LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

The Tagged Token Dataflow Architecture

Preliminary version for distribution in subject 6.83s
Massachusetts Institute of Technology
8 August 1983

Arvind
David E. Culler
Robert A. Iannucci
Vinod Kathail
Keshav Pingali
Robert E. Thomas

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for this project 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 and in part through various grants from the International Business Machines Corporation.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

The Tagged Token Dataflow Architecture

Preliminary version for distribution in subject 6.83s
Massachusetts Institute of Technology
5 August 1983

Arvind
David E. Culler
Robert A. Iannucci
Vinod Kathail
Keshav Pingali
Robert E. Thomas

Abstract

A detailed account of the Tagged Token Dataflow Architecture is presented. In this paper, we describe the Tagged Token Dataflow Machine being constructed at M.I.T. by the Functional Languages and Architectures group. The first two sections discuss the nature of tags, and describe how programs are mapped on the machine. The third section describes the Processing Element and the instruction set in detail. We conclude with a description of the current status of the project and our goals for the future.

Key words and phrases: computer architecture, data flow, multiprocessors, resource management

The Tagged Token Dataflow Architecture

1. Architectural Strategy

The aim of this paper is to give a detailed account of the Tagged Token Dataflow Achitecture being developed by the Functional Languages and Architectures group at M.I.T. This project began as an effort to realize the U-interpreter model of execution in a multiprocessor setting; however, the result is an interesting multiprocessor system, quite independent of the U-interpreter. The gist of this project has been discussed extensively elsewhere [5, 6, 3], and the reader is presumed to have a basic understanding of the dataflow approach to parallel computation. For answers to the question "Why dataflow ?" and other such questions related to the tagged-token dataflow model of computation, the reader is referred to [6]. In this paper, we focus on the salient points of the architecture of the machine.

The Tagged Token Dataflow Architecture comprises a number of identical Processing Elements (PE's) which are connected together by an n-cube network. A high-level view of a PE is shown in Figure 1-1. It performs five basic functions, as listed below.

1. *Waiting-Matching section*: Detects when an instruction is enabled. Operations are restricted to having one or two inputs, thus if a token requires a partner it waits in the waiting-matching section until its partner arrives. Tokens that do not require partners bypass this stage.

2. *Instruction-Fetch section*: Fetches an instruction from program memory. We do not support virtual memory for the program store.

3. *ALU*: Executes the instruction. The ALU receives a stream of packets comprising operations and operands. It may be as powerful as the surrounding subsystems can support. Currently, it is not pipelined internally.

4. *Output section*: Generates the result tokens. It produces tokens with data from the ALU and tags generated from destination instruction pointers and parameters that specify how programs are mapped in the system.

5. *I-Structure storage*: Provides storage for data structures. It solves the read-before-write problem [6].

The PE can be thought of as an asynchronous pipeline in which given enough work, all sections will computing simutaneously.

Each of the modules in the PE can be internally pipelined, if necessary. It is hoped that such a system will be scalable upto a large number of PE's - *i.e.*, show a real increase in processing power as more PE's are added to the system.

From the view-point of the machine, a dataflow program is a set of *code-blocks*. A code-block is a collection of dataflow instructions which results from the compilation of an Id loop or a procedure. A code-block may have within it instructions that invoke other code-blocks. The machine is

