THE USC ROVING EMULATOR

*FRED COHEN

DIGITAL INTEGRATED SYSTEMS CENTER REPORT
DISC/82-8

DEPARTMENT OF ELECTRICAL ENGINEERING-SYSTEMS
UNIVERSITY OF SOUTHERN CALIFORNIA
LOS ANGELES, CALIFORNIA 90089-0781

JANUARY 1983

*This work was supported in part by the National Science Foundation under Grant MCS-8023485 and by the US Naval Electronics Command under Contract No. N00039-80-C-0641.

Table of Contents

1 Background 1
- 1.1 Summary of the Report 1
- 1.2 The Function of the Roving Emulator 1
- 1.3 A Top Down Scenario of the Roving Emulation in Use 3
- 1.4 Some Other Hardware Emulation Engines 4
  - 1.4.1 The IBM Emulation Engine 4
  - 1.4.2 The Bell Labs Emulation Engine 4
  - 1.4.3 The C-MU RTM Engine 5
- 1.5 A Review of Simulation Techniques 7
  - 1.5.1 Compiled Code Simulation 7
  - 1.5.2 Table Driven Simulation 7
  - 1.5.3 Event Driven Simulation 8
- 1.6 Foreseeable Bottlenecks and Their Avoidance 8
- 1.7 The Description of the Emulated System 10
  - 1.7.1 ISP 10
  - 1.7.2 DDL 11
  - 1.7.3 MIMOLA 12
  - 1.7.4 TI-HDL 13
  - 1.7.5 Conclusions 14
- 1.8 Desirable Description Language Properties 15

2 Theory of Operation 18
- 2.1 A General Model Of Computation 18
- 2.2 An Architecture for Data Flow 20
- 2.3 A Special Structure For Deadlock Prevention 23
- 2.4 The Resource Accesser Structure As A Generalized Pipeline 27
- 2.5 The Emulator as a Resource Accesser Structure 32
- 2.6 How the Architecture Processes Delta Nets 33

3 The Register Transfer Level Design 36
- 3.1 System Level Architecture 36
- 3.2 Custom Chip Decisions 38
- 3.3 The Memory Processor 39
- 3.4 The Allocation Processor 39
- 3.5 Processing Element Family of Processors 40
- 3.6 Queue chip 41
- 3.7 Bus Watcher 43
- 3.8 Bus Arbiters 43

4 Low Level System Operation 44
- 4.1 Observing the Subsystem Under Test 44
- 4.2 The Memory Processor 44
- 4.3 The Allocation Processor 46
- 4.4 The Processing Elements 48
- 4.5 Queuing of Data 48

5 Design for Testability and Performance Features 49
- 5.1 Allocation Processor 50
- 5.2 Processing Element 50
- 5.3 Queue Chip 51
- 5.4 Bus Watcher 51
- 5.5 Arbiters 52
- 5.6 Self Test During Emulation 52
- 5.7 Optimizing Hardware Utilization and Performance 53

6 Conclusions 55

I. Simulations of the Emulator 58

I.1 The Roving Emulator Intermediate Languages 58
I.2 The delta-net System Level Simulator 59
I.3 The Processing Element Parser and Simulator 67

II. An Implementation Scenario 71

II.1 Sample HDL Description of the PLA 71
II.2 The PLA in Processing Element Intermediate Form 73
II.3 Parsing and Simulating Intermediate Form 74
II.4 Analysis of a Configuration 75
II.5 The Encoder Emulation in Operation 79
II.6 A Simple Bus Watcher 80

III. Figures 81

1 Background

1.1 Summary of the Report

This report summarizes the major results of a 6 month research effort dealing with the design of the U.S.C. roving emulator (RE). The concept of roving emulation is based on the idea of having a set of models capable of acting as any other unit in a system. By observing the initial state, behavior, and final state of a unit under test and emulating it using this information, the results from the golden unit can be used to detect errors in the unit under test. Roving emulation is done by 'roving' a general purpose golden unit from one unit under test to another. The latency of error detection has been studied for such systems [Bre-1 82], and the design presented here has been applied in a case study [Bre-2 82].

1.2 The Function of the Roving Emulator

The RE is intended to be a general purpose facility for online testing of computer systems during normal operation, without significant performance degradation. The basic idea is to compare the execution of the system under test with a model of its intended performance. In operation, the RE would be either a board or large chip in a larger computer system. The REs job is to rove through the system under test watching each subsystem for some period of time, keeping data on its initial state, final state, and interactions over that period. It then emulates that subsystem using the initial state and input data, comparing its output and final state data to observation. If there is a difference then an error in the subsystem is detected and appropriate action is indicated. The emulator then continues to cycle through the various subsystems. Figure 1a illustrates a roving emulator connected to a general switching network. Figure 1b illustrates a bus architecture with a roving emulator.

The major design goals of the emulator are listed here:

- Testing during normal operation (the system under test should be able to continue to operate while being tested)

- Accurate emulation of any device (any system should be emulatable with a single facility)

- Low system overhead (the system under test should suffer as little degradation as possible due to the addition of the roving emulation capability)

- Minimal slowdown factor (faster emulation improves coverage and reduces error latency)

- Minimal size (the emulator should be able to be added to a small personal computer as well as a large production facility)

- Self testing (the emulator should be able to tell whether differences in results are due to a failure in itself or the unit under test)

- Fault tolerant (it should be able to continue to run accurately in the presence of many internal faults)

- Non Interfering (it should not be able to interfere with the system under test even if it detects faults since an error in it might cause damage to an otherwise working system)

- Optimum fault location (good fault location makes reconfiguration, repair, and recovery more efficient)

1.3 A Top Down Scenario of the Roving Emulation in Use

Let us suppose that a user has an HDL or program description of a set of subsystems to be emulated. To prepare for the use of the roving emulator on these subsystems the following steps must be taken:

1. The subsystems may have to be modified to allow their state to be dumped by the emulator. In many cases this facility may already be available. As examples, system using set/scan or LSSD registers already have this basic facility.

2. The designer may have to design an interface to the RE that allows it to observe the subsystems' I/O. Again, in many cases this may already have been done. For example, in a system with a bus, the interface to the bus need only be designed once. Any components on that bus can then be observed without redesign.

3. The designer must decide on a configuration of the RE which will most appropriately perform the task at hand. In general, any configuration with sufficient memory will be able to perform any emulation. In order to achieve high performance, low power, minimal board area, or other design goals, a specific configuration may be preferred. Software might be written to help determine a proper configuration for a given requirement.

4. The HDL or programatic descriptions must be translated into the U.S.C. intermediate form (IF) which can drive the emulator. This involves the use of either hand translation or a compiler. We have already designed a small compiler that takes logic level descriptions of subsystems and produces IF code for our generic processing elements (PE)s. In large scale use more sophisticated tools would have to be developed for efficiency.

5. The IF description would normally be tested by a simulator to perform both design verification of the system under test and to verify that the emulator is performing properly with the compiled code. It should be noted here that most HDLs make implicit assumptions about the nature of default values for various parameters including such things as unit gate delay and synchronous operation. The emulator itself makes no such assumptions, and may therefore produce output which reflects otherwise undetected race conditions in the system design. Additional information may have to be embedded in HDL descriptions or compilers to make explicit these assumptions.

6. The IF description would then be transformed into machine code for the emulator. EPROMs or PROMs should be burned to reflect the final emulation code and configuration restrictions, and inserted into the emulator. Alternatively, the emulator could be off loaded with the relevant information as it is needed.

1.4 Some Other Hardware Emulation Engines

Two low level hardware emulation engines have previously been designed, one by Bell Labs and the other by IBM. Although only the IBM engine has been fully implemented, both of these designs have significant features of note. A related design which could also be used for emulation is being implemented by C-MU.

1.4.1 The IBM Emulation Engine

The IBM emulator [Pfi 82] is essentially a compiled code system which specifies the execution of a logic engine implemented in hardware. Although the slowdown problems in many pure compiled code simulators are severe, the IBM system compensates for this with the speed of its special hardware and its extreme parallelism. It is thus a significant improvement over compiled code simulation of logic circuits executing on standard commercial processors.

The IBM system was designed for analysis of designs rather than for field work. It is very large and power consumptive, requires additional hardware for each 10000 gates simulated, requires an extensive compilation procedure to optimize processor utilization, and has no link to external processing. In addition, this engine is strictly a gate level simulator and therefore may be unreasonably slow for emulation of large functional units even though they may have relatively simple logical transformations. The use of a gate level simulator also requires explicit knowledge of the internal design of a system which may not be available or required to perform the task of roving emulation.

1.4.2 The Bell Labs Emulation Engine

The Bell Labs emulator [Abr 81] consists of a pipelined control unit used to supervise event driven simulation, and a set of special hardware execution units used to perform high speed emulation of events. Although the design is not yet solidified, it currently includes an array processor and several general purpose logic simulation chips as execution units. This system holds promise of achieving a very good slowdown factor.

There are several drawbacks to the use of this processor for roving emulation. It requires several boards and significant power utilization, is severely limited in the number of gates it can emulate at any given moment, and is not arbitrarily expandable for emulation of large systems. Its scheduling is all done at compilation time which prohibits the use of arbitrary configurations at runtime, and it may have memory collision problems when all processing elements are operating. Another drawback is its single control unit which might become a performance bottleneck in processes when its execution units perform very simple tasks, and a waste of processing power when it nearly always waits for results. It would also become a reliability bottleneck since a single failure could halt the entire system, or even produce erroneous results without detection. Again, this processor was not built for the roving emulation task and could hardly be expected to meet its requirements.

1.4.3 The C-MU RTM Engine

The C-MU emulator implements an RTM engine. RTM is the intermediate form used in the ISP simulator [Bar-1 80] developed at C-MU over the last decade. Their design is based on a rather high level multiprocessing representation and relies on a shared memory configuration with a VAX for its execution. Although this engine would be very inefficient for gate level emulation, it has good possibilities for higher level applications.

It is estimated that initial slowdown factors will be on the order of 100,000 to 1 and that no matter what techniques are used ISP slowdown will never be better than 100 to 1. The C-MU emulator is being designed as a single board subsystem in a larger computer system, but acts symbioticly with its host rather than as an observer.

In summary, the major drawbacks to previous designs are:

1. Little or no fault tolerance in the designs themselves. This may lead to faulty emulation without detection.

2. No internal self test capabilities are provided. This lack of design for testability can lead to very long test times and difficult verification of operation.

3. Inefficient hardware utilization in certain tasks. As has already been pointed out, special purpose hardware is best at special tasks. Since it is desired that the emulator be very good at a large variety of tasks, extreme specialization is to be avoided wherever unnecessary

4. Minimal configurations are too large. In many cases, a very small configuration is desirable at the expense of performance.

5. Inefficient or inadequate means for translating system descriptions into internal forms quickly and reliably. Software development is needed to improve this situation.

1.5 A Review of Simulation Techniques

The major software simulation techniques in use today are:

- Compiled Code
- Table Driven
- Event Driven

In addition, there is a promising technique under development called 'Time Warp Simulation' which will not be included here [Jef 82].

1.5.1 Compiled Code Simulation

Compiled code simulation consists of compiling a description of the system to be simulated into either an intermediate code form or the machine code for a specific processor. The compiled code is then executed as a program which transforms inputs to outputs for each subsystem every time a data value for any subsystem changes. This tends to be very slow since the code often spends most of its time in evaluating elements whose signal values are not changing.

1.5.2 Table Driven Simulation

Table driven simulation uses an uncompiled representation of the system to be simulated. The structural and behavioral aspects of the system (e.g. delay, logic values, fanin, fanout) are represented in tables. During simulation, the tables are traversed each time step, and relevant information is evaluated. Each component is evaluated at each time step. This is often done regardless of whether a given component is actually changing during that time. As a result, simple table driven simulation has a severe overhead in wasted effort despite the low simulation overhead achieved by the simplicity of the scheme.

1.5.3 Event Driven Simulation

Event driven simulation consists of scheduling simulations of components only when their input changes imply potential output or state changes. The schedule is kept in time ordered queues so that during time steps when no processing needs to be done no processing is done, and only components which require simulation at any given time are simulated at that time.

Many existing simulators use combinations of these techniques to achieve their design goals. The RE should provide facilities to implement any or all of these techniques as needed.

1.6 Foreseeable Bottlenecks and Their Avoidance

In sequential emulation engines performance is limited because a sequential method is being used to emulate a parallel system. Since nearly every current system has a high degree of parallelism, this is a severe bottleneck. There is no way to make a sequential process perform a parallel task faster than the sum of the times required to emulate each task at the level of their description. Any efforts to improve sequential engine performance must come from improvements in system modeling, description language translation, and special purpose hardware for executing the translated description more quickly.

In a parallel emulation engine this bottleneck may be eliminated to a large degree, but parallel processors have memory and bus bandwidth bottlenecks. This is because of the requirement to share information between parallel processes modeling physically connected components. Shared information may either be duplicated for simultaneous access or accessed sequentially. If information is duplicated, information must be transferred between processors when data changes, thus increasing the required bus bandwidth. If it is not duplicated, sequential access control is necessary, thus increasing the required memory bandwidth. There are ways to reduce this problem, but no currently available ideas eliminate it.

In exchange for attaining a high degree of parallelism, the ability to perform simple synchronization is given up. If a given result is needed to process the next problem, the entire system must wait for the slowest task to be completed before continuing the emulation. This weakest link problem may be reduced by emulating several subsystems at once in the hope that their bottlenecks don't overlap. The hope is that while one subsystem emulation slows down awaiting a critical result, another will be sped up due to the additional availability of resources.

The extent to which these problems effect the performance of the emulator may vary depending on the partitioning used to model the system being emulated, inherent properties of the system under test and the emulator, and many other considerations. No single design can hope to optimize the whole set of systems of interest.

One approach to this design problem is to design the RE as a family of architectures which can be structured to optimize the bus and processing bandwidth requirements of the systems being emulated. In this approach, if a given resource is a bottleneck to system performance, another similar resource may be added in parallel to it to alleviate the problem. This type of design may also enhance the fault tolerance of the emulator. If redundant hardware fails, the system may be reconfigured to eliminate the failed hardware at a reduction in performance. Self test with redundancy is also well understood. A further advantage of such a scheme is that performance and cost can be traded off for each implementation without redesigning the system or regenerating IF descriptions.

1.7 The Description of the Emulated System

Emulation is based on a description of the system being modeled. The representation used to describe the system has a strong effect on its emulation. Although many languages exist for specifying hardware design, few have actually been implemented for simulation. A major concern in the design of the RE is that it fulfill the needs of a wide variety of tasks. In order to fulfill these needs, we first looked at the use of current HDLs which were representative of the state of the art, with the hopes of finding one suitable for an optimized hardware implementation. Languages considered for the RE were:

- Instruction Set Processor (C-MU) [Bar-2 80]
- Design Description Language (Hewlett Packard) [Dul 68]
- MIMOLA (Universitat Kiel, W. Germany) [Zim 80]
- TI-Hardware Description Language (Texas Instruments) [Tex 82]

1.7.1 ISP

ISP has several advantages in that a considerable effort has been made to utilize the system for automatic generation and verification of systems in use as well as under design. A full ISP simulator already exists, and compilers for transforming an ISP description into an RTM engine description are in use at this time. Unfortunately, the ISP system has some substantial drawbacks which make it poor for low level simulation. In particular, its low level timing dependencies are difficult to use and must be coded in a program form rather than as a network list. There is a module that comes with ISP called PMS which allows ISP modules to be combined in a processor-memory-switch description similar to a network list in its characteristics.

ISP is designed to describe instruction set processors. Although it is general enough to model any processing, it may be quite inefficient in many cases. Many subsystems of interest do not process instruction sets at all, but rather transform data directly, or perhaps execute different instruction sets at different times. This may present an additional problem to the use of ISP.

According to the designers of ISP, it will probably never achieve emulation at faster than a 100 to 1 slowdown factor. It is difficult to do logic level emulation with ISP due to its design assumptions. One further disadvantage is that the IF used by ISP (the RTM engine) is at a multiprocessing level, and is thus difficult, although not impossible, to design and implement in hardware as a special purpose parallel processor.

Many systems are already described in ISP and considerable fault simulation has been done with them. ISP is publicly available, written in Pascal, and quite well tested. The 'ISP Simulator' is an event driven simulation with substantial multiprocessing capabilities.

1.7.2 DDL

A design described in the 'Design Description Language' is simulated using a time driven simulator without any IF between the register transfer level and execution. Stanford has a simulator for DDL, and according to some sources it runs very slowly. The program is implemented in pascal which makes it very transportable, but there is no easy way to implement it on a parallel processor without more detailed encoding. It appears that DDL lends itself to a single level description of a system, and as such is severely limited in its primitive capabilities. It is sufficient for use in the RE, but the general unavailability of information is not a good sign.

1.7.3 MIMOLA

The MIMOLA description language is a multilevel integrated design language capable of translating very high levels of description into low level designs with human aide. In a sense, the highest level of description allowed in MIMOLA is the level used by most applications software designers. This level allows software macroing, routine calls, and complex block structured conditionals. The lowest level is essentially a standard *RTL.

One of the major features of MIMOLA is the ability of the designer to perform dynamic analysis of design tradeoffs in conjunction with the system in order to evaluate performance and a variety of cost functions in search of a minimum configuration. In addition to the hardware design capabilities, the language is capable of modeling the execution of various types of software on the system under design to allow operating systems and other frequently used software to be designed and tested prior to hardware availability. This may in the long run have the effect of producing higher quality systems than any of the other languages mentioned here.

While the system allows the designer to produce an algorithm very quickly it may be advantageous from the standpoint of optimizing hardware to produce better algorithms more slowly. It appears that the design of MIMOLA is far in advance of any of the other HDLs presented here, and is indicative of the trend towards higher level and more sophisticated design aides.

From the standpoint of use in the RE, the high level description allows the use of very high level functional emulation, and this improves performance characteristics of the emulator significantly. There are a few problems with MIMOLA that should be mentioned. It is not very good at handling very low level

*Register Transfer Language.

special purpose designs because it is essentially based on the development of a hardware facility to implement a microcode sequence. In many cases, instruction sets may not be relevant to the computation being done in a subsystem. In addition, the system as currently implemented is not capable of handling multiprocessor designs or networks very efficiently. MIMOLA is written in Pascal and as such is relatively transportable to a variety of machines. The MIMOLA system produces many intermediate code forms and maintains a design database with extensive compilation and redesign capabilities, and as such would be very amenable to hardware implementation at many levels. There is painfully little simulation capability in the MIMOLA system as it currently stands, and this is a significant drawback to any widespread use.

1.7.4 TI-HDL

The 'Texas Instruments Hardware Description Language' is a hierarchical description language consisting of structural and behavioral descriptions at multiple levels in a hierarchy. This is a redundant form which can be used for design verification as well as test generation and design automation. TI-HDL can be processed using an event driven simulator with no intermediate description language which makes it relatively efficient, but difficult to design a hardware engine for. TI-HDL allows for concurrency at the block level only. Since a given system is described as a hierarchy of blocks, this should allow parallelism at multiple levels depending on the level of the description and the transformations used to emulate it in hardware. TI-HDL seems to have the widest range of emulation levels, extending from the register transfer level down to the mask level. One drawback of TI-HDL is that the levels available for use are fixed at one of 5 levels (block, register, gate, transistor, mask).

Actual implementation is completed from the transistor level up. The system is currently used internally at Texas Instruments for VLSI design. A Pascal version of the system may be made available in the near future which extends from the gate level up, and it is being considered for limited university distribution at this time.

No performance data is currently available on TI-HDL, but according to some reports it is very slow. This of course depends on the level of detail being used in the simulation. According to sources at T.I. they have been doing transistor level simulations of 70,000 transistor chips which would explain their performance difficulties.

1.7.5 Conclusions

Initially we investigated the possibility of using an available hardware description language as our core form of description, with the idea of implementing an engine to execute its code. The results of this investigation follow this section. What we found was that many designers used each language, and that none of the languages produced code which would be easily implemented as an emulation engine. Our conclusion was that we should define an intermediate form which could be used to describe virtually any emulation, and implement compilers from the other HDLs into it. This has the advantages of allowing any description presently in use to be emulated without rewriting the description, and allowing future languages to be added to the emulator without redesign.

