LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

A Critique of Multiprocessing von Neumann Style

Computation Structures Group Memo 226
6 April 1983

Arvind

Robert A. Iannucci

to be presented at the 10th International Symposium
on Computer Architecture
Stockholm, Sweden
June 14-17, 1983

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0661. The second author is supported by the International Business Machines Corporation.

Source File = CRITIQUE.MSS.69, Last updated 6 April 1983 at 5:25pm

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Abstract

In recent years, there have been many attempts to construct multiple-processor computer systems. The majority of these systems are based on von Neumann style uniprocessors. To exploit the parallelism in algorithms, any high performance multiprocessor system must, however, address two very basic issues - the ability to tolerate long latencies for memory requests and the ability to achieve unconstrained, yet synchronized, access to shared data. In this paper, we define these two problems, and examine the ways in which they are addressed by some of the current and past von Neumann multiprocessor projects. We then proceed to hypothesize that the problems *cannot* be solved in a von Neumann context. We offer the data flow model as one possible alternative, and we describe our research in this area.

Key words and phrases: caches, cache coherence, data flow, multiported memories, multiprocessors, packet communication, von Neumann architecture

Table of Contents

List of Figures

A Critique of Multiprocessing von Neumann Style

1. Fundamental Issues in Multiprocessor Architectures

In the early days of computing, the issue of constructing a multiple-processor computing system was viewed as little more than an interesting intellectual exercise; after all, it seemed clear that machines could be made to operate faster simply by increasing the speed of the underlying technology. Given this view, it seemed that the style of machine organization for potential multiprocessing was not of overriding importance. The von Neumann organization, because of its sequential nature, was conceptually simple and easy to realize. Hence, it is not at all surprising that an entire community (and industry) was born with a built-in bias towards sequential computing. While understandable, this tacit assumption about machine organization has inherent limits which, form our present vantage point, sit just beyond the horizon.

Attention has been focused in the recent past on constructing multiprocessor systems [19, 20, 21, 22, 23, 11] - attention derived from a desire for more performance, greater fault tolerance, the ability to exploit the rather curious economics of a single-chip computer, or whatever. What has *not* been done sufficiently is a re-evaluation of the assumptions that led us to where we are now. This is painfully clear when one observes that von Neumann style uniprocessors still form the building block for the majority of multiprocessor projects or proposals. Many variations on the von Neumann theme have been explored (*e.g.*, pipelining, multiple functional units, vector instructions), but not much has been done with the sequential control required for instruction execution. There are good reasons to believe that this re-evaluation is overdue.

In this paper, we will use the term *multiprocessor* to refer to a computing system that exploits parallelism in programs through replication of resources. Although we will examine some special purpose machines, our primary interest here is in *general purpose parallel computers*, *i.e.*, computers that can exploit parallelism, when present, in any algorithm. Further, we are interested in the property of *scalability* - that the system can be made incrementally more powerful by adding incrementally more hardware resources (and without reprogramming). The basic building blocks of a multiprocessor are

- **Processing elements:** These perform arithmetic and logical operations, and are provided with an interface to communication and memory elements;

- **Communication elements:** These are used to construct an interconnection network. The resulting network has a number of *ports*, each with a bounded *bandwidth*. The purpose of the network is to interconnect various processing elements with each other and/or with memory elements; and

- **Memory elements:** Collectively, these provide the repository for program and data storage. Processing elements can, either directly or indirectly, access the data in the various memory elements via the network.

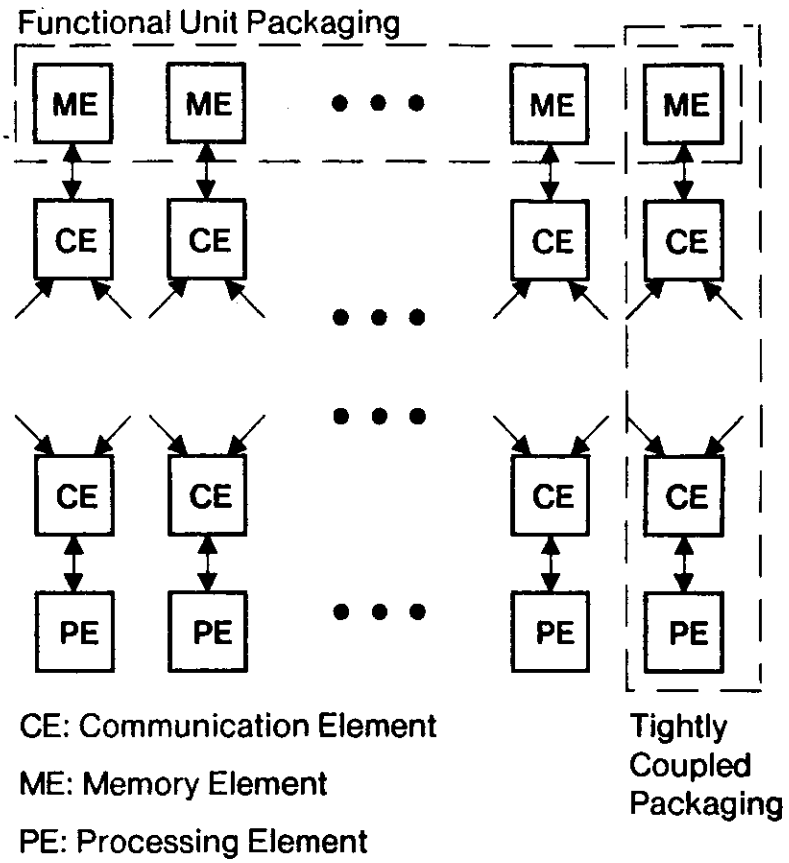

Figure 1-1: An Abstract Multiprocessor

Scaling of the multiprocessor involves incremental additions to all of the above. Modularity, an important design characteristic, allows scaling without redesign, *i.e.*, modules are constructed for each of the above, and an arbitrarily large machine (within the physical limits of packaging) is built by simply plugging these together. See Figure 1-1.

Note that we have said nothing about the programmer's view of the system (*e.g.*, explicit vs. implicit specification of parallelism and communication), the programming methodology, the network topology, packaging, etc. For our purposes here, we are concerned with the shared memory multiprocessor model, and we will consider tightly coupled memory/processor pairs which use a message passing methodology as shared memory machines. An equivalence between the shared memory model and the message passing model based on storage for messages, timing, and synchronization can be established.

1.1. The Issues

Let us review for a moment some of the issues which seem fundamental, at least from an engineering point of view, in the construction of any multiprocessor - von Neumann or not.