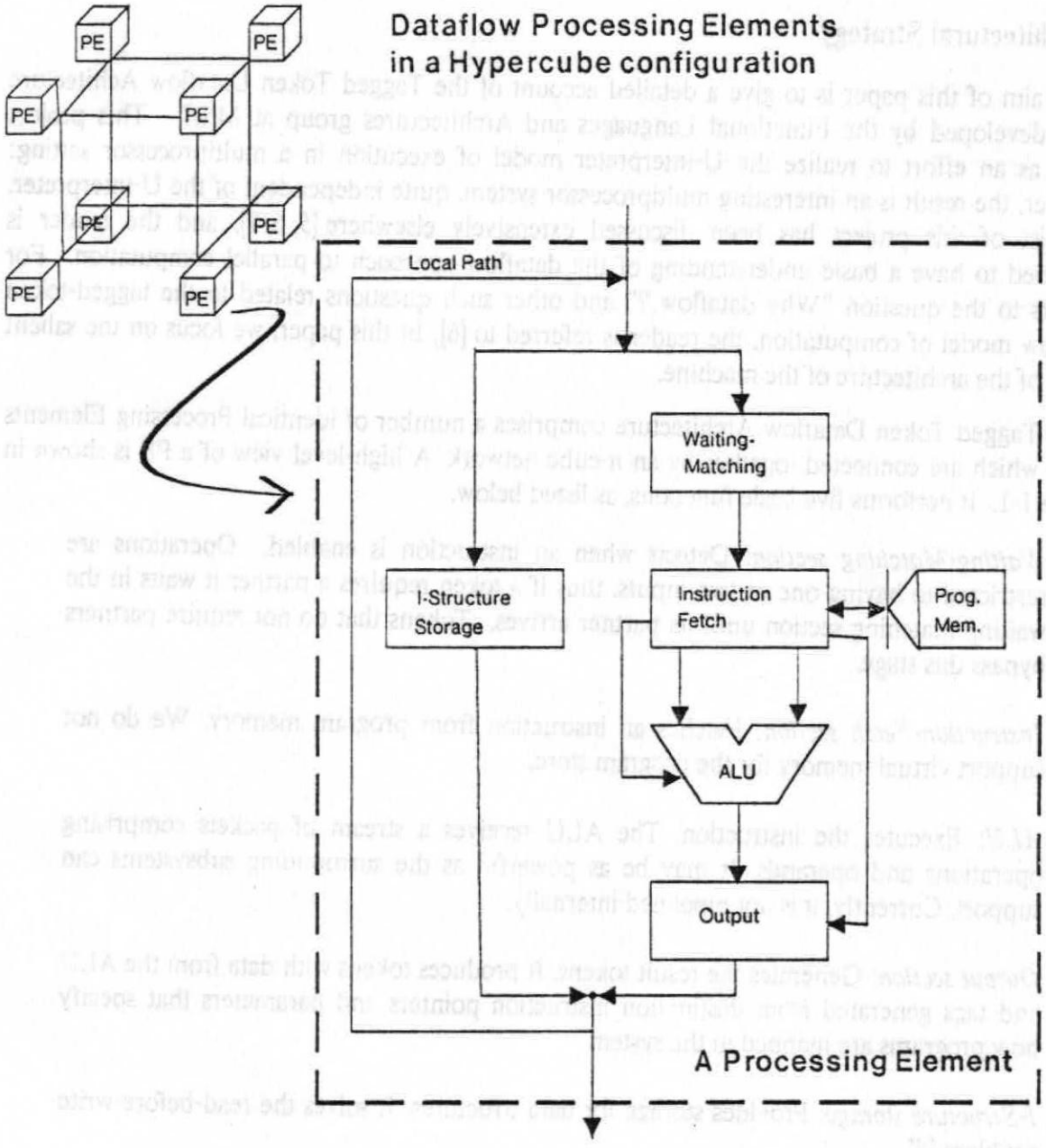

Figure 1-1: A Block Diagram of the Tagged Token Dataflow System

designed to support the dynamic invocation of code-blocks - *i.e.*, it is not necessary to load all code-blocks into physical memory before beginning the execution of the program. In order to permit the sharing of code by multiple invocations of a given code block, all tokens are *tagged* so as to keep tokens belonging to different instructions and different instantiations of an instruction separate. The precise nature of this tag will be explained in Section 2.

2. Tags and Resource Management

A *code-block* is a directed[1] graph of *instructions* resulting from the compilation of a loop or a procedure. Before a code-block can be *activated*, it must be loaded into the program memories of one or more PE's. Once the loading of a code-block has been accomplished, it is *activated* by placing tokens on its input arcs. The set of tokens generated by activating a code-block is termed a *code-block activation*. Tokens must specify their destination instruction. Once a code-block has been loaded, the tuple <PE#, address> identifies the destination instruction uniquely. Accordingly, we will require that all tokens carry the <PE#, address> of their destination instructions. Since a copy of the program graph may be shared by many different activations simultaneously, tokens must carry some additional information so that tokens from different activations are not confused. This can be done by associating a field called *hue* with every activation of a code block. By requiring that all tokens belonging to one code-block activation have the same hue and assigning different hues to different activations of the same copy of the code-block, we can ensure that tokens belonging to different code-block invocations are not confused. Therefore, in addition to a data value, every token must carry the 3-tuple <PE#, address, hue>. We will refer to this 3-tuple as the *tag* of a token.