Our current plan is to take descriptions in the form of HDLs as input and compile them into a set of intermediate forms (IFs) which represent a processable model for emulation. At this level, simulation using a special purpose simulator can be done to verify design and test emulator configurations for the specific description. When emulation on the engine is finally desired, the IFs are assembled into machine codes for the particular configuration being used.

Because of the large number of description languages and other techniques used to represent systems it would be limiting to force the user to duplicate the description effort to use our system. In addition to the high overhead in time required to redescribe a system, there is considerable room for error in a redescription. Our decision was to implement an IF which could be translated into from most HDLs. In this way, we could optimize our design for the functional utility perceived in the use of the emulator, while providing the ability to emulate systems designed or modeled in any desired language. Optimization of the HDL description for emulation would then consist of only the optimal compilation of HDL into the IF.

1.8 Desirable Description Language Properties

There are some desirable properties of description languages which we feel would improve the eventual performance of an emulation and the overall utility of the descriptions. Although these need not be incorporated in the HDL being compiled from, they are included here as a guideline to future investigation. These properties are:

- Arbitrary hierarchies of blocks
- Optional behavioral descriptions of blocks
- Independent alterable data and control networks between blocks

Arbitrary hierarchies of blocks allow a system to be described and modeled at any level or set of levels the designer deems appropriate. This allows the level of description to be set by the user rather than the inherent structure of the description language. It allows arbitrary bottom up combinations of modules to be described very easily. In addition, it allows top down design to proceed without prematurely tying the hands of the designer to a given bottom level implementation.

Optional behavioral descriptions of blocks allows the user to describe blocks in terms of other blocks, algorithmic descriptions, or combinations of the two. This permits efficient descriptions of partially understood systems, design verification of behavioral descriptions with lower levels of design, and emulation at multiple levels, depending on available information. In the case of multilevel emulation, hierarchical fault location may be done directly through emulation, thus providing a powerful diagnostic tool as well as a detection mechanism.

Independent alterable data and control networks between blocks allows the designer and emulator to discriminate between signals that effect only input states of blocks, and signals which cause activation of components within the blocks. It also allows these properties to be changed so that a given connection may be a control connection when it will cause component activation, and a data connection when its control function is disabled. This produces a significant improvement in efficiency of event driven emulation without loss of generality. It also permits the system to be described simply with network lists, the most commonly used representation at this time.

Another interesting result came from our survey. It would seem from current trends and for efficiency reasons, that HDLs will eventually be distinguishable from other high level programming languages in only the following ways.

1. HDLs will not allow dynamic memory allocation except through the explicit reconfiguration of finite and bounded memory elements. This is because hardware is a physical reality with physical limits, and it cannot logically allocate what physically doesn't exist.

2. Data flow and control dependencies will dominate the description of hardware to a much larger degree than software. This is because a-priori explicit tradeoffs must be made in hardware design since hardware is a permanent resource while process space in software is a temporary resource.

3. HDLs will probably not use very high level mathematical descriptions of operations until significant advances in understanding design tradeoffs are made. Hardware designers currently specify or supervise the exact details of execution and hardware utilization rather than allowing arbitrary tradeoffs to be made by optimization techniques that are out of their control. When these techniques become better than designers, higher levels of description will become dominant.

2 Theory of Operation

The theoretical model used to explain the operation of the roving emulator, maps through design decisions, into the implementation architecture chosen for the requirements at hand. This section describes a new model of computation, and models a data flow network with it. Using a special case of this model, a deadlock free hardware structure is arrived at. This is shown to be a generalization of a pipeline. The emulator is designed as such a structure, and the elements in the pipeline are defined so as to implement the general model of computation in hardware.

2.1 A General Model Of Computation

The RE is intended to implement a general model of computation derived from the theory of Petri-Nets [Pat 81]. Petri-nets themselves are in fact inadequate to represent computation in general because of the lack of mathematical completeness. With the simple addition of an inhibitory input they become general enough to define any computation. Although elegant and powerful, Petri-Nets are too unwieldy to be of general use except from a theoretical standpoint.

The mathematical structure of a Petri-net with inhibition takes the following form:

```
Petri-net :=    (Place,Transition,Input,Nature,Output)
        :=      (P,T,I,N,O)

        Place :=        (state - number of things in it)
        Transition :=   (places marked upon firing)
        Input :=        (source of marking)
        Nature :=       (inhibitory or enhancement)
        Output :=       (transitions effected by firing)

Firing Rules:
        1       T is enabled iff
                        (for all P with O to T there is
                        a marking for each link to T)
                and     (no inhibitory O from any
                        marked P goes to T)

        2       Any enabled T can fire, but only one at a time

        3       The firing of T removes one marking from T
                for each O from any P to T and places a marking
                for each I to any P from T
```

An improvement in the form of allowing arbitrary computational resources to be placed within any given place in the net allows more intelligence to be associated with a place in the net. This produces a mix between the advantages of practical computational machines, and the generality of Petri-nets for representing computation. The intelligent places will be referred to as *resources*, while the components providing connectivity between and activation of these resources (often represented as the links in a network) will be referred to as *accessers*.

The particular network invocation chosen for our design is a generalized Petri-net with control and data inputs, arbitrarily specifiable delays, and intelligent resources. It is implemented with the following firing rules and named a *'delta-net'*:

```
Delta-Net := (Resource,State,Accessers,Type,Delay,Effect)
       :=   (R,S,A,T,D,E)

       Resource :=      (Physical or logical entity)
       State :=         (Values of all retained data)
       Delay :=         (Time till effect)
       Accesser :=      (Physical or logical links)
       Type :=          (Control or Data Value)
       Effect :=        (Accessers changed)

Firing Rule:
       if       (for any (R,A,T=Control) A is activated)
       then     for each change in (R,A,T = Data)
                {E(S,A) => (S,A) at time t+D(S,A)}
```

The activation of a resource involves the execution of the data associated with that resource. In the case of physical resources, this involves the physical transformation of data from input to output. In the case of logical resources, it involves the activation of a physical resource on that logical resource. Physical resources are permanent over the execution of processing (although they may change due to physical changes in the system), while logical resources are temporary (in the sense that they are activated and then used up through their processing into effects).

Figure 2a illustrates a sample delta-net element. Figure 2b illustrates a simple network of these elements.

2.2 An Architecture for Data Flow

Using this general model of computation and some assumptions about an implementation architecture, proofs can be generated with regard to the stopping conditions of machines, their ability to utilize hardware resources effectively, and their ability to perform tasks of various types. The most general architecture consists of the implementation of physical resources and accessers which connect them. In the case of the RE and many other systems, getting the most hardware utilization for the investment is an important design goal. It is felt that through maximum utilization, maximum throughput can be achieved. Another critical factor in our design is the prevention and detection of deadlock due to the allocation of physical resources towards a goal. The research has led to the following description of three generic classes of resources and their accessers.

- A terminal resource is distinguished by the fact that there is a single input and output accesser. This is a terminal resource since it cannot pass a portion of its processing on to another resource and await its return without a recursive loop.
- A critical resource is distinguished by the fact that all resources accessing it must pass through it in order to access any other resource accessing it. This makes this type of resource a reliability and performance bottleneck and is to be avoided if at all possible.
- An embedded resource is distinguished by the fact that it can access multiple inputs and outputs, and in addition it is not the only means of accessing any given resource connected to it from any other resource connected to it. This type of resource can be used to access other resources without creating the bottlenecks created by a central resource.

Figure 3 illustrates the three types of resources.

Assuming a system can have a large number of resources capable of executing simultaneously, the effective parallelism of computation can be maximized by maximizing the utilization of the resources for processing. Let us suppose that we had a non ending stream of information to be processed through resources. If we could design these resources and their accessers so as to allow nearly 100% resource utilization, it would be a big step towards optimizing effective parallelism. Such a design is possible if some simple assumptions are made. The assumptions are:

- An arbitrary number of each processing resource is available at implementation time, so that given a ratio of resource requirements by type of resource, sufficient quantities of each resource in the right proportions can be implemented.

- Each resource must finish its task in finite time.

100% utilization is attained as long as:

- There is always data awaiting processing by each resource.

- Each resource always has a place to store its results so that it can process its next input as soon as it has finished its last output.

If there is always data awaiting processing, then every resource always has something to do. If it can always release its previous results, it never has to wait to output results. Unless designed to waste time, 100% utilization will be achieved.

If the accessers are designed as FIFO queues and there is always enough room in the queue for more data, then the output requirements of this system are satisfied. If there is always data awaiting processing in each input queue, and the queue is designed to always pass information on at the soonest possible moment, then the input requirements of this system are satisfied. In short, if the accessers are implemented as FIFO queues and are never allowed to get full or empty, then the processing resources will operate at 100% utilization. In practice this could very nearly be attained assuming only a modest queue size and a reasonable distribution of resource processing times.

Figure 4 illustrates some examples of these types of resources connected to FIFO accessers so that 100% resource utilization is possible.

2.3 A Special Structure For Deadlock Prevention

According to Brinch-Hansen [Bri 80], there are necessary conditions for deadlock in the cases of both permanent and temporary resources. If these conditions are prevented from conception, or localized and made detectable, they can be systematically prevented or corrected for, and a deadlock free system can be designed which dynamically allocates resources. The necessary, but not sufficient conditions for deadlock in permanent resources are:

1. Mutual exclusion of access to resources
2. Non Pre-emptive scheduling
3. Partial allocation
4. Circular Waiting

In the case of temporary resources, the necessary conditions required for the prevention of deadlock are:

1. The producer of the resource and its consumer must agree as to which acts next or else the consumer might try to consume what has not yet been produced.
2. The producer must eventually produce, and the consumer must eventually consume or the system will empty or fill.
3. The items produced must be sufficient to allow some consumption to take place or circular dependencies may result.
4. A product requiring consumption may only be produced when a consumer is able to consume it, or its requirement of consumption may not be met.

Let us examine these conditions on a case by case basis to determine which, if any of them, can be fulfilled appropriately to localize and detect or prevent deadlock.

The resources and accessers as presented make no explicit assumptions about mutual exclusion or non preemptive scheduling of permanent resources. It is however extremely complex to design these capabilities into hardware compared to the design of comparable systems without these features. In addition, including such features consumes time and chip area and introduces significant additional room for error in the design of a system. If these two conditions can be allowed while still preventing deadlock (as they can), it would be advantageous.

Complete allocation and prevention of circular waiting at a local level may be used to force the occurrence of deadlock conditions to occur only at a specific place in the system. If this is done, special provisions can be made to detect the deadlock case in that locality and take appropriate action. This is the solution taken in the emulator, and it is a general property of a whole class of data flow machines defined below. Deadlock due to lack of sufficient space for solution of the problem can be detected at runtime, and reordering of execution can be attempted. If this fails or is undesirable due to performance requirements or configuration changes, aborting the task can be used instead.

In the case of partial allocation, it was found that, if a resource is always provided with a packet of data to be processed which is self contained in the sense that no other information is required to process that data, several problems can be averted. This prevents the possibility of producing a product which cannot be consumed (assuming it is sent to a resource capable of consuming it), and guarantees that any product produced can be consumed in its entirety by the resource processing it (with the possible side effect of that consumer producing another temporary resource which must then be consumed). This of course assumes that the consumer is capable of completing any consumption in finite time. If this is to be guaranteed, the hardware must prevent the possibility of infinite loops in any given resource.

In the case of circular waiting, the same type of prevention must be carried out at the inter resource level. Unfortunately global looping cannot be prevented while maintaining the ability of the system to provide any possible processing. It can however be localized to one type of resource. If all other resources are designed so that all global loops always pass through a resource capable of detecting and preventing or recovering from deadlock, the system will be deadlock free. Although a given computation may not be able to finish, the system will continue.

The local prevention of circular waiting and infinite looping is easily done at the expense of local processing generality. This is perfectly acceptable since the system can be designed so that the generality is embedded in the inter resource execution. The simplest solution is to prevent any type of branching in each resource. This has the pleasant side effect of producing finite run times, but the unpleasant effect of preventing any testing and conditional action at the local level. The more intelligent we make the local processor, the less we burden the global net with the details of simple testing, and the faster we will make the processing. It is for this reason that forward branching only would be an acceptable middle ground. With a little bit of effort it is possible to design finite looping in hardware so that it is impossible to have infinite loops. This allows all the power of decision making and looping while forcing a finite run time at the local level.

The finite run time is simply proven by the fact that with no branching the runtime is the finite sum of finite runtimes of individual instructions which is finite. In the forward branching case, a forward branch can only prevent the execution of a portion of a finite number of finite time processes, and since it cannot force any of them to be repeated, it can only reduce the finite runtime. Thus the runtime for forward branching programs is finite. Strictly finite loops also have finite runtimes, and thus a relatively powerful processor can be implemented as a deadlock free resource.

If we look back at the original conditions for deadlock, only the global case of system wide circular waiting remains. As stated earlier, this is not in general removable, but is localizable if one forces all global loops to go through a deadlock detecting resource capable of taking appropriate action. In the case where there are no critical resources, only embedded and terminal resources remain candidates. Embedded resources may be usable for this purpose (although the proof is not completed at this time it is believed to be impossible to use embedded resources in this way). In the case of the emulator this is not done. The solution is to force all global loops through a terminal resource designed to detect the condition of being unable to store incoming results due to a lack of space.

If this permanent resource simply consumes incoming temporary resources without producing any new resources, the system will eventually empty (which can also be detected due to the finite and bounded completion time of each permanent resource) at the expense of lowered utilization over a short period of time and loss of a portion of the current computation. Appropriate recovery can then be performed. In the case of emulation, partial results provide reduced coverage, but can still be used as a valid indicator. This same detection mechanism can be used to detect slowdowns in the hardware and other errors as well as completion of time locked operations.

This leads to the conclusion that such a system will be most effective when this type of mass synchronization happens as infrequently as possible. Further, the communication overhead associated with the movement of data through the structure would suggest that the longer the resource runtime, the less time will be spent in overhead. Again, the use of the general capabilities of the network should be restricted as much as possible to attain high performance.

Figure 5 indicates a generalized structure for deadlock prevention.

2.4 The Resource Accesser Structure As A Generalized Pipeline

In order to get a better understanding of the resource accesser structure, it will be represented in light of another high hardware utilization structure with many similar properties, the pipelined processor. Let us examine a typical reconfigurable pipelined architecture operating under a small variety of circumstances (see figure 5). A simple 5 element pipeline will suffice for the purposes of this example. It should be pointed out that the problems demonstrated here are not concocted examples made to make pipelines look bad, but rather illustrate the range of behavior generally expected from a pipeline.

Given the pipeline shown in Figure 6 with two reconfigurable loops, the problem is to determine the delay and utilization of the hardware under a variety of tasks, and the overhead associated with the implementation of those tasks. The first problem comes from the fact that each element in the pipeline has an execution time that may not be exactly matched to the other elements execution times. In this case, it is necessary to determine clock speeds so that in any given configuration, the longest clock period required is as small as possible. This is given by:

$$Tc = \max(Ta,...,Te)$$

The minimal delay of the pipeline (assuming no strictly linear neighbors are combined and no loops are used) is 5*Tc. If there is looping, the end to end delay may be higher.

Scheduling of the pipeline is quite complex and must be done on an a-priori basis to assure that no conflicting utilization can occur (invalidating the results). The components of a pipeline generally have fixed transforms, and a small number of possible configurations are usually allowed. The effective parallelism of a pipeline varies depending on the utilization pattern of each component. Typical scheduling techniques are used below to demonstrate the 4 possible configurations of this pipeline:

```
Case 0 - Best Case - as t -> infinity
Parallelism -> 5; Utilization -> 100%
1 result / 1 clock cycle
pipeline delay = 5 clock cycles
 -1--2--3--4--5--6--7--8--9--0--1--2--3--4--5- t
1 A  B  C  D  E  F  G  .  I  J  K
2    A  B  C  D  E  F  G  .  I  J  K
3       A  B  C  D  E  F  G  .  I  J  K
4          A  B  C  D  E  F  G  .  I  J  K
5             A  B  C  D  E  F  G  .  I  J  K

Case 1 - Good Case - as t -> infinity
Parallelism -> 3; Utilization -> 60%
3 results / 11 clock cycles
pipeline delay = 11
 -1--2--3--4--5--6--7--8--9--0--1--2--3--4--5- t
1 A  B  C  A  B  C  A  B  C  D  E  F  D  E  F
2    A  B  C  A  B  C  A  B  C  D  E  F  D  E
3       A  B  C  A  B  C  A  B  C  D  E  F  D
4                            A  B  C
5                               A  B  C

Case 2 - Poor Case - as t -> infinity
Parallelism -> 1.5; Utilization -> 30%
1 result / 6 clock cycles
pipeline delay = 9
 -1--2--3--4--5--6--7--8--9--0--1--2--3--4--5- t
1 A              B              C
2    A              B              C
3       A              B              C
4          A     A     A  B     B     B  C
5             A     A     A  B     B     B  C
```

```
Case 2 + 1 - a bit better - as t -> infinity
Parallelism -> 1.75; Utilization -> 35%
1 result / 6 clock cycles
pipeline delay = 15
 -1--2--3--4--5--6--7--8--9--0--1--2--3--4--5- t
1 A        A     B  A     B     C  B     C
2    A        A     B  A     B     C  B     C
3       A        A        A     B     C  B
4                            A     A     A  B
5                               A     A     A
```

As is quite clear from the above example, scheduling may be nontrivial, and utilization and parallelism vary widely. Delays are limited by the number of components passed through and the system clock speed required for the slowest component in the pipeline. As the resources in this pipeline become more complex, their delays may become quite high and problem dependent. This makes the scheduling and allocation problem still more complex.

In addition, a pipeline is very difficult to design redundantly so that when a resource becomes inoperable, the performance will only slightly degrade. These types of pipelines have the advantage of low delay, but vary widely in their utilization, depending on the problem being solved. They require complete emptying of the pipeline in most cases before the next group of problems can be processed. This reduces their utility for applications where multiple configurations are often used in an interleaved mode.

By contrast, let us now look at the same pipeline implemented as a set of queued resources (see figure 7).

In this implementation, each of the resources used in the previous design are connected by queues. This simply allows them to process data at their own rates, therefore speeding up the processing of data significantly in case where the speed of one of the resource is the limiting factor. Several copies of the limiting resource can be placed in the system to compensate for its slow speed through parallel processing. In effect, this is the same as the previous pipeline in the sense that if the same streaming mode is used for a very long time period the delays will average to nearly the same constant (a little longer due to the overhead of the queues) as before. If the queues are eventually filled up by processing awaiting the limiting resource, the system will act exactly like the previous pipeline.

When a different configuration comes into use, things are much better for the new structure. Instead of awaiting the completion of all pending processing and then reconfiguring the system, the new data along with its processing specification can be fed in directly. In the case of a very long pipeline this has considerable advantages. In the case of pipelines which are reconfigured often, the advantage is also noteworthy. In the case where the different configurations utilize resources in different proportions, the effect of increasing the average resource utilization may be quite pronounced.

By example, let us suppose that all of the elements are the same speed, and the system is being switched between the configurations used in the cases of the pipeline above. If we suppose that the queues are large enough to store several switching intervals worth of configurations, the average utilization for the four examples above will be the average number of resource uses per unit time. In the first example each resource is always used. In case 1, resources 1,2, and 3 are used 3 times as much as 4 and 5. In case 2, resources 4 and 5 are used 3 times as much as resources 1,2,and 3. In case 1+2 all resources are used equally. Note that the reason the pipeline doesn't get high utilization in many cases is because of conflicts in timing rather than utilization.

In the queued system this does not happen. The average utilization over a long period for all four configurations running simultaneously is 100%! In fact, case 1+2 running with queued resources will achieve 100% utilization in the long run even though the best that can be done without queuing is 35%. This is because queues remove timing dependencies from resource utilization. An additional advantage of the queued resource is the ability to use an arbitrary configuration through the switching of data. The major disadvantage is the additional hardware required for sharing resources.

Let us then examine the next step in the generalization of the pipeline by replacing the specific hardware resources used in the previous example with a set of generic resources which can perform any of the functions of the previous example at a slight performance degradation from the more tailored system. Figure 8 shows the general form of a generic resource pipeline.