Issue 1: Ability to Tolerate Memory Latency

Latency is the time between issuing a memory request and getting a response. As the number of ports in the network increases, it stands to reason that the average latency of a memory request will have to increase if only due to the switching time in the network. Given the rate at which a processing element can issue memory requests and the bandwidth of each memory element (bits per second per port), it is clear that at some point, the memory units, on the average, will not be able to respond to each processor request without causing the processor to idle.

The key observation, then, is that it is absolutely necessary that each processor be able to issue multiple memory requests in succession without intervening memory responses. It is also quite reasonable to assume that, because of contention, the memory responses may arrive out of order unless some constraints are placed on the memories and network interconnections. The problem is unsolvable for a truly scalable multiprocessor unless some assumptions about program behavior are made. The kind of assumptions we have in mind are that the distribution of memory references with respect to the distance from the issuing processor does not get worse as the number of processors and memory elements increases.

Traditional ways of circumventing the latency problem attempt to exploit spatial locality. Instruction prefetching is the most successful technique: the inherent sequentiality of the von Neumann machine assures some measure of locality. Exploiting spatial locality in data references is more difficult. General register architectures subscribe to the notion that the user will somehow know what data will be referenced most frequently; however, it is well known that optimal register usage essentially amounts to solving the coloring problem on large graphs [16], and is very difficult to achieve.

A dynamic scheme for exploiting locality is the (demand) cache for main memory. This scheme is difficult to apply in a multiprocessor context due to the cache coherence problem. Censier and Feautrier [7] define the problem as follows: *"A memory scheme is coherent if the value returned on a* LOAD *instruction is always the value given by the latest* STORE *instruction with the same address."* In a multiprocessor context, it is easy to see that this may lead to difficulties. Suppose we have a two-processor system tightly coupled through a single main memory. Each processor has its own cache, to which it has exclusive access. Suppose further that two tasks are running, one on each processor; and we know that the tasks are designed to communicate through one or more shared memory cells. In the absence of caches, this scheme can be made to work. However, if it so happens that the shared address is present in both caches, the individual processors can read and write the address and *never* see any changes caused by the other processor. Using a store-through design instead of a store-in design does not completely solve the problem either. What is logically required is a mechanism which, upon the occurrence of a write to location x, invalidates all other cached copies of location x wherever they may occur, and guarantees that subsequent LOADs will get the most recent (cached) value. This can incur significant overhead and complexity.

Several approximate solutions for keeping cache coherence exist, but all such schemes inevitably introduce overhead and/or decrease parallelism (*e.g.*, due to lockout of page-sized areas of memory). Schemes have been proposed to explicitly interlock writing or to bypass the cache (and flush it if necessary) on a write; in either case, the complexity goes up and the performance goes down rapidly as the machine is scaled.

Another attempt at tolerating memory request latency is by performing context switching at a

very low level (analogous to task switching when an I/O request is made in a multiprogrammed operating system). Thus, while one computation waits for the memory to respond, the processor resumes another, parallel computation. Of course, the scheme works only if the context switching itself does not generate any memory references. This is done by duplicating programmer-visible registers (*i.e.*, the processor state). The problem here is again the increase in processor complexity as the machine is scaled.

Uniprocessors such as the Xerox Alto [24], the Xerox Dorado [15], and the Symbolics 3600 have used the technique of microcode-level context switching to allow sharing of the CPU resource by the I/O device adapters. This is very different than the type of context switching needed in multiprocessors. While both schemes rely on replication of hardware registers and other state information, the uniprocessor application requires only a fixed number of disjoint contexts - established by the number of I/O adapters plus the emulator itself. In the multiprocessor case, it will be necessary to have an unbounded number of tasks to achieve scalability. To illustrate this, consider the problem of expanding such a system. As memory elements are added, the depth of the communication network will grow. Hence, the number of low-level contexts to be maintained will also have to increase to match the increase in memory latency time.

Issue 2: Ability to Share Data without Constraining Parallelism

A much more troublesome issue is that of sharing data between two or more processes while maintaining proper synchronization. One manifestation of this is the *write-write race* - two processes attempt to write data into the same location. The meaningfulness of this is not at all clear, and the problem can be properly avoided by some software convention, perhaps assisted by some run-time checking. Functional languages avoid this problem totally because *updating* the value of a variable has no meaning in such languages.

The real problem, however, is not a pathological case like the write-write race: it is the real problem of the *read-before-write* race. To illustrate this problem, consider two routines running on two different processors, both accessing a two-dimensional array of numbers. One routine is creating the elements, in order, and writing them into the array. The other routine is waiting to read the elements. One possible way of avoiding a read-before-write race would be to allow the *entire* array to be written prior to allowing the consumer routine to begin processing. By this simpleminded transfer of control, there is no synchronization problem, but neither is there any chance for parallelism. On a single processor, the computation cannot be made faster by overlapping the production and consumption of a data structure, and this sort of scheme works well. It defeats the purpose of multiprocessing, however.

A more common scheme is to synchronize on a per-row or per-column basis of the array (as appropriate); this incurs more overhead, but constrains parallelism less. The extreme approach would be to synchronize the two routines on a per-element basis. It should be obvious that doing so is impractical with current methods and requires fundamental changes at the hardware level.

This example oversimplifies the real situation - consider the case where the elements are not produced in a regular (*i.e.*, row order or column order) way, or the case of a nonuniform data structure. The question remains: is it necessary to sacrifice parallelism for proper synchronization of reads-before-writes? We will show in section 2 that synchronization can be achieved with no loss of parallelism.

1.2. A Survey of von Neumann Multiprocessors

We examine a few of the planned or existing von Neumann machines that are described in the literature. Each one fails to satisfy at least one of the above two basic issues. We have ignored commercially-available 2 or 4 processor systems because, to the best of our knowledge, such systems have not been used in a manner in which processors cooperate on solving one problem. The main motivation for such systems has been higher performance in a multiprogrammed environment, and even in that limited area the performance of these systems has been far from impressive.

In our discussions, one figure of merit that is used in evaluating multiprocessor systems is ALU utilization/idle time. This assumption orients our discussion toward favoring machines designed for solving problems which can be expressed in numerical terms. Another important class of computation is the *symbolic* type in which the ALU utilization metric is less useful. One multiprocessor in this category is the Connection Machine. While it does not fit our model, we examine it in some detail because it is an interesting second-generation SIMD architecture.

1.2.1. C.mmp

C.mmp was a tightly-coupled multiprocessor built on a base of PDP-11 minicomputers connected into a single global memory through a high-speed crossbar switch [23]. The processors ran asynchronously, and could use local memory without interfering with the global memory. The switch speed was comparable to the speed of a local memory reference, but the cost of building a larger switch which maintains the same performance level grows at least quadratically. This reliance on technology doesn't solve the memory latency problem; it merely circumvents it. However, the investigators were clear on the point that the machine was exploratory in nature, and felt that scalability was only a secondary goal.

