processor messages are *actually* routed between processors, and on which of the (possibly numerous) interconnection networks the messages are traveling.

The above-listed services are walled off from the architecture designer, as they are artifacts of the M.E.F. structure itself, not the architecture under emulation. This frees the experimenter from low-level emulation details, as well as creating a "clean" development environment in which the low-level details are kept from creeping into the high-level design. For example, details of how communications packets are routed between physical processors on an M.E.F. should have no bearing on the design of an experimental processor using the M.E.F. as a tool.

2.1.4 Metering

An M.E.F. is useless if it does not allow some sort of metering functionality to the emulation experimenter. Therefore, in addition to the T.E. execution and state models, an M.E.F. must support metering. An M.E.F. system should implement two types of meters, which are used in two separate ways:

- *System-wide* meters provide a way to monitor some activity as it occurs on *every* T.E. in the system. For example, if we are interpreting some machine's instruction set, we might want to meter how many emulated instructions were executed during an entire emulation experiment.

- *Per-T.E.* meters are much like T.E. state variables, in that they record activity within a single T.E. *only*. These are useful, for instance, for measuring memory faults on a per-emulated-processor basis.

Either of these metering abstractions is necessary to allow the user to store, interpret, and display the vital statistics of an experiment. For instance, it might be important for the designer to know exactly how much of his or her inter-processor buffering is being used, both dynamically (during emulation experimentation) and, more quantitatively, statically at the end of an experiment. The user should be able to clear, update, and check on the values of meters both at run time and post-experimentation time, and display them dynamically in a run-time console graph or statically via his or her own methods.

2.1.5 Control Panel

The last, and most user visible portion, of the M.E.F. abstraction is the MEF Control Panel. In fact, this provides the *only* user interface between the M.E.F. proper and the architectural designer, as well as between the user of the emulated machine and the running emulation experiment. Hosted on any available physical processor that is capable of communicating with every other P.P. of an M.E.F. system,* the Control Panel provides means of configuring and partitioning the available M.E.F. physical processors into an M.E.F. system, loading and initiating the selected target architecture (emulation experiment) on those P.P.'s, and monitoring and debugging the running experiment.

---

* Under the MIT MEF system, the Control Panel is hosted on a Lisp Machine, just as any other part of the MEF system.

Besides acting as a "bootstrap processor," or "virtual front panel," for the M.E.F., the Control Panel takes the place of the bootstrap processor for the *target* (or *emulated*) machine, including primitives for initiating and booting the T.E.'s of an experiment as well as injecting a starting message from "outside" of the emulated system.

Figure 5, below, gives a user's operational view of the structure of an M.E.F., in terms of what the user must implement and how it interfaces with the substrate.

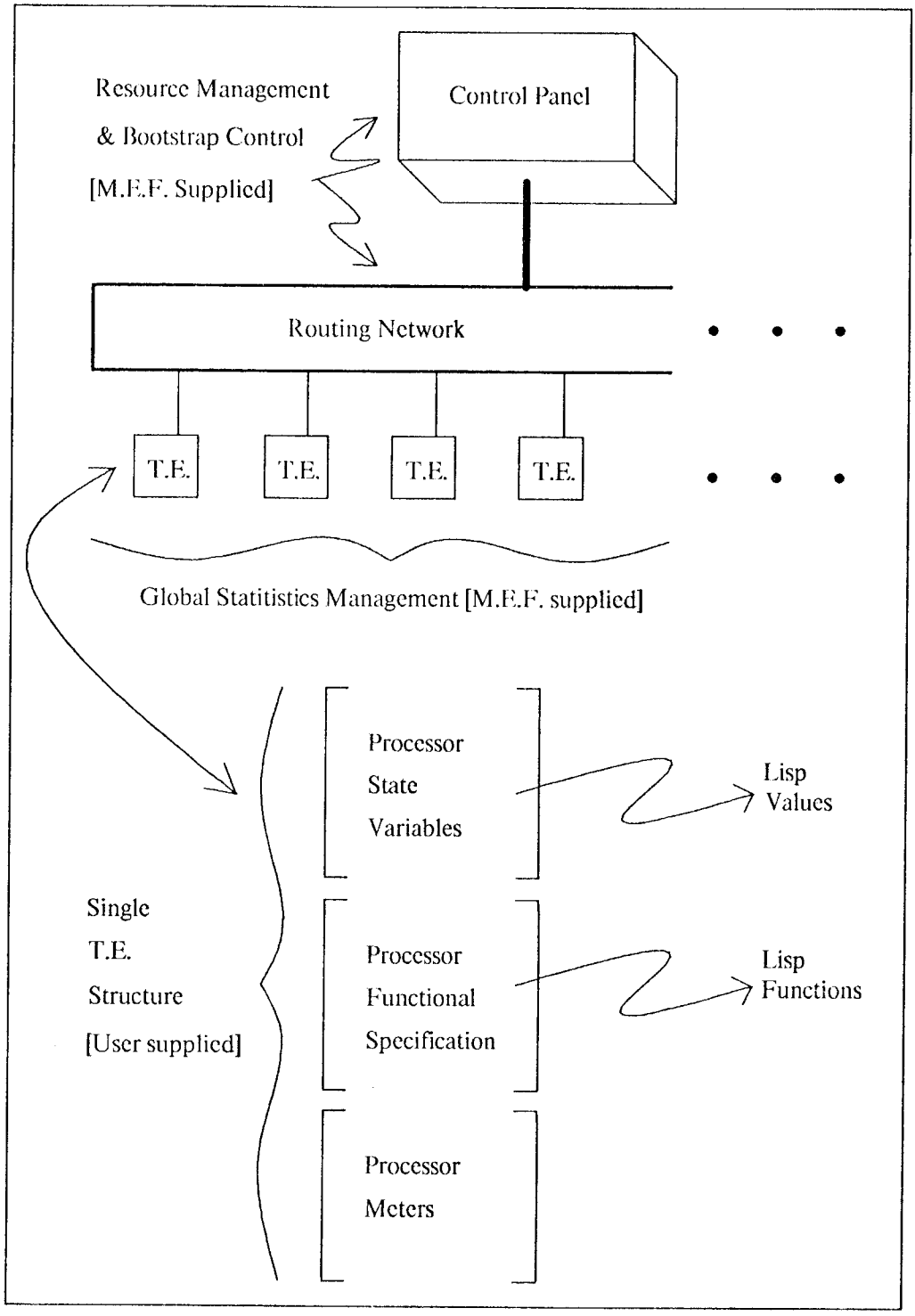

Figure 5: Operational View of an M.E.F. Emulation Experiment

2.2 M.E.F. Software Design Decisions

The abstract M.E.F. structure discussed above does not necessarily translate directly to a real multiprocessor architecture. This section discusses a particular implementation of an M.E.F. to which we alluded previously. The MIT MEF, developed at the M.I.T. Laboratory for Computer Science, was implemented under the Lisp Machine environment,[21] connected via a collection of packet-switched data networks. The Symbolics 3600 Lisp machine was used for development, while Texas Instruments Explorers were used for most of the facility's processing. The software systems available on these two machines are both based on the work of the M.I.T. Artificial Intelligence Lisp Machine Project[21] and are thus comparable; the structure and design decisions of the MIT MEF are portable across both as well.

2.2.1 Logical Structure

The MIT MEF, as it is implemented on general purpose Lisp Machines, is written in Lisp Machine Lisp and structured in a hierarchical fashion, outlined in figure 6. On top of the resident Lisp Machine system software are two basic portions of the MEF, the *static* section and the *dynamic* section. One may think of these as the *definitional* and *runtime* support parts of a modern computer language, since in essence an M.E.F. implements a functional description language for computer architectures. Above this, a MEF system user constructs his or her functional model of an architecture, on top of which end users (architecture explorers) conduct experimentation. Above the MEF system, as in any computer system, we find two levels of "users;" the application (or *systems*) programmer, and the end-user of the application program executing on the system.

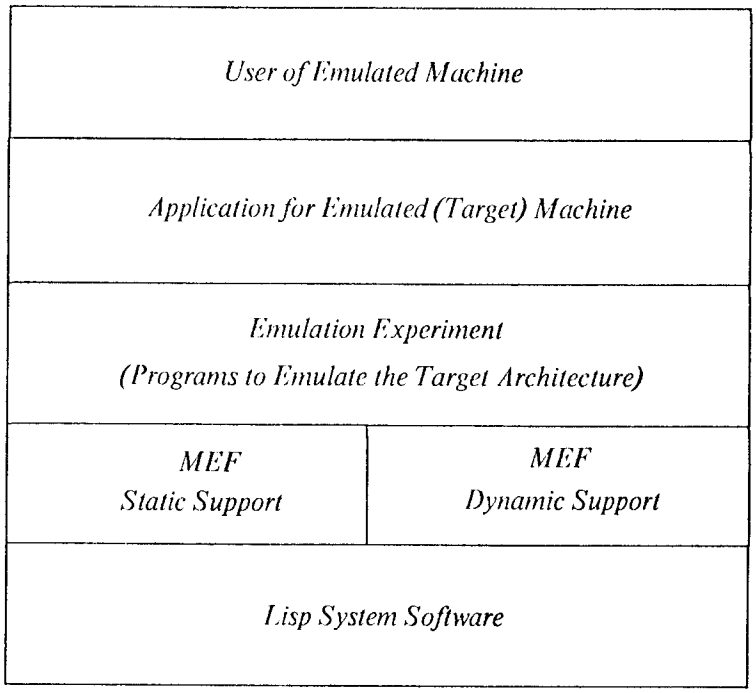

Figure 6: Logical Structure of the MIT MEF

This level of the model is straightforward; however, many design choices necessary to implement the M.E.F. abstract model in the MIT MEF were not so obvious.

2.2.2 The Target Element State Abstraction

In particular, the target element (T.E.) abstract model—a procedural box with an output stream, an input stream, and some non-shared state—presented many choices for implementation under the Lisp Machine environment. First we must note the requirements of such an implementation.

First and foremost, the implementation of the T.E. abstraction *must not* require any knowledge on the part of the emulation experimenter of whether his or her T.E. definitions are to be executed on the same machine, or on multiple machines. In particular, an implementation under a global-state machine (such as global variables in the Lisp Machine sense) must provide a namespace "wall" between T.E.'s. As an example, multiple processes

under the Lisp Machine operating system share global variable names (or *static* variables); thus, implementation on a Lisp Machine using global variables for T.E. state storage would be an incorrect implementation, since T.E.'s executing on the same machine would overlap state, while T.E.'s executing on different physical machines would not.

In addition, the architectural experimenter should not need to know how many T.E.'s are executing on any given physical processor, or in particular which of his or her T.E. models is executing on which processor. This means that the M.E.F. system must automatically choose connection paths, *as well as decide any issues of code copying* between physical processors participating in an emulation.

We must stress that an M.E.F. should support any number of different *kinds* of T.E.'s, and any number of each kind of T.E. For example, we may wish to model a shared memory processor as **N** *memory* units and **M** *processing* units; this would entail the description of two different *kinds* of T.E.'s. One description would describe the activities of a *memory* units while the second would functionally describe a *processing* unit. Then, an M.E.F. would be directed to execute a model composed of **N** of the first and **M** of the second.

The real issue is that care must be taken in implementation of an M.E.F. to preserve the abstraction of the T.E. model. The T.E. state model chosen for the MIT MEF system allows the user to think of the state of a T.E. as global (static) variables *which are local to each T.E.* As was noted above, this model would not serve the purposes of an M.E.F. if T.E. state were actually implemented in this way, but we felt that this abstraction was the simplest possible model.

Below we discuss various possible implementation schemes for T.E. state under the Lisp Machine environment, addressing such issues as code copying, complexity of the model from the user's viewpoint, and performance. All of these schemes assume that the M.E.F. user is required to list the names of all of the "T.E. state variables" (or *registers*) used, so that the M.E.F. can know what variable names a given T.E. model must be "closed over."

2.2.3 T.E. Abstraction via Flavors

Readers familiar with the Lisp Machine architecture at this point invariably suggest the use of *flavors* to implement T.E. state. Flavors[21] are a powerful object-oriented programming concept, combining multiple inheritance and dynamically linked functionality in a message-passing syntax. They implement a user-definable language type structure in which types define function and state templates, much like the CLU language *clusters*,[22] except that the functions allowed to operate on a given type may be dynamically altered, and new functions added.

From the T.E. state model point of view, flavors are the perfect implementation vehicle. They support a template view of programming that well fits the philosophy of a group of shared functions (e.g., a T.E. functional definition) operating on multiple similar aggregates of state (e.g., T.E. registers). Unfortuantely, the dynamic feature of the Lisp Machine flavor implementation introduces great function call overhead. Since the functions that operate on a given flavor can be added to, deleted from, or otherwise altered at any time, calling a function to act on a particular instantiation of a flavor (an *instance*) requires a hash-table lookup. Since all of the functions defined to operate on instances of a flavor are not known at compile time, and since Lisp variables are not typed at compile time, calls on an instance cannot be compiled down to a machine function call. Thus flavors were ruled out as an implementation mechanism for the T.E. abstraction.

2.2.4 T.E. Abstraction via Global Variables

Another possibility was to use Lisp global (static) variables. Unfortunately, although this would be a straightforward implementation scheme and would support very good performance, due to the reasons outlined in the beginning of this section (overlap of variable references between executing T.E. definitions) this scheme could not support more than one T.E. per physical processor. Since we need an M.E.F. to potentially support any number of T.E.'s per physical processor, this idea was quickly scrapped.

2.2.5 T.E. Abstraction via Dynamic Variables

There is, however, an extension of the Lisp Machine global variable concept. Under the Lisp Machine operating system, global variables may be *"bound"* to different values during execution of a particular Lisp Machine *process*. Thus, T.E. functional models each could be executed inside its own *process*, with the values of its registers bound, and thus separated from the registers of other T.E.'s executing on the same physical processor.

Unfortunately this scheme introduces an enormous performance penalty. It requires that the Lisp Machine process scheduler must insure that bound variables for a blocked process must be unbound when another process is awakened for execution; in addition, of course, the bound variables of the awakened process must be made current at process-switch time. This scheme was attempted as the T.E. abstraction implementation for the MIT MEF, with remarkably poor results due to this process switching overhead (particularly with more than three T.E.'s executing on a physical processor).

2.2.6 T.E. Abstraction via Packages

Another promising feature of the Lisp Machine that was considered was the *package* model. This feature allows separation of variable namespaces at function definition or load times, to provide some small amount of name separation under an operating system that is basically supports a monolithic namespace and address space. We think of this scheme as a "poor man's segmentation," as it solves some of the problems of naming that were addressed by the Multics segmentation scheme.[23] It must be noted that Lisp Machine packages are completely unrelated to Ada packages,[24] which are closer in spirit to the *flavor* concept.

Using the Lisp Machine package feature, each T.E. would be loaded into its own separate namespace, and would simply address its registers as global variables inside that namespace. Unfortunately, even T.E.'s of the same *type* would be unable to share code (T.E. function definitions), since each T.E. would need its code to address different namespaces. The paging and virtual memory waste accompanying such unnecessary code copying ruled out this concept for the MIT MEF T.E. implementation.

2.2.7 T.E. Abstraction via Arrays Referenced (Indexed) Off a Dynamic Variable

Finally, a hybrid scheme was chosen that used some of the concepts of flavors with a dynamically bound global variable. The MIT MEF uses a *single* bound global variable which points to an array (*state block*) which contains the current values of all T.E. registers for the currently-executing T.E. At process-switch time, this *single* variable is unbound and bound to the new context, a minor introduction of overhead. Accesses to T.E. registers are altered (via macro expansion) to offsets into the current state block at *compile time*. This can be done because users are required to list all of the registers to be used by each T.E. model. Run-time access to T.E. register values is then accomplished by adding an offset (established at compile-time) to a global variable—i.e., a one-memory-reference overhead.

This memory reference overhead is the only disadvantage of such a scheme. Unfortunately, however, since T.E.'s are presumed to make constant access to their state, this overhead can be tremendous, as it introduces a 100% overhead in state memory reference. Fortunately, this overhead can be alleviated by a caching scheme, in which the current state block pointer is saved (at process-switch time, or at the time of the first T.E. register access) in the CPU. This scheme will be utilized with the Texas Instruments Lisp Machines with the introduction of a small amount of microcode to support the caching of special virtual memory pointers on the processor board of the machine. This method of "T.E. procedure switching" is the analog of the standard operating systems approach to multiprogramming process switching.

2.2.8 The T.E. Execution Abstraction

Besides modeling the state of executing T.E.'s, we must somehow model the actual execution of T.E.'s based on users' functional descriptions. As we have alluded above, these functional descriptions must be written in Lisp Machine Lisp for the MIT MEF; execution of the descriptions, at first glance, simply entails the execution of interpreted or compiled Lisp forms using references to T.E. state as was discussed above.

However, there is a choice for implementation of these T.E. execution "threads," or "processes." The most obvious choice, in the case of the Lisp Machine, is to use the Lisp Machine scheduler and *process* abstraction, since it directly implements the T.E. execution abstraction, running threads of Lisp code on separate stacks, blocking on input/output operations, etc. Therefore, by default the MIT MEF system calls the Lisp Machine system software to create and run processes as the implementation of the T.E. abstraction. This allows the emulation experiment writer to model T.E.'s in a fully general way, as individual machines that do I/O and computation in no pre-determined order.