In this case, since each resource is capable of performing each processing task, there is no need to force configuration information into the system at all. Each data item can carry with it all of the information needed to process it in the desired manner, and can be processed by any available processing resource at any time, either in parts or as a whole. By processing it in larger chunks, the performance lost in the more generic implementation can be regained by queuing fewer intermediate steps. The connectivity network can be reduced significantly from a cross point switch to a simple set of forks and joins. Again queues can be used at the input and output to improve processor utilization. In this case, any configuration one desires for the pipeline can be achieved, and full module level utilization is obtainable regardless of the distribution of configurations and their changes over time.

The tradeoffs are quite straight forward here. Generalized processing facilities are a more expensive resource but more flexible. More information must be sent with the data to process it in the generalized resource model. The arrival of results is not guaranteed to be in the order of the problems presented (this is solvable through adding dependencies to the system). The end to end delay is higher for queued resources in many cases, but the resource utilization is extremely high and long term throughput can be significantly increased. The overheads associated with this type of computation suggests that it is more optimal for larger processing tasks (due to the lower relative overhead). Finally, the expense of switching circuits between resources can be significantly reduced by sharing data paths over time. Since the resources are queued, waiting for shared resources can be done at the accesser rather than at the resource without reducing the effective resource utilization.

2.5 The Emulator as a Resource Accesser Structure

The major design problem facing the implementation of such a system is the requirement to connect arbitrary sets of resources through accessers with a reasonable amount of hardware. Since the use of a cross point switch is extremely expensive in area, power, delay, and reliability, it is unacceptable for the emulator design. Since the average bandwidth of the data flow is more important to long term utilization than the peak data rate, a fully connected network would be a severe over design. There is in fact no performance penalty for slight delays in transmission so long as they don't force queues to become empty or full.

As a result of the preceding analysis, the architecture shown in figure 9 was conceived.

A bus architecture was chosen instead of a complex switching system for several reasons. First, by using multiple busses between resources, any resource is easily accessible from any other resource with a minimal delay. In addition, if a bus becomes inoperable, another bus may be used for the same purpose at the possible price of performance degradation. This allows graceful degradation in hardware without software intervention. Depending on the ratio of processing to accessing, the number of busses may be increased or decreased to tailor the system to the application without redesign or software modification. Busses are quite fast at performing actual transfers, are easily implemented on or between integrated circuits, and allow the addition of external devices to the system in a general way. The access to these busses is in the form of arbitrated packet transmission from queues being emptied to queues being filled. The other ends of these queues are connected to processing resources which produce or consume logical resources. Figure 10 illustrates this bus architecture.

2.6 How the Architecture Processes Delta Nets

The architecture processes delta-nets by passing successively smaller portions of information through a generalized pipeline. It strips information at each step and produces an environment of partial results which eventually gets reflected back into the memory as changes. Specifically, if we assume that the memory starts out with a description of some logical resource requiring evaluation, it has a structure of the form: (R,S,A,T,D,E). The memory evaluates the links going to a given resource, and replaces the generic state information with the specific state to be executed. Since the accessers to the resource are no longer of any interest once their values have been mapped into state information, they are no longer needed. It passes the new structure (R,S,D,E) on to the next level of processing.

The allocation processor uses the information regarding physical resource requirements associated with R and the current utilizations of available physical resources to allocate processing to a physical processor of the right type. In some cases the logical resource R may have been previously loaded into a particular physical resource, and that portion of the information can be passed as a single instruction. The allocation processor also allocates a block of space in its memory for handling the returned information of the logical resource R, and adds a descriptor to the packet being forwarded. It then passes a modified structure (S,D,E) or (R,S,D,E), depending on the previous contents of the physical resource, on to the next level of processing.

The next level of processing evaluates the low level operations to be performed on the state information, possibly updates its internal state to reflect changed values, and passes any changed values along with their computed delays back to the allocation processor that requested the computation. This comes in the form of any number of packets with the structure (D,E).

The returned values are only the external values that have changed. If they are of the control type they will cause further processing to be invoked. If they are of the data type, they will simply update values in the memory. Either one only happens after the appropriate delay. A value of the data type can have the optional side effect of changing an effect from the data type to the control type or from the control type to the data type. The allocation processor is responsible for rerouting the returned information to the appropriate memory processor (decided by the address at run time), and awaiting the completion of the execution of each logical resource on a physical one. Once the logical resource is executed to completion, the memory in the allocation processor can be freed up for other storage. In all cases, information flowing from a terminal processor takes precedence over information flowing to it. In this way, emptying the pipeline has priority over filling it.

Finally, the information reaches the appropriate memory processor in the form of a structure containing (D,E) where it is stored for invocation at the appropriate time. An effect of the data type, cannot invoke further resources, while an effect of the control type always invokes further resources.

In this scheme of operation, the memory and processing elements are terminal resources, while the allocation processors are embedded resources. The queues can also be thought of as terminal resources with an identity transform.

3 The Register Transfer Level Design

The current design of the RE is completed at the architectural level with some portions completed to the register transfer level. Eventually it is hoped that the emulator will be implemented on a full wafer of silicon as a maximally configured system with external memory additions possible. In this state, the system will configure itself through self test so as to make the maximum possible number of resources available.

3.1 System Level Architecture

The RE family of computer architectures consists of a group of independent subsystems. Each subsystem is designed for implementation on a single VHSIC chip. The subsystems can be structured in sets to provide performance and reliability tradeoffs. A system consists of the following sets of subsystems.

- A set of memory processors (MP)s which maintain and dispatch global information regarding the state and description of the system being emulated to the allocation processors. They also store information from the bus watcher providing the data necessary for emulation, and forward information to an output port to allow the results of emulation to be externally observed. A timing wheel is provided for time based simulation, and system state information is maintained in tables.

- A set of allocation processors (AP)s which supervise the emulation by allocating code from memory descriptions to processing resources. This implements the operating system features necessary to execute code.

- A set of processing elements (PE)s which perform evaluation of code. The PEs themselves are strictly forward branching sequential instruction and/or limited looping processors. Detection hardware in the PEs allows them to detect data changes requiring network processing.

- A set of queueing elements which act as input and output buffers between resources and busses with arbiters. These are simply FIFO queues implemented in hardware.

- A set of input and output busses with arbiters. This provides physical links for passing information between system resources.

- A bus watcher for observing emulated subsystems. This provides the data via observation of emulated subsystems to the roving emulator.

- An output port for transmitting the results of emulation to the system under test or external observers.

The current plan will allow for:

- Multiple byte length data transfers between resources

- Up to 16 of each type of system resource (MP, AP, PE)

- Up to 16 busses and arbiters for supervising information transfers

- Queued storage of information awaiting processing

- Externally expandable resource memories

- Independent or shared processor clocks and/or asynchronous resources

In its minimal configuration the emulator consists of one MP with external memory, two input and output busses, a single general purpose PE, four queues, two bus arbiters, a bus watcher, an AP, and an output port. This comes out to a modest 11 component configuration which is capable of emulating virtually any computer system at a considerable slowdown factor. Configurations of this size could easily be used in home microprocessors or automatic test equipment in the field without substantial power, weight, or space utilization, and without significant performance degradation to the system under test.

In its maximal configuration, the current design would require over 200 resources, and could still fit on a large board in many commercial processors or on a single full wafer implementation with external memory. It is customizable to the performance characteristics of the system being emulated. From the standpoint of reliability, the maximal system could survive a large number of hardware faults and still be capable of operation with reduced performance. The performance characteristics of a full scale system depend heavily on the system being emulated, the description of that system, and the software used to generate code for the emulator. There is no performance data available at the present time.

3.2 Custom Chip Decisions

Several commercial chips have been examined for possible use in the RE, but after careful consideration it was decided that in order to get the high performance characteristics desired in this system, custom processors would have to be used for all but the MPs and perhaps special purpose PEs. Special purpose PEs are a distinct possibility. The present interface can be connected to most parallel I/O ports. The only restrictions to external processor use are the requirements to augment the power of HDL to IF compilation, to include the allocation of special purpose processors to tasks they are best at, and the requirement to make connections to them available off wafer. A custom MP is still under consideration at this time.

3.3 The Memory Processor

The memory processor (MP) (see figure 11) is entrusted with the mapping of logical network nodes and links into physical storage, the retrieval and storage of these nodes, links, and values associated with them, global time keeping, and system state detection. It is the general purpose nature of this processing which indicates the advantage of using a general purpose processor from an off the shelf manufacturer. Although all of the facilities normally used in such systems (such as floating point hardware) are not needed, it would certainly be advisable to consider previous designs before implementing a custom design, at least in the first design pass. It may be that this presents a bottleneck to system performance and requires custom circuitry for a high performance or special purpose application, but this depends heavily on the performance characteristics which are not yet available.

3.4 The Allocation Processor

The allocation processor (AP) (see figure 12) is given the tasks normally associated with an operating system, i.e. those of managing the use of resources to maximize efficiency. Like many operating systems, the AP has a notion of processes and processors, but unlike most operating systems, it is not required to do the complex types of scheduling required for general purpose applications. This makes it very amenable to special purpose implementation. The nature of the processing performed consists mainly of simple calculations that are very easily and efficiently implemented on custom integrated circuits, and which are quite slow when implemented on a general purpose machine.

The specific tasks assigned to APs are the routing of information to and from PEs, and the maintenance of high PE utilization through routing choices.

3.5 Processing Element Family of Processors

The processing element(PE) family of chips is designed to perform the major data processing portion of emulation. Depending on the level of the emulation, these element may consist of things as simple as gates or as complex as entire processors. As an example, it is relatively clear that the same circuit that is efficient at emulating an adder, may not be efficient as a Hamming code checker. For this reason, the PEs are currently designed as a family of chips with a standard interface capable of variable numbers of bytes of input and output. An example design of a PE is the generic PE shown in figure 13.

The generic PE is a prototype of a very simple PE. It has been designed to demonstrate the concept and provide a basis for simulation. The prototype is designed as a simple 16 bit stack machine with an ALU, a result register, input and output ports with protocols identical to the queue chip, a micro control engine, and an externally extendible local memory.

The generic PE accepts variable byte length data from a queue and either executes it as calls to microroutines or interprets it as input data. When the PE is ready to accept data, it reads from the queue. Once the PE gets the data it performs its designed transform and makes results available to the output queue. When the output queue is ready to accept data, it loads from the PE freeing it to process the next entry.

It is currently assumed that the PE has limited looping and branching capability. This is a basic philosophical decision in the emulators separation of control and processing, and permits deadlock free operation. This is of major interest since testability and reliability are critical issues in the implementation of an emulator, and this affords significant improvements to both.

In operation, the micro sequencer simply reads a sequence of instructions into the control store and performs a routine call to them. It places output into the output queue until the code is completed, and then awaits the next sequence of instructions. Included in the repertoire is the capability to define micro routines for later use.

3.6 Queue chip

The queue chip is used to enqueue processing tasks and results of processing between the system busses and the PEs. The reason for a hardware queue is that without it, each allocation processor would have to wait for an appropriate PE to become available for a given task, even though there might be other tasks which could proceed simultaneously in other PEs. By providing hardware queues, many tasks for each PE can be prearranged. A further advantage is that the ability to fill queues as data requires processing reduces the peak bus utilization and probability of having to wait for an available bus. This tends to increase the throughput of the system, and reduces the number of busses required for given performance characteristics. Similarly, the output queue allows a PE to process the next available data before the previous results have been read by the allocation processor. This increases the hardware utilization of the PE and thus increases system throughput.

The queue chip also provides performance data to the allocation processors, which allows them to optimize hardware utilization by spreading processing tasks evenly throughout the system. As an example, in a system with two 64 bit addition PEs which requires 3 126 bit additions, the allocation processors can queue up 2 additions to one processor and 1 to the other. As the results of the low order bytes are returned, the high order bytes can be queued up to whichever processor has the fewest tasks awaiting it. Thus rather than dedicating a PE to a given addition, the additions may be split up so that each PE processes 3 64 bit additions (parallelism = 2).

In this simple case, what would have taken one PE 4 additions and the other 2 now takes 3 each, thus saving 25% of the real time required to complete the task. In the case where one of the processors was already busy with 2 tasks in its queue, splitting the addition up so that the empty PE performs 4 additions and the loaded one 2, will result in an elapsed time of 4 additions whereas splitting it up 3 each would take 5 addition times due to the two previously queued additions. This case results in savings of 20% of the real time required to complete the task. From this example it is clear that without feedback, the scheduling of processing may be somewhat inefficient. Thus feedback is provided in the form of the current queue length for each PE.

The queue chip is implemented using a very simple finite state machine, a local memory, two mod N adders, a comparitor, and input and output busses (see figure 14). This forms a very simple integrated circuit which implements a ring based FIFO stack in hardware. Because the input and output sides of the queue chip are identical in their signaling, it is easy to enlarge a queue by simply appending one chip onto another. The simplicity also allows a large amount of space for a large queue memory, and simplifies testing to a great degree.

There are also many other applications for the queuing chip outside the realm of the RE. Because of its relatively standard external interface and the general applicability of queues, it could be used to optimize data transfers between many systems currently in use.

3.7 Bus Watcher

The bus watcher is designed to observe the external system bus and store the I/O bus data associated with the subsystem to be emulated. This provides the information required to decide whether the emulated system is performing as expected, and provides the input and output vectors required to drive the emulation. The bus watcher also provides the mechanism for dumping the state of the emulated processors at the beginning and the end of each emulation period. The actual design of the bus watcher is dependent on the bus being interfaced to, and the manner in which the state information of the processors being emulated is dumped. Our design only provides the mechanism for interfacing this information to the roving emulator.

A generic bus watcher has been designed for interface to a generic bus (see figure 15). For any specific application this design will have to be further specified before it will be usable.

3.8 Bus Arbiters

Bus arbiters are the major means for supervising communication between the resources in the system. A major design decision was made in the use of multiple busses with arbiters rather than a fully connected architecture. The reason for this decision was that the hardware necessary to implement a cross point switch is too extensive to justify the chip area when the bandwidth limitation of multiple busses is acceptable. In the case of the emulator, this bandwidth limitation is not severe because the number of busses can be increased sufficiently to allow any reasonable bandwidth, and because the queueing of data enables bursts of packets to be used, rather than requiring that processing resources waste cycles waiting for busses. The architecture of such a bus watcher is shown in figure 16.

4 Low Level System Operation

At the architectural level, the operation of the emulator has been explained in very general terms. This section details the events that occur during an actual emulation at a much lower level.

4.1 Observing the Subsystem Under Test

The first task for the emulator before starting the emulation of any given subsystem is to dump the initial state of the subsystem, gather its I/O data over a period of time, and dump the final state of the subsystem. Once this data is available the emulator can then proceed to verify the operation of the subsystem. The RE uses its bus watcher, which watches an external bus, to decide which I/O involves the subsystem being watched. Whenever the applicable subsystem is in use, the bus watcher observes the data transfers and send the relevant information to an MP in the RE. The exact details of how this is done depends on the physical interface to the system being emulated. In general the bus watcher responds to a set of bus addresses corresponding to the subsystem of interest. Any interaction with that subsystem is stored. The bus watcher can also observe external devices connected to the module provided it has the proper hardware interface to their signals.

4.2 The Memory Processor

The internal description of a subsystem about to be emulated is initially kept by the MPs. This consists of a set of packets, each containing the necessary information about one node in the delta-net description of the system being emulated. The values of links in the network are kept in a separate array structure. These structures are distributed throughout the set of MPs so that each set of nodes, and all values required for their firing are in the same MP. In this way, any data necessary for the execution of a given node can be guaranteed to be in the same memory as that node, and communication between MPs need not be done for this purpose.

If it is impossible to form this partition naturally, special nodes for passing changes can be added to cause the same effect. These special nodes are also used to synchronize timing wheel scheduling between MPs, and to perform self test for timeouts of packets.

A list of the nodes activated by a change in any signal value is also kept. When a value changes, events are scheduled by simply placing all activated nodes in the MP's event queue. At the appropriate time, based upon a node's delay, the results of the change will cause the appropriate processing. Effects returning to the memory processor can optionally cause other effects to change between the control type and the data type. This changes the activation list dynamically. In this way, every effect needn't cause activation of further processing. Very efficient data flow control can be performed through this feature as well.

In addition to the current values of internal signals, the MPs must store the observed data from the bus watcher as it arrives. This is used when all other activity in the system indicates that the time has come for the input to be read, or the output to be written. Whenever input is to be read, the values associated with this input are replaced by new values, without running the emulation until all changes have been made, and effects scheduled. This has the effect of synchronizing the execution of any inputs occurring at the same time. In the case of output, the output values given by the emulation are compared to the observed values. If they disagree, either a fault has been detected, or the emulator has made an error. This situation can be handled in several ways.

The temptation is to repeat the emulation of the possibly erroneous operation to assure that the error wasn't a transient in the emulator, verifying that the emulated result is identical in both cases. If the result is verified, only a long term transient, a permanent emulator fault, or an observation error could have been responsible. An assortment of techniques can be used to determine further details.

Once the emulation gets under way, the MP simply acts upon packets indicating changes with associated delays by scheduling the changes internally. When a scheduled event requires emulation, the MP forwards the emulation problem to an allocation processor. The forwarding involves the replacement of any tokens representing the current value of a given accesser, with the actual value in the memory. Since all of the nodes have their current values local to their MP, this is done by simple table lookup. The packet is then forwarded to an AP.

It is also possible to include a table in the MP indicating the current memory configuration of the PEs. This allows the replacement of PE code with a specification of the code block to be used, and the set of PEs on which the code block exists. The AP can then allocate packets with special requirements to special sets of processors. This facility is also used to allow the implementation of special purpose PEs.

4.3 The Allocation Processor

Once a packet is sent to an AP, it must be routed to an appropriate PE. This routing is done on the basis of the relative utilization of PEs (as determined by their queue lengths) and the special requirements of the packet (as specified by special routing information). A packet is normally routed to the

PE which is performing the best historically, fulfills all special requirements, has the shortest queue at the time of transmission, and is least suspected of having internal errors. This is determined by a set of parameters kept in an internal routing table, and updated by the experience of execution. Packets going to the PEs no longer need information about resource requirements, and this field is not sent on to the PEs.

In addition to routing packets to PEs, the allocation processors must keep track of processes sent from it pending completion on PEs. This is done in part by adding a descriptor to the beginning of the packet which indicates the AP which sent the packet out. Each packet sent from a PE is routed to the AP which created the process being executed. When the process completes, it sends a packet indicating its completion, which allows the AP to release any space associated with remembering that process.

Packets returning from the PE are only used to indicate changed outputs of the node being emulated. Thus, the PE suppresses any outputs which, although they are reevaluated, have not changed. These packets consist of the new value, the token of the effected value, and the delay after which this change takes effect. Since the token indicates which MP must get the new value, additional routing information is not needed. The AP simply forwards the new value and token to the proper recipient. Timeouts, output checks, forced process termination, and any number of other features can be added to the AP as desired.

4.4 The Processing Elements

The PEs take in a packet, which indicates a possibly complex sequence of instructions to be executed, and data to be processed by these instructions. It is the responsibility of the PE to perform the actual calculations, determine what if any changes in data take place, and forward changes and their delays to the network. In practice, this is implemented by the code which is included in the packet transmitted to the PE. The PE need only read the packet into its memory and start executing it at its beginning. The packet contains all instructions and calls to resident routines needed to achieve the desired effect.

4.5 Queuing of Data

Data is not sent directly between system resources, but rather is placed in queues awaiting processing. Once processed, it is placed in queues of processed data. Assuming that these queues are empty and the resource is available, the data will pass through very quickly with a minimum of overhead. If the resource is busy or the queue is not empty, data may have to wait for the resource before it can be evaluated. This removes a considerable burden from the system resources since only preparation, scheduling, external overseeing, and evaluation must be dealt with explicitly, and not the current state of every resource in the system. In addition, the queues allow the continued allocation of tasks while resources are processing their data, thus increasing hardware utilization. There can of course come a point where the queues are saturated and the memory and allocation resources must simply wait for queues to become available before producing new output. By providing enough hardware, this can be avoided to a very large extent.

5 Design for Testability and Performance Features

Q:`Who shall guard the guards?'

A:`They shall guard each other!'

The RE is a system for testing another system. It is therefore important to be able to verify its proper operation so as to know whether a difference in results from the two systems is due to a fault in one or the other of them. It is also important to be able to locate the fault in order to facilitate repair, and to degrade gracefully.