Interprocessor communication was facilitated by the shared memory and a cross-processor interrupt scheme. Synchronization was performed at several levels due to the recognized need to keep the overhead of this operation very low. High-level semaphores were maintained by Hydra on data objects to properly handle synchronized sharing. It is clear that the performance cost of this relative to, say, an ALU operation is rather high unless some potential parallelism is traded away.

The original design called a local cache for each processor, but only one processor in the machine was ever fitted with one (and it was never used by Hydra). The reason is, quite simply, the cache coherence problem.

1.2.2. Cm*

Cm* differed from its predecessor in several ways; notably, that the machine used a kind of hierarchical network to interconnect a number of microprocessors, each with its own memory [20, 21]. One might have guessed that this physical locality when combined with spatial locality in programs and data would have resulted in less communication overhead; the idea was intuitively appealing. This hope manifested itself in the design of the communication strategy - any processor making a nonlocal memory reference would idle until the reference was completed. Because of the hierarchical structure, this meant that greater interprocessor distances translated into longer memory reference times and decreased processor utilization.

Packet switching instead of circuit switching was used in the communication network (another difference from C.mmp) to avoid processor deadlock. Kmap, the communications controller, was

actually a context-switching processor which could tolerate the long-latency remote memory references. Unfortunately, the processors (LSI-11s) could not perform similar low-level context switches during a remote reference. It would be interesting to speculate on the behavior of Cm* if micro-tasking processors had been used.

In fact, Cm* demonstrated quite clearly the importance of Issue 1; the effect of processor idle time put an upper limit on the number of processors that could cooperate on even highly parallel programs (*e.g.*, chaotic relaxation) [9]. As far as Issue 2 was concerned, synchronization was based on a message system, but the underlying mechanism was the shared-memory, lockable segment.

1.2.3. The NYU Ultracomputer

Another shared memory multiprocessor is the NYU Ultracomputer [11]. Their solution to the synchronization problem is the atomic FETCH-AND-ADD instruction (sometimes called REPLACE-ADD). The instruction format is FETCH-AND-ADD(*address*,*value*), and works as follows: suppose two processors, i and j, simultaneously execute FETCH-AND-ADD instructions with arguments (A,v_i) and (A,v_j) respectively. After one instruction cycle, the contents of A will become $(A)+v_i+v_j$. Processors i and j will receive, respectively, either (A) and $(A)+v_i$, or $(A)+v_j$ and (A) as results. Indeterminacy is a direct consequence of the race to update memory cell A. The implementation of FETCH-AND-ADD calls for a synchronous packet communication network which connects n processors to an n-port memory. If two packets collide, say FETCH-AND-ADD(A,x) and FETCH-AND-ADD(A,y), the switch extracts the values x and y, forms a new packet (FETCH-AND-ADD($A,x+y$)), forwards it to the memory, and stores the value of x temporarily. When the memory returns the old value of location A, the switch returns two values ((A) and $(A)+x$). Hence, one memory reference may involve as many as $\log_2 n$ additions, and implies substantial hardware complexity.

The issue of processor latency has not been specifically addressed, and there are serious questions in our minds as to whether such a network ($t_{switch} \ll t_{memory}$) is realizable. We also do not understand the implications of basing a whole programming model on such an instruction as FETCH-AND-ADD.

1.2.4. VLIW Architectures - ELI-512 and the Polycyclic Processor

Another approach which seems to be coming into vogue these days is the horizontally-microprogrammed processor in which a smart compiler (or a patient and talented human) is able to fold many parallel operations into a single machine cycle. Examples of this are the ELI-512 [10], the ESL Polycyclic processor [17], the AP-120B [8], and numerous programmable signal processing machines.

While these machines are able to resolve run-time sharing conflicts (by moving them to compile time) and are usually able to plan memory references and control transfers in advance of the need (*e.g.*, the delayed jump), these machines suffer from their special-purpose nature. Except in the simplest of cases, compilers require "hints" from the programmer or, in some cases, rely on luck in doing the code generation. Clearly, these machines are not suited at all to real-time multiuser multiprogramming, interrupt handling, or anything which relies on the ability to efficiently switch contexts.

We believe that this technique is effective in its currently-realized context - special purpose

computation with small scale (4 to 8) parallelism, but the technique is not sufficiently general as to allow significant scaling up.

1.2.5. SIMD Revisited: the Connection Machine

Recently, there have been proposals to revive and to improve upon SIMD architectures (*e.g.*, Illiac IV [6]). The Connection Machine [13] proposal can be viewed in this context although it is intended for an entirely different class of problems than Illiac IV. It generalizes the communications scheme of Illiac IV, but still uses a single instruction stream with each processor having the capability of not participating in an instruction.

Illiac IV had 64 fairly powerful processors connected in an 8×8 rectangular, end-around grid topology. Each processor could directly access only its local memory (2K words). Using communication registers (one per processor) data could be shifted in one step to any of the four neighboring processors. Thus, in seven steps, a processor could access data from any other processor. However, because a single instruction controlled all processors, every processor had to wait even if one processor needed data from nonlocal memory. Also, if one processor wanted to transmit (shift) data to the processor to its east and another to its west, two machine instructions had to be executed. The need for such communication was poorly understood at that time, and Illiac IV executed a very small subset of scientific problems efficiently.

The Connection Machine proposal envisions a million processors, each consisting of 12 32-bit registers, some flag bits, and one 1-bit ALU. The processors[1] are connected in an Illiac IV-like grid with similar restrictions. However, groups of 64 processors are also connected by a 14 dimensional hypercube - there are 2^{14} groups. The bit-serial communication through the hypercube links is packet oriented; each processor can transmit or receive a message to/from any other processor. In the absence of conflicts, a message will reach its destination in at most 14 steps; but, because of conflicts, some messages will take significantly more steps than the required minimum number. A global flag is raised when all processors are done communicating, and only then can the next instruction begin.

It is clear that the speed of one bit ALU operations is irrelevant because it will be insignificant in comparison with the communication time - a processor will spend almost all (90%?, 99%?) of its time communicating. This machine, because of its single instruction sequence nature, also does not fit the model of the multiprocessor presented in this paper. Our model implicitly assumes that the goal of a general purpose parallel machine is to execute ALU operations (*e.g.*, addition, multiplication, comparison) as efficiently as possible by overlapping ALU operations with communications. However, for the class of applications in which communication predominates computation, our framework and metrics are not applicable. Many such situations occur in applied artificial intelligence programs where more time is spent exploring the connectivity of a large graph rather than computing with the information found at the nodes of the graph. The relevance of Issue 1 for the Connection Machine is not clear, and Issue 2 does not arise in a SIMD architecture.