Unfortunately, there is an overhead associated with process switching under the Lisp Machine system, which rises linearly with the number of processes (and thus emulated T.E.'s). This is caused by the need to flush and re-fill stack buffers and possible paging overhead to reinstate a previously blocked process.

The MIT MEF allows users interested in maximum performance to use a slightly less general model to attain some speedup by avoiding process switching. In this model, T.E.'s are required to be functional blocks that take a single message as input, do some computation, output zero or more messages, *and then return to the caller.* This allows the MEF system to do its own "T.E. scheduling" by directly *calling* the Lisp implementation of a T.E. in the same process as the MEF system, rescheduling another (or even the same) T.E. on return of the model.

Although this system is not completely general, it does fill the needs of many emulation experiments that can be modelled easily in this more restrictive manner. We must note, however, that this method of T.E. implementation carries another disadvantage: the M.E.F. system will block *forever* until a T.E. abstraction returns, since there is no "block and reschedule" facility within MEF. Thus, errors in T.E. implementation have the capacity to halt execution of an entire emulation unless T.E. definitions are not allowed to call potentially blocking routines.

2.2.9 T.E. and System Metering

The implementation of the metering abstraction under the MIT MEF system is two-fold, to support the two different types of meters. Per-T.E. meters are implemented under the MIT MEF system in much the same way as T.E. state registers, that is, as an array of meter values stored with the state array. System-wide meters, however, present a range of implementation choices.

The most obvious choice would be to simply centrally record the values of all system-wide meters on one of the physical processors taking part in an emulation run. However, this increases interconnection network overhead for no particular reason, since the values do not need to be centrally collected until the user actually asks for the value of a system meter.

Therefore, the MIT MEF implementation collects the values of system-wide meters in a distributed manner, incrementing local copies of each meter on each physical processor in the M.E.F. When the user (via the system control panel) asks for the value of a particular meter, a meter read request is sent to each physical processor taking part in the current M.E.F. experiment; the results are then tabulated centrally and presented to the user.

2.2.10 Communications Software

This brings to light the issue of interconnection schemes for an M.E.F. The abstract definition of an M.E.F. notes only that the details of packet communication between T.E.'s should be totally transparent. This means that:

- From the T.E. procedure's point of view, the network *send* primitive must return "immediately;" i.e., no further work on the part of the T.E. may be required in order to get the message to the destination T.E. This implies that the implementation of the *send* primitive must automatically queue messages, backoff and retry, acknowledge if necessary, or perform any other network management functions necessary to transmit messages.

- The actual interconnection network(s) should be accessed in a uniform way, through a uniform *send* primitive (i.e., there should *not* be a *send-via-Ethernet* function as well as a *send-via-Hypercube* function, just one generic *send* function).

- Inter-T.E. messaging between T.E.'s executing on the same physical processor should be accessed via the same *send* primitive as network messaging, as the experiment writer is not aware of the actual physical processing configuration of an emulation.

- Network *input* must be via a single *receive* function, regardless of the physical and/or logical locations of the sending T.E.

The MIT MEF actually implements simultaneous access, via a single queueing *send* primitive function, to four inter-T.E. communications media. The first is for T.E.'s residing on the same physical processor, which communicate through a queueing mechanism much like Unix pipes.[25] All communications requests, in both directions, are handled by the local distributed portion of the M.E.F. software system, called *outposts*. An outline of this procedure is presented later in this chapter.

The other three network interfaces connect to three different high-speed inter-processor media, including a ten-megabit per second ring-topology Ethernet (executing Chaos[26] protocols); a three-megabyte per second circuit switched hypercube-topology network based on the Bolt, Beranek, & Newman Butterfly switch;[27] and a four-megabyte per second packet switched hypercube-topology network developed at M.I.T. in cooperation with the International Business Machines Corporation.*

Regardless of the communications path followed by the transmission of a single MEF inter-T.E. message, the MEF sending and receiving functions provide a packet-switched abstraction to all communicating T.E.'s. Thus, though a message might follow any of four different types of routing based on the distance between the physical machines supporting two communicating T.E.'s, and the size of the message being sent, the same functions are used *without retry* to complete that communication.

---

* The last two of these networks are under development at this time.

2.2.11 Main Control Panel

Tying the entire MIT MEF system together is a *Control Panel* program that also executes under the Lisp Machine operating system, either on one of the physical Lisp Machine processors participating in an active emulation experiment, or on another (dedicated) processor. This "Control Panel" implements a bootstrap processor that oversees the activities of the current M.E.F. configuration of physical processors. Users issue this program commands such as:

- **CONFIGURE:** Set up a certain group of physical processors as an M.E.F. This command begins a small overseer program on each P.P. participating in the emulation, and establishes all necessary network connections between each physical processor.

- **START-EMULATION:** Given an emulation name, loads and begins the emulation named, instating T.E.'s described by that emulation on the participating physical processors and establishing the logical connectivity between the T.E.'s as described by the emulation experiment.

- **SET-TRACING:** Enable a system-wide tracing facility to direct tracing information (specified by T.E. function descriptions) to the screens of each participating physical processor, or to the Control Panel.

- **STATUS:** Get the status of a running (or idle) T.E. on any physical processor.

- **DEBUG-TE:** Temporarily halt a T.E.'s running Lisp code and allow the experimenter to walk through that T.E.'s active state, checking and/or altering variable values and restarting computation at any point.

- **SHUTDOWN:** Shutdown the current emulation experiment, much like a bootstrap processor shutting down a mainframe machine. Halts all T.E.'s and returns their storage to the Lisp heap.

Figure 7 shows the overall multiple physical processor architecture of an emulation running under the MIT MEF system. In this example, there are four participating physical processors, one of which is also hosting the Control Panel. The running emulation experiment is comprised of eight target elements.

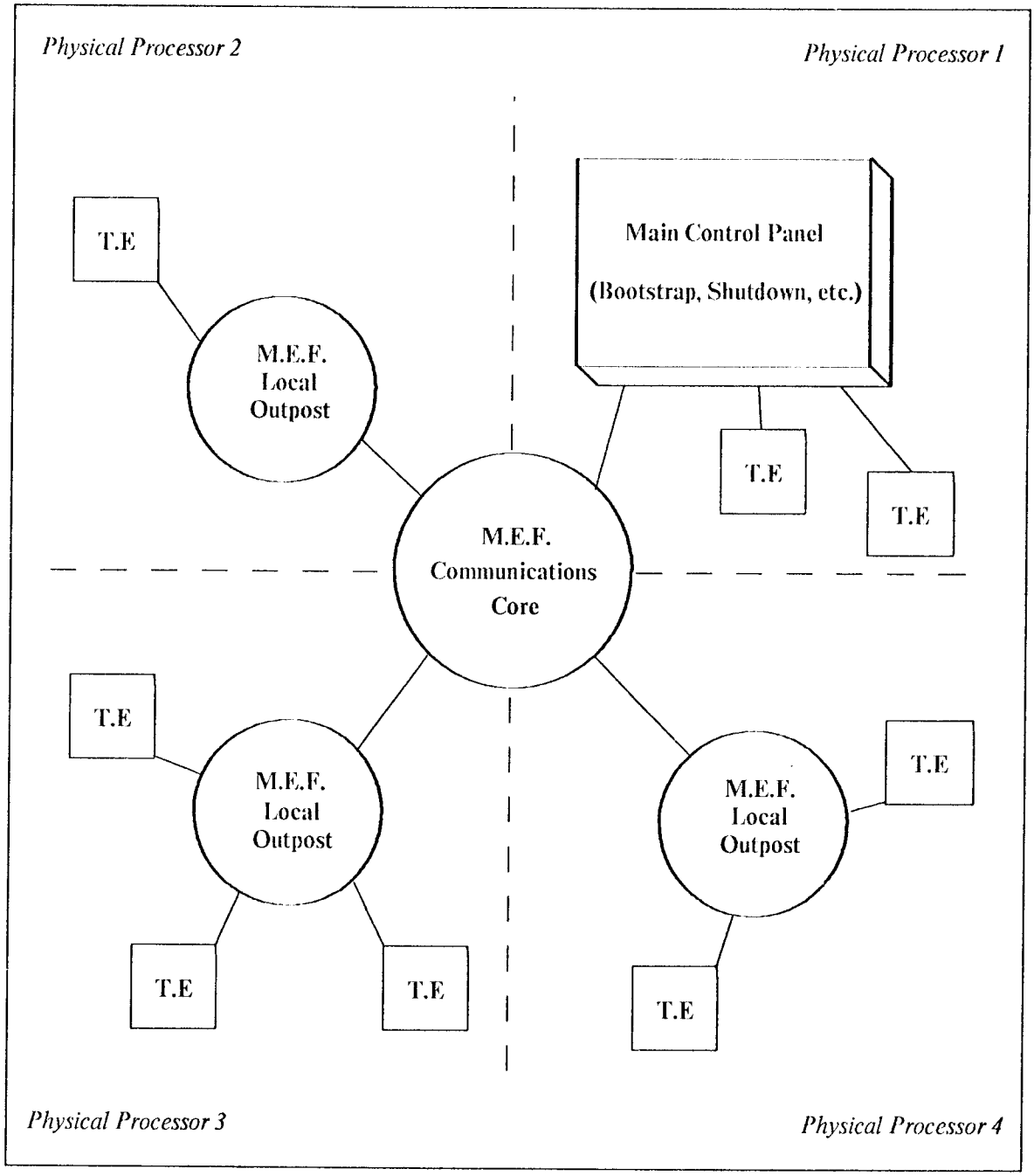

Figure 7: Multiple Processor Dynamic Structure of the MIT MEF

2.3 Program Outline

The MIT MEF software consists of three major parts, including (1) the Control Panel, (2) interfaces to the M.E.F. for use by emulation experiment implementations, and (3) the M.E.F. *outposts*, or overseers, executing on each P.P. of the M.E.F. This last portion is the most interesting, as it represents the distributed "operating system" portion of the M.E.F. software, executing configuration and control commands emanating from the Control Panel and the user.*

One particularly interesting detail of the operation of the MEF *outposts* is the paradigm for interconnection. The Lisp Machine architecture presents the Ethernet interconnection medium to the application programmer as a bidirectional stream connected to a listening process (another stream) on the foreign physical host. Therefore, for interconnection between outposts on the Ethernet medium, each outpost needs a stream connecting to each other outpost.

It would seem that the simplest method to gain this array of interconnections would be to simply have each outpost initiate a connection to each other outpost; however, the bidirectional nature of the resulting streams would make such a scheme quite wasteful. Therefore, each outpost uses its own outpost identification number (which ranges from zero to one less than the number of outposts, or P.P.'s) as a sort of *"quicksort comb,"* passively waiting for connection requests from lower-numbered outposts, and actively initiating connection requests to all high-numbered outposts.

Besides simplifying the connection process, this scheme also insures that no outpost will issue a connection request to a physical processor that is not yet executing the outpost code, since the outposts are initiated on the P.P.'s in ascending outpost-number order.

Outpost operations are roughly outlined in the program below.

---

* Specifications for use of the MEF Control Panel and emulation experiment interfaces to the MIT MEF system are presented in the appendix of this thesis.

```
/* Local M.E.F. Controller (OUTPOST), to be executed on each
   P.P. of an M.E.F. configuration.  This routine manages
   the T.E. models executing on its local processor, while
   managing communication to and from these T.E.'s as well
   as M.E.F. control messages from the M.E.F. Control Panel.
*/

procedure MEF_Outpost (Command_Stream) begin

    /* First, read local P.P. number (as allocated by
       the Control Panel) and the total number of P.P.'s. */

    Local_PP  := Read_Byte (Command_Stream);
    Total_PPs := Read_Byte (Command_Stream);

    /* Now set up communications with all other P.P.'s.
       We use a "triangular matrix" technique: each Outpost
       simultaneously passively waits for contacts from each
       lower-numbered P.P., while initiating contact with
       each higher-numbered P.P. See above for details. */

    cobegin
        for Listen_PP from 0 below Local_PP
            Routing_Table (Listen_PP) :=
                Await_Contact (Listen_PP);

        for Talk_TE from Local_PP + 1 below Total_PPs
            Routing_Table (Listen_PP) :=
                Initiate_Contact (Listen_PP);
    end;
```

```
    /* Communications are now set up.  We can expect
       commands to arrive from the Control Panel, as
       well as requests for communications from local
       and foreign T.E. models. */

    do forever
        Input_Stream := Await_Input (Command_Stream,
                                     Routing_Table,
                                     TE_Table);

        if Input_Stream = Command_Stream begin
            /* Input is from the Control Panel. */

            case Read_Byte (Command_Stream)
                if 'M' then Forward_Message (Command_Stream);
                if 'E' then Experiment :=
                          Setup_Emulation (Command_Stream);
                if 'K' then Shutdown_TE (Command_Stream);
                if 'S' then Shutdown ();
                if 'C' the Create_TE (Experiment,
                                      Command_Stream);

                /* Other local notification and control
                   functions are performed here as well. */
            end case;
        end;

        /* Input is from a foreign or local T.E. model. */
        else Forward_Message (Input_Stream);

    /* Schedule any TE's that having pending input.  This
       will cause the procedural definition of an TE with
       pending input to be invoked in the environment of
       that particular TE. */
    Experiment.Scheduling_Paradigm (Experiment);

    end do forever;
end MEF_Outpost;
```

```
/* Forward a message to a local or non-local T.E. */
procedure Forward_Message (From_Stream) begin

    Destination_TE := Read_Byte (From_Stream);
    Message_Length := Read_Word (From_Stream);
    Message := Read_String (From_Stream, Message_Length);
    TE_Descriptor := Lookup_TE (Destination_TE);

    /* If local message, place in local T.E.'s incoming
       message queue.  Else forward to proper P.P. */

    if Destination_TE element_of Local_TE_List
        then Queue (Message,
                    TE_Descriptor.Incoming_Messages);
        else Send (Message_Length, Message, Destination_TE,
                   Routing_Table (TE_Descriptor.PP));
end Forward_Message;


/* Create and initialize a Target Element. */
procedure Create_TE (Experiment, Control_Stream) begin

    TE_Type := Lookup_Experiment_TE (Experiment,
                   Read_String (Control_Stream);
    TE := Create_TE (TE_Type, Experiment);
    Add_TE_to_Running_Profile (TE, Experiment);

    /* Call the user-specified initialization procedure. */
    In_TE_Environment (TE_Type.Init ());

end Create_TE;
```

Chapter 3

Using an M.E.F. as an Interpreter

3.1 An Educational Experiment: The von Neumann Machine Interpreter

A multiprocessor emulation facility is fundamentally a tool for multiprocessor prototyping. In this and the the following chapter, we present possible uses of an MEF to show the need for such a tool, and the use of such a tool. These particular prototypical uses of an MEF were actually written and executed on the MIT MEF;[18] some results of those interpretation & emulation experiments will also be presented.

The simplest multiprocessor configuration in use today is the von Neumann style *uniprocessor,* composed of one *central processing unit,* and one *"memory bank."* Although this is a uniprocessor in today's sense, it is actually composed of two fully parallel processing units, activating each other via synchronous or asynchronous messages over some type of computer *bus.* Although the parallel portions of this configuration (we will call them simply the CPU and the MEMORY) often lock, waiting for the completion of an operation in the other processor, there is generally no possibility of deadlock, and there can be some overlap of operation if the datapaths of the CPU allow work to be done while a bus read or write cycle is active.

Therefore we present a von Neumann style uniprocessor, viewed at the bus level, as the world's simplest multiprocessor architecture. Figure 8 shows a view of the the processing elements and their trivial connectivity.

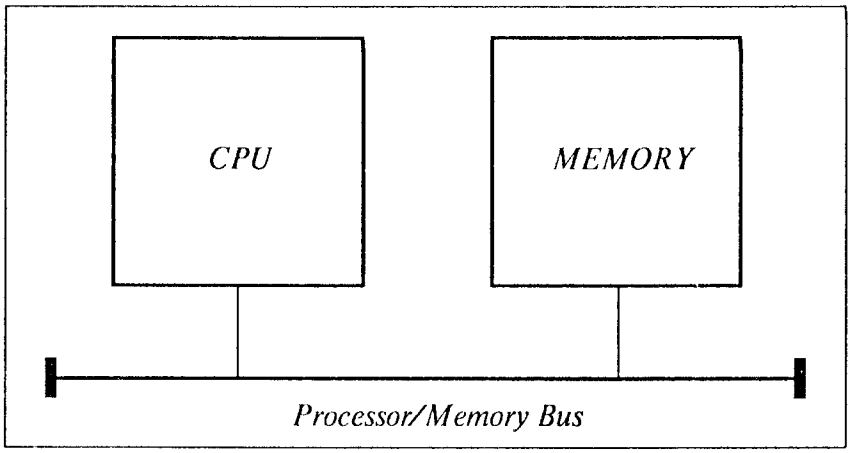

Figure 8: A Simple von Neumann Processor

As can be seen in this simple figure, we view the *asynchronous* communications network of the Emulation Facility as a single, synchronous connection (i.e., a bus) between the two parallel processors. We say *synchronous* because all messages between these two processors are queued by the communications medium; there is no "interrupt level" activity. For example, a "memory write cycle looks something like this:

CPU	*(message direction)*	*MEMORY*
WRITE	→	
ADDRESS	→	
	←	*DATA*

For testing purposes, we might alter this simple "protocol" by inserting an acknowledgment by the memory:

CPU	*(message direction)*	*MEMORY*
WRITE	→	
	←	*WRITE ACK*
ADDRESS	→	
	←	*DATA*

This kind of modification is trivial to make within the framework of an Emulation Facility; the CPU processor simply waits for an acknowledgment after transmitting a *write* request (i.e., reads from the MEF an incoming message from MEMORY), and the MEMORY processor

needs to generate an acknowledgment after receiving a *write* request (i.e., it must write to the MEF a message directed at the CPU).