The RE is designed to make it possible to emulate other systems with varying degrees of performance by varying the configuration of the emulator itself. This is done in such a manner as to make nearly every aspect of the emulator redundant. As a result, the RE can be configured to operate in the presence of a large number of errors. It would clearly be advantageous to be able to perform graceful degradation online, especially if it comes with little overhead.

In addition to any self test capability, each component of the RE can be individually tested off line to assure proper operation before installation, and during repair of partially localized faults. The test procedure for each component is outlined next, followed by some sample scenarios that could allow graceful degradation of the system in place.

5.1 Allocation Processor

Testing the AP chip requires explicit details as to its internal configuration. These details have not yet been specified and therefore explicit tests have not yet been investigated.

5.2 Processing Element

The sample PE described in this report is quite easy to test because of its simple data paths, but in general the testing of PEs will have to be done on a case by case basis.

To test the PE, a memory pattern must be generated using some relatively standard techniques, and fed into the system to verify the proper operation of the memory and portions of the microstore. This includes loading the microstore with memory patterns and extracting them via micro code calls. Depending on the memory size this takes varying amounts of time.

Assuming that this test has been completed, the data paths can be exhaustively tested by performing every legal operation on every pair of data values. This sounds expensive, but in an 8 bit ALU only 2E16 such values exist for each of the approximately 8 primitive operations. It is of course possible to get a much smaller test set by analyzing the particular implementation, and verifying the ALU operation one bit slice at a time. Assuming a datum can be processed every microsecond (quite conservative for current technology), the exhaustive test of the ALU would require only one second.

It should be noted here that since the PE has restricted looping capability it may be impossible to program the entire test internally. It is however possible to perform testing through the normal flow of system information in the same way as an emulation would be programmed.

5.3 Queue Chip

The queue chip is perhaps the easiest to test of any circuit in the system. The finite state machine has very little complexity and can easily be run through its states very quickly. The initialization input makes it very easy to put it into a known state, and from that point it is easy to verify the state of the machine and memories.

Assuming that the circuit properly moves through its states, the memory is the only remaining thing to be tested. By using normal memory testing techniques, the memory can be tested simultaneously with the finite state machine, by storing and retrieving appropriate test patterns to and from the FIFO stack. Again, a stack is easily tested with standard ATEs, and can also be tested by simply passing test patterns as data through the PE to which it is attached.

It should be noted that in a test of the queues and PE it is possible to have results that are not sufficient to locate the fault inside one or the other of the chips. In the case of on line testing it is sufficient to have a go/no-go test of the group as a whole since no single part of it is useable independently. It is of course possible to remove the parts for off line testing.

5.4 Bus Watcher

The bus watcher is dependent on external systems for its design, and therefore tests for those portions not strictly internal must be done on a case by case basis.

5.5 Arbiters

Because of the relative simplicity of the arbiters in this system it is quite reasonable to design them as duplicated processors with internal error detection through comparison.

5.6 Self Test During Emulation

Because of the redundancy in the system it may be possible to perform many different types of test on the emulator to achieve varying degrees of coverage, test times, and frequency of tests. Although this is still a research issue, some sample techniques include duplication or triplication of processing in PEs, performing tests when elements are not in use, processing test vectors interleaved with emulation tasks, over configuring the system to allow testing without significant performance degradation, and graceful degradation based on test results.

In the case where a highly parallel emulation is under way, there are alternatives to self test during emulation. One solution is to interleave test vectors in the same packets as emulation. This has the effect of slowing the emulation down by a constant factor, but the advantage of testing for errors in the emulator before concluding that the observed system is good or faulty. Another case is that only a few of the subsystems under test require the full processing power of the emulator. In these cases, self test during the low utilization can be used, and the verification can be forfeited during high utilization periods. There is an entire spectrum of self test solutions available with this system, and its flexibility permits the designer to use the means which seems most suitable for the case at hand.

Given that this self test locates an error in the emulator, the question which immediately comes to mind is how the emulator can be reconfigured to continue emulation in a degraded state without errors. This is relatively easy to accomplish by informing all of the working components in the system that the failure has been detected. By ignoring input from the faulty component and not sending further processing to it, many faulty combinations of components can be configured out of the system. The major exceptions to this are the bus watcher, which provides the input data upon which emulation is based, and the output port, which reports results to the external world. In these cases the components must have built in redundancy in order to compensate for errors.

In any case, the results of the emulation need not force the system into removing its components. As an example, the system might choose to configure the questionable component out of the configuration, perform tests on it, and only after it has passed a full test set allow it to continue operation. A system with several emulators could even configure emulators out for testing without loss of emulation. Only an increase in error latency would result.

Again it should be pointed out that these techniques involve research issues which must be addressed in more detail before a meaningful analysis can be performed.

5.7 Optimizing Hardware Utilization and Performance

One way to optimize hardware utilization is by making sure that once the system is initialized, no input or output queue is ever full or empty. Assuming that this is the case, every processing resource will always have data awaiting processing. Thus whenever the current task is completed another will be waiting. In addition, no processing resource will ever have to await the emptying of an output queue before processing its next data. Thus all of the hardware will be in use all of the time, and the system will be at its peak performance all of the time.

The question that remains to be answered is how to achieve this goal. The solution that we use is to allocate enough processing resources of the proper types so that no type of resource is a bottleneck, and to always emulate a large enough set of modules or time slices from the same module so that the utilization pattern of hardware is relatively stable. In addition, to fill otherwise wasted time, self test can always be executed to maintain complete hardware utilization. Assuming that there can be a relatively stable state of hardware utilization, the problem of allocating hardware is relatively straight forward. Balance processing resources with fuller queues by adding more resources of their types, and balance PEs with emptier queues by removing processing resources of their type.

Although these ideas are straight forward, their implementation will require extensive off line simulation or actual emulation time under a variety of hardware configurations. In the case of the general purpose processing element, considerable performance advantages may be obtainable through on line feedback to the control processors. By loading code for heavily used blocks on a large number of PEs and less often used blocks on a small number of PEs the same effect can be obtained.

6 Conclusions

Several important aspects of developing a workable roving emulator have been explored, and many of the results may have much more widespread implications than were originally expected. In particular:

- A design was completed at the architectural level for a highly parallel roving emulator which fulfills all of the design goals.
- A theory of computation was developed (delta-nets) which has representational capabilities of wide applicability. It was found to be a good model for data flow systems as well as more conventional machines.
- A hardware means was found to assure deadlock detection and avoidance.
- A generalization of pipelines was found which demonstrates that very highly parallel pipelines are easily realizable, and that they can be made to degrade gracefully and have very high processor utilization.
- A very flexible set of self test and graceful degradation capabilities were designed into the system from its inception. As a result, the emulator has widespread application as a research tool in the design of gracefully changing systems.
- A simulator was developed to test out the principals and theories developed. It was found to be very flexible in its applicability and very useful for developing models of many aspects of systems.

Further work is needed to test out the theories and techniques developed in the course of this research. In particular, several lines of research and development are strongly indicated at this time. They are:

- Implementing a prototype system using off the shelf microprocessor development systems and evaluating performance of the system under actual operation will be necessary steps for the completion of this research.

- Significant theoretical work must still be done to achieve a better understanding of the entire class of computers of which the emulator is only a specific example.

- The theory of computation developed in the course of this research should be further explored to determine its potential for modeling a much broader class of machines.

- The emulator structure may be very useful as a very reliable high performance machine in its own right and as a testbed for concepts in design for testability. As such, it should be built, instrumented, and explored to as great a degree as is possible.

- The emulator has great potential for use as a remote diagnostic tool, a general purpose mobile ATE, and many other applications. Production of this type of equipment must await the results of the prototype.

As already indicated, there is much work to be done before the RE becomes a viable production level tool. The following areas are thought to be critical to its eventual widespread use:

- The development of a more extensive theory of operation
- Performance analysis of the family of architectures
- Development of high quality simulators for the emulator
- Development of software tools for translating HDLs into emulator code
- Development of software tools to aid designers in configuring the emulator for each system
- Theory and test generation for self testing the emulator during operation
- Coding of self test algorithms into the emulation compilers
- Design and development of a prototype using standard microprocessors
- Design of special purpose PEs for high speed emulation of special purpose transforms
- Theory and evaluation of graceful degradation of the emulator
- Implementation of a roving emulator chip set or a single wafer system

I. Simulations of the Emulator

The RE attempts to circumvent all of the major problems with previous emulation engines with respect to the task of performing board or chip level emulation of systems during normal operation. Because of the large amount of memory and processing power required to do the many tasks required for a full system emulation from an HDL description, it was decided that the emulation engine would require external support in the form of significant preprocessing and simulation.

I.1 The Roving Emulator Intermediate Languages

Rather than attempt to design a hardware description language for the emulator itself, we have decided to describe an intermediate form (IF) which may be compiled down to from any of the currently available HDLs as well as any future HDLs. In this way, the emulator will be able to survive significant changes in understanding and description of processing without redesign and reimplementation. In addition, because of the nature of the emulator's design, it can be configured to trade off performance, hardware, reliability, cost, area, power utilization, and other factors without changes in the description of the system being emulated. This allows implementation tradeoffs to be optimized for each application freely and enhances the capability of graceful change during operation.

As a result of this significant preprocessing and complexity of configuration it would be very difficult to evaluate performance, reliability, configuration dependencies, and other tradeoffs without some aides to analysis. Theoretical work has been done to assure to as great a degree as is possible that the system will perform its intended functions properly, but a great deal of further theory must be developed before it will serve the immediate needs of the system under implementation. As an aide to understanding the theory, analyzing the emulator, and implementing test software, two simulators were implemented.

The IF actually consists of two independent languages, a 'delta-net' language used to describe the flow of data and control in subsystems, and a data transform language used to describe the transformation of data when passed through PEs. The syntax of the IF for the delta-net is still undefined although a simulator has been written in Interlisp which implements its current form. A syntax and semantics for the sample PE has been completed and a parser has been implemented as a front end to the PE simulator.

I.2 The delta-net System Level Simulator

The delta-net system level simulator is a general purpose simulation package capable of modeling synchronous and asynchronous activity, serial or parallel processing, data dependent timing, arbitrary control and data dependencies, and virtually any other capabilities implied by the delta-net model of computation. It is implemented in Interlisp on a VAX 11/750 under Unix^{tm}. It should be noted that the actual emulator is a parallel processor while the simulations run under Unix^{tm} on a VAX-11/750. As such, actual processing in the simulator is synchronous and special provisions must be made to model asynchronous activity between multiple elements for accurate performance evaluation.

There is no preprocessing for the simulator at this time, and as such it is the users responsibility to describe processing properly. This simulator is not for general purpose use because it isn't fast enough to be used for many tasks (i.e. a gate level description of a 100,000 component system). It was designed as a high level simulator for performing functional level analysis of delta-net structures and their theoretical implications under arbitrary implementations, and serves only an auxiliary role in the testing of design ideas for the emulator.

The following description gives a flavor for the use of the simulator in a task designed to test the simulation mechanisms rather than to emulate a particular design. The nodes in this example map into the resources and delays in the delta-net structure, while the links map into both links and effects. The type of link is derived directly from their effects.

```
the variable LINKS takes the form:
        (linkname linkname ... linkname)
where:
        each name is the linkname of a link


each link takes the form:
        (value effect effect ... effect)
where:
        there can be any number of effects including 0
        each value and effect is an expression

the variable NODES takes the form:
        ((nodename delay) ... (nodename delay))
where:
        each delay is an expression returning an integer
        delay is relative to the current time
        effects not in this list are assumed to be of delay 0

each effect takes the form:
        (nodename argument ... argument)
where:
        arguments are evaluated by the function nodename
        there can be any number of arguments including 0

The simulation output takes the form:
Every relevant event time is indicated:
        TIME NOW =12
Changed links are shown taking new values:
        A<=1
Effects caused by change are printed:
        =>((TESTA (QUOTE B)) (PRINT (QUOTE (JUST RAN A)) SIMOUT))
Any output from currently executing links is printed:
        (JUST RAN A)
Any of the last three may continue indefinitely until all activity
for that time has ceased. The next event time is then handled.
```

This particular example demonstrates the many capabilities of the simulator in a very simple way. It involves two links which effect resources in such a way as to force them to oscillate execution. There exists a terminating condition, a link with no effects, an effect with no delay, an effect with a delay which is a function of its inputs, and effects with constant delay. In addition, event delays are such that there is a case where the delays of two effects cause them to be simultaneously executed. The example goes like this:

```
(DEFINEQ (TESTA
        (LAMBDA (NAME)
                (* SUBTRACT ONE FROM NAMED LINK INVOKING CHANGES TESTB)
                (LINK NAME (SUB1 (GETLINK NAME)))
)))
(ADDNODE 'TESTA '(IQUOTIENT (GETLINK 'B) 2))

(DEFINEQ (TESTB
        (LAMBDA (NAME1 NAME2)
                (* ADD ONE TO NAME1 TILL > NAME2 INVOKING CHANGE)
                (COND ((ILESSP (GETLINK NAME1) (GETLINK NAME2))
                        (LINK NAME1 (ADD1 (GETLINK NAME1))))
                (T      'DONE))
)))
(ADDNODE 'TESTB 2)

(DEFINEQ (TESTC
        (LAMBDA (COUNTER)
                (* ADD ONE TO LINK AND INVOKE NOTHING)
                (LINK COUNTER (ADD1 (GETLINK COUNTER)))
)))

(ADDLINK 'A)
(SETQ A
(QUOTE (10 (TESTA (QUOTE B))
        (PRINT (QUOTE (JUST RAN A)) SIMOUT)
)))

(ADDLINK 'B)
(SETQ B
(QUOTE (10 (TESTB 'A 'B) (TESTC 'C))
))

(ADDLINK 'C)
(SETQ C (QUOTE (0)))

(CLEARDNET)
(LINK 'A 0)
```

Link A is initially at the same value as B and the system is stable with no effects scheduled. The system is perturbed by causing A to take on the value of 0. The effect of a change in A is to cause a message to be printed out (JUST RAN A) and to schedule TESTA to evaluate the current value of B. This evaluation take place after a delay which is a function of B (in this case int(B/2)). It should be noted that although this simple example uses integer values for links, the values are not restricted to integers or even numbers, they can be arbitrary list structures including the descriptions of nodes or links, and the simulation will operate in the same manner (assuming that the changes are not to the simulator itself).

TESTA evaluates B by subtracting one from it. This change in the link B has the effect of causing TESTB to evaluate the current values of TESTA and TESTB and the effect of causing TESTC to evaluate the current value of C. TESTC simply adds one to link C with a constant delay, and since link C has no effects associated with it, this causes no further activity. The evaluation of TESTB however is to compare the values of link A to link B. Unless B < A, TESTB causes A to be incremented by one with a constant delay. This then causes a cycle in the simulation which will force A and B to approach each other until B < A, each step proceeding faster than the previous one due to the reduced delay of TESTB (which is data dependent on the value of B). When the system stabilizes and the dust settles the simulation terminates. Here is the output:

```
TIME NOW =1
C<=1
TIME NOW =12
A<=1
        =>((TESTA (QUOTE B)) (PRINT (QUOTE (JUST RAN A)) SIMOUT))
(JUST RAN A)
TIME NOW =15
B<=24
        =>((TESTB (QUOTE A) (QUOTE B)) (TESTC (QUOTE C)))
TIME NOW =16
C<=2
TIME NOW =27
A<=2
        =>((TESTA (QUOTE B)) (PRINT (QUOTE (JUST RAN A)) SIMOUT))
(JUST RAN A)
.
.
.
TIME NOW =156
A<=13
        =>((TESTA (QUOTE B)) (PRINT (QUOTE (JUST RAN A)) SIMOUT))
(JUST RAN A)
TIME NOW =159
B<=12
        =>((TESTB (QUOTE A) (QUOTE B)) (TESTC (QUOTE C)))
TIME NOW =160
C<=14
TIME NOW =165
```

Although this simple example does not exemplify any significant transformation of data, it does serve quite well to demonstrate the general capabilities of the delta-net model of computation and the ability of the simulator to simulate a great variety of systems of current interest.

Let's now look at a structure that many simulators have problems handling. An extendible one shot circuit commonly used for initialization at power up is considered difficult to simulate because it invokes itself due to capacitive delays in its feedback. This is further complicated by the ability of a hold signal to recharge the capacitor, thus causing an arbitrary length delay which is unknown when it is initially scheduled.

There are several solutions to this problem, one of the easiest and least efficient being the use of a resetable counter to functionally simulate the one shot. The counter counts down a variable every clock time (or if the simulation of the capacitance is important it performs logrithmic subtractions according to circuit parameters) and triggers the end of initialization when it reaches zero. The hold signal is simulated by externally resetting the value of the counter to the appropriate value. This will work quite well, but is extremely inefficient considering that for short clock times and long initializations the system could run a large number of processes simply to cause a constant delay.

The preferred solution is to have a link which contains the time for termination of the oneshot. Upon initial instantiation, this value is changed to the delay of the oneshot. This link has the effect of scheduling the oneshot in the event queue for that time. Upon reaching that time, the oneshot checks the value of the link to make sure it is the same as the current simulation time. If so, it terminates initialization, if not it does nothing. The hold circuit is simulated by adding an appropriate delay to the value of the link. This change in the link will cause the oneshot to be scheduled at the new time (without the complexity of removing the previous scheduling). Here is the code and execution of this simulation with comments:

```
(* A Delta-net model of a One Shot)
(SETQ LINKS '(CAPACITOR INITDONE))
(* the links are CAPACITOR and INITDONE where
CAPACITOR holds the current termination time of the oneshot
and INITDONE is the output from the initialization circuit
)
(SETQ NODES '(
        (ONESHOT (IDIFFERENCE (GETLINK 'CAPACITOR) TIMENOW))
        (HOLD 0)))
(* the nodes are ONESHOT and HOLD - ONESHOT gets evoked at
a delay that depends on the value of the capacitor, HOLD
gets evoked with delay 0
)
(SETQ CAPACITOR '(0 (ONESHOT) (LINK 'INITDONE NIL)))
(* CAPACITOR initially at 0, effects ONESHOT and
the link INITDONE when it changes
)
(SETQ INITDONE '(T))
(* INITDONE is initially TRUE and effects nothing)

(DEFINEQ (ONESHOT
        (LAMBDA ()      (* The oneshot itself)
                (COND   (* If TIMENOW=capacitor)
                ((EQ TIMENOW (GETLINK 'CAPACITOR))
                 (LINK 'INITDONE T)) (* initdone = T)
                (T      T))))) (* else do nothing)

(DEFINEQ (HOLD
        (LAMBDA ()      (* The hold circuit)
        (COND           (* if hold extends the oneshot)
        ((ILESSP (GETLINK 'CAPACITOR) (* change capacitor)
                 (IPLUS TIMENOW HOLDTIME))
        (LINK 'CAPACITOR (IPLUS TIMENOW HOLDTIME)))
                (T T)))))

(SETQ HOLDTIME 3)       (* a hold delays by at least 3)
(SETQ PENDING NIL)      (* start out with nothing pending)
(SETQ SOON '((3 ((HOLD))))) (* schedule a hold for time 3)
(SETQ TIMENOW 0)        (* set time now to be time 0)
(LINK 'CAPACITOR 5)
(* trigger the oneshot with the capacitor charged to 5)
CAPACITOR<=5    (* capacitor changed)
        =>((ONESHOT) (LINK (QUOTE INITDONE) NIL))
                (* effect is scheduling the oneshot for
                time 5 and changing INITDONE to NIL)
```

```
(RUN 10)          (* run the simulation for 10 time steps)
                  (* started at TIMENOW =0)
INITDONE<=NIL     (* initdone became NIL)
                  (* which triggers nothing in this test)
TIME NOW =3       (* the time is now 3)
CAPACITOR<=6      (* the hold at time 3 changed capacitor to 6)
        =>((ONESHOT) (LINK (QUOTE INITDONE) NIL))
                  (* the change in capacitor effects ONESHOT
                  but doesn't change INITDONE from nil)
TIME NOW =5       (* at time 5 the original ONESHOT does nothing)
TIME NOW =6       (* at time 6 the held ONESHOT sees capacitor=6)
INITDONE<=T       (* this changes INITDONE to T, and since
                     INITDONE effects nothing simulation ends)
T                 (* successful termination of simulation)
```

I.3 The Processing Element Parser and Simulator

The PE parser and simulator is an integrated software package intended to simulate the generic PE specified earlier in this report. It serves several purposes including the solidification of the design of that PE, demonstration of a parser for the PE, testing of sample descriptions in the intermediate syntax, and demonstrating the feasibility of the generic resource for general purpose processing. The following description outlines the syntax and semantics of the parser/interpreter.