---

[1] It is probably better to view these processors as cells of a "smart" memory.

2. Proposed Solutions: Data Flow Architectures

In this section we present a memory structure which allows efficient synchronization at the atomic level, and we also present an architecture which offers the ultimate in flexibility in issuing overlapped memory requests.

2.1. I-Structure Storage

If we associate with each memory cell in a machine special flags (called *presence* bits) which indicate the memory cell's status - written or unwritten - we have the ability to solve the read-before-write race problem as follows: assume that a memory module has just received a request to read a particular memory location and to forward the contents to instruction x. The memory module interrogates the presence bits associated with that location. If the bits indicate that the cell has already been written into, the contents are retrieved and forwarded to instruction x. If the bits indicate that the location is empty, the memory module puts the read request aside, and marks the empty location to indicate that a read request is outstanding.[2]

Now, when a write request for that location arrives at some time in the future, the memory module notices the pending read request, and forwards the newly-arrived datum to instruction x (as well as writing it into memory and setting the presence bits accordingly). Note that the memory module must maintain a list of deferred read requests (see Figure 2-1) as there may be more than one *read* of a particular address before the corresponding *write*. We call this type of memory *I-Structure Storage*. The issues involved with building such a memory, and the design for an I-Structure memory controller are discussed extensively in [12].

This mechanism, when coupled with a processor which is able to issue multiple, overlapped memory requests and which can tolerate out-of-order responses, allows the uncoupling of memory latency from the performance of a multiprocessor. The penalty of such a scheme in terms of the demands placed on memory elements is not excessive. A read operation is as efficient as in a traditional memory. Write operations take twice as long, however, due to the prefetching of presence bits. Many different implementation strategies are possible which can largely eliminate this penalty.

2.2. Data Flow Processing Element

When one desires to build a machine capable of issuing multiple memory requests and of tolerating long latencies, the most troublesome aspect of von Neumann architecture is the built-in sequentiality (*viz.* the program counter). By eliminating the notion of control flow for program sequencing, we can circumvent this problem directly. One alternative to sequential control flow is *data flow*, where the execution of instructions is triggered solely by the availability of the operands. In order to explain the operation of a data flow processor, we must digress for a moment to discuss program and data representation.

---

[2] The idea of associating a status bit with each memory cell is not new - the Denelcor HEP multiprocessor [18] uses this idea to synchronize cooperating parallel processes which share registers and/or memory cells. Unsatisfiable requests result in a busy-waiting condition - *i.e.*, there is no such thing as a *deferred read* list.

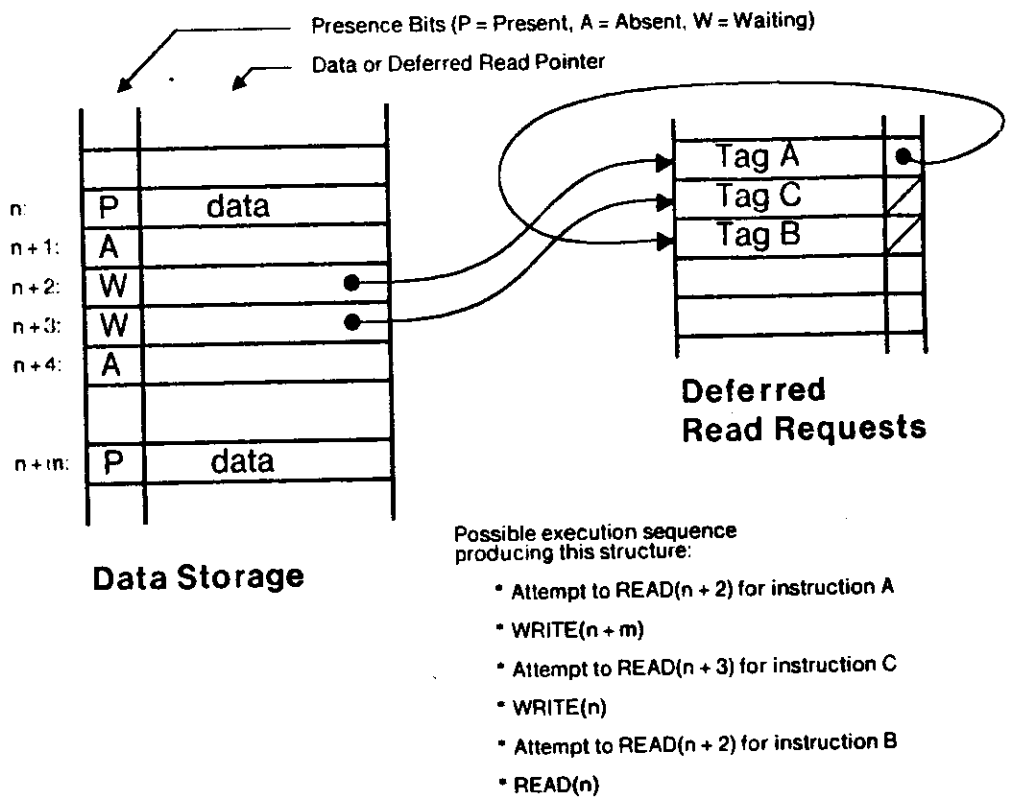

Figure 2-1: I-Structure Storage

2.2.1. Program Representation

Data flow compilers translate high-level programs into directed graphs; vertices in the graph correspond to machine instructions, and edges correspond to the data dependencies which exist between the instructions.