If the hue field is very large (say 64 bits), then it is possible to support an arbitrary number of activations of a copy of a code-block. In that case, hues need not be reused. However, unless memory need never be reused, the termination of all activations of a copy of a code-block must be detected. Therefore, there are two options - very large address space and a large hue field, or limited address space and a restricted hue field. A large *physical* address space is prohibitively expensive, while a large *virtual* address space is complicated and may result in unacceptable delays. Hence, we have opted to keep the size of the hue field limited, and reuse both physical memory and hues. A natural view of tags is as points in the 3-dimensional address space shown below. Since program memory and hues are resources of the machine, it is convenient to think of them as being allocated for invocations of code-blocks. The management of these resources must be done by an agent that has a global view of the system resources. An alternative scenario is a group of agents that communicate with each other to perform allocations.

We now address the issue of how tags are generated for result tokens by an activity. In order to produce the tag of a result token, the activity must be able to generate the PE#, address and hue of the destination activity. Let us first consider the simple case of an activity which wants to send a result token to another activity that belongs to the same code-block invocation. Clearly, the hues of both the input tokens and the result tokens of this activity must be the same. In order for this activity to generate the PE# and address of the destination instruction, it must know how the code-block has been mapped in physical memory. This can be achieved by associating some *mapping information* with the code-block. The instruction itself contains the relative address within the code-block of the destination instruction. The mapping information together with the relative address of the destination instruction is sufficient to generate the PE# and address of the destination instruction. Thus, the tag of the result tokens can be generated from information

---

[1] For the moment this should be considered to be an acyclic graph. Once the notion of operators that directly manipulate the tag is introduced, it will make sense to allow cyclic graphs as long as every cycle includes one such special operator.

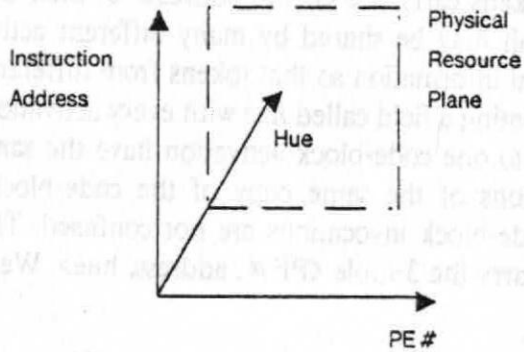

Figure 2-1: Tags as a 3-dimensional Address Space

available to the activity. In sections 2.2 and 2.3, we will explain how code-blocks are mapped on the machine. This will enable us to specify the precise nature of the mapping information that must be associated with each loading of the code-block.

The other case to be considered is when an activity wants to send a token to an activity which belongs to a *new* invocation of a (possibly different) code-block. This arises when a code-block invokes another code-block, or when a code-block is recursive. In such a case, resources must be allocated to the new invocation. For example, a copy of the invoked code-block may need to be loaded into memory, and a hue allocated for the invocation. Since the allocation of resources requires communication with a global agent, code-block invocation is, in general, fairly complicated and is described in detail in section 2.4.

A useful optimization in this case is to treat the case of self-recursive code-blocks in a special manner. Such code blocks arise from the compilation of Id loops and self-recursive procedures. Invocations of such code-blocks can be divided into *external* invocations (*i.e.*, when some other code block invokes it) and *internal* invocations (*i.e.*, when it invokes itself). Each internal invocation will result in the allocation of a new hue for that invocation[2] . To lower the amount of communication with the resource manager, it is possible to allocate a bunch of hues to each external invocation. A simple approach is to divide hue into two fields - *color* and *initiation-number*. Each external invocation of the code-block is given a color, and can use the full range of the initiation-number field for internal invocations. Allocation of the block of tags among the internal invocations is done by special operators that manipulate hue directly. By making such code-blocks cyclic with

[2]Of course, we may run out of hues, in which case a new copy of the code block will have to be loaded elsewhere in the machine

one such special operator in each cycle, the generation of tags for internal invocations can be done without communication with the resource manager. Of course, since the block of hues allocated to each external invocation of such a code-block is finite, it is possible to run out of hues. In such a case, there must, once again, be in communication with the resource manager.