From the standpoint of a systems architect (in this example, perhaps a bus/backplane designer) the flexibility to quickly and simply change an interconnection strategy and processor interface *without touching a soldering iron* is unparalleled. Validation of a parallel-processing design can be performed in a manner much more like the exercise of modern programming: a tight edit/compile/debug prototyping loop, followed by thorough specification and implementation after the underlying structure is well understood.[28]

We began this chapter by discussing a supposed "von Neumann, single-processor" machine, and ended the last section with a multiprocessor view. We must stress that a Multiprocessor Emulation Facility is a *general tool* for architectural design with all types of parallelism, such as pipeline stages, distributed concurrent database systems, or multiple processing elements. The power in the idea is the inherent ability to emulate *any* complex system.

3.2 A Multiprocessor Experiment: The Tagged Token Dataflow Emulator

The preceding example use of the emulation facility is indeed instructive, but it does not exercise the full abilities of the M.E.F., nor does it represent a particularly new or experimental design. The experiment discussed below fulfills both of these ends.

3.2.1 The Dataflow Model of Computation

Though the basic novel ideas (in comparison to the von Neumann machine structure) of the dataflow model are rather old in computer science terms,[29] only lately have several different variations on the old theme shown great promise.[18, 12] In the dataflow computer model, asynchrony and functionality form the key to a highly parallel computational model in which programs can be run in a parallel fashion without programmer specification of parallelism.[7]

The major problems plaguing dataflow computation models today are the same twin problems that have always plagued computer science and engineering, namely (1) where do we put things, and (2) how do we get them from here to there. These *resource management* questions apply to program and data storage allocation, processor structure for maintaining

computation state, communications bandwidth compression, processor time-sharing, and so on. Basically, dataflow models engender more than their share of the problem of how and where to apply computation storage and computational power, and how to communicate such data among structurally and geographically separated machines.[30, 31]

Unfortunately, these huge problems never keep people from designing real hardware to realize particular variants of the dataflow model, or even to use expensive (in monetary, temporal, and human terms) silicon design technology in a quest for a parallel dataflow architecture. Several dataflow architectures have been shelved after great expenditures,[32, 33] even though specialized simulation techniques were used before committing to hardware.

Obviously, a more flexible tool for dataflow machine experimentation is needed to avoid the financial and human expense; an M.E.F. exactly fits that need. In fact, the work reported in this paper was originally prompted by a need to quickly and flexibly emulate a variant of the Tagged-Token Dataflow Architecture, a machine realization of the Irvine/MIT U-Interpreter abstract machine model.[18]

3.2.2 The Tagged-Token Dataflow Machine

The Tagged-Token Dataflow Machine is an architectural realization of an abstract machine model developed at the University of California at Irvine, named the U-Interpreter.[8] This architecture utilizes an extremely low computational granularity, on the order of single instructions on standard processors (i.e., addition and subtraction), to realize the maximum parallelism in general programs without programmer specification of parallelism. In addition, the memory model of this architecture inherently tolerates long memory latencies, as well as multiple access to data memories without unduly constraining parallel execution. Both of these capabilities are thought to be unattainable within the von Neumann machine framework.[18, 34]

3.2.3 Emulation of the Tagged-Token Machine

The basic system architecture of the TTDA machine matches quite closely the abstract M.E.F. model. An N-dimensional hypercube network topology is used to interconnect any number of processing elements, each of which is a rather general-purpose pipelined CPU and memory management unit. Figure 9 outlines the overall structure of the TTDA.

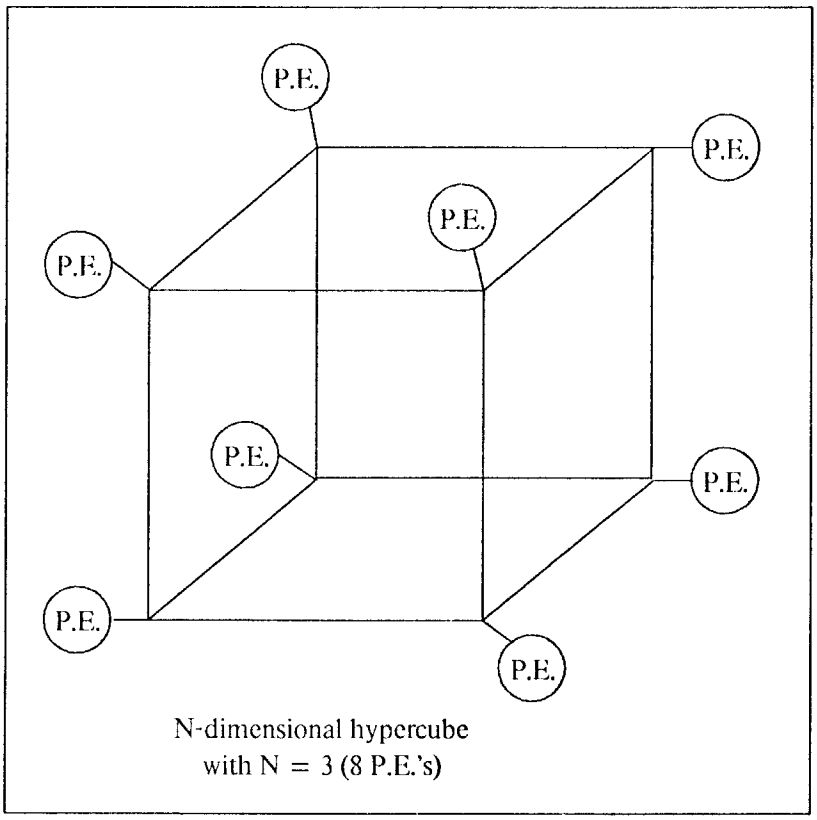

Figure 9: Structure of the Tagged-Token Dataflow Machine

3.3 TTDA Implementation under the MIT MEF

The TTDA machine was actually implemented under the MIT MEF system at the MIT Laboratory for Computer Science, as the first major emulation experiment available. Functional models of all levels of the TTDA were emulated, as well as interconnection protocols as specified by an agreed specification.[35, 18] Early emulation runs with this model pointed out a severe lack of deep understanding in the area of decentralized processor cycle and memory space resource allocation and deallocation. As a first curative step, an extra processing element was added to the architectural definition (and thus the emulation experiment) to support centralized resource control; to allocate or deallocate any system resources, each T.E. had to make a request of the resource management T.E. Obviously, this became the main system bottleneck; at this writing, more experimentation is in progress to solve the problem.

This solution is a good example of a "quick fix" to an architecture used to circumvent unknowns in the problem and concentrate on the parts of the architecture that need immediate attention. In the case of the TTDA machine, the first question to be answered by a M.E.F. was the validity of the processing approach; therefore, the resource management problem was bypassed for future study without slowing down work on the processing units of the architecture. An overall sketch of the implementation functions for the TTDA looks like this:

```
procedure TTDA Processing Unit:
    loop begin
        Incoming_Token := Read_Message ();

        case Incoming_Token.D_Type

        if 0 then begin /* D=0, an ALU token.
                        Try to match with token in wait_match.
                        If single input operation, or match found,
                        call ALU to dispatch on operation type. */
            Match? := Wait_Match_Control (Incoming_Token);
            if Match? begin
                Alu_Operation := Fetch (Incoming_Token.Address);
                Outgoing_Tokens := Alu_Control (Alu_Operation);
                for each Outgoing_Token in Outgoing_Tokens do
                    Write_Message (Outgoing_Token.Destination_PE,
                                   Outgoing_Token);
                end;
            end begin;

        if 1 then begin  /* D=1, an I-structure token.
                        Dispatch on I-structure request type, satisfy
                        read/write request and return answer to
                        requesting P.E. */
            Outgoing_Token := I_Structure_Control (Incoming_Token);
            Write_Message (Outgoing_Token.Destination_PE,
                        Outgoing_Token);
            end begin;

        if 2 then begin /* D=2, a PE-control token.
                        Perform local housekeeping function. */
            Outgoing_Token := Pe_Control (Incoming_Token);
            Write_Message (Outgoing_Token.Destination_PE,
                        Outgoing_Token);
            end begin;
        end case;
    end loop;
end procedure;
```

```
procedure TTDA Resource Management Unit:
    loop begin
        Incoming_Token := Read_Message ();
        case Incoming_Token.D_Type

        if 0 then begin /* D=0, a system-manager token.  Dispatch
                           on request type, reading system state
                           to find free resources for request. */

            Outgoing_Tokens := System_Manager (Incoming_Token);
            for each Outgoing_Token in Outgoing_Tokens
                Write_Message (Outgoing_Token.Destination_Pe,
                               Outgoing_Token);
            end begin;

        if 2 then begin /* D=2, a PE-control token.
                           Perform local housekeeping function. */

            Outgoing_Token := Pe_Control (Incoming_Token);
            Write_Message (Outgoing_Token.Destination_PE,
                           Outgoing_Token);
            end begin;

        end case;
    end loop;
end procedure;
```

At emulation start-up time, some number of *TTDA Processing Units* are initialized along with a single *TTDA Resource Management Unit* to control all system resources.

Some results that show the magnitude of emulation experimentation that can proceed on a particular M.E.F., the MIT MEF, are documented below. It must be noted that the MIT MEF, at this writing, is still using low-speed (ten megabits/second) bus communications (i.e., Ethernet with Chaos protocols).

Lines of Lisp code to implement the TTDA machine:	12420
Raw computation speed (dataflow operations/second):	500
Wait/match section buffer size (tokens/PE):	1024
Program memory size (words/PE):	16384
Maximum operations computed (at this writing):	800000

This first major emulation quickly pointed out the power of an emulation facility, particularly the ease of alteration of a model experiment. This writer's favorite example of this ability centers on an anecdote: early in the project, the TTDA emulation did not support 64-bit IEEE standard floating point computation, although the TTDA definition demanded such functionality. A new programmer was put to the task of adding this ability, with only a day's familiarity of the source code (though he already understood basic dataflow concepts). He was able to add (and debug) the ALU functional code for 64-bit floating point in *one day* of work. It is precisely this reconfigurability and flexibility that one wants from an M.E.F.

3.4 Levels of Interpretation

For the implementation of the TTDA Dataflow machine under the prototype MIT MEF the author chose a level of emulation modelling each P.E. of the TTDA machine as a single T.E. in the M.E.F. domain. Therefore, each dataflow T.E. was modelled as a single sequential process emulating the TTDA instruction set and commucating with other processing elements of the emulated machines.

Obviously, this is not the only level of abstraction at which such a machine may be modelled. For example, the author could have chosen an approach in which the entire multiprocessor was modelled as a single sequential thread, with no communication necessary between elements. The implementation of such a machine simulation might have looked something like this:

```
procedure TTDA_Machine begin
    Token := Head (Tokens_Awaiting_Processing);
    Tokens_Awaiting_Processing :=
        Tail (Tokens_Awaiting_Processing);
    PE := Token.Destination_PE;

    /* Here simulate the operation of processing element
       number PE, leaving any output tokens on the queue
       Tokens_Awaiting_Processing. */
end procedure;
```

This program has no interesting properties from a multiprocessing point of view. Although it will exhibit the behavior of a TTDA machine as viewed by a *user,* it simulates none of the internal interfaces of the elements of the dataflow architecture, and is therefore not particularly useful for prototypical implementations. However, the simple single-thread approach to machine simulation is an acceptable input to an emulation facility.

Instead of going further *away* from the parallelism inherant in the dataflow design, however, we can move toward it by modelling the internal pipelining of each processing element of the TTDA. A pipelined hardware implementation of a TTDA processing element would include, for example:

- An incoming token queue manager.
- A Waiting/Matching associative token storage unit.
- An instruction fetch/ALU processing unit.
- A storage management unit.
- A PE controller (system interface) unit.
- An outgoing token queue manager.

Each of these components may be modelled as an independant process, participating states in the pipelined microarchitecture of a TTDA processing element. Each of *these* T.E.'s would actually represent only a small part of the overall machine being emulated.

Again, this model may be used to prototype a TTDA machine under a M.E.F. architecture. Although it would require more communication (and probably processing) overhead, it is closer to the true design of a TTDA machine, and therefore perhaps more useful for some studies of dataflow computation. Figure 10 outlines the three levels of interpretation noted above, with some notes on the advantages and disadvantages of each approach.

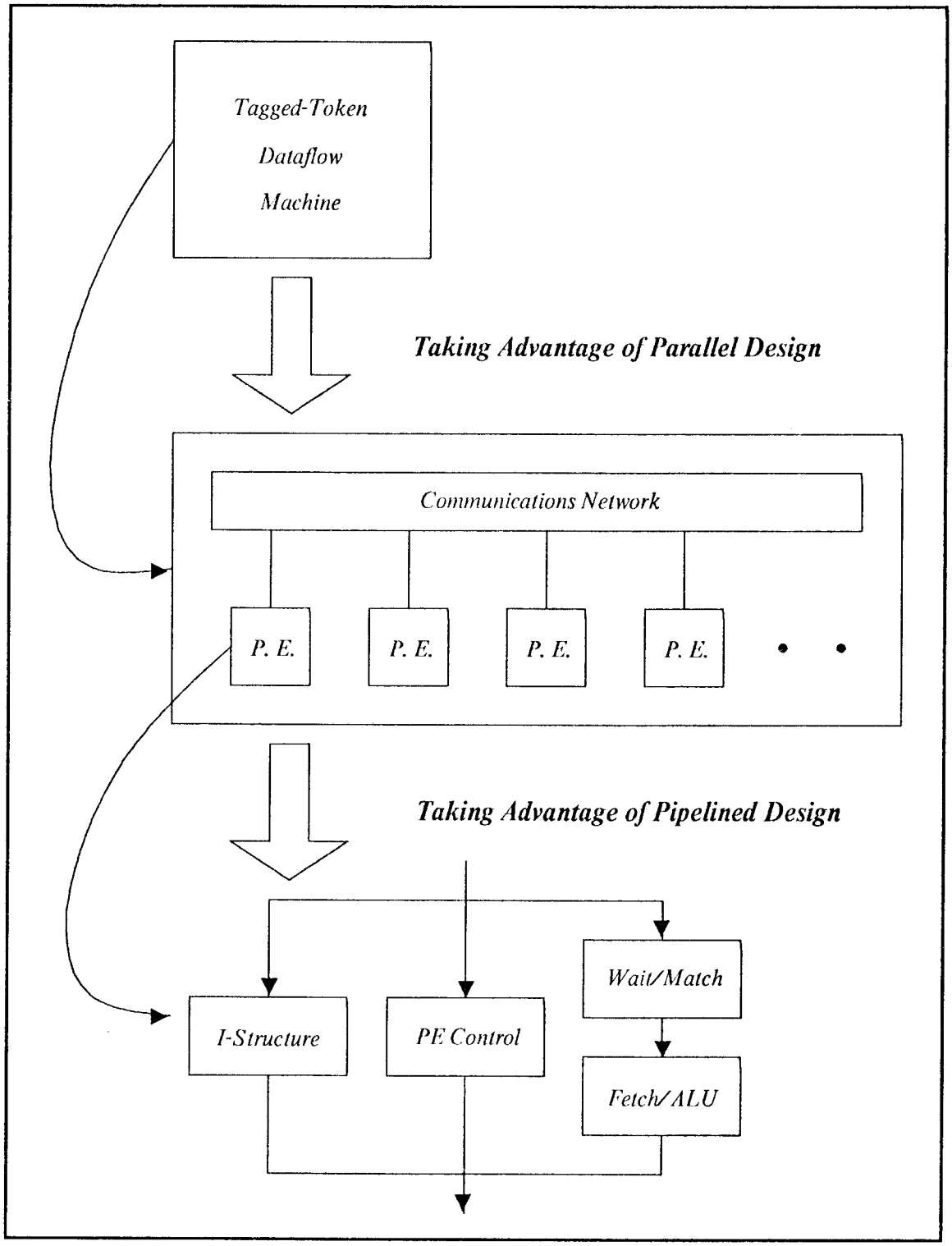

Figure 10: Three Different Approaches To TTDA Simulation

Chapter 4

Using an M.E.F. as an Emulator

In the preceding chapters, we used the term "emulation" quite loosely to represent any use of an M.E.F. to interpret an architectural definition outlined by a machine designer. In this and following chapters, we will narrow this definition to a particular method of functional and time simulations of systems.

4.1 Interpretation, Simulation, and Emulation

As was noted in the last chapter, a system may be *interpreted* at any level. In the case of the Tagged-Token Dataflow Architecture, this means that we may (1) interpret the machine code for each target element; or, at a higher level, we may (2) interpret the machine as a whole by executing high-level dataflow languages such as Id;[8] or, at a lower level, we may (3) interpret the intra-machine flow of control and data between the pipeline stages of a dataflow processing element. However, this choice does not affect the *scheduling* of the simulated activities of a system (i.e., the T.E.'s) during simulation; thus, it does not force such a simulation to divulge timing information about a machine model executing under an M.E.F. environment.