The implication is, quite simply, that instructions which depend on other instructions should be sequenced accordingly; but where no dependence (edge) exists, instructions can be executed in parallel. A simple example of this graphical translation is shown in Figure 2-2, compiled from the following ID program which integrates a function f from a to b over n intervals of size h by the trapezoidal rule:

```
(initial s ← (f(a)+f(b))/2;
        x ← a+h
 for i from 1 to n-1 do
    new x ← x+h;
    new s ← s+f(x)
 return s)*h
```

The graph shown is somewhat stylized; the box marked *f* represents the subgraph necessary for

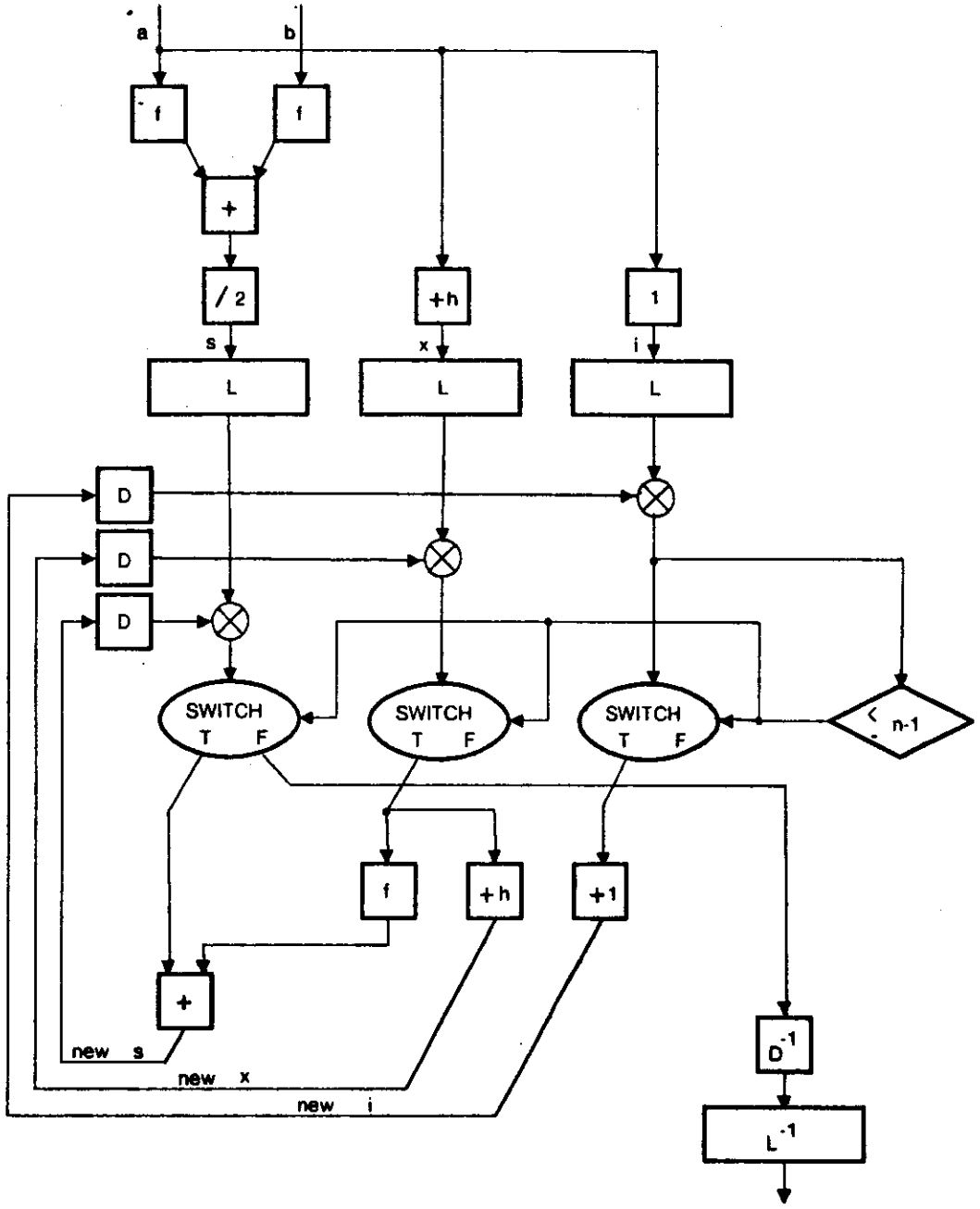

Figure 2-2: Compilation of the Loop Expression for the Trapezoidal Rule

invoking function f (which is, itself, a graph). Instructions D, D^{-1}, L, and L^{-1} are included to provide proper entry, iteration, and exit by manipulating context-identifying information (discussed in the next section). The remainder of the operators are arithmetic, relational, and conditional instructions whose function should be self-evident. The graph generated by the compiler is reentrant.

2.2.2. Data Representation

It is the processor's task to propagate data values through this graph, triggering instructions when the operands are available. Data values are carried on logical entities called *tokens*; a token contains not only a data value but also the name of the instruction to which it belongs. Conceptually, tokens move about on the vertices of the graph. Instructions are enabled for execution when tokens are present on all input vertices. Upon execution, the instruction absorbs the input tokens, and produces an output token for the next instruction in the graph. A program is said to *terminate* when no enabled instructions are left.

Our execution model allows more than one token to be present on an arc; and, therefore, the next-instruction label also contains some dynamic, or context-sensitive information. In their full generality, these next-instruction labels or *activity names* contain four parts:

- u: The *context* field, which uniquely identifies the context in which a code block is invoked. The context itself is specified by an activity name, thus making the definition recursive.
- c: The *code block* name. Each procedure and each loop has a unique code block name.
- s: The *statement* (instruction) number within the code block.
- i: The *initiation* number, which identifies the loop iteration in which this activity occurs. This field is 1 if the activity occurs outside a loop.

Activity names, then, define an unbounded namespace. Names in this space are mapped dynamically into a finite namespace. The activity name plus some mapping information uniquely define the runtime *tag* and processing element (PE) number.

Since instructions may have more than one input operand, we also include two more pieces of information on each token: the *total* number of operands required by its target instruction (called *nt* - number of tokens), and an index value (called the *port*) which specifies the operand number associated with this token. Tokens of this type are called *normal* tokens, abbreviated as $d=0$.[3] A more complete discussion of formats is given in [3]. The complete token, then, looks like this:

<d=0,PE,tag,nt,port,data>

2.2.3. Structure of the Processing Element

Figure 2-3 is a block diagram of an abstract data flow machine and its processing element. Assume that the program to be executed has been compiled into a directed graph, and an encoding of this graph is stored in the *program memory*. As tokens arrive at the machine's input, they are classified according to type. The d=0 tokens (above) which require partners ($nt \geq 2$) are routed to the the *waiting - matching* section.

Since each token carries the name of its target instruction, we can match up related tokens (*e.g.*,

[3] The reason for this notation has been lost in historical obscurity.