As of this time, full macroprocessing capabilities exist in the processing element simulator. This is helping solidify the architecture of the system and the IF, simulate emulation concepts, and evaluate performance on a limited basis.

The current description of a data transform consists of a logical expression using the common operators of boolean algebra augmented by a few commonly used ALU and switching functions. It is hoped that a means will be provided for emulating finite state machines and programmable logic arrays in the eventual implementation, but no plan for this is currently under implementation. It is felt that the higher the level used for description, the faster the emulator will execute. This will lower the latency time for fault detection and provide better coverage over a given period of emulation time. As an example, it is a significant saving in effort to express adders as adders rather than as gate descriptions when the functional behavior is the only result of interest. This also holds for more complex transforms. In addition, the low level operations of the actual implementation need not be known in order to generate emulation code. At the same time it is necessary to be capable of describing special purpose transforms of a general type, and thus low level operations are made available.

A BNF like description of the data syntax is presented here in easily readable form. This description was taken from the code that implements the parser (written in yacc under Unix^{tm}):

```
tokens: GATE GIND GMEM LIND LMEM PUSH POP
        NOOP IN OUT INIT TERMIN DIGIT

operators in order of precedence:
        NOOP IN OUT GATE LMEM GMEM LIND
        GIND PUSH POP INIT

list    :       /* empty */
        |list TERMIN
        |list stat TERMIN
        |list error TERMIN
        ;

stat    :PUSH stat POP
        |complex
        |IN int
        |OUT stat
        |int
        |NOOP
        |INIT
        ;

complex :LIND seper stat
        |GIND seper stat
        |stat seper GATE seper stat
        |stat seper GMEM seper stat
        |stat seper LMEM seper stat
        ;

seper   :
        ;

int     :DIGIT
        |int DIGIT
        ;
```

A lexical scanner (lex under Unix^{tm}) is used to map the following characters into the tokens of the BNF like description above:

Symbol	Use	Token
!	followed by anything is a	NOOP
@	global indirection	GIND
.@	local indirection	LIND
>	global storage	GMEM
.>	local storage	LMEM
(or {	parenthetical expressions	PUSH
) or }	end of parentheses	POP
INIT	initialize processor	INIT
IN	input from input port	IN
OUT	output to output port	OUT
AND	a gate called AND	GATE
OR	a gate called OR	GATE
NEQ	a gate called NEQ	GATE
NOR	a gate called NOR	GATE
NAND	a gate called NAND	GATE
+	a gate called +	GATE
-	a gate called -	GATE
LSH	a gate called LSH	GATE
RSH	a gate called RSH	GATE
*	a gate called *	GATE
/	a gate called /	GATE
?	a gate called ?	GATE
~	a gate called ~	GATE
EOL	end of line is a terminator	TERMIN
anything else is ignored		

A sample description of a design is provided in detail in the implementation scenario for the rs/232 encoder chip presented in a later section of this report, along with the results of parsing and simulating its PLA.

II. An Implementation Scenario

This scenario is a simple overview of the addition of a roving emulator to a system implementing the 255/223 rs-encoder circuit using Berlekamp's bit serial multiplier [Tru 82]. The examples include the description of a portion of the chip in a HDL, the translation of the HDL example into our IF, sample runs from our PE parser and simulator, a description of the system in operation testing a board level implementation of this small computer, an informal analysis of configurations and slowdown factors for that scenario and their good and bad points, and a description of the implementation of a bus watcher for that system.

II.1 Sample HDL Description of the PLA

The encoder chip has a rather large PLA which is used to form partial products of a Galois field multiplication used in the Berlekamp multiplier circuit. There are a total of 33 bits determined by this PLA, but the task is substantially reduced by the fact that 16 of the bits are duplicated symmetrically in the output from the input, and many of the partial product terms have constant zero values. For this reason, emulation can be done quite quickly if the correct mathematical representation is given for the PLA. Each bit in the PLA output is the result of the 'xor' of a set of 'ands' of constants with the input vector (8 bits). Since all zero valued constants result in a zero 'and' output, and since the 'xor' of 0 with anything results in identity, these terms can be ignored. In addition, the constant values of the 'ands' are half zeros. Since there are only 18 distinct outputs from this circuit (16 of the 34 are copies of other values) only 18 sets of 'xors' must be done for each of 4 'ands' (average number of 'ands' per element). The boolean expressions for these 34 outputs are given below.

$T_0 = Z_0;$
$T_1 = Z_0xZ_1xZ_3xZ_4xZ_6;$
$T_2 = Z_0xZ_1xZ_2xZ_3xZ_4xZ_5xZ_6;$
$T_3 = Z_1xZ_2xZ_4xZ_6;$
$T_4 = Z_4;$
$T_5 = Z_1xZ_2xZ_3xZ_4;$
$T_6 = Z_0xZ_2xZ_3;$
$T_7 = Z_0xZ_1xZ_3xZ_5xZ_6xZ_7;$
$T_8 = Z_0xZ_5xZ_6;$
$T_9 = Z_0xZ_2xZ_5xZ_7;$
$T_{10} = Z_3;$
$T_{11} = Z_1xZ_3xZ_5;$
$T_{12} = Z_1xZ_2xZ_4xZ_5;$
$T_{13} = Z_1xZ_2xZ_4xZ_6;$
$T_{14} = Z_0xZ_1xZ_3xZ_5xZ_7;$
$T_{15} = Z_5;$
$T_{16} = Z_0xZ_4xZ_5xZ_6;$
$T_{17} = T_{15};$
$T_{18} = T_{14};$
$T_{19} = T_{13};$
$T_{20} = T_{12};$
$T_{21} = T_{11};$
$T_{22} = T_{10};$
$T_{23} = T_9;$
$T_{24} = T_8;$
$T_{25} = T_7;$
$T_{26} = T_6;$
$T_{27} = T_5;$
$T_{28} = T_4;$
$T_{29} = T_3;$
$T_{30} = T_2;$
$T_{31} = T_1;$
$T_{32} = T_0;$
$T_{33} = Z_0xZ_1xZ_2xZ_7;$

II.2 The PLA in Processing Element Intermediate Form

Fortunately, formula of this form are very easily expressed in the PE-IF as follows:

```
Z0 > 0
(Z0 neq (Z1 neq (Z3 neq (Z4 neq Z6)))) > 1
(Z0 neq (Z1 neq (Z2 neq (Z3 neq (Z4 neq (Z5 neq Z6)))))) > 2
(Z1 neq (Z2 neq (Z4 neq Z6))) > 3
Z4 > 4
(Z1 neq (Z2 neq (Z3 neq Z4))) > 5
(Z0 neq (Z2 neq Z3)) > 6
(Z0 neq (Z1 neq (Z3 neq (Z5 neq (Z6 neq Z7))))) > 7
(Z0 neq (Z5 neq Z6)) > 10
(Z0 neq (Z2 neq (Z5 neq Z7))) > 11
Z3 > 12
(Z1 neq Z3) neq Z5 > 13
(Z1 neq Z2) neq (Z4 neq Z5) > 14
(Z1 neq (Z2 neq Z4) neq Z6) > 15
(Z0 neq (Z1 neq Z3) neq Z5) neq Z7 > 16
Z5 > 17
(Z0 neq Z4) neq (Z5 neq Z6) > 20
@17 > 21
@16 > 22
@15 > 23
@14 > 24
@13 > 25
@12 > 26
@11 > 27
@10 > 30
@7 > 31
@6 > 32
@5 > 33
@4 > 34
@3 > 35
@2 > 36
@1 > 37
@0 > 40
(Z0 neq Z1) neq (Z2 neq Z7) > 41
```

The neq operation is a bitwise exclusive or of two values,replacing the xor notation (if (a = not b)) = (if (a and not b) or (b and not a)).

II.3 Parsing and Simulating Intermediate Form

Rather than produce the entire parsed output of this PLA description, a sample parsing of a few of the entries will serve the purpose. The parser and simulator are currently implemented so as to execute concurrently with parsing, so the simulation will be included in the parsing.

In the parser/simulator output one will note the occurrence of the define command. This is used to simulate values input to the PE since simulators for the other portions of the emulator do not yet exist. This facility is actually part of the PE, and allows the element to off load program data from the memory and allocation processors so that program doesn't require reload over the period of the emulation (256 test case in this case). In addition, the results of each calculation are returned at the end of their parse for the specific input case. This demonstrates the output values attained by the current inputs. The prompt rove> indicates the program accepts the rest of that line as input.

```
rove>divert(-1) define(Z0,0) define(Z1,1) define(Z2,1)
define(Z3,1) define(Z4,0) define(Z5,1) define(Z6,1)
define(Z7,1) divert(0)
rove>Z0 > 0
PUSH LITERAL 0;PUSH LITERAL 0;POP TO GLOBAL MAR;
POP TO GLOBAL MEMORY;
0
rove>(Z0 neq (Z1 neq (Z3 neq (Z4 neq Z6)))) > 1
PUSH LITERAL 0;PUSH LITERAL 1;PUSH LITERAL 1;
PUSH LITERAL 0;PUSH LITERAL 1;NEQ;PUSH FROM RESULT;
NEQ;PUSH FROM RESULT;NEQ;PUSH FROM RESULT;NEQ;
PUSH FROM RESULT;PUSH LITERAL 1;POP TO GLOBAL MAR;
POP TO GLOBAL MEMORY;
1
rove>
(Z0 neq (Z1 neq (Z2 neq (Z3 neq (Z4 neq (Z5 neq Z6)))))) > 2
PUSH LITERAL 0;PUSH LITERAL 1;PUSH LITERAL 1;
PUSH LITERAL 1;PUSH LITERAL 0;PUSH LITERAL 1;
PUSH LITERAL 1;NEQ;PUSH FROM RESULT;NEQ;PUSH FROM RESULT;
NEQ;PUSH FROM RESULT;NEQ;PUSH FROM RESULT;NEQ;
PUSH FROM RESULT;NEQ;PUSH FROM RESULT;PUSH LITERAL 2;
POP TO GLOBAL MAR; POP TO GLOBAL MEMORY;
1
.
.
.
rove>@1 > 37
PUSH LITERAL 1;POP TO GLOBAL MAR; PUSH FROM GLOBAL MEMORY;
PUSH LITERAL 37;POP TO GLOBAL MAR; POP TO GLOBAL MEMORY;
1
rove>@0 > 40
PUSH LITERAL 0;POP TO GLOBAL MAR; PUSH FROM GLOBAL MEMORY;
PUSH LITERAL 40;POP TO GLOBAL MAR; POP TO GLOBAL MEMORY;
0
rove>(Z0 neq Z1) neq (Z2 neq Z7) > 41
PUSH LITERAL 0;PUSH LITERAL 1;NEQ;PUSH FROM RESULT;
PUSH LITERAL 1;PUSH LITERAL 1;NEQ;PUSH FROM RESULT;NEQ;
PUSH FROM RESULT;PUSH LITERAL 41;POP TO GLOBAL MAR;
POP TO GLOBAL MEMORY;
1
rove>exit
```

II.4 Analysis of a Configuration

As an example of the analysis of a configuration, we have taken the design of the encoder and broken it up into the modules from which it was designed. This is an example of a circuit which up until now has been a multiple component system, and as such exemplifies the application of the emulator to such systems as well as its possible ramifications to testing chips with visibility busses during normal use. A major assumption used here is that the emulator has the same clock rate as the system under emulation. This may not be the case in many examples. For instance, if the emulator is implemented on a single wafer, it will be about 100 to 1000 times faster than it would be on a prototype bread board with standard microprocessors emulating system components. Similarly, the encoder will probably operate 100 times faster on a chip than when made of discrete components on several boards with TTL technology. For this reason, the slowdown factors presented here must be taken with a 'grain of salt'.

The encoder chip is composed of 5 subunits:

1. Product Unit - the PLA described above
2. Remainder Unit - 33 8 bit shift registers chained through xors
3. Quotient Unit - 2 8 bit shift registers with parallel load
4. Control Unit - combinational logic and an 8 bit counter
5. IO Unit - 2 registers and a switch

In operation, the emulator goes through 256 clock cycles at a time, translating input to output. At the beginning of each 256 bit cycle, the system clears its internal state, leaving all state information at 0s. This is rather handy from the point of view of the emulator since there is no need to dump the initial state of the system. Even if it was desired to dump the initial state, because of the large memory in the remainder unit, it would take almost 256 cycles to dump the state without the addition of a large number of pins to the circuit. This could be done, but has no advantage in this case, and in fact incurs a significant disadvantage since by removing the requirement to dump state, the system being emulated can execute with no performance degradation. In this case, we decided not to use the final state of the system either since it would incur a large overhead with very little potential advantage. Because of this initialization behavior, windows of more than 256 cycles are meaningless, thus the window size is restricted to from 1 to 256 cycles of the encoder clock.

Each of the modules in the system will also have their own slowdown factors for emulation. As an example, the PLA can do all 204 operations required to simulate it in one clock cycle. Doing this sequentially would clearly take a great deal more time. This slowdown factor is also effected by the configuration of the emulator, since with 16 PEs, it may be possible to do 204 operations in only 14 emulator clock cycles. Even more interestingly, the emulator might be configured to have enough memory to store the entire PLA transform in its memory, thus emulating the operation with a single table lookup. Each of these possibilities requires a cost performance tradeoff which must be analyzed in the light of external factors such as cost, power, area, and reliability.

The remainder unit is an example of a subsystem composed almost entirely of memory. It is usually very difficult to emulate memory because it requires at least as much memory in the emulator as in the system and a dump of the entire initial state. In this case however, the memory is not so large that it is unemulatable, and the state is initialized so as to be known every 256 cycles of the machine. For this reason, it is reasonable to emulate the memory using 33 8 bit memory locations, shifting each and performing the required xors on a single PE. This much memory is available on our simple PE, and the capability to shift and mask properly is available. The actual machine does this entire process in a single clock cycle, whereas it requires at least a shift and 2 masks per register in our simple PE. For this reason, we could expect a slowdown factor of about 90 for the emulation of this special purpose memory. Again, it would be easy to use table lookups at the expense of space, or even to use a better PE than our simple one to produce significant improvements.

The quotient unit is simply a pair of shift registers with parallel load and output. This requires very little processing power in the emulator, and could easily be accomplished with 4 emulation cycles per encoder cycle. Since this is so easy to accomplish, and the slowdown factor is so good, there is no reason to do otherwise.

The control unit has only 3 bits of state (used for the divide by 8 counter which produces the c2 output), and would be relatively easy to emulate. Its purpose is the generation of nonoverlapping clocks, and as such it is an asynchronous device which, although it could be modeled synchronously with slight loss of information, would best be modeled using the full power of the emulators delta-net capability. This would produce a slowdown factor of nearly 20 for the entire control circuit as opposed to a slowdown of only 4 or 5 for a synchronous model. Again, depending on the requirement for fault location, this might or might not be a reasonable tradeoff. In the case of a chip, the entire chip is faulty if any component is faulty, and must therefore be switched out of operation (a relatively simple addition to the encode would permit the input to be tied to the output with an external control signal). If this is the case, then the system will no doubt produce other errors during operation, and the emulation of the control unit would probably not be very advantageous. In the case of the controller being a board in a larger system, it would be necessary to model the controller in more detail to get good fault location.

The IO unit is extremely simple to emulate, requiring only two memory bits and a gate. In this case, the emulator would produce a slowdown factor of only 3 or 4, and no optimization would be worthwhile.

The previous analysis does not consider several points, those being the overhead associated with dumping state in many systems, the design tradeoffs required to analyze any designed system, and the control overhead incurred by switching between emulation of different components and loading their data into the PEs. Because of the queuing used in the emulator and the ability to off load sequential processing to the PEs this overhead should be expected to average about 4 to 5 cycles per operation, and this constant should be added to each emulation time for a good estimate.

Much more importantly, the emulator is a data flow machine, and the performance of this type of machine is not a well understood phenomena. In order to perform any accurate analysis, considerable theoretical work must be done.

II.5 The Encoder Emulation in Operation

In operation, the emulator would rove through each of the 5 modules of the encoder system emulating each for a predetermines window of time. The actual window for each subsystem is a complex function of the module and the requirements placed on the system for detection. In our example analysis, the error latency for on line testing of an implementation of this system comes to about .1 sec (based on a 2MHz system clock and a requirement of 200000 clock cycles average to detect a single stuck at fault). In the case of the encoder system, most of the test time (about 70%) is spent in testing the internal memory of the encoder as designed, and this time could be significantly improved by using other means of self testing or additional observation points in the design as implemented. In the case of the encoder, it is possible to perform the emulation testing with no interference whatsoever in the operation of the system (strictly from external observation).

II.6 A Simple Bus Watcher

A simple bus watcher would serve for the case of the encoder system, consisting only of a local memory, interface to the emulator via a port into one of the system busses, and external interface via clocked parallel load registers and a single multiplexer. This simple architecture is pictured in figure II.1. It must be noted that many systems will not require specialized interface to the emulator, but rather will interface directly through system busses. In special cases, however, it is relatively easy to handcraft an interface to the emulator as required.

REFERENCES

[Abr 81] M. Abramovici, Y. H. Levendel, and P. R. Menon.
A Logic Simulation Machine.
Bell Labs, Naperville, Il 60566, 1981.

[Bar-1 80] M. Barbacci, A. Nagel, and J.D. Northcutt.
An ISPS Simulator.
Carnegie-Mellon University, Pgh. Pa. 15213, 1980.

[Bar-2 80] M. Barbacci, G. E. Barnes, R. G. Cattell, and D. P. Siewiorek.
The ISPS Computer Description Language.
Carnegie-Mellon University, Pgh. Pa. 15213, 1980.

[Bre-1 82] M. A. Breuer and A. A. Ismeal.
Roving Emulation as a Fault Detection Mechanism.
DISC Report No. 82-7, Electrical Engineering Dept., University of Southern California, 1982.

[Bre-2 82] M. A. Breuer, F. Cohen,and A. A. Ismaeel.
Roving Emulation as Applied to a (255,223) RS-encoder system.
DISC Report No. 82-6, Electrical Engineering Dept., University of Southern California, 1982.

[Bri 80] P. Brinch-Hansen.
Operating Systems Principals.
, N.Y., 1980.

[Dul 68] J.R. Duley and D.L. Dietmeyer.
A Digital System Design Language (DDL).
IEEE Trans Comp., 1968.

[Jef 82] D. Jefferson and H. Sowizral.
Fast Concurrent Simulation Using Time Warps, Part 1 - Local Control.
Unpublished - Technical Report Rand Corporation, Santa Monica Ca., 1982.

[Pat 81] Paterson.
Petri-Net Theory and the Modeling of Systems.
Prentice Hall, N.Y., 1981.

[Pfi 82] Pfister.
The Yorktown Simulation Engine.
19th Anual IEEE Conference on Design Automation Conference, N.Y., 1982.

[Tex 82] unknown.
The Hardware Description Language - Language Summary.
Unpublished - Technical Report Texas Instruments, Dallas Tx., 1982.

[Tru 82] T.K. Truong, I.S. Reed, I.S. Hsu, K. Wang, and C.S. Yeh.
The VLSI Implementation of a Reed-Solomon Encoder using Berlekamp's Bit-Serial Multiplier Algorithm.
Submitted to IEEE Trans. on Computers, 1982.

[Zim 80] G. Zimmermann.
The Mimola Design System (2 papers).
4th International Conference on Computer Hardware Description Languages and their applications, 1980.