In this and the succeeding chapter, we will discuss two different general methods for garnering timing information from emulation experiments along with the simple functional execution of the machine model. We divide the term *"simulation"* into three concepts:

- *Interpretation,* as outlined above, specifies only that a functional specification of a machine is executed. No implicit timing information collection is suggested.

- *Simulation,* as will be outlined in the following chapter, specifies a system interpretation that explicitly keeps timing information about the

functional units of a simulated machine (T.E.'s) while the simulation is taking place.

- *Emulation,* the topic of this chapter, specifies a system interpretation that keeps no *explicit* timing information, but gathers such information *implicitly* through use the of explicit scheduling of T.E. activity.

Figure 11 outlines the hierarchy of simulation methods as this thesis views it.

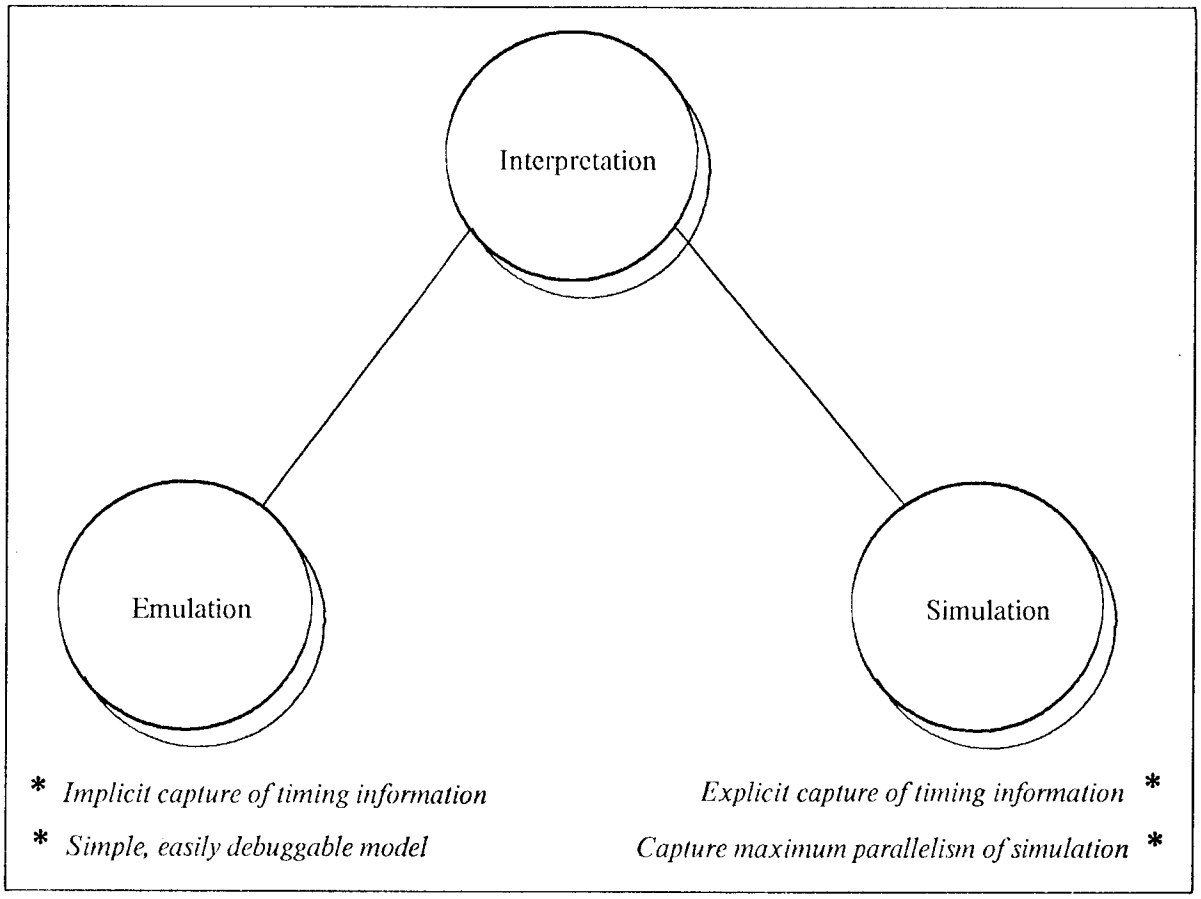

Figure 11: Types of System Interpretation/Emulation/Simulation

The definition of emulation that we use in this chapter is best observed in such systems as the Yorktown Simulation Engine (YSE).[36] This system, though it provided architectural designers with exact timing analyses of their logic circuit, had no explicit overhead involved in maintaining timing information during simulation. Instead, the YSE

relied on timing constraints "programmed into" the circuit entered for simulation. Such combinatorial logic circuits consisted of multiple stages of logic; a single stage would be simulated before the next stage was begun. Thus, timing and inter-element dependence information was implied by the structure of the circuit as it was described to the machine.

Likewise, an M.E.F. views emulation as a simulation activity for which it need keep no timing information.* Instead, an exact schedule of events necessary for the emulation of a system are specified with the definition of an emulation experiment, along with the functional definition of the target elements of the emulation. This simulation schedule can be as simple as a round-robin operating system-like process scheduler, or as complex as a full load-balancing priority scheduler. A complete M.E.F. implementation will allow any user-specified schedule for T.E.'s executing on a single physical processor, along with a method of specifying which T.E.'s will execute on each P.P.**

4.2 Scheduling as an Approach to Emulation

In fact, a simple round-robin scheduling scheme is exactly the right model for the emulation of certain systems. For example, emulation of a balanced pipeline, in which each pipeline stage takes the same amount of real time (or at least a set amount of real time) to execute is a perfect target architecture for emulation-style modeling. Figure 12 outlines an execution schedule for a three-processor system, each processor of which is composed of a balanced three-stage pipeline. Representing each stage of the system as a T.E., a trivial scheduler for each triplet of T.E.'s running on a P.P. would simply execute the functional description of each stage one after the other, and then repeat. The time that the real system would have taken to execute the pipeline four times, for instance, is easily calculated as four times the runtime of the pipeline; the time taken to *emulate* the pipeline is irrelevant.

---

* Note, however, that several methodologies for maintaining simulated timing information automatically will be advanced in the following chapter.

** Note that by this definition, at the current time the MIT MEF is not a complete M.E.F. implementation.

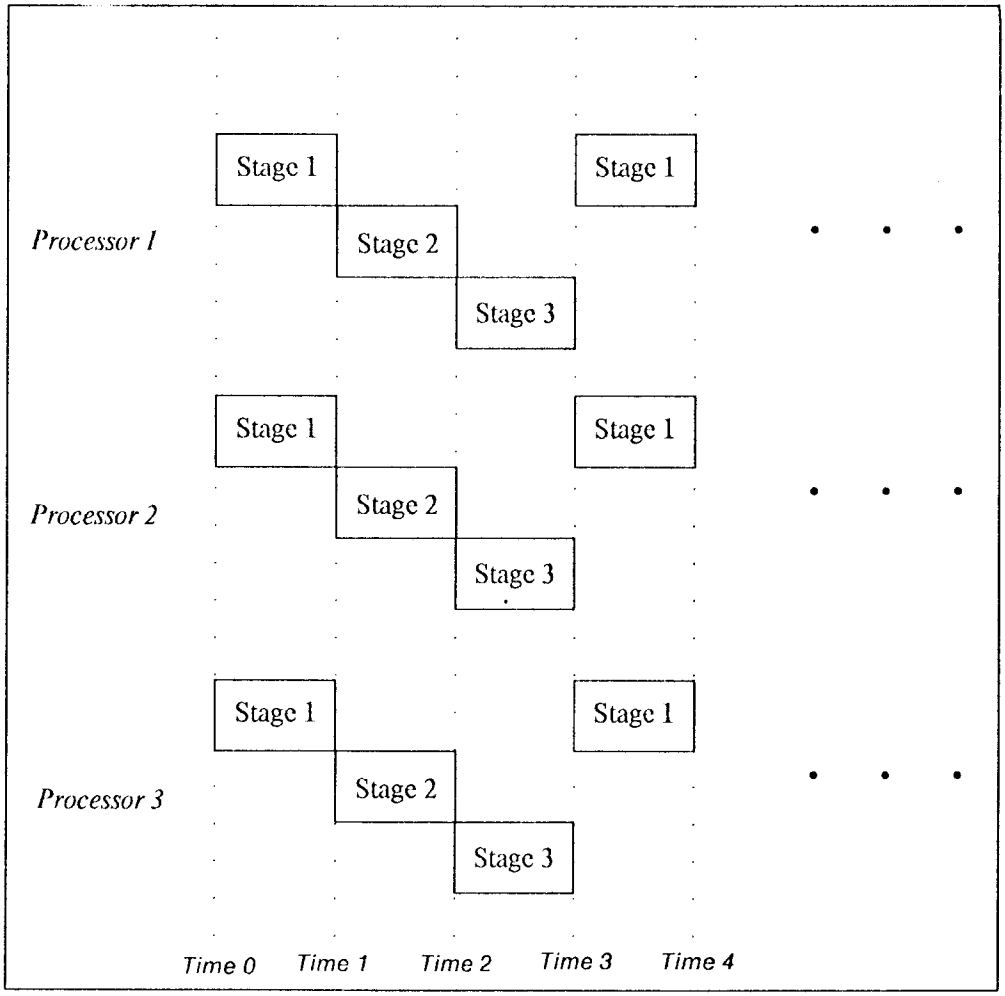

Figure 12: Scheduling Stages of a Balanced Pipeline

The main disadvantage of this model of interpretation is the inherent mismatch between the M.E.F. implementation model and any shared-resource (e.g., shared memory) multiprocessor model. However, there are quite attractive reasons for using system emulation as a simulation methodology. Emulation under the M.E.F. environment is a good "impedance match" between the architecture of an M.E.F. and the general emulation architecture. The M.E.F. is composed of general-purpose multiprogramming processors connected via some selection of high-speed networks; a real pipelined multiprocessor is also a set of multiprogramming (although space-division instead of time-division) processors communicating via some (hopefully) high-speed network. Therefore we should be able to

keep the overhead of emulation rather low, allowing emulation of relatively complex machines executing large problems. This is exactly what is necessary to make a multiprocessor architecture successful; one must build the machine and convince enough applications programmers that the overhead of attempting their application on the machine is worth the time necessary to rewrite their applications for the new environment.

Chapter 5

Using an M.E.F. as a Simulator

The uses of a facility such as that outlined in this thesis continue far beyond the two interpretation experiments presented in previous chapters. As even the von Neumann machine experiment showed, modeling of various quite different levels of parallelism can be accomplished, including architectures that do not seem to map directly to the abstract architecture of the facility itself.

The abstract structure outlined in this paper was chosen as the most general of multiprocessor configurations, sort of the Turing machine of the multiprocessor world. The skeptic will immediately note that it does not, however, directly support such multiprocessor design ideas as synchronous processors (like the Connection Machine, or Illiac IV)[15, 4] or a shared memory model (e.g., C.mmp).[5] However, this model is abstract enough to allow prototypical implementation of even these diverse models of parallel computation; generally, another virtual processing element is added to the emulation experiment definition to model this shared resource (e.g., clock or memory) of the system. This is actually quite close to the real hardware implementation of such a system, in which a central clock for synchronous operation or a central shared memory will actually be a separate subsystem of the architecture. In addition, interconnection schemes other than packet-switching networks can be simulated on top of our packet switch by using the packets as individual items in a continuous stream communication scheme, and so on.

5.1 A Globally Synchronized Architecture

As an example of an extension, we present a particular method of simulating a globally clocked architecture on top of an M.E.F. Other methods are possible, of course, but the most obvious relies on excessive constraints on message passing between logical processors. To alleviate the expense of this method, we present an design which corresponds quite neatly to a globally clocked architecture on an intuitive level.

We first refer the reader back to figure 4, in which the general simulation or emulation experiment schema is displayed. Below, in figure 13, is a modified copy of the schema, with a new processing element, named *clock,* added.

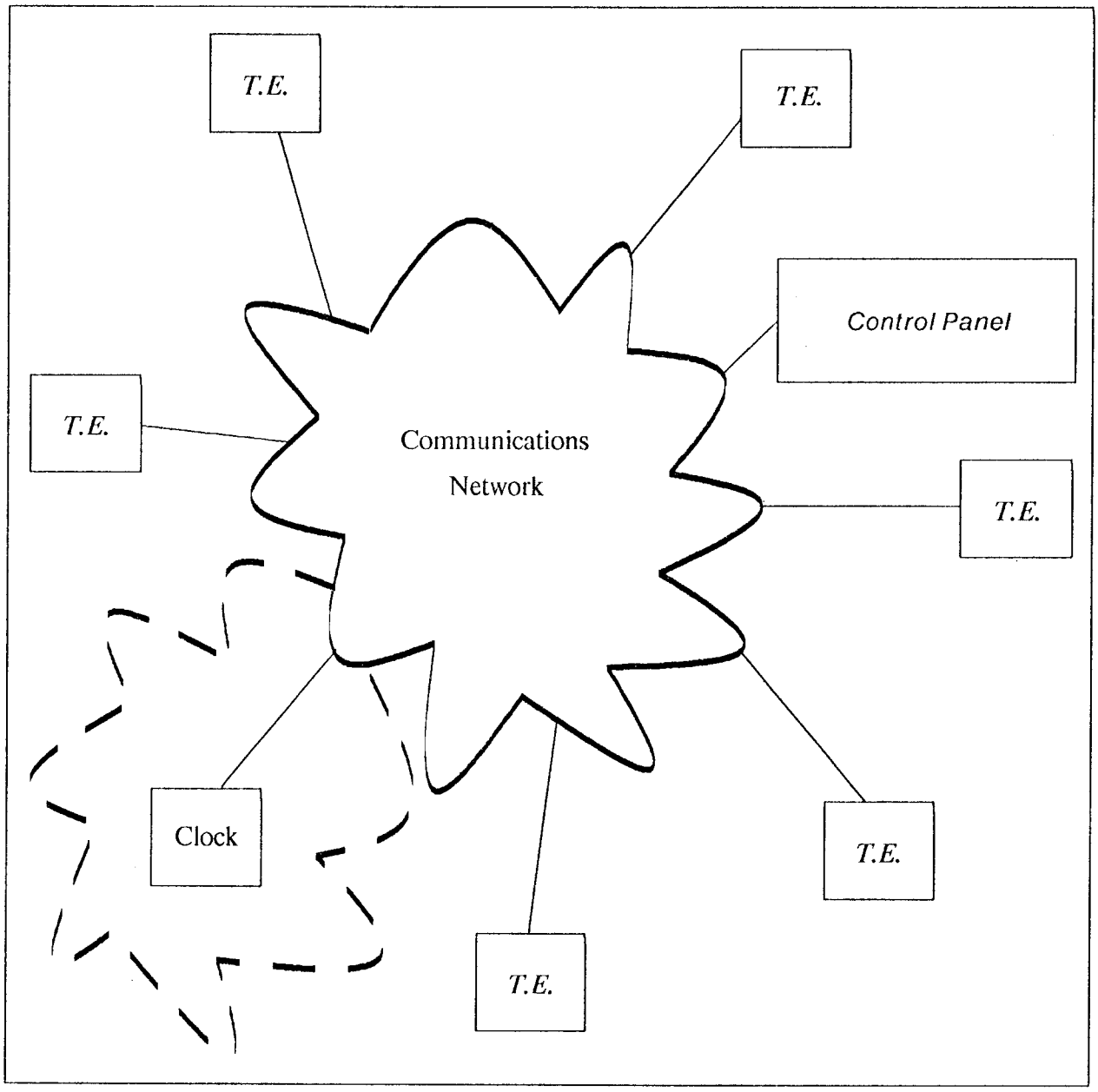

Figure 13: General Emulation Scheme for a Clocked Architecture

In figure 13, each *T.E.* corresponds to a processing element in an emulation experiment's definition, as usual. However, one special-purpose processing element, the *clock,* is added for global synchronization of processing. The clock T.E. has a simple functional definition:

```
do forever begin
     for each TE in {all normal T.E.'s} do
          send (pe, "Clock Pulse");

     {Wait for each T.E. to acknowledge the pulse.}
end
```

Each normal T.E. in the emulation must then be sure to wait for a *"Clock Pulse"* message before beginning any single synchronized computation, and then send an acknowledgment to the Clock T.E. when the computation is complete. As long as this protocol is adhered to, all activity in the emulation experiment will occur in a globally synchronized manner.

In a similar manner, any prototypical architecture with *central resource needs,* such as a global clock, may be emulated on an M.E.F. For instance, a central (*shared*) memory system may be emulated by adding a single special T.E. (perhaps named *memory*) which emulates a memory subsystem by receiving read/write requests and processing them in a synchronized manner, much as the memory processing element in chapter 3.

Not to belabor the point, this scheme of virtual expansion of the basic M.E.F. architecture can be accomplished for arbitrarily centralized or synchronized systems. For instance, figure 14 is a good emulated realization of a physically separated telecommunications system, with multiple independent processing elements on each end (for instance, for some complex image processing task), with each of the processor groups under the control of a central synchronization clock.