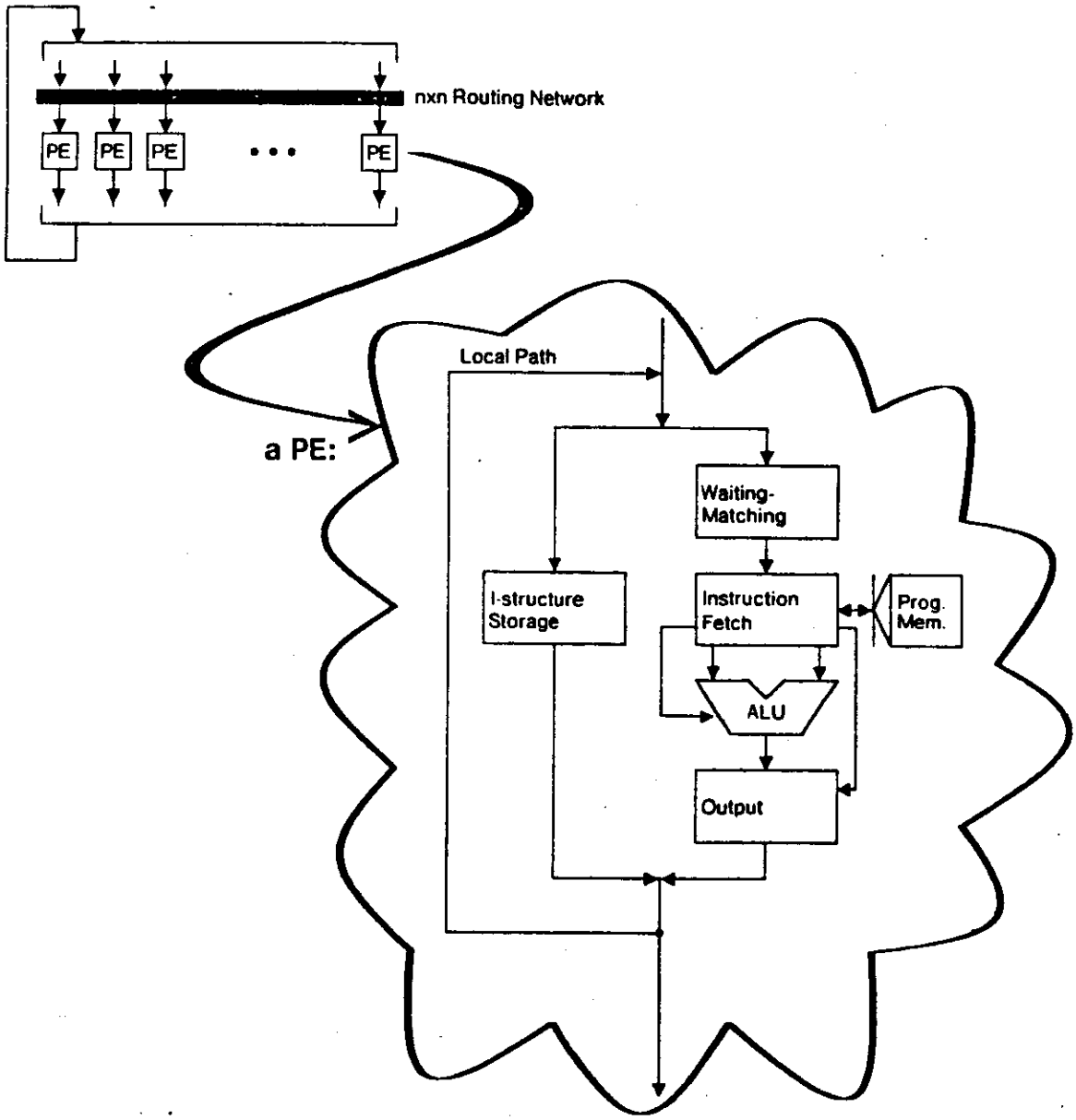

Figure 2-3: Organization of the Tagged-Token Data Flow Machine

the two input operands for an addition) by comparing the tags that they carry. This is the function of the waiting - matching section. When a match is found, the pair is passed on to the *instruction fetch* unit. When a match is expected but not found, the token remains in the waiting - matching unit's associative memory until its partner arrives. The instruction fetch unit also directly receives $d = 0$ tokens which require no partners ($nt = 1$).

The instruction fetch unit looks up the operation code and other information associated with the

token-carried names, and passes this enabled instruction on to the *ALU*. At this point, no other information is needed to carry out the operation save that which is in this enabled instruction packet.

The ALU output represents a datum which is ready to move off to its target instruction; but first, it has to be put in a token. We build this output token by computing a new tag, using the old tag along with information stored in the instruction itself. The *output* section handles these operations.