3. Mapping of Activities

Clearly, to exploit parallelism it must be possible to distribute activities over the various PE's, and this distribution must be in concert with the structure of the program to be effective. Code-block invocations are the obvious unit of distribution. In this section we consider how the activities are distributed over the PE's.

3.1. Multiple PE's co-operating in a code-block invocation

The activities belonging to a code-block invocation are distributed over many PE's by spilitting and distributing the code. However, to generate tags automatically, given a small set of mapping parameters, we place constraints on how the program memory may be allocated.

Constraint 1: A code-block will be loaded across a set of one or more consecutively addressed PE, using the same region of program memory in each PE and a single Hue throughout.

With this constraint, allocation for a code-block invocation reduces to assigning a rectangular window across a sequence of PE. It will support as many simultaneous activations as there are Hues. If a code-block is split into m parts of length l and loaded onto m PE consecutively then a new tag can be generated with the four parameters m, l, PE_{base}, *Base-local-address*, as indicated in Figure 3-1.

Suppose an activity with Hue H must send a result to an instruction at S. The tag for the result token is given by:

PE#	$= PE_{base} + \lfloor S/l \rfloor$
INSTRUCTIONADDRESS	$=$ *Base-Local-Address* $+ S \bmod l$
HUE	$= H$

The port and number of tokens for enabling are copied from the instruction.

This is successful with respect to generating tags, but it complicates resource management. Such a window would be expensive to locate, and would tend to fragment memory. We must further restrict the usage of resources to make this approach tractable.

Constraint 2: The collection of PE in the system are divided into a set of disjoint *Physical Domains* (PD), each Domain being a set of consecutively addressed PE (see Figure 3-2). This division will not change during the execution of a program.

A couple of implementation details should be noted at this point. We restrict l to be a power of 2 to simplify the calculation of *mod* and *floor* functions. Also, the hardware is much simpler if we assume the instruction under consideration is present in the local PE. Hence the code for an instruction must not be split across two PE. We solve this problem at the compiler level; the compiler chooses the largest m allowed for each code-block and assures that no instruction is split