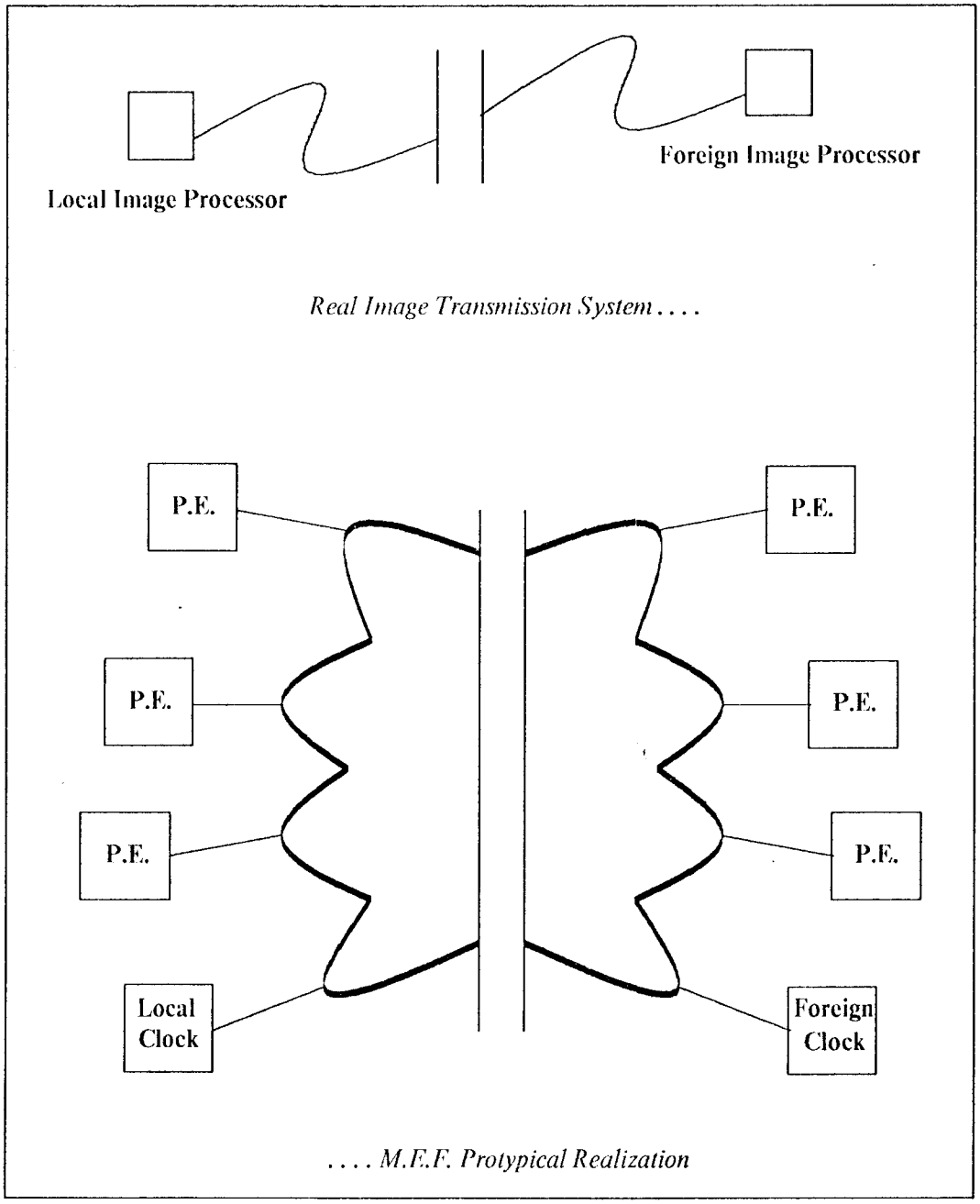

Figure 14: Emulation of a Physically Separated Two-Domain System

Emulation of systems such as that cursorily outlined in figure 14, with two physically separate but cooperating multiprocessor systems operating in two independent clock domains, can help reveal (and thus solve) major skewing problems. For instance, the simple global clocking emulation protocol mentioned above for a single time domain system certainly

applies to each of the processor groupings (each clock synchronizes the actions of its local processing elements). However, it does not solve the usual problems of multi-domain clock skew that can be introduced by simulation speed differences between the two processor groupings.

For example, suppose that the *Local* grouping of figure 14, because of the nature of its task or its particular implementation as an emulation experiment, acted far faster than the emulated computation speed of the *Foreign* grouping. An M.E.F. could quickly become totally overrun by queued messages pending for processing elements in the *Foreign* grouping; this is essentially the cognate of clock skew in real multi-clocked hardware realizations.

As in the real hardware case, however, this can be solved; the solution that is simplest for implementation in an M.E.F. environment suggests hardware implementations for system clock skew. Basically, a new protocol is added between the local and foreign *clock T.E.'s* of the emulation to cause each clock to temporarily stop pulsing whenever the other clock is skewed by more than some arbitrary number of pulses, waiting for the foreign activity to "catch up." Our simple clock T.E. definition from above would be changed to:

```
integer local_pulse_number, foreign_pulse_number

when {Foreign Clock Pulse Received}
     Foreign_Pulse_Number = {Clock Pulse Number Received}

Local_Pulse_Number = 1

do forever begin
     for each TE in {all normal T.E.'s} do
          Send (pe, "Clock Pulse");

     {Wait for each T.E. to acknowledge the pulse.}

     send (foreign_clock_pe, Local_Pulse_Number)

     Local_Pulse_Number = Local_Pulse_Number + 1

     wait until
          Local_Pulse_Number - Foreign_Pulse_Number < Max_Skew
end
```

The ease with which our earlier clocking example can be changed in the light of a new situation, and the close mapping of that change to hardware realizations of solutions of like problems, is the exact purpose of an M.E.F.

5.2 Distributed Simulation Approach to Synchronization

The last section presented a simple paradigm for modeling globally clocked architectures. Though the presented system is easy to understand and implement, it obviously wastes much of the parallelism available in an M.E.F. Since some of the tasks performed by simulated processes (T.E.'s) will be completed faster than others, some T.E.'s will remain idle, waiting for clock pulses, which could in turn cause part of the M.E.F. substrate (physical processors) to idle.*

Another solution to the problem of simulation of system-wide clocking is available, which can be used to potentially speed up simulations of architectures in an M.E.F. setting. There is of course a trade-off, namely the amount of communications overhead. This scheme is used in distributed simulation systems to increase simulation throughput (not simulat*ed* throughput, but the throughput of the simulation iteself) by distributing the clocking of the simulated system.

Basically, the scheme as outlined by Bryant[37, 38] consists of a per-T.E. current view of the global clock value, kept up-to-date by the local T.E. with no central control. In addition, a per-T.E. queue of simulation activities that the T.E. is not yet prepared to simulate is kept. Each data message transmitted from one T.E. to another carries either a *stimulus* (event to be simulated), or a *timestamp*, which records the earliest time that that message could have been transmitted in the real system which is being simulated. When a T.E. acts upon an incoming message, it

1. Calculates the output stimulus message to be issued, and

---

* Again we must be careful to separate the *simulated machine* from the real globally clocked machine, the latter of which *must* idle some of its hardware if it is implemented using standard clocking architectures.

2. Calculates a new clock value for itself, which may also issued to T.E.'s downstream.

The new clock value within a T.E. is the earliest time that the global clock could have counted to within that T.E.; in other words, it is the earliest possible simulation time for any *unreceived messages to be simulated.* Thus, it provides the time against which already-received messages must be compared to decide if they are "safe" to simulate. An event is "safe" to simulate if it is the next event that would have occurred in the real system.

Note that this methodology does *not* provide a way to simulate events of a real system out-of-order; that cannot be done in the general case in which T.E.'s are non-functional (i.e., have local state). It *does,* however, provide a method of sequencing events in the T.E.'s in a way that preserves time ordering but requires no system-wide communications or shared resources (besides the communications network itself). Nevertheless, the method avoids deadlocking.[38] A definition of a T.E. which follows the basic schema of Bryant follows:

```
/* Definition of the initial source (bootstrap) T.E. */
procedure Source T.E. begin

    /* Send out the initial messages. */
    for each Message in Initial_Stimulus_Messages do
        Send_Message (Message_Type=Event_Stimulus, Message);

    /* Notify all TE's immediately downstream we're done. */
    for each TE in Downstream_TEs do
        Send_Message (TE, Message_Type=Increment_Clock,
                          Source_TE=Local_TE,
                          New_Time=INFINITY);
end procedure;
```

```
/* Standard simulation element definition . */
procedure T.E. Definition:
static integer Clock initial 0,
       integer Old_Clock initial -1,
       integer Upstream_Clocks (Number_of_Upstream_TEs)
               initial 0,
       list    Pending_Events initial NIL;

while Clock < INFINITY begin

    /* Process all messages representing simulation
       events which are now safe to simulate. */
    while Pending_Events not equal NIL do begin
        Event := Pop_List (Pending_Events);
        Clock := Simulate_and_Send_Output_Messages (Event);
    end;

    /* If the simulation of pending events moved the clock,
       send messages to all downstream TE's to notify. */
    if Clock > Old_Clock then begin
        for each TE in Downstream_TEs do
            Send_Message (TE, Message_Type=Increment_Clock,
                              Source_TE=Local_TE,
                              New_Time=Clock+Delay(TE));
        Old_Clock := Clock
    end;

    /* Process any incoming messages from upstream TE's. */
    for each Message in Receive_Messages () do begin
        if Message.Message_Type = Event_Stimulus then
            Add_to_List (Message, Pending_Events);
        else begin
            Upstream_Clocks(Message.Source_TE)
                := Message.New_Time;
            Clock := Max (Clock,
                          Minimum_of_Array (Upstream_Clocks))
        end
    end;
end while;
end procedure;
```

The added clock delay apparent in the clock incrementation code for a regular (non-source) T.E. above is a symptom of a problem with this methodology for distributed simulation. Given the existence of cycles in the interconnection of T.E.'s (which would represent feedback loops and the like in real systems), the local clock values in any two T.E.'s of a cycle cannot differ be more than the sum of the event delays around the cycle, which might unnecessarily reduce asynchrony and thus parallelism of the simulation. In addition, given the possibility of zero-time simulation events, it might be possible that a cycle of T.E.'s might get "stuck" at a particular clock value if a pending event of zero simulation time was triggered in the cycle. A zero-time simulation event would trigger no update of the local clock, and therefore might not allow further event simulation in the local T.E. nor in other T.E.'s in the cycle. Thus, an artificial delay may be introduced for any inter-T.E. connection arc which "breaks" this possible zero-length cycle. In other words, the artificial delay factors are introduced so that the minmum delay around any given cycle in the network of simulated T.E.'s is greater than zero, preventing deadlock.

This scheme, though it allows much more parallelism in the simulation of a timed architecture, does have the disadvantage of adding communication overhead which can become cumbersome, especially as the time skew between local T.E. values of the global clock becomes large. However, the scheme has the useful feature of provable deadlock avoidance, given the assumptions that (1) processes only output messages at firing time, and then remain silent until the next set of inputs are available; and (2) unbounded output buffering is available. Given these assumptions, it has been shown[38, 39] that such a simulation system can only deadlock when all processes are waiting for input, which cannot happen. In addition, there is a variant of this scheme, named *Time Acceleration,*[37], which requires static analysis of the interconnections of T.E.'s in a system to be simulated and guarantees far more asynchrony in the simulation of a system model.

Chapter 6

Conclusions: Future Directions

6.1 Other Uses of an M.E.F.

The ease with which emulation experiments can be altered and re-executed on an M.E.F. hints at another, related, capability. Because of its malleability, an M.E.F. can be used to emulate *any* process or system that can be modeled as a group of communicating processes. The scope of such systems is huge; we present here some examples, with some discussion of implementation designs under an M.E.F. scheme.

6.2 Message Passing Computational Models

A common theme in much multiprocessor research today, particularly in the artificial intelligence community, is representation of computation by *message-passing* "agents," each executing on separate processing elements and communicating via some interconnection network. A good example of this approach is the Actor II language and Apiary archicture of Hewitt.[17] This model of computation is an excellent analog of the M.E.F. general model, as it represents computation as a set of communicating sequential processors. A.I. applications written as message passing activities with no shared state fit exactly the M.E.F. general abstraction.

6.3 Protocol Testers

Besides the obvious multiprocessor scheme noted, Chapter 3 hinted at another good use for an M.E.F. In that chapter, we discussed a trivial computer architecture, viewing the M.E.F. communications substrate as a interconnection bus. We introduced a simple protocol for CPU/memory communication based on that substrate, and noted how expansion of such a protocol could proceed.

In fact, an M.E.F. does not limit us to such a simple interconnection scheme or protocol, as could be clearly seen in the hypercube network topology presented for dataflow experimentation in Chapter 3. In fact, arbitrary interconnection schemes transmitting via arbitrary protocols can be emulated under an M.E.F. environment, to allow objective measurement of performance, simplicity, and leanness of a protocol design. It has been noted that techniques for protocol specification and validation, though under development, are currently quite primitive;[40] we believe that an M.E.F. can help alleviate this problem.

We envision use of an M.E.F. to emulate bus protocols, network protocols, and even *internet* protocols, viewing processing nodes as communication hosts, bridges, gateways, or even *other networks* in internet experimentation. In the simple host to host case, an M.E.F. can be used to test, validate, and evaluate protocol implementations at any network interface layer.

Figure 15 shows a test configuration for validation of interconnection protocols. The processing element labeled *"validation processor"* watches a stream of commands and requests passing between the processing elements implementing a protocol under test. This validation processor architecture can be can be used as part of a standards specification, with competing implementations executing as the processor under test, thus forming a fast, reliable checkpoint for standards validation without high implementation expenses for vendors attempting to provide protocols.

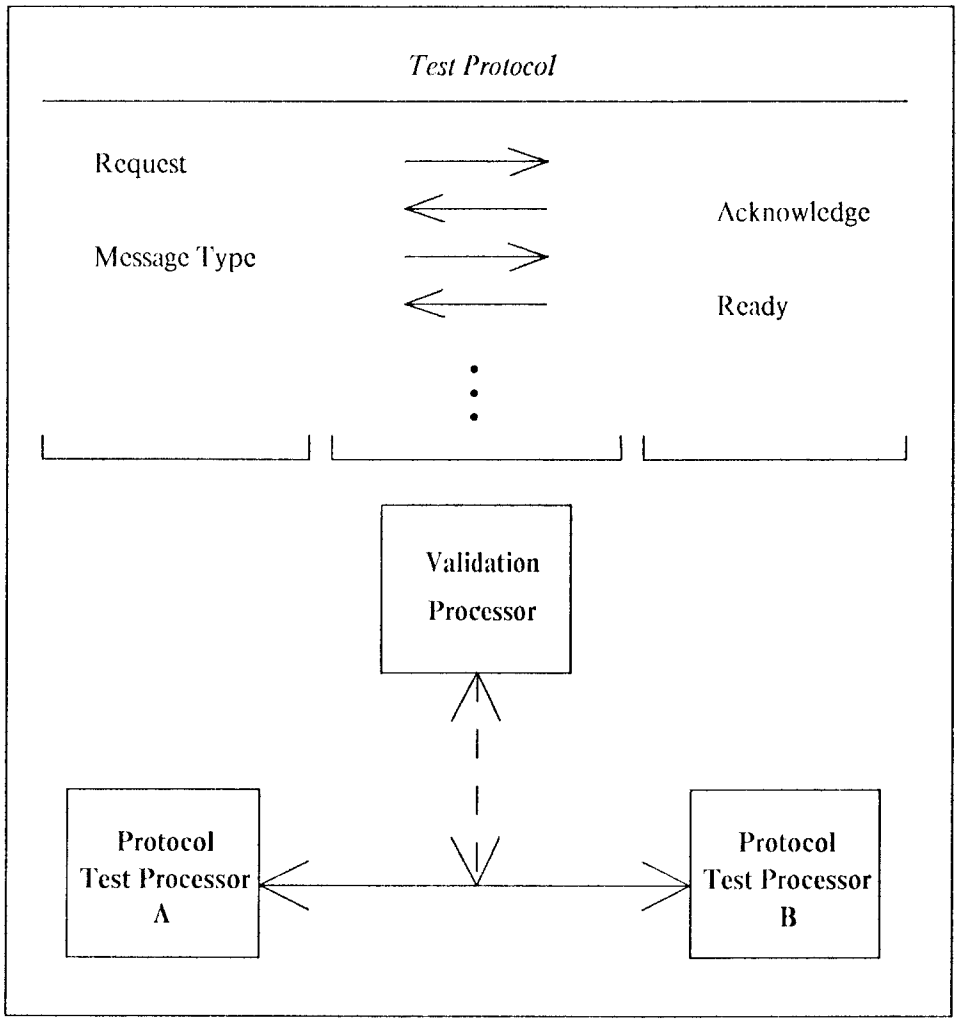

Figure 15: Validation of a Test Protocol Implementation

6.4 Education

An M.E.F. is also a perfect instructional tool for use by students of computer architecture. It provides a cheap, fast tool for construction and test of architectures by students of computer science. For example, students can be given example computer architectures as projects to implement under an M.E.F., thus gaining true understanding of architectural features such as addressing, microarchitectures, and so forth. Parts of such architectures can be supplied as separate T.E.'s in an emulation, supplying basic functionaility

that instructors do not wish to see in students' work. For example, a student can implement the instruction fetch and effective address decode portion of an architecture to learn a complex addressing scheme, while the instructor supplies a basic CPU function and a memory processor that accepts only absolute addresses, such as the one described in Chapter 3.