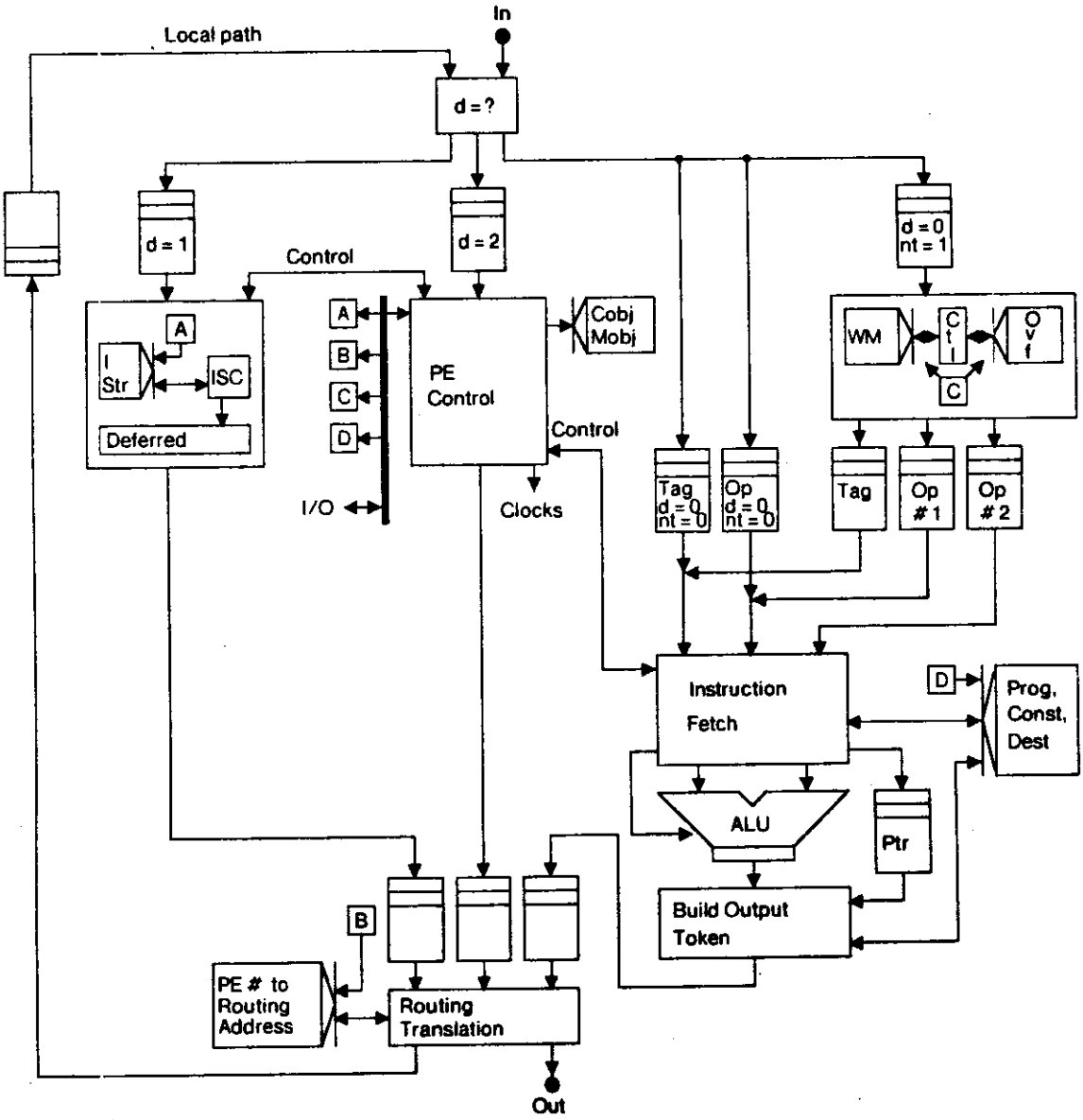

Figure 2-4: Data Flow Processing Element

The output section also computes the PE number for the new token. A routing translation table

turns this PE number into a network routing address. See Figure 2-4 for a more detailed view of the PE.

Other paths through the processing element provide for the cases where an incoming token is destined for the I-Structure Storage (d=1), or is destined for the PE Controller (d=2).

2.2.4. References to Data Structures

Conceptually, in a data flow graph, even a data structure is represented by a single token which is replicated on encountering a fork. The two most common operations on data structures are SELECT (to select a specific element of a data structure) and APPEND (to generate a new data structure which differs from the input structure in one selected position). In any reasonable implementation, however, data structures reside in some storage (I-structure storage in our machine), and tokens carry only pointers to the structure. Thus, a SELECT operation becomes a FETCH instruction while an APPEND operation becomes a STORE instruction.[4]

Now, to understand how I-structure storage references are processed consider an enabled FETCH instruction in the data flow graph. This instruction will be sent the ALU for processing. The ALU constructs a special token (d=1) which contains the address of the I-structure element to be read, the name of the PE on which this element resides, the FETCH opcode, and the name (tag and PE number) of the instruction which is to receive the result of the FETCH. The token is then forwarded to the proper I-structure controller, and the ALU proceeds to process the next enabled instruction.

This token makes its way through the network and is routed to the I-structure controller on the appropriate PE. The actual FETCH is performed as described previously, possibly after being deferred. When the FETCH operation is complete, a new token is formed to forward the fetched datum to the target instruction.

STORE operations are similar - a STORE instruction in the graph is executed, causing a data value and an address to be built into a d=1 token. The new token is forwarded to the appropriate PE which completes the operation by writing the data value, checking for and processing any deferred read requests, and setting the presence bits.

2.3. Review of the Issues

As we have seen, data flow provides a means whereby a processing element can issue many simultaneous memory requests, can tolerate long latencies (given that the program being executed is sufficiently parallel), and can deal with responses that arrive out of order. The mechanisms which make this work are

- **Tagged tokens:** By having each datum carry context-identifying information with it, no time-ordering ambiguities can arise.

- **Associative pairing:** Enabled instructions are detected by matching of these tagged

---

[4] There is a minor complication in that some APPENDs can cause a new copy of a data structure to be created. This is not germane to the main discussion here; the interested reader may refer to [5].

tokens.

- **I-structure storage:** By providing low-level synchronization bits per word of memory and by enforcing a discipline which defers unsatisfiable requests, data can be shared between producers and consumers with no performance overhead and with no loss of parallelism.

Many other issues are involved with the design of a tagged token data flow machine, and many other benefits accrue. The interested reader is directed to the literature [2, 5, 4, 1].

3. Construction of a Testbed

Many people find data flow an interesting and exciting research direction for a number of reasons. We feel that, because data flow addresses the fundamental problems that have haunted von Neumann multiprocessors, it offers a fresh perspective and the hope that we will be able to exploit the thousand-fold parallelism "grail" after which so many have sought.

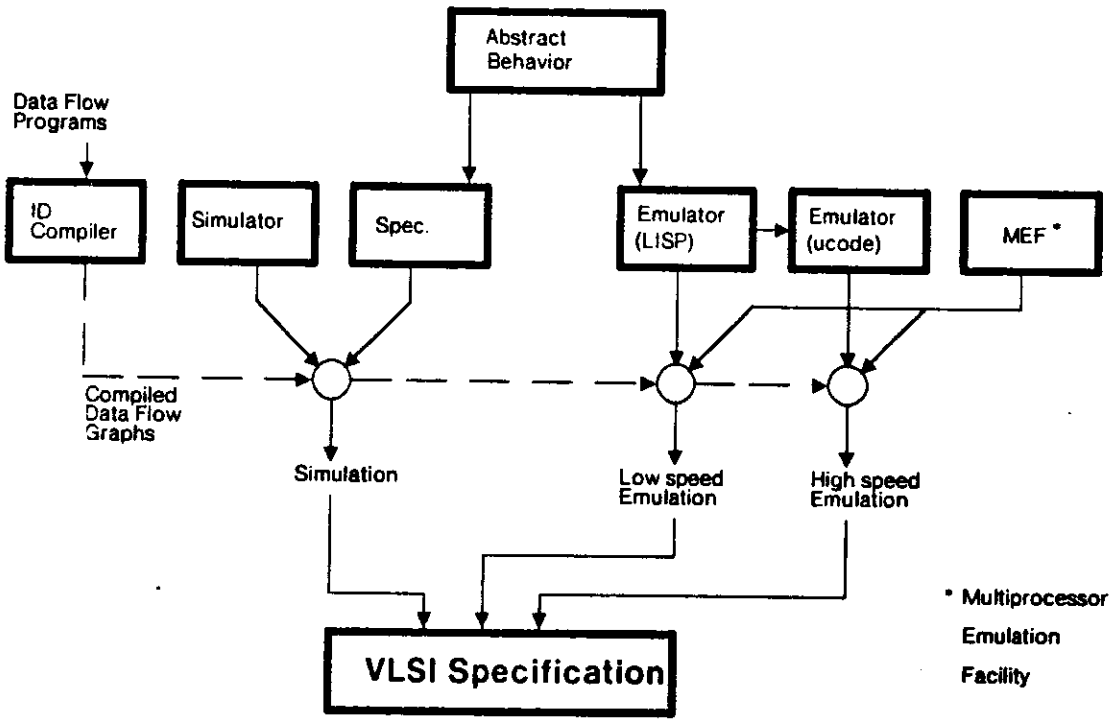

Figure 3-1: Development Plan for the Tagged-Token Machine

In our group, we are currently pursuing a two-pronged attack on the problem of constructing a practical prototype machine which embodies these principles (refer to Figure 3-1):

- **Simulation:** We have built a detailed simulation model of the tagged-token machine which interprets graphs produced by our compiler for the Irvine Dataflow (ID) language. The model accounts for communication as well as processing *simulated* time. We are installing an IBM 4341 MG2 processor which we will dedicate to the task of running these simulations of realistic data flow programs.