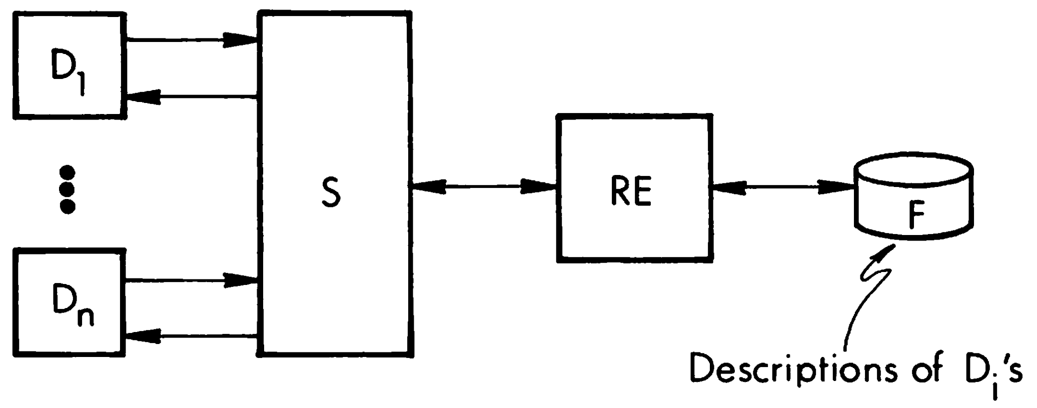


FIGURE 1A

A ROVING EMULATOR IN A GENERAL SYSTEM

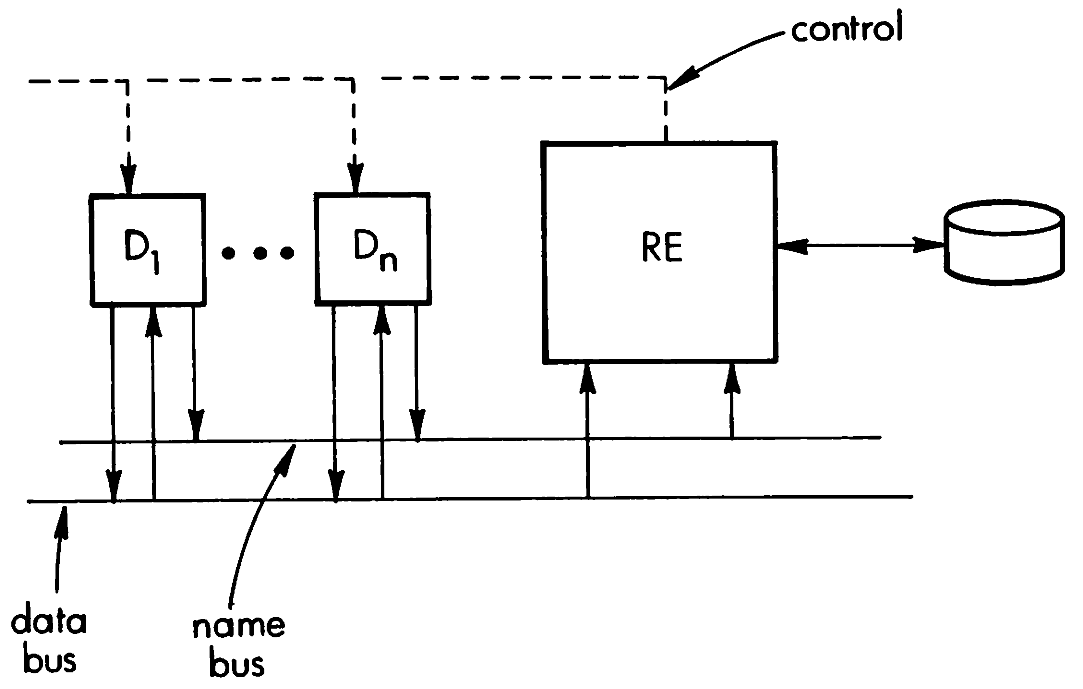


FIGURE 1B

A BUS ARCHITECTURE WITH A ROVING EMULATOR

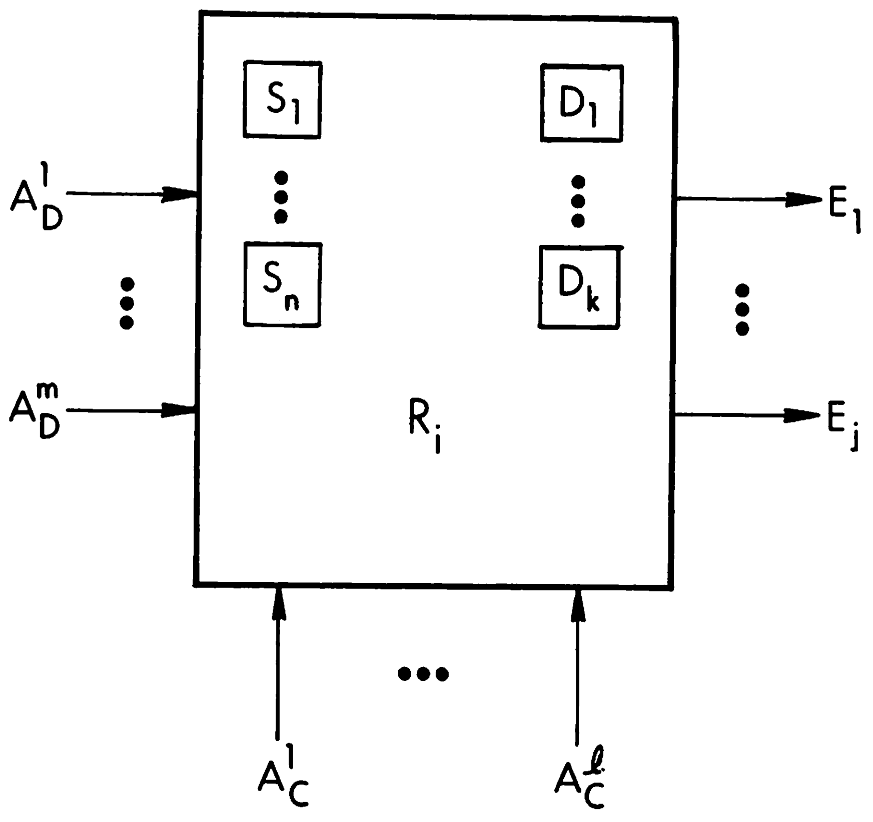


FIGURE 2A

A SAMPLE DELTA-NET ELEMENT

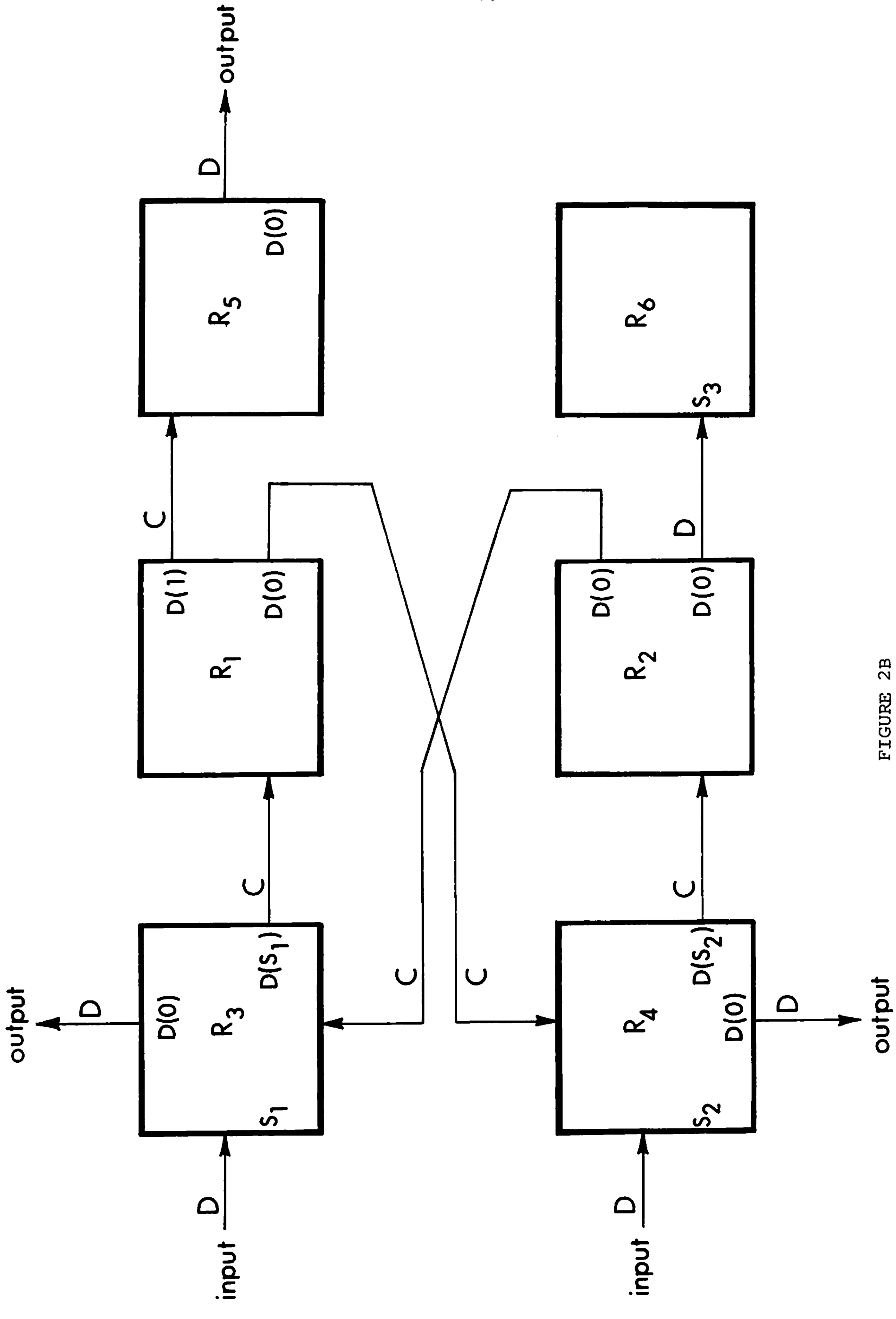


FIGURE 2B

A SIMPLE DELTA-NET NETWORK

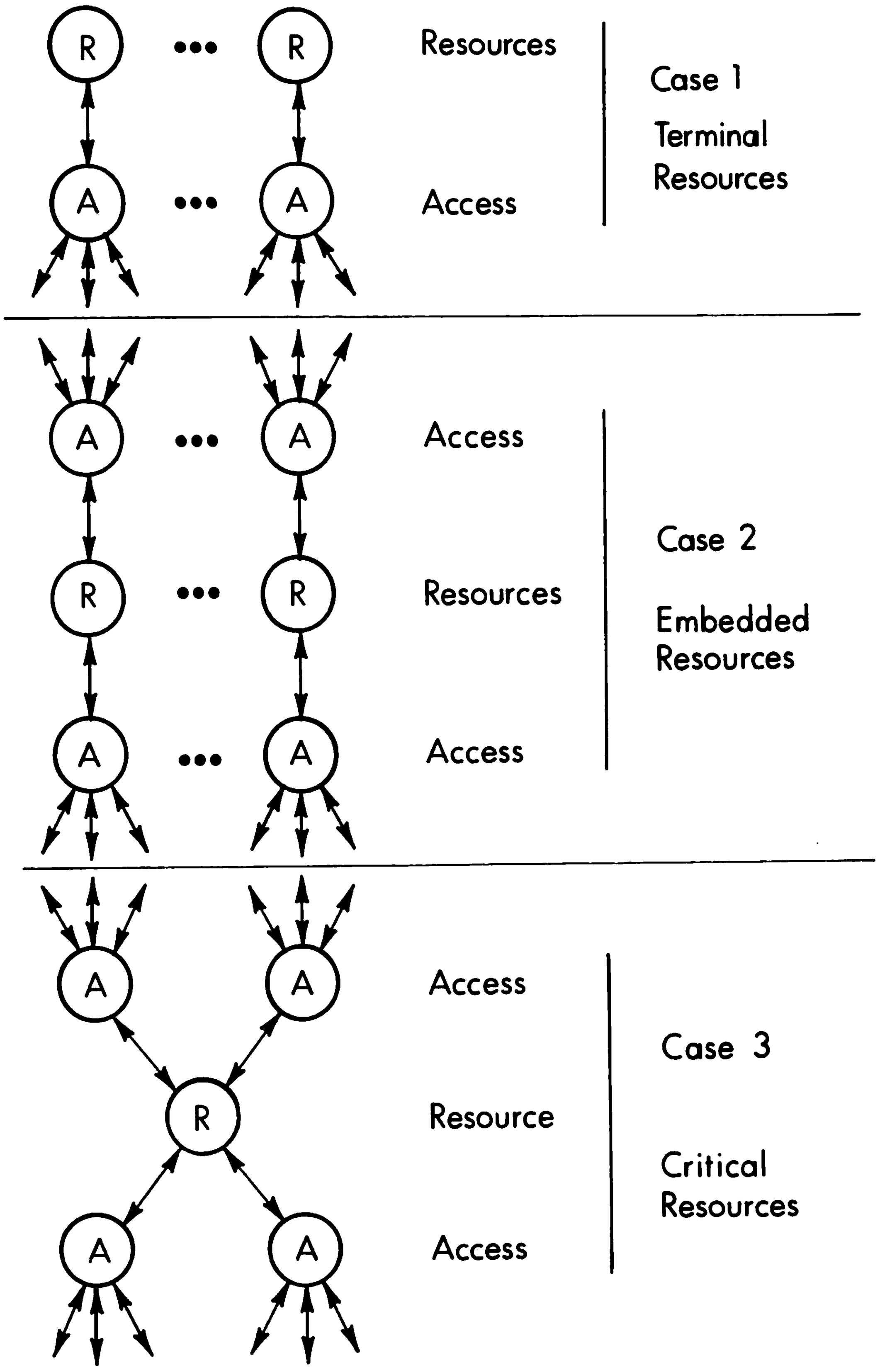


FIGURE 3

THREE TYPES OF RESOURCES

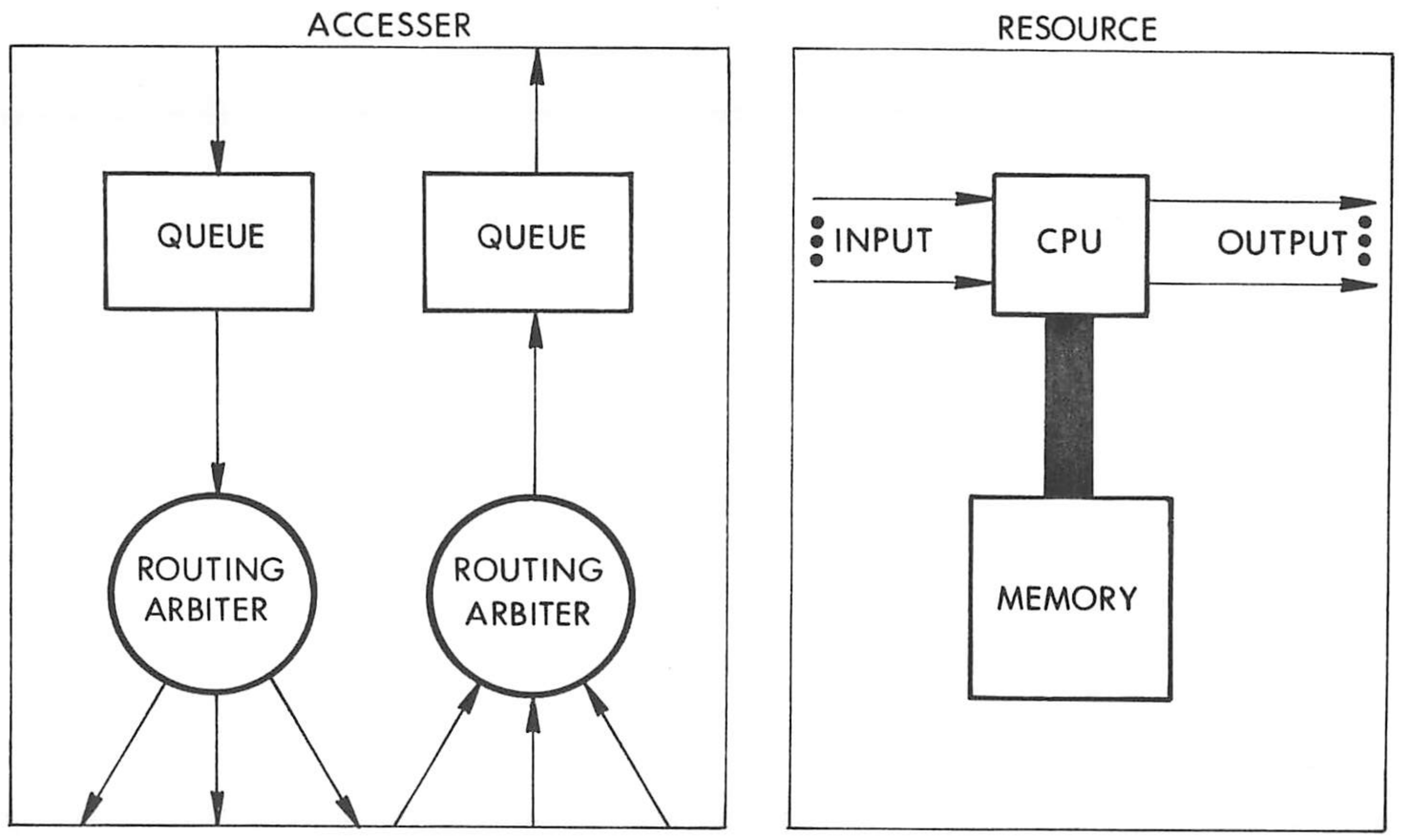


FIGURE 4

RESOURCES AND ACCESSERS

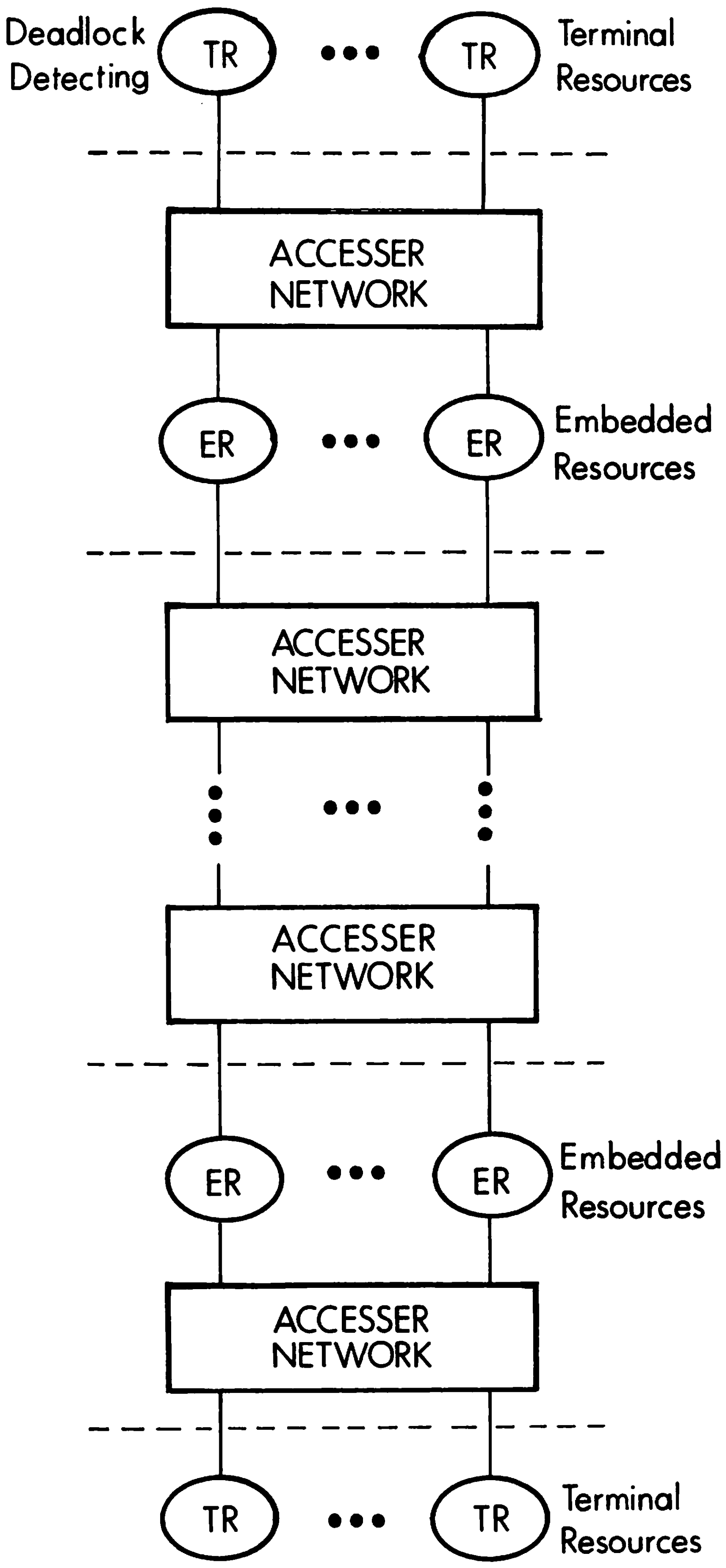


FIGURE 5