In addition, validation techniques similar to those mentioned in the section above can be used to administer testing and grading of student projects, by watching and evaluating the runtime behavior of architectural implementations. Since descriptions of machine architectures can be given and debugged quickly and easily in an M.E.F. environment, students can gain hands-on experience of the basic building blocks of computer science.

6.5 Other Distributed System Uses

Many other computer architectures, particularly distributed ones, can be emulated, and thus evaluated, in an M.E.F. environment. These include distributed database systems, fault tolerant systems, communications networks (such as telephone systems), and so on. The list is endless. Each can be described in terms of processes communicating via fixed or alterable communications paths; this meta-architecture maps directly to the M.E.F. abstract model of processing elements communicating via the M.E.F. substrate.

6.6 Human Networks and Other Systems

In fact, there is no reason that the uses of an M.E.F. must be limited to high technology applications. Many real-life situations in other sciences can be modelled in terms of communicating processes. For instance, the asynchronous nature of an M.E.F. lends itself to studies of stochastic processes and other statistical studies.

Outside of the realm of the "hard sciences," an M.E.F. can still be useful in such studies as economic systems, geopolitical forecasting, and any other application requiring large and granular modelling. The ability to alter emulations with little or no overhead especially makes such a use of an M.E.F. desirable.

6.7 Conclusion

The thesis of this work is that an advanced tool for the emulation of computer architectures (particularly multiprocessor architectures) is an extremely helpful component of an effort in computer architectural design. Our examples of emulation experiments actually designed for and executed on the MIT MEF clearly outline the successful application of fast protyping methods and flexible debugging and testing features to the design and implementation of complex systems.

This work has already led to the completion of a small, slow emulation facility; we hope that it will also lead to larger facilities with more computational horsepower, enhanced functionality, and more simple interfaces for architectural experimenters. Certainly the extremely complex design demands in the realm of superfast multiprocessor computers call for more wide-reaching tools for testing new ideas in a timely manner, and we believe that the M.E.F. idea is the foundation of such a set of tools.

6.8 Impact on Future Machine Design

The September, 1984 issue of *Spectrum* magazine included a feature series entitled, *In Pursuit of the One-Month Chip,* which outlined the methods and hopes for the future in the area of very fast prototyping of integrated circuits (VLSI) to implement various processor architectures and other electronic designs.[41] However, it completely shrugged off the entire higher-level design problem, stating merely that "a well-understood design must be used." In fact, the tools to accomplish such a higher-level design do not exist at the level of complexity needed to complement the other circuit and chip design tools presented in the article.

The M.E.F. described in this thesis performs exactly those functions, providing a "multiprocessor sandbox" in which ideas can be quickly invented, attempted, and either discarded or completed in very little time. Though more work, particularly in the area of user-controlled dynamic network load scheduling and other resource allocation issues, is needed to complete the ideas of M.E.F. structure, the structure outlined in this thesis provides a strong basis for a useful and viable alternative and addition to the architectural design schemes of the present. The impact on future computer design projects is potentially great, as the full capabilities of prototypical emulations are realized.

References

1. M. Bardon, et. al., *A National Computing Environment for Academic Research,* National Science Foundation, July, 1983.

2. R. M. Russell, The Cray-1 Computer System, *Comm. of the ACM,* Vol. 21, No. 1, January 1978.

3. M. J. Kascic, *Vector Processing on the Cyber 200,* Control Data Corporation, 1979.

4. W. J. Bouknight, S. A. Denenberg, D. E. McIntyre, J. M. Randall, A. H. Sameh, D. L. Slotnick, The Illiac IV System, *Proc. of the IEEE,* Vol. 60, No. 4, April 1972.

5. H. H. Mashburn, The C.mmp/Hydra Project: An Architectural Overview, in *Computer Structures: Principles and Examples,* D. P. Siewiorek, C. G. Bell, A. Newell (Eds.), McGraw-Hill, New York, 1982.

6. A. W. Burks, H. H. Goldsine, J. von Neumann, Preliminary discussion of the logical design of an electronic computing instrument, from *Collected Works of John von Neumann,* Volume 5, A. H. Taub (Ed.), MacMillan, New York, 1963.

7. T. Agerwala and Arvind, Data Flow Systems, *Computer,* February 1982.

8. Arvind, K. P. Gostelow, and W. E. Plouffe, *An Asynchronous Programming Language and Computing Machine,* Dept. of Information and Computer Science Report TR 114a, University of California, Irvine, December 1978.

9. G. M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, *Proc. 1967 AFIPS Spring Joint Computer Conf.,* March 1967.

10. J. E. Thornton, *Design of a Computer, The Control Data 6600,* Scott, Foresman and Co., Glenview, Ill., 1970.

11. D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, Dependence Graphs and Compiler Optimizations, *Proc. 8th ACM Symp. Principles Programming Languages*, January 1981.

12. J. B. Dennis, Data Flow Supercomputers, *Computer*, November 1980.

13. Arvind, V. Kathail, and K. Pingali, *A Data Flow Architecture with Tagged Tokens*, Laboratory for Computer Science, Technical Memo 174, MIT, Cambridge, MA, September 1980.

14. J. Rattner, W. W. Latten, Ada determines architecture of 32-bit microprocessor, *Electronics*, February 24, 1981.

15. W. D. Hillis, *The Connection Machine (Computer Architecture for the New Wave)*, M.I.T. Artificial Intelligence Lab Memo 646, September 1981.

16. T. W. Malone, R. E. Fikes, M. T. Howard, *Enterprise: A Market-like Task Scheduler for Distributed Computing Environments*, Working Paper, Xerox PARC, October 1983.

17. C. E. Hewitt, The Apiary Network Architecture for Knowledgeable Systems, *Conference Record of the 1980 Lisp Conference*, Stanford, 1980.

18. Arvind, M. L. Dertouzos, and R. A. Iannucci, A Multiprocessor Emulation Facility, TR-302, Laboratory for Computer Science, MIT, Cambridge, MA, October 1983.

19. Charles A. Hornig, *private communication* on the subject of failure of a Honeywell Multics development system in a multiprocessing environment for the first time.

20. Gregory M. Papadopoulos, *A NuBus Channel Adapter*, Tanglewood Design Note 13, M.I.T. Laboratory for Computer Science, May 1985.

21. D. Weinreb, and D. Moon, *Lisp Machine Manual*, M.I.T. Artificial Intelligence Laboratory, July 1981.

22. Barbara Liskov, Alan Snyder, Russel Atkinson, and Crai Schaffert, Abstraction Mechanisms in CLU, *Communications of the ACM*, Volume 20, Number 8, August 1977.

23. A. Bensoussan, C.T. Clingen, and R.C. Daley, The Multics Virtual Memory: Concepts and Design, *Comm. of the ACM*, Vol. 15, No. 5, May 1972.

24. *Reference Manual for the Ada Programming Language, Proposed Standard Document*, United States Department of Defense, July 1980.

25. K. Thompson & D. M. Ritchie, The UNIX Time-sharing System, *Communications of the ACM*, Vol. 17, July 1974.

26. Symbolics, Inc., *PROT: Networks and Protocols*, Vol. 7 of Release 5.0 documentation set, March 1984.

27. Rettberg, R., C. Wyman, D. Hunt, M. Hoffman, P. Carvey, B. Hyde, W. Clark, and M. Kraley, *Development of a Voice Funnel System: Design Report*, Tech. Report 4098, Bolt Beranek and Newman, Inc., August, 1979.

28. G. L. Steele, *The Definition and Implementation of a Computer Programming Language Based on Constraints*, M.I.T. Dept. of EE & CS Ph.D. Thesis, August 1980.

29. J.E. Rodriguez, *A Graph Model for Parallel Computations*, MAC-TR-64, Laboratory for Computer Science, MIT, Cambridge, MA, September 1969.

30. David E. Culler, *Resource Management for the Tagged-Token Dataflow Architecture*, S. M. Thesis, MIT Dept. of EE & CS, Cambridge, MA, December 1984.

31. A. J. Catto, & J. R. Gurd, Resource Management in Dataflow, *Proc. 1981 ACM Conf. Functional Programming Lang. and Computer Arch.*, October 18-22, 1981.

32. Donald W. Oxley, Motivation for a Combined Data Flow - Control Flow Processor, *Proc. SPIE 25th Annual Symp.*, 1981.

33. D. Comte, N. Hifdi, & J. C. Syre, The Data Driven LAU Multiprocessor System: Results & Perspectives, *Information Processing 80*, S. H. Lavington (Ed.), North-Holland, 1980.

34. Arvind & Robert A. Iannucci, A Critique of Multiprocessing von Neumann Style, *Proc. of the 10th Int'l. Symp. on Computer Architecture*, June, 1983.

35. Arvind & Robert A. Iannucci, *Instruction Set Definition for a Tagged-Token Data Flow Machine,* Computation Structure Group Memo 212-3, MIT Laboratory for Computer Science, February 1983.

36. G. F. Pfister, The Yorktown Simulation Engine: Introduction, *Proc. 19th Design Automation Conference,* Las Vegas, 1982.

37. Randal E. Bryant, *Simulation on a Distributed System,* Computation Structures Group Memo 182, MIT Laboratory for Computer Science, July 1979.

38. Randal E. Bryant, *Simulation of Packet Communication Architecture Computer Systems,* M.I.T. Lab. for Computer Science Technical Report 188, November 1977.

39. K. Mani Chandy, and Jayadev Misra, Distributed Simulation: A Case Study in Design and Verification of Distributed Programs, *IEEE Trans. Software Eng.,* Vol. SE-5, No. 5, September 1979.

40. Louis Pousin & Hubert Zimmerman, A Tutorial on Protocols, *Proc. IEEE,* Volume 66, Number 11, November 1978.

41. F. Guterl, In Pursuit of the One-Month Chip, *IEEE Spectrum Magazine,* Volume 21, Number 9, September 1984, New York, New York.

Appendix

Reference Manual:
The MIT Multiprocessor Emulation Facility

This appendix is a reference manual for the M.E.F. implemented by the author at the M.I.T. Laboratory for Computer Science, in the Functional Languages and Architectures group, as part of the MIT MEF project. In the discussions below, we assume familiarity with the Lisp Machine operating system and the Lisp Machine dialect of Lisp. The abstraction and implementation of the MIT MEF are discussed elsewhere in this document.

1. Emulation Experiment Description

In order to emulate a particular computer architecture, the MIT MEF must have some *static* experiment definitions as well as some *dynamic* run-time communications support. The *Emulation Experiment Description* describes the toplevel block structure of an architecture to be emulated. The following functions are defined:

(define-emulation-experiment *experiment-name &rest keywords*)

This function gives the MEF all of the basic information about emulation experiment *experiment-name*. The following alternating keyword/value pairs may be used:

:interactive <T / NIL> — Specifies whether this experiment is to be interactively executed, or via remote batch facility. The number of processors in an experiment may only be left for actual execution time if this keyword is set to **T.**

:long-name <name> — Specifies a "pretty name" for this experiment, for use in output labelling.

:number-of-processors <number / :read> — Specifies the number of logical (target) processing elements (T.E.'s) that should be emulated by the MEF for this experiment. The value ":read" specifies that this number should be prompted for at emulation execution time.

:configuration <config> — Specifies a configuration map for

this emulation experiment. The configuration map notes how many T.E.'s of each T.E. type in the current experiment should be executed, and to what logical "destination addresses" they should respond. The format of <config> is a list, each element of which is a two-element list. The first element of this sublist should be either a logical destination number, or a list of (*from to*), where *from* and *to* are expressions denoting a range of logical destination numbers. These expressions may be arithmetic expressions on the variable "*:N*", which will be "bound" to the total number of T.E.'s that are being executed.

For example, a configuration map such as "*((0 CPU) (1 MEMORY))*" specifies that there will be two T.E.'s total; at logical address *0* is a T.E. of type *CPU*, while at logical address *1* there will be a T.E. of type *MEMORY*. A more advanced map is used by the Tagged-Token Dataflow emulation; it uses

((0 (- :n 2)) normal-process) ((- :n 1) manager-process))

which means, allocate *:N — 2 "normal-process"* T.E.'s at addresses *0* through *:N — 2*, and allocate a single *"manager-process"* T.E. at address *:N — 1*.

:message-handler <function> — Specifies that control panel *user messages* (defined below) should be handled by the function <function>.

:shutdown-handler <function> — Specifies that the function <function> should be called when the current emulation is completed, and the user has requested that the MEF shutdown operations. This hook can be used to collect statistics, etc., when an experiment is completed.

:left-graph (<label> <scale> {<nbars>}) —
:right-graph (<label> <scale> {<nbars>}) — Requests that the left (right) bar graph display area of the MEF control panel be reserved, with the label <label>, a bottom-to-top scale of <scale>, and a number of bars to be displayed equal to <nbars>. The default number of bars if not specified is equal to the number of T.E.'s emulated.

:routing-paradigm <type> — Notes that this emulation will follow the given T.E. to T.E. routing paradigm. The MEF will simulate a particular message-passing paradigm only to the extent of recording how many messages are forwarded by each node of a simulated network. The <type> specification can be *:NONE*, meaning that routing will not be simulated; *:STATIC*, meaning that the routing table given by the *:routing-map* option should be used, or *:DYNAMIC*, which notes that the routing information is not stable, and should be recomputed from the

:routing-map function at each message-pass.

:routing-map <map> — Specifies the map of T.E. to T.E. routing to be simulated by this emulation experiment. The <map> may be a complete route specification, such as:

((1 2) (1 2 3) (2 3 1) (2 3) (3 1) (3 1 2))

which specifies the routes from every T.E. to every other T.E. (for example, to get from T.E. number two to T.E. number one, the route is T.E. 2 —> T.E. 3 —> T.E. 1); or it may be a function. This function takes three arguments (sending-T.E., destination-T.E., total number of T.E.'s in this experiment) and should return a list noting the T.E.'s through which the message should travel.

:pe-implementation <type> — Specifies in which manner T.E.'s should be executed. The *:NORMAL* option notes that T.E.'s should be represented as Lisp Machine processes. The faster, but less general, *:FUNCTIONAL* option requests that T.E.'s be implemented merely as calls to functions, which return after processing an incoming message (thereby bypassing process-switching overhead). The fastest scheme, *:ONE-PER-PROCESSOR*, requests that T.E.'s be represented as direct calls to functional definitions within the MEF overseer process, with state represented as Lisp global variables. This last option, though it causes emulated machine execution to proceed more quickly, requires that no more than one T.E. may be executed on any physical processor.

(define-processor-variables *variable-set-name &rest variables***)**

Defines a new set of T.E. state variables (*registers*) for use by T.E. functional definitions. The *variable-set-name* may be specified in following *define-processor* forms to note what registers are used by such T.E. definitions; in the Lisp code for those T.E.'s, the register names noted in the accompanying *define-processor-variables* form may be used freely as Lisp global variables. The syntax of the *variables* portion of this form are variable names or lists of the form (*variable-name initialization*).

(define-processor *processor-type emulation-name variable-set start-function status-function relative-load-factor &rest pe-meters***)**

Defines a new T.E. type for the *emulation-name* emulation experiment, named *processor-type* for which a functional definition will be specified. The T.E. registers to be used by this definition are declared to be the set *variable-set.* The function called to initiate this type of T.E. is *start-function;* this function will be called with the logical address of the T.E. being started and the total number of T.E.'s in the running emulation. The *status-function* is called by the MEF to obtain the current status of a running T.E.; it is called in the scope of that T.E.'s registers, and must return a string noting the status of the T.E. The *relative-load-factor* should be a number between

zero and one noting the relative CPU load for this type of T.E.; this information is used to do per-physical-processor load balancing. The *pe-meters* named will be noted as per-T.E. meters for each T.E. of this type.

(define-meter *meter-name emulation-name***)**
Defines a system-wide meter named *meter-name* accessable to (and shared by) all T.E.'s in emulation experiment *emulation-name.*

(defmef *function-name experiment-name argument-list &body body***)**
Defines a function *function-name* as part of the functional definition of one of the T.E. types of *experiment-name.* The *argument-list* and *body* arguments are the Lisp function definition.

2. Communication and Metering Functions

In addition to the above purely static (definitional) support functions, the MEF system includes many Lisp functions to interface to run-time communications and metering facilities. These may be broken down into two classes: functions used by T.E. functional definitions, and functions to interface to the MEF Control Panel. We begin by listing the MEF functions available to T.E. function definitions.

(write-message *destination-processor array &optional start end***)**
Sends a message to the logical T.E. numbered *destination processor.* The message to be sent should be in *array,* an **art-8b** or **art-string** Lisp array. *Start* and *end,* if given, should be an inclusive starting index of the message in *array* and an exclusive ending index in *array,* in the standard Lisp Machine Lisp style. Note that the receiving T.E. may be *any* logical T.E. in the current emulation, regardless of on which physical processor such T.E. is executing or what interconnection schemes are necessary to transmit the message. The sender also need not wait for message transmission completion or retransmit requests, as all transmission details are handled by the MEF. It must be noted that the receiving T.E. does *not* get any information from the MEF as to which T.E. sent the message; if such information is necessary, the transmitting T.E. should include its logical address in the message.