- **Emulation:** We are in the process of constructing a high-speed multiple processor emulation facility out of a set of 32 to 128 user-microprogrammable processors interconnected by a programmable, high-bandwidth, fault-tolerant packet communication network [14]. This emulator will also interpret the graphs generated by our compiler, but at much higher speeds. What is lost is the detailed internal timings of the abstract data flow machine; but what is gained is the ability to run very large application programs to learn about the behavior of programs in a multiple processor data flow environment.

  The key element to building the emulation facility is the packet communication network module, one of which is integrated with each microprogrammable processor. The network topology will be a seven dimensional hypercube with each connection implemented as a 4 megabyte per second bit-serial link. This topology was chosen for its flexibility. Each switch module also includes a routing table which allows the experimenter to specify any *emulated* topology which can be mapped onto the hypercube. The hardware has the capability of exploiting the redundancy in the hypercube network for message routing and for fault tolerance. Table-based routing also allows the facility to be statically partitioned into two or more smaller emulation machines.

  The idea of closely associating a switch module and a microprogrammable processor allows much of the low-level maintenance and fault recovery logic to reside in microcode. The hardware provides the essential fault-detection mechanisms, and is flexible enough to allow for simple error recovery under the control of a microcode task.

  It is our belief that this microcodable emulation facility will provide enough flexibility to actually study the effects of the two issues raised in this paper.

References

1. Arvind, and K. P. Gostelow. The U-Interpreter. *Computer* (February 1982).

2. Arvind, K. P. Gostelow, and W. Plouffe. An Asynchronous Programming Language and Computing Machine. Tech. Rep. 114a, Department of Information and Computer Science, University of California, Irvine, California, December, 1978.

3. Arvind, and R. A. Iannucci. Instruction Set Definition for a Tagged-Token Data Flow Machine. Memo 212, Computation Structures Group, Laboratory for Computer Science, MIT, Cambridge, Mass., December, 1981. revised February, 1983

4. Arvind, and V. Kathail. A Multiple Processor Dataflow Machine That Supports Generalized Procedures. The 8th Annual Symposium on Computer Architecture, May, 1981, pp. 291-302.

5. Arvind, and R. E. Thomas. I-Structures: An Efficient Data Type for Functional Languages. Tech. Rep. TM-178, Laboratory for Computer Science, MIT, Cambridge, Mass., September, 1980.

6. Bouknight, W. J., S. A. Denenberg, D. E. Mcintyre, J. M. Randall, A. H. Sameh, and D. L. Slotnick. The ILLIAC IV System. *Proc. of the IEEE 60*, 4 (April 1972).

7. Censier, L. M., and P. Feautrier. A New Solution to the Coherence Problems in Multicache Systems. *IEEE Transactions on Computers C-27*, 12 (December 1978), 1112-1118.

8. Charlesworth, A. E. An Approach to Scientific Array Processing: the Architectural Design of the AP-120B/FPS-164 Family. *Computer 14*, 9 (September 1981), 18-27.

9. Deminet, J. Experience with Multiprocessor Algorithms. *IEEE Transactions on Computers C-31*, 4 (April 1982), 278-288.

10. Fisher, J. A. Very Long Instruction Word Architectures and the ELI-512. Tech. Rep. 253, Yale University, Department of Computer Science, December, 1982.

11. Gottlieb. A., R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer. *IEEE Transactions on Computers C-32*, 2 (February 1983), 175-189.

12. Heller, S. K. An I-Structure Memory Controller. Master Th., Dept. of Electrical Engineering and Computer Science, MIT, Cambridge, Mass., June, 1983.

13. Hillis, W. D. The Connection Machine: Computer Architectue for the New Wave. Tech. Rep. 646, Artificial Intelligence Laboratory, MIT, Cambridge, Mass., September, 1981.

14. Iannucci, R. A. Packet Communication Switch for a Multiprocessor Computer Architecture Emulation Facility. Memo 220, Computation Structures Group, Laboratory for Computer Science, MIT, Cambridge, Mass., October, 1982.

15. Lampson, B. W., and K. A, Pier. A Processor for a High-Performance Personal Computer. Xerox Palo Alto Research Center, January, 1981.

16. Radin, G. The 801 Minicomputer. Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems, March, 1982. Same as Computer Architecture News 10,2 and SIGPLAN Notices 17,4

17. Rau, B., D. Glaeser, and E. Greenwalt. Architectural Support for the Efficient Generation of Code for Horizontal Architectures. Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems, March, 1982. Same as Computer Architecture News 10,2 and SIGPLAN Notices 17,4

18. Smith, B. J. A Pipelined, Shared Resource MIMD Computer. Proceedings of the 1978 International Conference on Parallel Processing, 1978, pp. 6-8.

19. Sullivan, H., and T. R. Bashkow. A Large Scale, Homogenous, Parallel Machine. The Fourth Annual Symposium on Computer Architecture, March, 1977.

20. Swan, R. J., S. H. Fuller, and D. P. Siewiorek. Cm* - A Modular Multiprocessor. Proceedings of the National Computer Conference, 1977.

21. Swan, R. J., A. Bechtolsheim, K-W. Lai, and and J. Ousterhout. The Implementation of the Cm* Multi-microprocessor. Proceedings of the National Computer Conference, 1977.

22. Widdoes, L. The S-1 Project: Developing High-Performance Digital Computers. COMPCON Spring '80, February, 1980, pp. 282-291.

23. Wulf, W. A., R. Levin, and S. P. Harbison. *Hydra/C.mmp: An Experimental Computer System.* McGraw-Hill, 1981.

24. *ALTO: A Personal Computer System - Hardware Manual.* Xerox Palo Alto Research Center, Palo Alto, California, 94304, 1979.