A GENERALIZED STRUCTURE FOR DEADLOCK PREVENTION

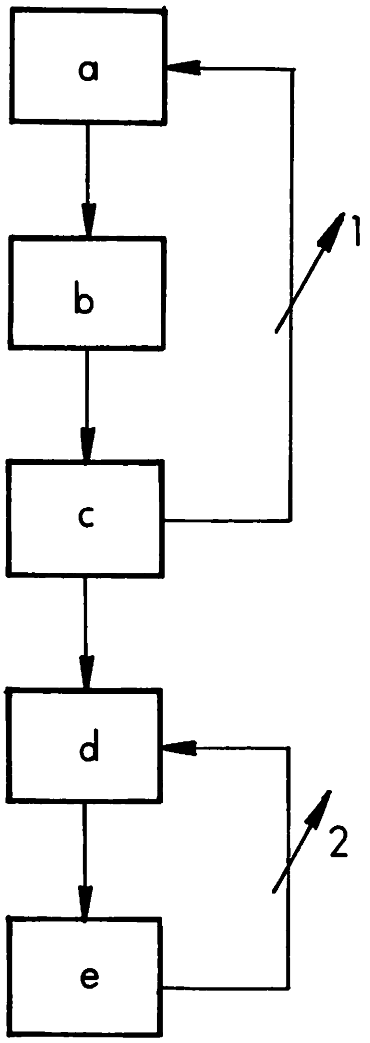


FIGURE 6

A 5 ELEMENT PIPELINE WITH TWO RECONFIGURABLE LOOPS

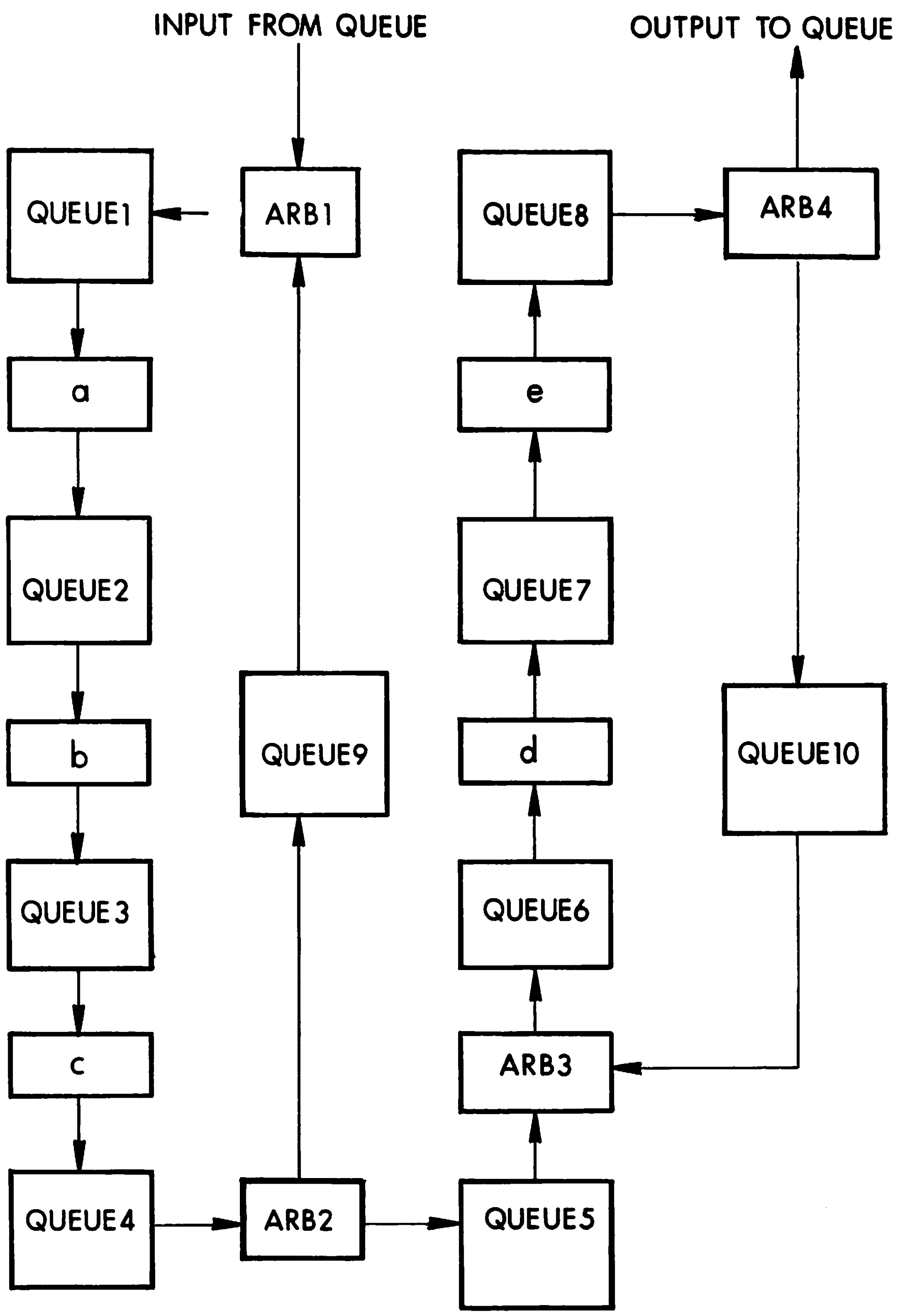


FIGURE 7

A DATA FLOW PIPELINE USING QUEUED RESOURCES

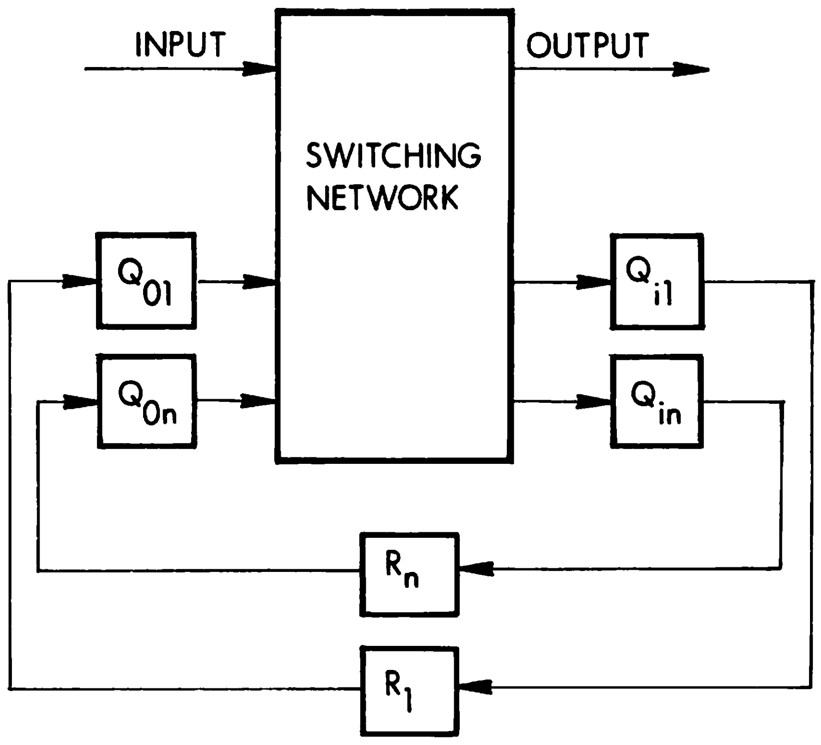


FIGURE 8

THE GENERAL FORM OF A GENERIC RESOURCE PIPELINE

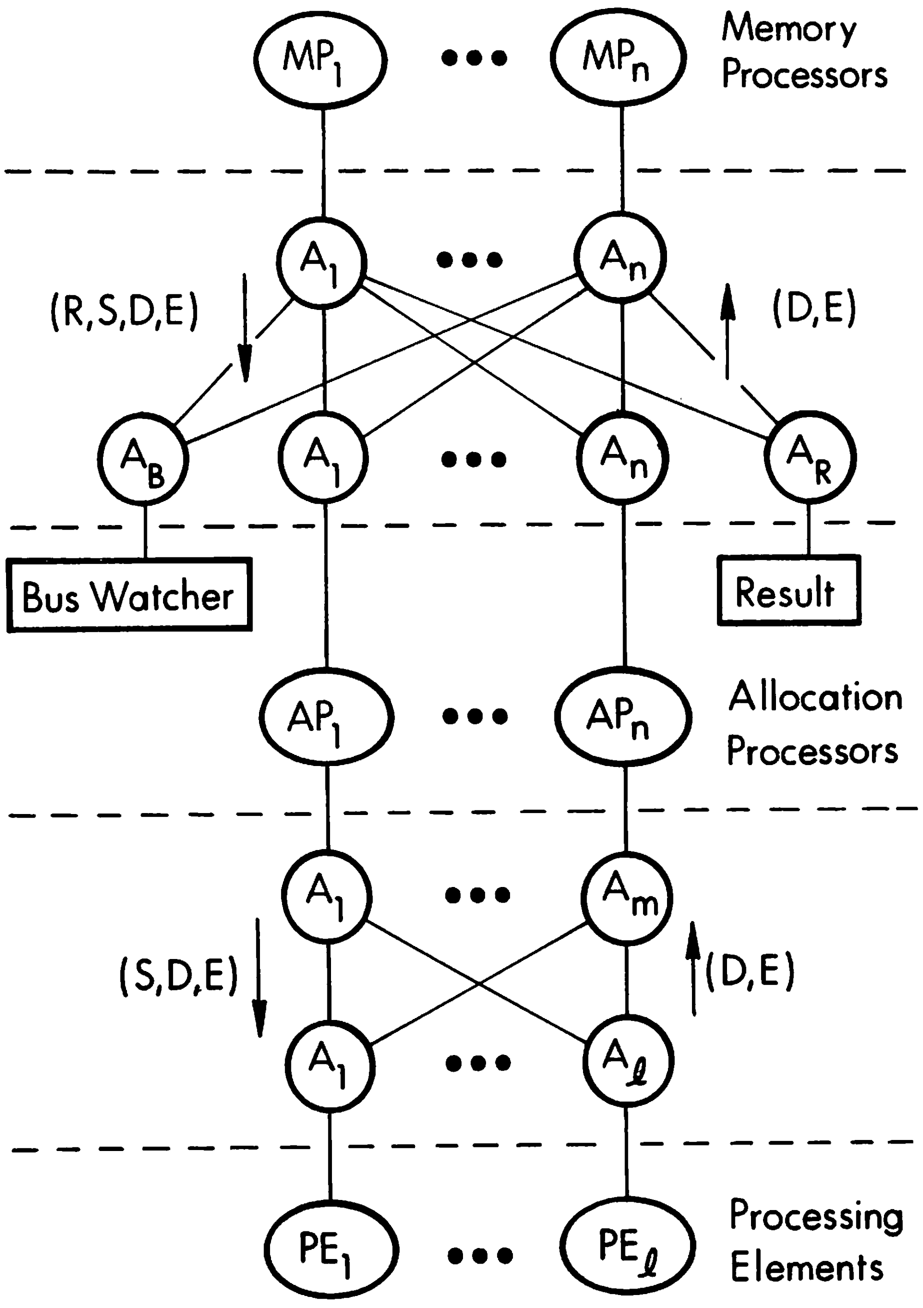


FIGURE 9

THE CONCEPTUAL STRUCTURE OF THE EMULATOR

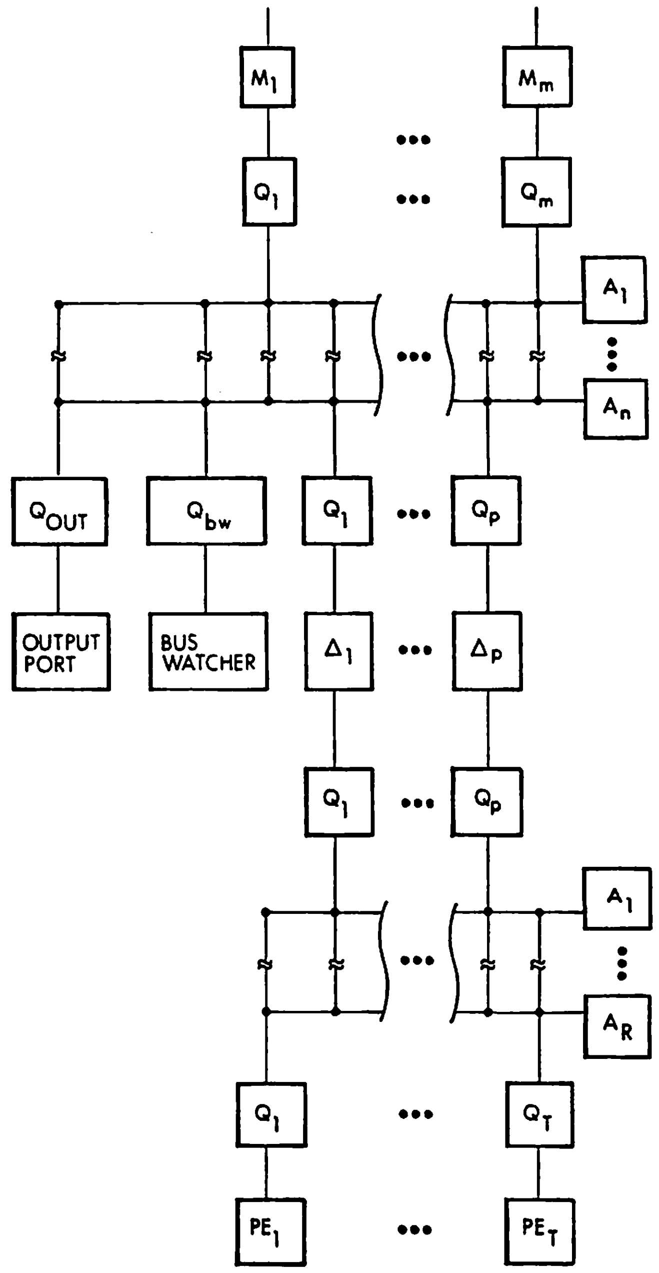


FIGURE 10

THE EMULATOR USING BUSSES FOR CONNECTIVITY