(read-message-byte)
Reads the next available incoming byte from any logical T.E. transmitting to the current T.E. This is useful to dispatch on packet types, etc. This function will block on pending input if there is no input available.

(read-message *array &optional start end***)**
Read any available incoming message that the MEF has directed from another logical T.E. to the current T.E. The message will be read into *array,* which must be an **art-8b** or **art-string** array. If *start* and *end* are given, the incoming message will be read into *array* starting at the inclusive index *start* and ending at the exclusive index *end.*

Otherwise, any incoming message will be used to fill the entire array. This function will hang until the enough incoming bytes are received to fill the specified portion of *array*.

(increment-meter *meter-name*)

Increments (by one) the system-wide or local-T.E. meter *meter-name*. *Meter-name* must have been declared previously by **define-emulation-experiment** or **define-processor.**

(decrement-meter *meter-name*)

Decrements (by one) the system-wide or local-T.E. meter *meter-name*. *Meter-name* must have been declared previously by **define-emulation-experiment** or **define-processor.**

(clear-meter *meter-name*)

Clears (sets to zero) the system-wide or local-T.E. meter *meter-name*. *Meter-name* must have been declared previously by **define-emulation-experiment** or **define-processor.**

3. Use of the Control Panel

In order to provide a *"bootstrap processor"* environment for configuring, starting up, and shutting down emulated architectures, the MEF includes a *Control Panel* system, available to Lisp Machine users by selecting the MEF window (via <Select> — Period or the System Menu). The frame that will be displayed looks like figure 16.

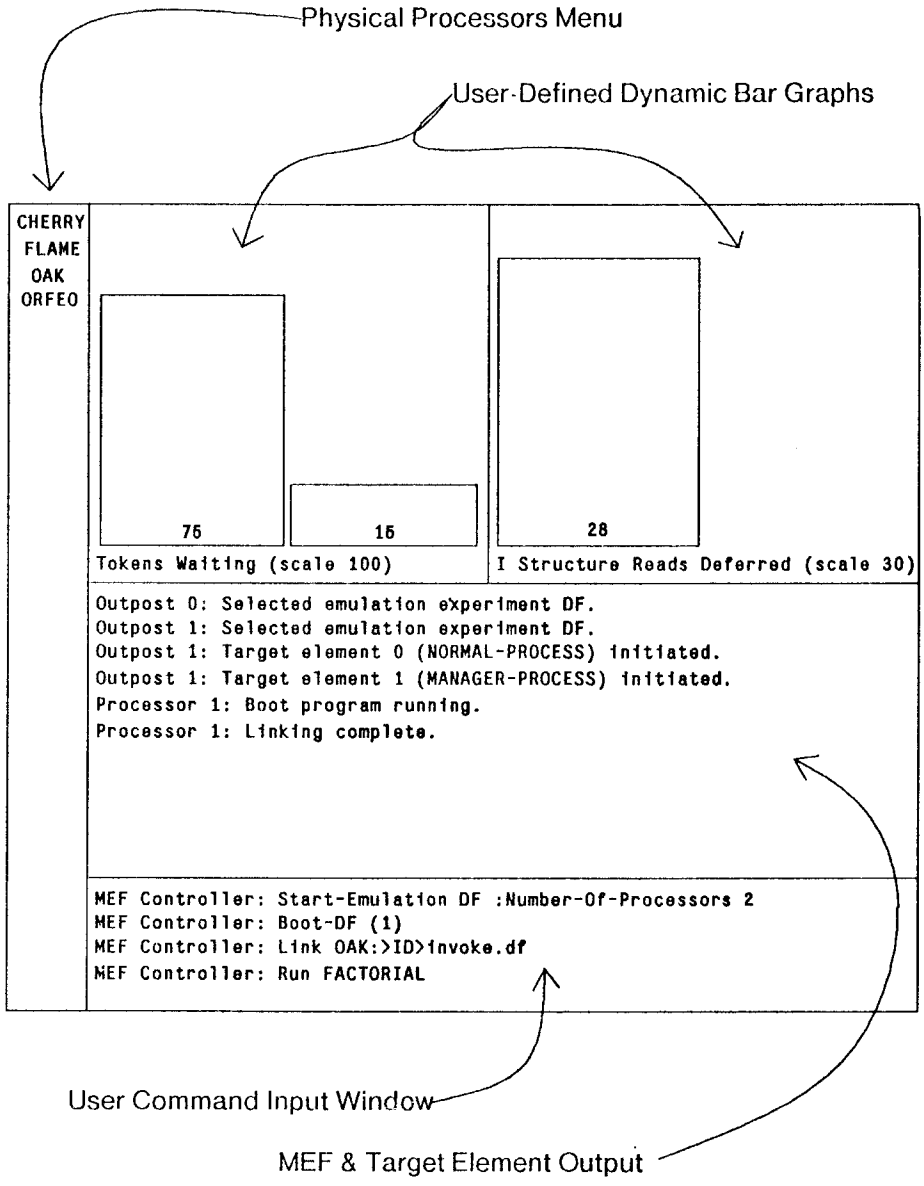

Figure 16: The MEF Control Panel

Clicking on the physical processor names in the menu on the left side allows the user to list the real Lisp processes' and emulated T.E. processes' status on foreign physical processors, or to terminate such processes. In addition, the following commands may be typed to the Control Panel's input window to use the MEF system:

Clear-Configuration

Clears any cached information about the physical configuration of the MEF system

(i.e., which physical processors the MEF system is currently using).

Configure *Emulation {:Number-Of-Processors Integer}*

Configures the physical description of a MEF system, allowing the user to specify the physical Lisp Machine processors that are to take part in the current emulation facility. This is the main interface for the support of the MEF's reconfigurability and partitionability. If the system has not yet been configured, this command will first ask for the source of a list of processors. Three answers are possible:

E (Enter List) — The names of physical processors to take part in this use of the MEF are taken from the keyword, one at a time. This list should be terminated by pressing the <End> key.

R (Read from File) — The names of physical processors to take part in this use of the MEF are taken from a file; the MEF will prompt for the pathname of the file. The file should contain physical machine names, one on a line. Lines beginning with semicolons are ignored, and may be used for comments.

F (Find Automatically) — All physical processors at the local development site that are capable of running MEF are automatically found and used as part of the current emulation facility configuration. This process takes a few minutes.

After the physical processors to be used in the current configuration have been specified, connections to each machine are made and a MEF system process is created on each participating machine. If an emulation name was specified in the command, the **Start-Emulation** command is executed with that name.

Debug-Processor *Address*

The logical T.E. executing with destination address *Address* is halted and forced into the Lisp debugger, with display routed automatically to the MEF Control Panel, allowing the user to inspect and modify the internals of the T.E. as well as restart or terminate its function.

More-Processing *[:On | :Off]*

Turns more processing in the (central) MEF output window of the Control Panel on or off, to allow leisurely inspection of its output or to ignore it.

Outpost-Login *User-Id Host-Name*

Prompts for a password (which is not echoed), and causes all foreign processors participating in the current emulation to log in to *Host-Name* with the login identifier *User-Id,* so that foreign machines may freely use file systems other than their own.

Set-Bar-Graph *[:On | :Off]*

Turns the Control Panel bar graph displays on or off, to allow faster operation of the MEF.

Set-Tracing *[:Off | :Local | :Central]*

Reroutes all tracing messages sent by logical T.E.'s. If *:Off* is selected, no tracing messages are displayed anywhere; *:Central* specifies that all messages should appear in the output window of the Control Panel, and *:Local* requests that tracing messages appear on the screens of originating physical processors.

Shutdown *:Complete*

Shuts down the current emulation experiment, terminating all communications and T.E. implementations. If *:Complete* is specified, all foreign MEF processes are also destroyed and inter-machine communciations dropped.

Start-Emulation *Emulation :Number-Of-Processors Integer*

Begins the emulation experiment named *Emulation*, which must have been defined previously via **define-emulation-experiment.** If the *:Number-Of-Processors* option is used, then the number of T.E.'s specified will be used; otherwise, if the emulation does not specify a number of T.E.'s to be used in this emulation, the user is requested to supply a number.

Status *[Pe-Number | :All]*

Sends status messages to the T.E. at logical address *Pe-Number*, or to all T.E.'s if *:All* is specified. The MEF system actually handles this request by calling the user-supplied status function in the context of running T.E.'s.

4. Functional Interfaces to the Control Panel

The following Lisp functions may be used by T.E. implementation functions to interact with the MEF Control Panel functions:

(write-error-message *format-control-string &rest format-arguments*)

Write a logical processor error message to the MEF controller for display. The message, generated by the **format** arguments given, is displayed in the output window of the Control Panel prefixed by the sender's T.E. logical address.

(write-display-message *format-control-string &rest format-arguments*)

Write a logical processor message, not denoting any error, to the MEF controller for display. The message, generated by the **format** arguments given, is displayed in the output window of the Control Panel prefixed by the sender's T.E. logical address.

(write-user-message *array &optional start end*)

Write a "user message" to the emulation experiment-defined message handler

residing in the controller. When this message is received by the Control Panel, the "user message" handler (defined by the *:message-handler* option to **define-emulation-experiment**) is invoked with the message as an argument. An additional argument, a stream on which to do output, also is handed to the message handler when it is invoked.

(send-tracing-message *format-control-string &rest format-arguments*)
Print a tracing message on the current tracing output stream, which is set via the **Set-Tracing** Control Panel command.

(send-graph-message *graph-display-select bar-number height*)
Sends a message to the Control Panel updating one of the bar graph displays. *Graph-display-select* may be zero or one, signifying the left and right bar graph displays. *Bar-number* selects the bar to update, while *height* specifies the relative height to which to set the bar. If the bar graph displays are disabled (via the **Set-Bar-Graph** Control Panel command), the bar graphs are not updated (and no communications overhead is incurred).

(inject-message *destination-processor array &optional start end*)
Send a message to a logical processor, pretending to be from another logical processor. This function may be used by the bootstrap functions of an emulation experiment to start up an emulated machine.

(read-meter *emulation-meter-name*)
Reads the current value of the system-wide meter *emulation-meter-name* and returns it as a Lisp integer.

(average-meter *emulation-meter-name*)
Reads the current value of the system-wide meter *emulation-meter-name* and averages it over the number of logical T.E.'s executing in the current emulation experiment.

(read-processor-meter *processor-meter-name*)
Reads the current value of the per-T.E. meter *processor-meter-name* for each logical T.E., and returns an array of values returns as well as the number of T.E.'s that supplied values for that meter.