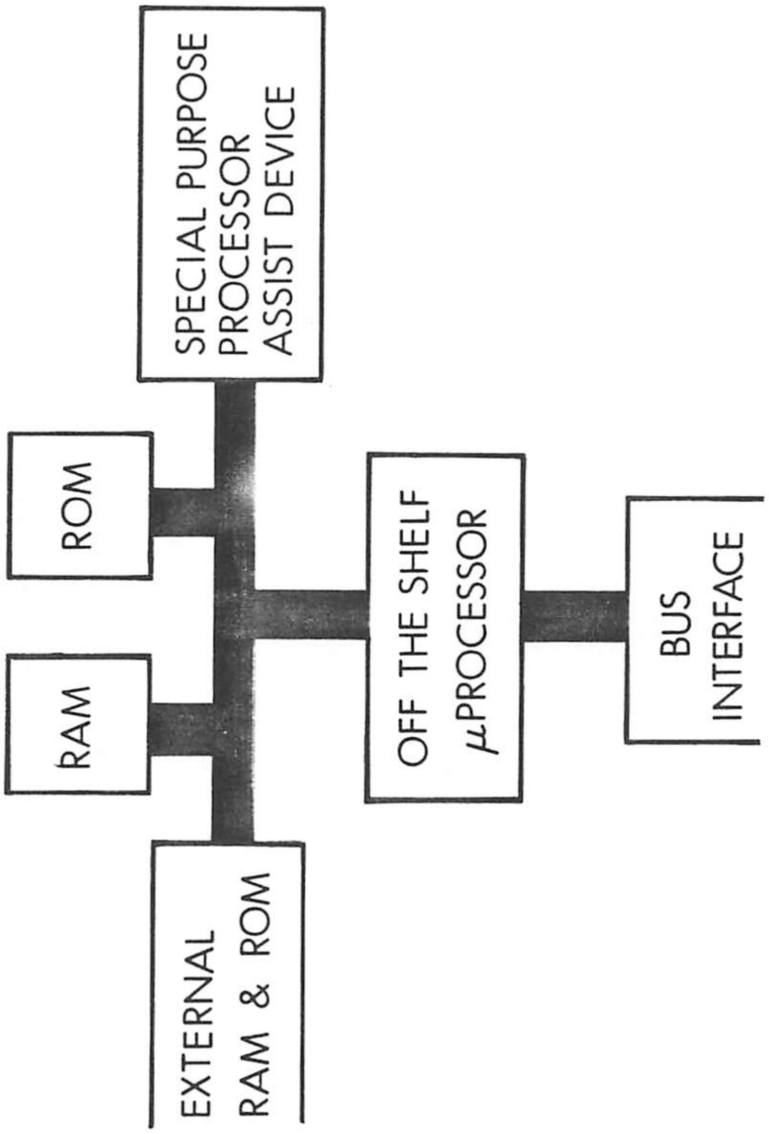


FIGURE 11

THE MEMORY PROCESSOR

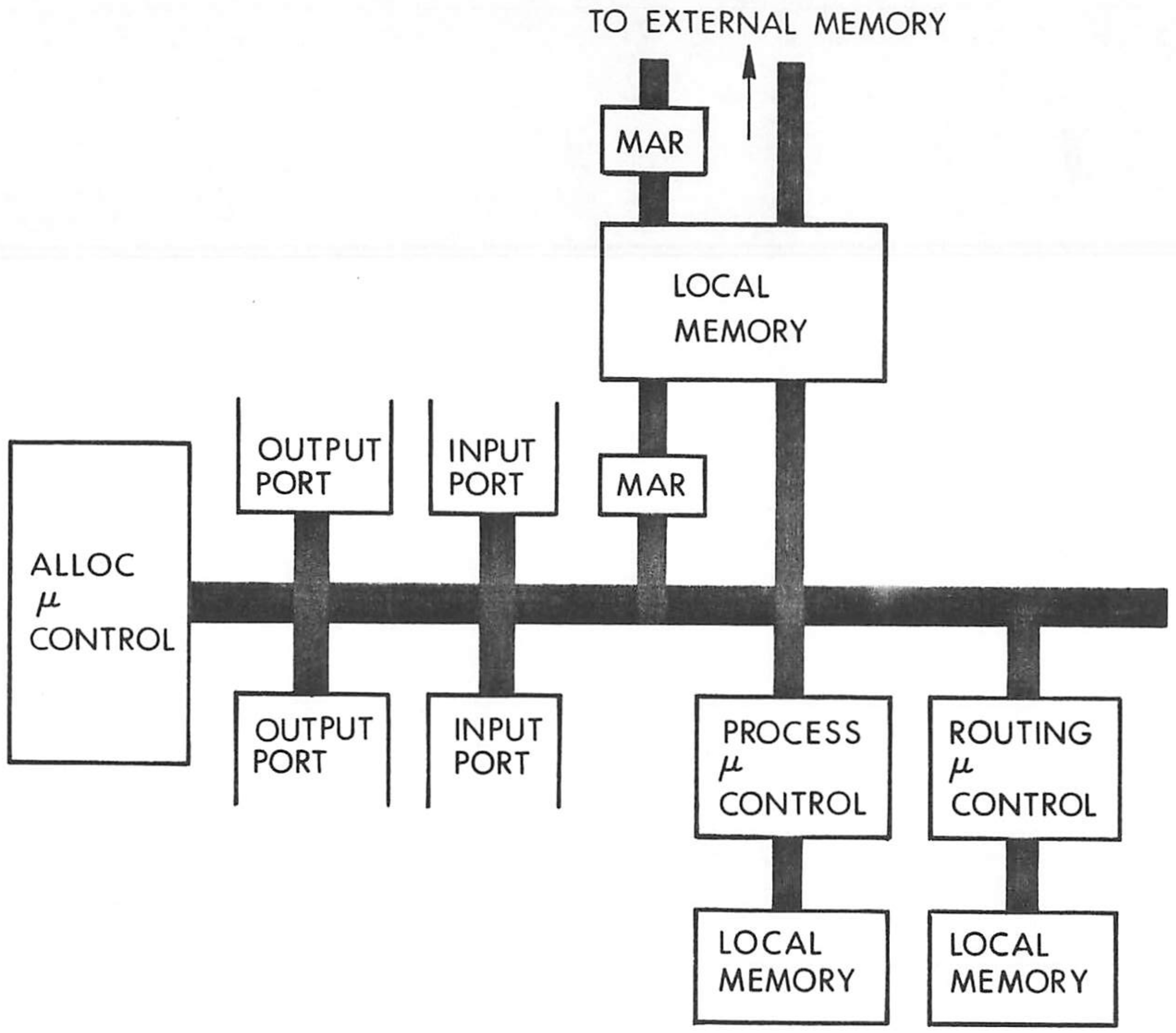


FIGURE 12

THE ALLOCATION PROCESSOR

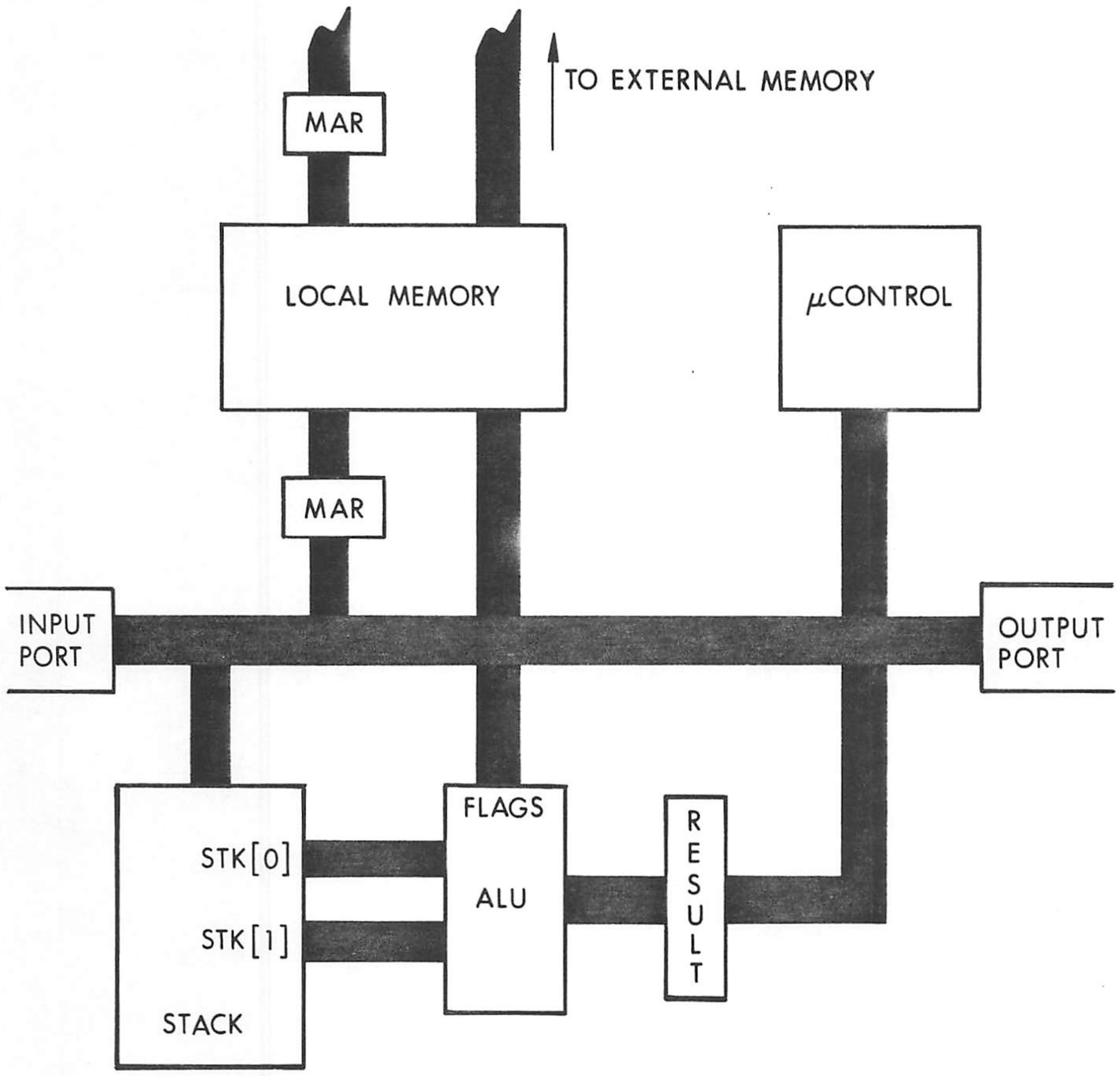
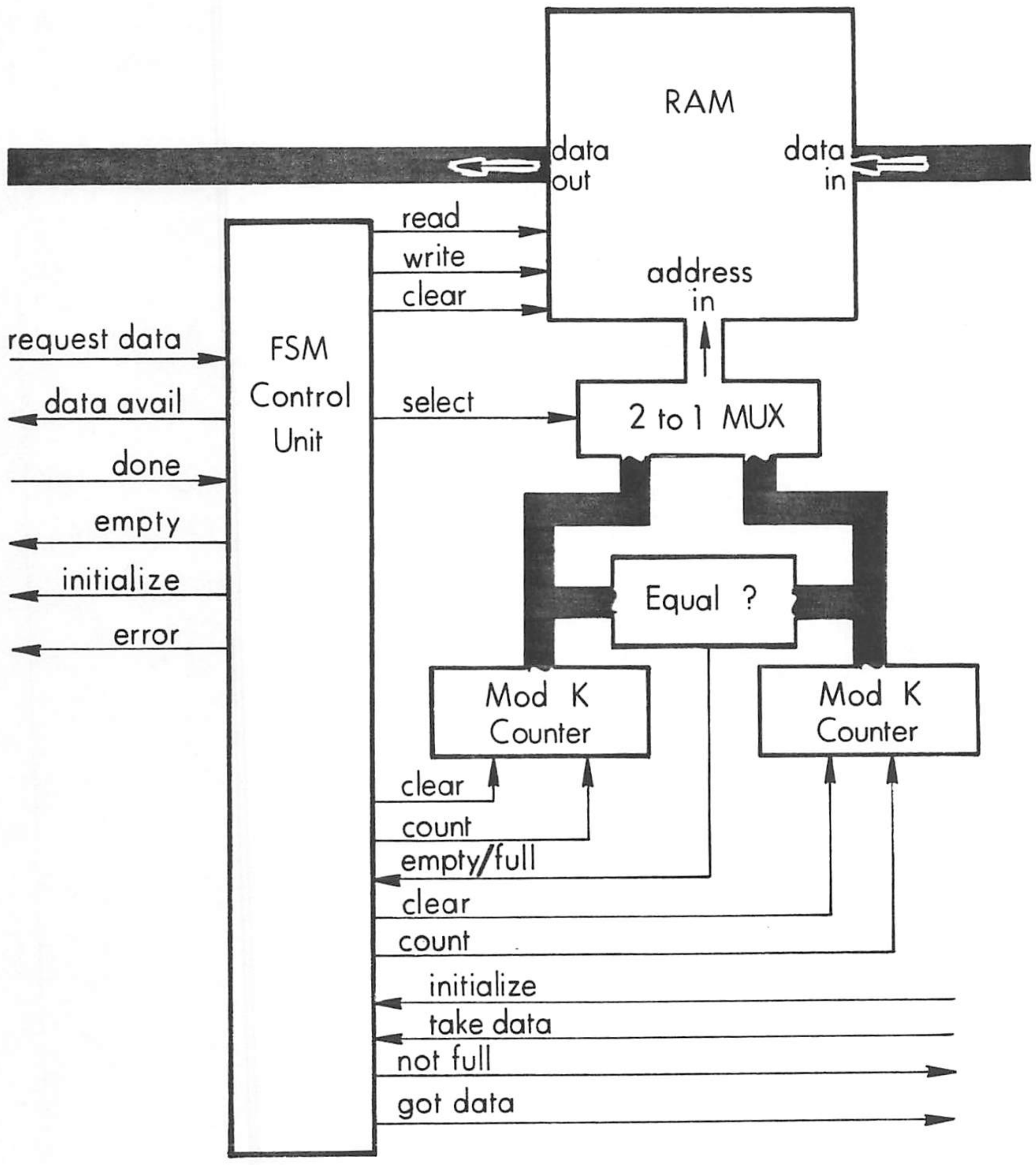


FIGURE 13

THE GENERIC PROCESSING ELEMENT



QUEUE MODULE — ROVING EMULATOR

FIGURE 14

THE QUEUE CHIP

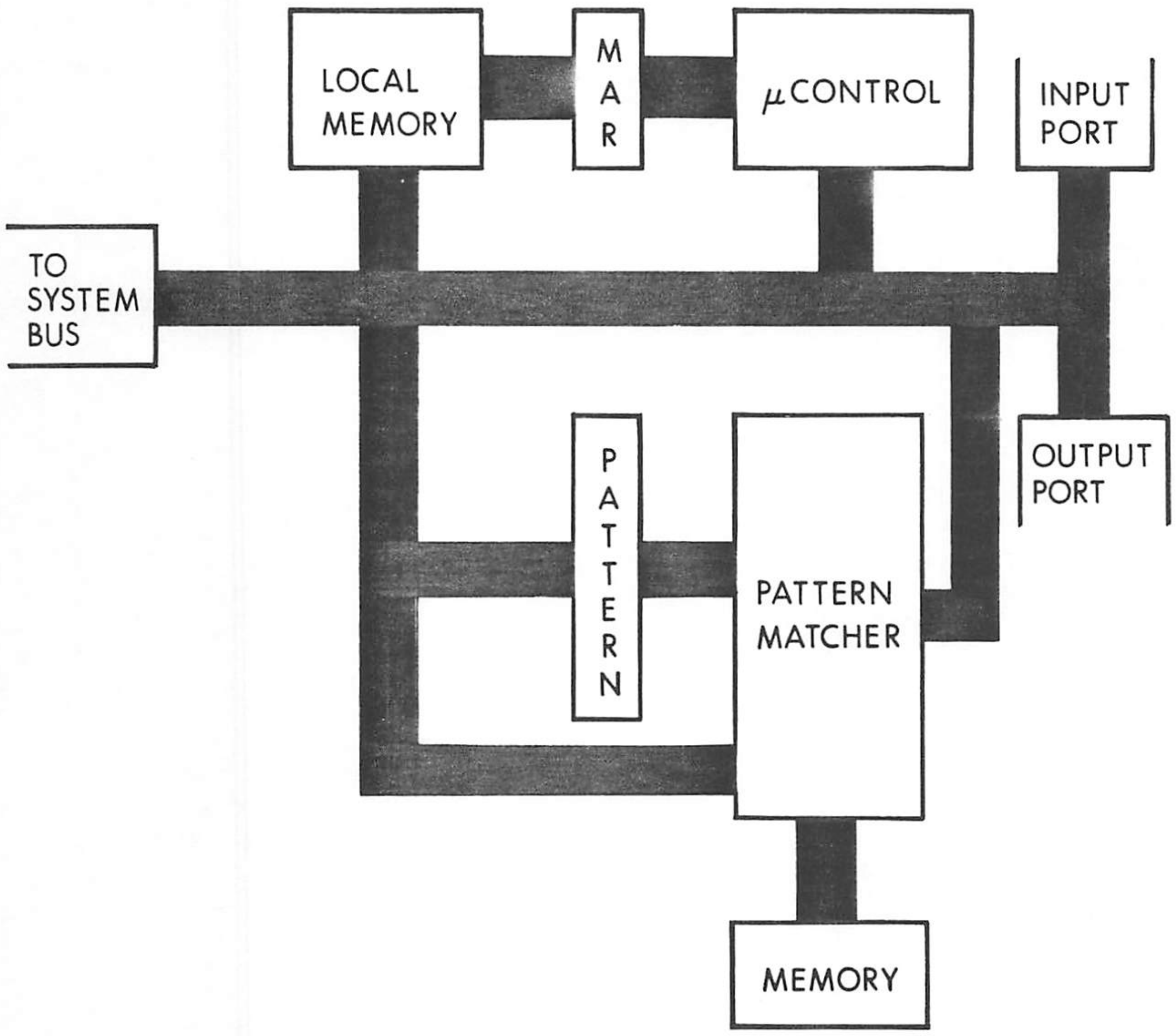


FIGURE 15

A GENERIC BUS WATCHER

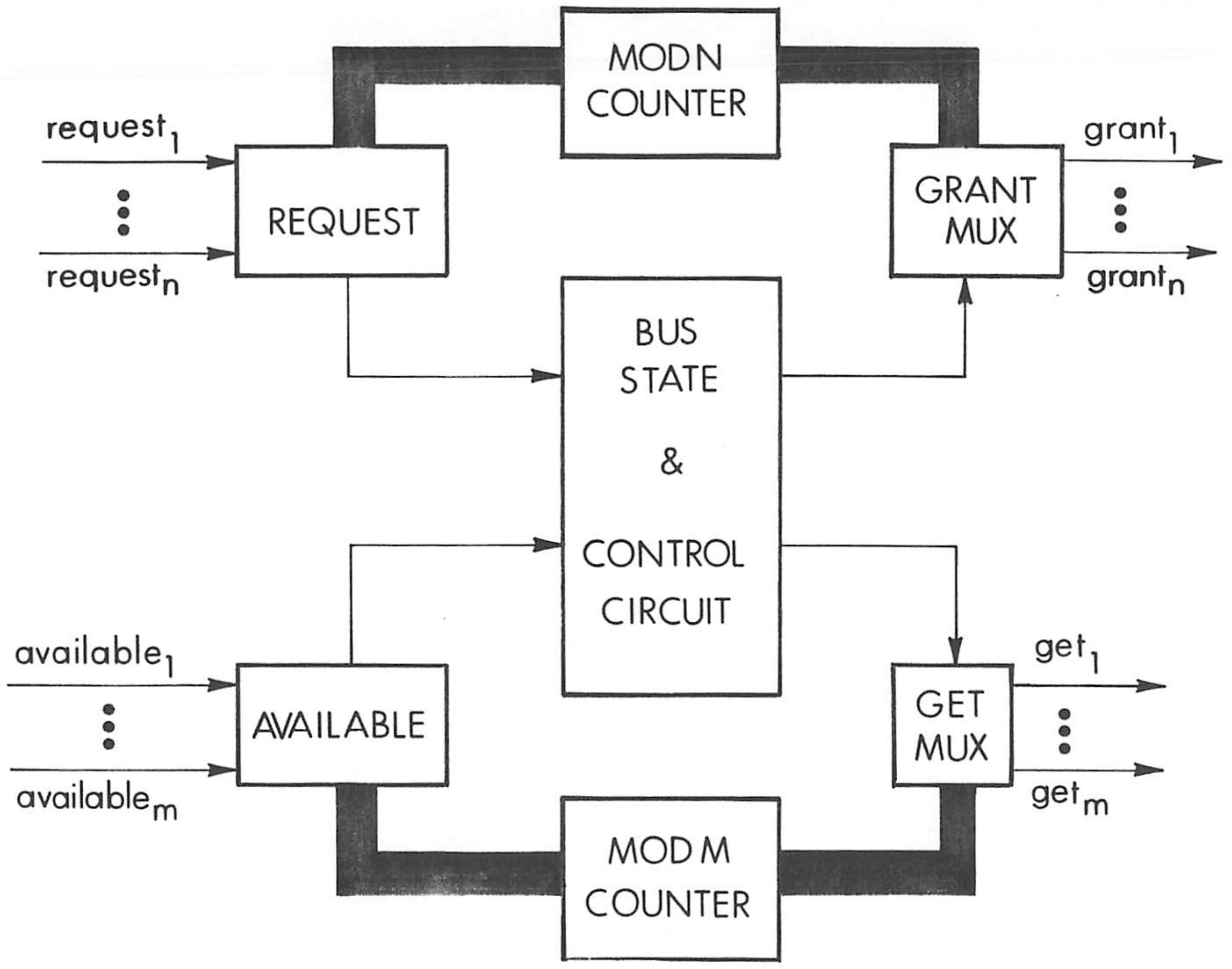


FIGURE 16

A BUS ARBITER

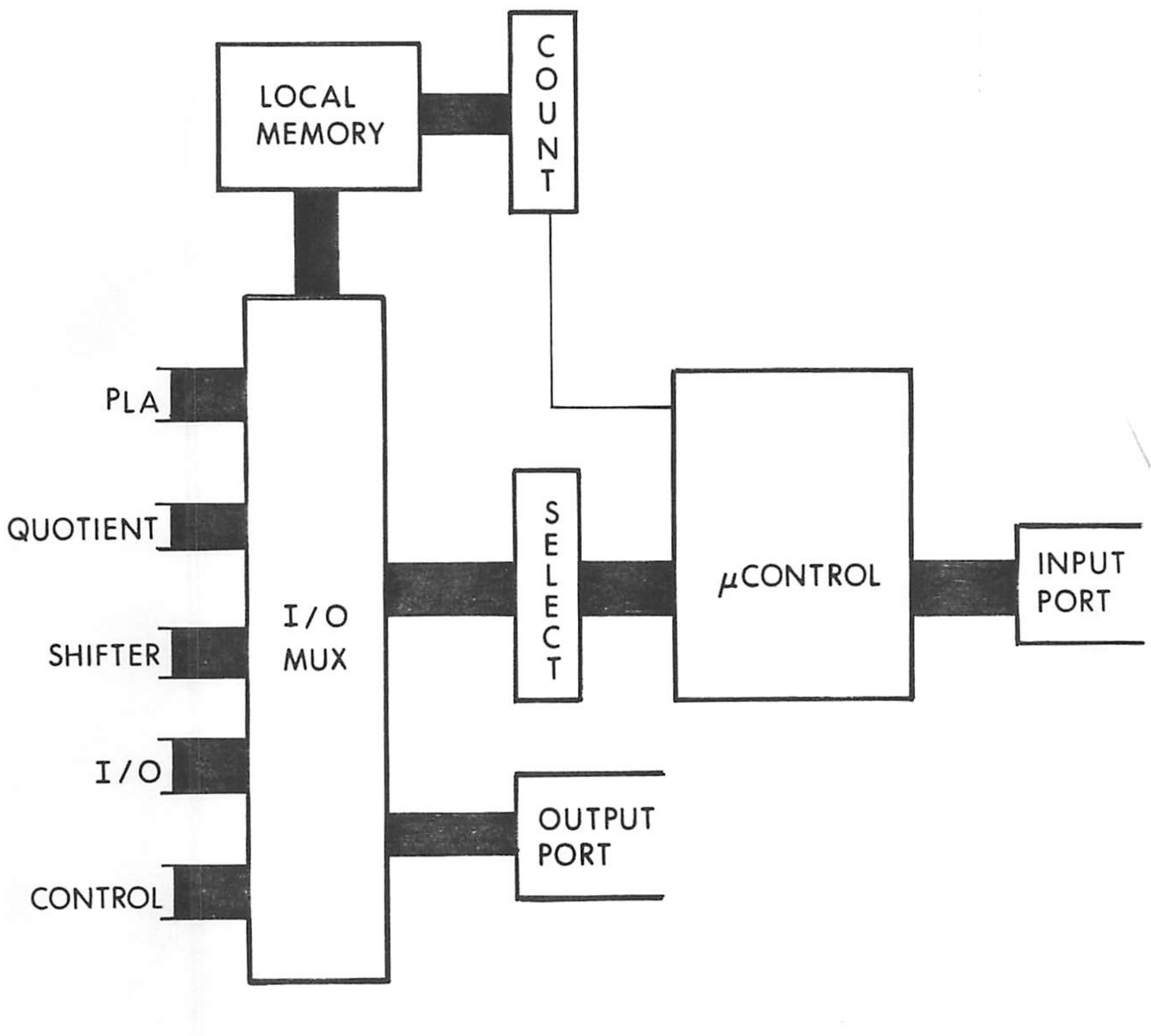


FIGURE II.1

A CUSTOM BUS WATCHER FOR THE ENCODER