5. Example

The following pages contain an example of an emulation experiment, named SIMPLE since it defines a trivial two-T.E. von-Neumann architecture comprised of a CPU and a memory box.

```
;;; -*- Mode: Lisp;  Package: (SIMPLE GLOBAL 1000.);  Base: 10. -*-
;;;
;;; Definition of the SIMPLE emulation experiment, which emulates
;;; a standard von Neumann architecture consisting of a CPU and a
;;; MEMORY, connected via a "bus" simulated by the MEF.

;;; Definitional interfaces to the MEF.
;;; This form specifies the name of the experiment and its configuration.
(MEF:define-emulation-experiment
  "SIMPLE" :interactive 't
           :long-name "Simple CPU//Memory Machine"
           :number-of-processors 2
           :configuration '((0 cpu) (1 memory))
           :left-graph '("Memory Accesses//Instruction" 30. 1)
           :right-graph '("Stack Depth" 100. 1))

;;; Where the stack begins in memory.  Stack grows upward.
(defconstant stack-base 2048.)

;;; The CPU part of the system has all of the standard parts.
;;; Note that we "cache the stack on the processor board;" i.e., the stack
;;; is separate from the memory.
(MEF:define-processor-variables cpu
  (pc 0)                ; Program counter.
  (cc 0)                ; Condition code.
  (fp stack-base)       ; Frame pointer.
  (sp stack-base)       ; Stack pointer.
  (registers (make-array 8. :type art-q :initial-value 0)))

;;; The MEMORY part of the system simply contains a small 32-bit wide memory.
(MEF:define-processor-variables memory
  (memory (make-array 4096. :type art-q :initial-value 0)))

;;; Notify the MEF of these two new kinds of processor.
(MEF:define-processor cpu simple cpu start-cpu cpu-status 1.0)
(MEF:define-processor memory simple memory start-memory memory-status 1.0)

;;; Definition of the INSTRUCTION abstraction; an instruction to the CPU
;;; stored in the MEMORY's memory.  An INSTRUCTION is composed of
;;; 6 bits of opcode and two addresses, each composed of
;;; 3 bits of address code, and 10 bits of address.
(defmacro instruction-opcode  (word) '(%logldb (byte  6. 26.) ,word))
(defmacro instruction-adcode1 (word) '(%logldb (byte  3. 23.) ,word))
(defmacro instruction-addr1   (word) '(%logldb (byte 10. 13.) ,word))
(defmacro instruction-adcode2 (word) '(%logldb (byte  3. 10.) ,word))
(defmacro instruction-addr2   (word) '(%logldb (byte 10.  0.) ,word))

(defmacro make-instruction (opcode adcode1 addr1 adcode2 addr2)
  '(%logdpb ,opcode (byte 6. 26.)
            (%logdpb ,adcode1 (byte 3. 23.)
                     (%logdpb ,addr1 (byte 10. 13.)
                              (%logdpb ,adcode2 (byte 3. 10.) ,addr2)))))

;;; These are the defined address codes, or simple addressing modes.
(defconstant register-adcode 0) ; Address is register number.
(defconstant memory-adcode 1)   ; Address is straight into memory.
(defconstant stack-adcode 2)    ; Address is stack offset.
(defconstant immediate-adcode 3); Address is immediate.
(defconstant fp-adcode 4)       ; Address is frame offset.
(defconstant arg-adcode 5)      ; Address is argument number.

;;; Functions for defining and storing CPU ALU functions.
(defvar alu-functions (make-array 64. :type art-q :initial-value nil))

(defmacro define-alu-function (name number arglist &body body)
  '(progn 'compile
          (setf (aref alu-functions ,number) ',name)
          (defun (:property ,name alu-function) ,arglist . ,body)
          (defprop ,name ,number alu-function-number)))
```

```
;;; Definition of the stack frame.  All numbers are positive offsets frame ptr (FP).
(defconstant old-fp 0)     (defconstant old-sp 1)      (defconstant old-cc 2)
(defconstant old-pc 3)     (defconstant frame-size 4)

;;; Constant PE numbers.
(defconstant cpu-pe 0)     (defconstant memory-pe 1)

;;; Condition codes
(defconstant cc-clear 0)   (defconstant cc-overflow 1) (defconstant cc-negative 2)
(defconstant cc-zero 3)    (defconstant cc-positive 4)

;;; Memory requests.
(defconstant memory-read-request #/R)
(defconstant memory-write-request #/W)
(defconstant memory-load-request #/L)

;;; Standard message buffers for inter-processor interaction.  These
;;; message buffers by convention are two words (eight bytes) long.
;;; Protocols are stored in the top byte, two byte fields in the
;;; low bytes of the first word, and word fields in the second word.
(defmacro message-type (message-buffer) '(aref ,message-buffer 3))

(defmacro put-two-byte-field (number message-buffer)
  (once-only (number)
    '(progn (setf (aref ,message-buffer 0) (ldb (byte 8 0) ,number))
            (setf (aref ,message-buffer 1) (ldb (byte 8 8) ,number)))))

(defmacro get-two-byte-field (message-buffer)
  '(dpb (aref ,message-buffer 1) (byte 8 8) (aref ,message-buffer 0)))

;;; The command interface for the controller screen while the emulation is running.
;;; Array to assemble SIMPLE code into.
(defresource assembly-array () :constructor (make-array 256. :type art-q)
        :initial-copies 1)

;;; The message buffer (see above) for the command interface.
(defvar inject-message-buffer (make-array 8 :type art-8b :initial-value 0))
(defvar inject-message-word (make-array 2 :type art-q
                                          :displaced-to inject-message-buffer))

;;; Command to link/load a file of SIMPLE code.
(define-command assemble simple "Assemble and load a function in SIMPLE code."
  ((:arguments (file :pathname "File to assemble and load." :noise-string "file")))
  (using-resource (array assembly-array)
    (let ((end (assemble file array 0))
          (bytes (make-array 1024. :type art-8b :displaced-to array)))
      (setf (message-type inject-message-buffer) memory-load-request)
      (put-two-byte-field 0 inject-message-buffer)
      (setf (aref inject-message-word 1) end)
      (MEF:inject-message memory-pe inject-message-buffer)
      (MEF:inject-message memory-pe bytes 0 (* end 4)))))

;;; Command to run the machine from a given PC with a set of arguments.
(define-command run simple "Run the SIMPLE emulation."
  ((:arguments (arguments :integer "Arguments to the loaded function."
                          :noise-string "arguments" :times (0 *)))
   (:control-arguments (start :pc :integer "Initiate program counter."
                              :noise-string "starts at" :default 0)))
  (format t "~&Starting at PC ~D, with ~D argument~:P.~%" start (length arguments))
  (setf (aref inject-message-buffer 0) (length arguments))
  (MEF:inject-message cpu-pe inject-message-buffer 0 1)
  (loop for number in (reverse arguments) doing
        (setf (aref inject-message-word 0) number)
        (MEF:inject-message cpu-pe inject-message-buffer 0 4))
  (setf (aref inject-message-buffer 0) start)
  (MEF:inject-message cpu-pe inject-message-buffer 0 1))
```

```
;;; Definition of the SIMPLE CPU processor.

;;; The CPU's message buffer (see above).
(defvar cpu-message-buffer (make-array 8 :type art-8b :initial-value 0))
(defvar cpu-message-word
  (make-array 2 :type art-q :displaced-to cpu-message-buffer))

;;; Total number of instructions/memory accesses
;;; executed since last RUN command.
(defvar total-instructions 0)
(defvar memory-accesses 0)

;;; This is the toplevel of the CPU.  It reads incoming messages,
;;; which at this level are assumed to be simply arguments and starting PC's.
(defun start-cpu (ignore ignore)
  (loop doing
        (*catch 'abort-cpu
          (setf sp stack-base fp stack-base cc 0 pc 0)
          (loop doing
                ;; Clear instruction count and access count.
                (setq total-instructions 1 memory-accesses 0)
                ;; Read in any arguments, push them on the stack.
                (loop repeat (MEF:read-message-byte) doing
                      (MEF:read-message cpu-message-buffer 0 4)
                      (stack-push (aref cpu-message-word 0)))
                ;; Read in the starting address, push phony stack frame.
                (initiate-call (MEF:read-message-byte))
                ;; Run.
                (*catch 'toplevel (run-cpu))
                ;; Pick up return value and display it.
                (MEF:send-graph-message 0 0 0)
                (MEF:send-graph-message 1 0 0)
                (MEF:write-display-message "The answer is ~D."
                                           (stack-pop))))))

;;; Standard function for ascertaining the status of a CPU processor.
(defun cpu-status ()
  (format nil "CPU. PC = ~D, FP = ~D, SP = ~D, CC = ~D" pc fp sp cc))

;;; Push a stack frame and update the PC if necessary.
(defun initiate-call (&optional new-pc)
  (memory-write (+ sp old-fp) fp)
  (memory-write (+ sp old-sp) sp)
  (memory-write (+ sp old-cc) cc)
  (memory-write (+ sp old-pc) pc)
  (setf fp sp sp (+ sp frame-size) cc 0)
  (and new-pc (setf pc new-pc)))

;;; Fetch a single word from memory.
(defun memory-fetch (address)
  (incf memory-accesses)
  (setf (message-type cpu-message-buffer) memory-read-request)
  (put-two-byte-field address cpu-message-buffer)
  (MEF:write-message memory-pe cpu-message-buffer 0 8)
  (MEF:read-message cpu-message-buffer 0 4)
  (aref cpu-message-word 0))

;;; Write a single word to memory.
(defun memory-write (address value)
  (incf memory-accesses)
  (setf (message-type cpu-message-buffer) memory-write-request)
  (put-two-byte-field address cpu-message-buffer)
  (setf (aref cpu-message-word 1) value)
  (MEF:write-message memory-pe cpu-message-buffer 0 8))
```

```
;;; Real instruction processor.  Loops on instruction at PC doing execution.
(defun run-cpu (&aux instruction)
  (loop doing
        ;; Update memory accesses/instruction and stack size bars.
        (MEF:send-graph-message
          0 0 (fixr (* 10.0 (// (float memory-accesses) total-instructions))))
        (MEF:send-graph-message 1 0 (- sp stack-base))

        ;; Fetch instruction at PC.
        (setq instruction (memory-fetch pc))
        (incf total-instructions)
        (incf pc) (decf memory-accesses)

        ;; Call the referenced function, bind results and branch if necessary.
        ;; We should get back an "answer" and a "disposition" which specifies
        ;; the kind of instruction which got executed.
        (multiple-value-bind (answer disposition)
            (funcall (get (aref alu-functions (instruction-opcode instruction))
                          'alu-function)
                     (dereference (instruction-adcode1 instruction)
                                  (instruction-addr1 instruction))
                     (dereference (instruction-adcode2 instruction)
                                  (instruction-addr2 instruction)))
          (selectq disposition
            (:store  (store-value answer
                                  (instruction-adcode1 instruction)
                                  (instruction-addr1 instruction)))
            (:branch (if (= answer 1)
                         (setf pc (instruction-addr1 instruction))
                         (setf pc (instruction-addr2 instruction))))))))

;;; Dereference an address-code/address pair into a value.
(defun dereference (adcode addr)
  (select adcode
    (memory-adcode      ; Memory reference.
     (memory-fetch addr))
    (register-adcode    ; Register reference.
     (aref registers addr))
    (stack-adcode       ; Stack reference.
     (memory-fetch (+ sp (sign-extend addr))))
    (immediate-adcode   ; Immediate value.
     (sign-extend addr))
    (fp-adcode          ; Stack reference offset from frame pointer.
     (memory-fetch (+ fp (sign-extend addr))))
    (arg-adcode         ; Stack reference to an argument.
     (memory-fetch (+ (- addr) fp -1)))
    (T (ferror "Unknown address code ~D." adcode))))

;;; Store a value at a location specified by an address-code/address pair.
(defun store-value (value adcode addr)
  (select adcode
    (memory-adcode      ; Memory reference.
     (memory-write addr value))
    (register-adcode    ; Register reference.
     (setf (aref registers addr) value))
    (stack-adcode       ; Stack reference.
     (memory-write (+ sp (sign-extend addr)) value))
    (immediate-adcode   ; Illegal immediate value.
     (ferror "You may not store into an immediate value."))
    (fp-adcode          ; Stack reference offset from frame pointer.
     (memory-write (+ fp (sign-extend addr)) value))
    (arg-adcode         ; Stack reference to an argument.
     (memory-write (+ (- addr) fp -1) value))
    (T (ferror "Unknown address code ~D." adcode))))

;;; Sign extend a ten bit address to a 32-bit LispMachine fixnum.
(defun sign-extend (10bit-number)
  (if (zerop (ldb (byte 1 9.) 10bit-number)) 10bit-number
      (%logdpb -1 (byte 22. 10.) 10bit-number)))
```

```
;;; The ALU functions that are to be emulated.
;;;
;;; LOAD and arithmetic functions.

(define-alu-function load 0 (ignore value) (values value :store))

(define-alu-function add 1 (value1 value2)
  (arithmetic-processor (+ value1 value2)))

(define-alu-function mult 2 (value1 value2)
  (arithmetic-processor (* value1 value2)))

(define-alu-function sub 3 (value1 value2)
  (arithmetic-processor (- value1 value2)))

(define-alu-function div 4 (value1 value2)
  (arithmetic-processor (fix (// (float value1) value2))))

(defun arithmetic-processor (number)
  (cond ((or (floatp number) (bigp number))
         (setf cc cc-overflow) (setq number 0))
        ((minusp number) (setf cc cc-negative))
        ((zerop number) (setf cc cc-zero))
        (T (setf cc cc-positive)))
  (values number :store))

;;; Stack operations.

(define-alu-function push 10. (value ignore) (stack-push value))

(define-alu-function pop 11. (ignore ignore) (values (stack-pop) :store))

(defun stack-push (value) (memory-write sp value) (incf sp))

(defun stack-pop () (decf sp) (memory-fetch sp))

;;; Branching functions based on input values.

(define-alu-function bneg 20. (value ignore) (if (< value 0) (values 2 :branch)))

(define-alu-function bpos 21. (value ignore) (if (> value 0) (values 2 :branch)))

(define-alu-function bzero 22. (value ignore) (if (zerop value) (values 2 :branch)))

;;; Branching function based on condition code.
(define-alu-function bcc 23. (condition ignore)
        (if (= condition cc) (values 2 :branch)))

;;; Call/Return and assorted.
(define-alu-function call 30. (ignore ignore) (initiate-call) (values 1 :branch))

(define-alu-function return 31. (value ignore)
  (let ((new-fp (memory-fetch (+ fp old-fp)))
        (new-sp (memory-fetch (+ fp old-sp)))
        (new-cc (memory-fetch (+ fp old-cc)))
        (new-pc (memory-fetch (+ fp old-pc))))
    (memory-write fp value)
    (setf fp new-fp sp (1+ new-sp) cc new-cc pc new-pc)
    (if (= new-fp stack-base) (*throw 'toplevel nil))
    nil))

;;; Stop the machine (by aborting the CPU processor).
(define-alu-function stop 40. (ignore ignore) (*throw 'abort-cpu nil))
```

```
;;; A character buffer.
(defvar string (make-array 32. :type art-string :leader-list '(0)))

;;; Print out a character (or two).
(define-alu-function print 41. (char1 char2)
  (array-push-extend string char1)
  (or (zerop char2) (array-push-extend string char2))
  nil)

;;; Finish print, sending to the controller console.
(define-alu-function terpri 42. (ignore ignore)
  (MEF:write-display-message string)
  (setf (array-leader string 0) 0)
  nil)

;;; Definition of the SIMPLE MEMORY processor.

;;; The memory processor's message buffer.
(defvar memory-message-buffer (make-array 8 :type art-8b :initial-value 0))
(defvar memory-message-word
  (make-array 2 :type art-q :displaced-to memory-message-buffer))

;;; The toplevel function for the memory processor.  Read memory requests,
;;; process, and return values if necessary.
(defun start-memory (ignore ignore)
  (loop doing
        (MEF:read-message memory-message-buffer 0 8)
        (select (message-type memory-message-buffer)
          (memory-read-request  (handle-read-request))
          (memory-write-request (handle-write-request))
          (memory-load-request  (handle-load-request))
          (T   (ferror "Illegal memory request.")))))

;;; Standard function for ascertaining the status of a memory processor.
(defun memory-status ()
  (format nil "MEMORY.  Word 0 = ~D" (aref memory 0)))

;;; Handle a memory READ request.
(defun handle-read-request ()
  (setf (aref memory-message-word 1)
        (aref memory (get-two-byte-field memory-message-buffer)))
  (MEF:write-message cpu-pe memory-message-buffer 4 8))

;;; Handle a memory WRITE request.
(defun handle-write-request (&aux address)
  (setq address (get-two-byte-field memory-message-buffer))
  (setf (aref memory address) (aref memory-message-word 1)))

;;; Handle a memory bulk LOAD request.
(defun handle-load-request (&aux start length)
  (setq start (get-two-byte-field memory-message-buffer)
        length (aref memory-message-word 1))
  (loop repeat length for offset from start doing
        (MEF:read-message memory-message-buffer 0 4)
        (setf (aref memory offset) (aref memory-message-word 0))))
```

```
;;; An assembler for SIMPLE code.

;;; Read code from PATHNAME, output code into ARRAY (art-q).
;;; Puts memory it uses after end of code.
(defun assemble (pathname array &optional (offset 0) &aux list tags)
  (pkg-bind 'SIMPLE
    (with-open-file (stream pathname :direction :in :characters 't)
      (loop with token and index and arg and count = offset
            as input = (send stream :line-in)
            while (and input (not (zerop (string-length input)))) doing
            (setq index (string-search-not-set '(#\space #\tab #\c-L) input))
            (when (and index (neq (aref input index) #/;))
              (multiple-value (token index)
                (read-from-string input nil index
                                  (string-search-char #/: input)))
              (when (= (aref input index) #/:)
                (push (list token count) tags)
                (multiple-value (token index)
                  (read-from-string input nil (1+ index))))
              (multiple-value (arg index) (read-argument input index))
              (incf count)
              (push (list token arg (read-argument input index)) list))))
    (loop for (inst addr1 addr2) in (nreverse list) and count from offset
          finally (return (1+ count)) do
          (setf (aref array count)
                (%logdpb (instruction-name->number inst) (byte 6. 26.)
                         (%logdpb (clear-address addr1 tags) (byte 13. 13.)
                                  (clear-address addr2 tags)))))))

;;; Change an address that could not be computed at scan time into an address
;;; (i.e., branch and call targets and the like).
(defun clear-address (address tags &aux lookup)
  (cond ((null address) (dpb register-adcode (byte 3 10.) 0))
        ((numberp address) address)
        ((setq lookup (assq address tags))
         (dpb memory-adcode (byte 3 10.) (second lookup)))
        (T (ferror "Unknown address: ~S." address))))

;;; Find the opcode for a given instruction.
(defun instruction-name->number (name) (get name 'alu-function-number))

;;; Tools for disassembling assembled portions of memory.
(defun unassemble (array &optional (from 0) to)
  (or to (setq to (array-length array)))
  (loop with word for index from from to to doing
        (or (setq word (aref array index)) (return))
        (format t "~&~:4D: ~8A" index
                (aref alu-functions (instruction-opcode word)))
        (format-address (instruction-adcode1 word) (instruction-addr1 word))
        (format t ", ")
        (format-address (instruction-adcode2 word) (instruction-addr2 word))
        (terpri)))

(defun format-address (adcode addr)
  (select adcode
    (memory-adcode (format t "~D" addr))
    (register-adcode (format t "R~D" addr))
    (stack-adcode (format t "SP|~D" (sign-extend addr)))
    (immediate-adcode (format t "#~D" (sign-extend addr)))
    (fp-adcode (format t "FP|~D" (sign-extend addr)))
    (arg-adcode (format t "ARG|~D" addr))
    (T (format t "UNKNOWN"))))
```

```
;;; Translate a symbolic address into an address-code/address pair.
(defun read-argument (string index &aux end)
  (setq index (string-search-not-set '(#\tab #\space) string index))
  (setq end (or (string-search-set '(#\tab #\space #/, #/;) string index)
                (string-length string)))
  (cond ((mem #'string-equal (substring string index end)
              '(r0 r1 r2 r3 r4 r5 r6 r7))
         ;; A REGISTER specification.
         (values (dpb register-adcode (byte 3 10.)
                      (- (aref string (1+ index)) #/0))
                 (1+ end)))
        ((string-equal "#CC" string 0 index 3 (+ index 3))
         ;; A CONDITION CODE specification.
         (let ((word (substring string (1+ index) end)))
           (values (dpb immediate-adcode (byte 3 10.)
                        (cond ((string-equal word "CC-NEGATIVE") cc-negative)
                              ((string-equal word "CC-POSITIVE") cc-positive)
                              ((string-equal word "CC-ZERO")     cc-zero)
                              ((string-equal word "CC-OVERFLOW") cc-overflow)
                              (T (ferror "Unrecognized condition: ~S" word))))
                   (1+ end))))
        ((= (aref string index) #/")
         ;; A CHARACTER specification.
         (values (dpb immediate-adcode (byte 3 10.)
                      (aref string (1+ index))) (1+ end)))
        ((= (aref string index) #/#)
         ;; An IMMEDIATE specificiation.
         (values (dpb immediate-adcode (byte 3 10.)
                      (read-from-string string nil (1+ index) end))
                 (1+ end)))
        ((string-equal "ARG|" string 0 index 4 (+ index 4))
         ;; An ARGUMENT specification.
         (values (dpb arg-adcode (byte 3 10.)
                      (read-from-string string nil (+ index 4) end))
                 (1+ end)))
        ((string-equal "SP|" string 0 index 3 (+ index 3))
         ;; An STACK OFFSET specification.
         (values (dpb stack-adcode (byte 3 10.)
                      (read-from-string string nil (+ index 3) end))
                 (1+ end)))
        ((string-equal "FP|" string 0 index 3 (+ index 3))
         ;; An STACK OFFSET FROM FRAME POINTER specification.
         (values (dpb fp-adcode (byte 3 10.)
                      (read-from-string string nil (+ index 3) end))
                 (1+ end)))
        (T
         ;; A SYMBOLIC ADDRESS (i.e., branch/call target).
         (values (read-from-string string nil index end) (1+ end)))))
```

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER MIT/LCS/TR-339	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Generic Software for Emulating Multiprocessor Architectures		5. TYPE OF REPORT & PERIOD COVERED Master's thesis May 1985
		6. PERFORMING ORG. REPORT NUMBER MIT/LCS/TR-339
7. AUTHOR(s) Richard Mark Soley		8. CONTRACT OR GRANT NUMBER(s) DARPA/DOD N00014-75-C-0661
9. PERFORMING ORGANIZATION NAME AND ADDRESS MIT Laboratory for Computer Science 545 Technology Square Cambridge, MA 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS DARPA/DOD 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE May 1985 13. NUMBER OF PAGES 96
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) ONR/Department of the Navy Information Systems Program Arlington, VA 22217		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for Public Release, distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Unlimited

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Computer architecture, emulation, simulation, dataflow

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The expense of designing, prototyping, and testing a new computer architecture (particularly non-traditional supercomputer architectures, such as the dataflow machine) is enormous. The relative inflexibility of hardware to experimental changes increases the need to fully test a new architectural idea.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

A software architecture for prototyping and testing single- and multiprocessor computer architectures is outlined. An overall design is discussed, noting the need for such a system, how it would be used to model and test various architectures, and possible implementation paths.

Various extensions and uses of such a generic prototyping system are also discussed, including extensions for modelling shared-resource systems, centrally synchronized systems, and distributed timing simulation systems. In addition, two uses of the system are presented, in particular the Tagged Token Dataflow Architecture, noting various methods in which such a machine may be simulated under a generic emulation package.