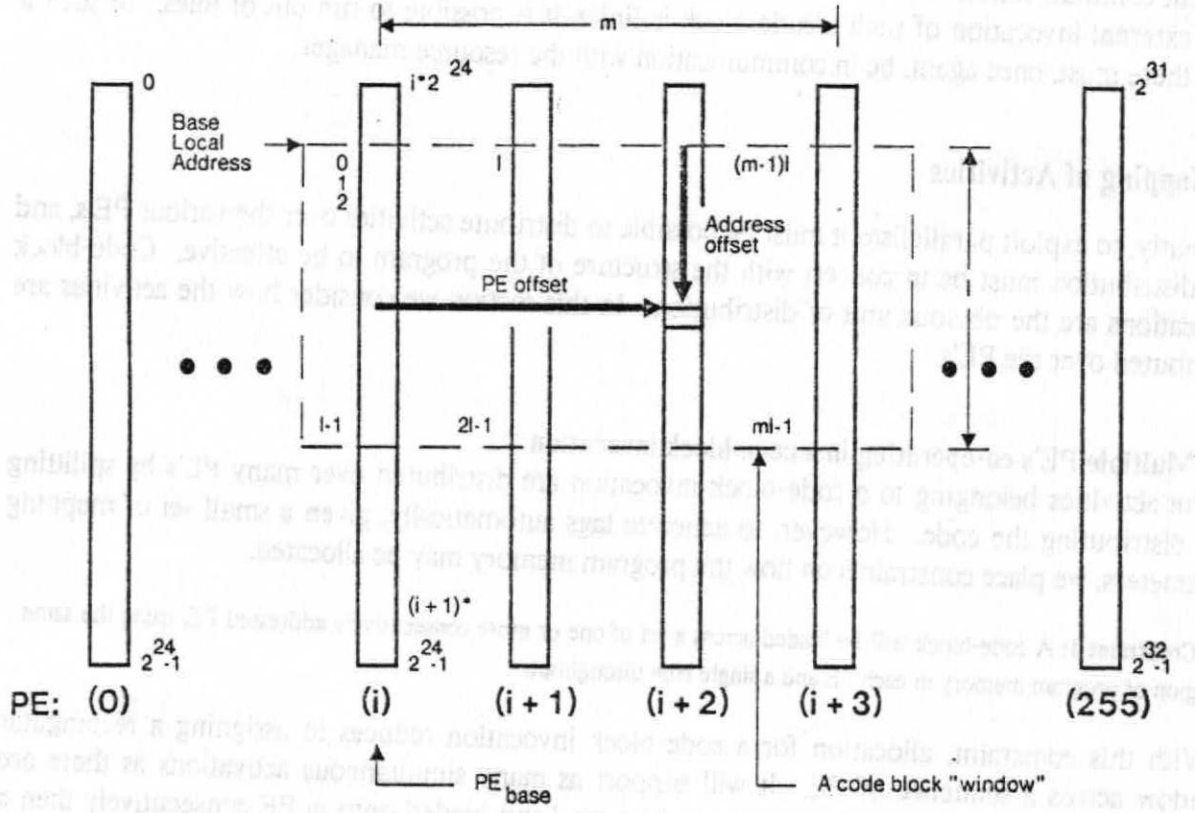

Figure 3-1: Destination Virtual Address Mapping

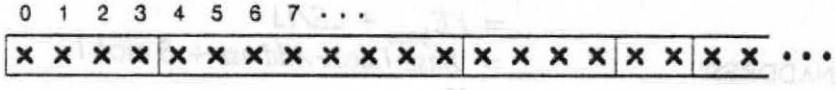

Figure 3-2: Physical Domains

when the code is divided into this many blocks. This *m* is a power of two, so any smaller power of two will work as well.

This approach is not the only solution to the problem. Another approach is to provide a relocation register for every code-block loaded onto a PE. This allows the block of memory used for a code-block to differ from PE to PE. However, one must either find a strip of relocation registers so the register index will be the same in each PE or include the code-block name in the tag and perform an associative match to find the relocation register. In either case, an extra level of indirection is required and the allocation problem is not significantly easier.

3.2. Distributing internal invocations

As discussed in Section 2 the generations of tags for *internal* invocations[3] of a code-block can be accomplished using special operators that manipulate hue field of the tags. Thus, it also makes sense to distribute initiations of a code-block automatically using certain mapping parameters. To make this idea tractable, we replace constraint 1 by a new constraint.

> Constraint 1': Each Domain is divided into a set of disjoint *Physical Subdomains* (PSD), each subdomain being a set of consecutively addressed PE. This subdivision is specified along with each window of code, so it may be different for different code-blocks in a given domain. A copy of the code-block is loaded into each subdomain as in Figure 3-1, where *m* is the width of the subdomain. The same COLOR is used throughout the domain for a given invocation.

Distributing initiations of a code-block must be approached with some care, however, because communication cost can outweigh the gain offered by parallel execution. Our studies [2] show that distributing initiations in blocks substantially reduces communication costs in many cases; data is often shared across a small number of initiations, usually one. To support this kind of distribution an additional parameter can be associated with an *external* invocation of a code-block so that initiations can be distributed in blocks.

One method of distributing initiations amongst subdomains is supported by the hardware using the following scheme (see Figure 3-3). The first activation, initiation 0, is assigned to the first subdomain of the domain. Given some mapping constant k, the j^{th} initiation is executed on the subdomain whose offset from the low end of the domain is given by

$$\langle \text{PSD-offset} \rangle = \lfloor j / k \rfloor \bmod \langle \#\text{PSD}/\text{PD} \rangle$$

In other words, the first k consecutive initiations are executed in the first subdomain, the next k initiation are executed on the second subdomain, and so on. If there are more initiations than k*(number of subdomains) then the assignment wraps around.

A method for distributing unused initiation numbers is provided for general recursion, but is not discussed here.

3.3. Code-Block Invocation

The Tagged Token Dataflow Architecture offers a completely general invocation mechanism, utilizing I-structures and the System Manager. It allows for any degree of strictness or non-strictness in passing arguments and in returning results. It allows mismatch in the number of arguments (or results) expected and the number actually transferred. By suitably configuring the graph, the compiler builds a calling sequence appropriate to the semantics of the high level languages. The general nature of the invocation process is presented below; the specifics of the calling sequence are presented in Appendix A.

Recall that a distinguished System Manager is created when the system is started and resides at some fixed location. It maintains the state of all system resources: program memory, code-block registers, colors, and I-structure storage. An *external* invocation of a code-block requires communication with this manager.

---

[3] Henceforth, initiation of a code-block will be used synonymously with *internal* invocation of a code-block.

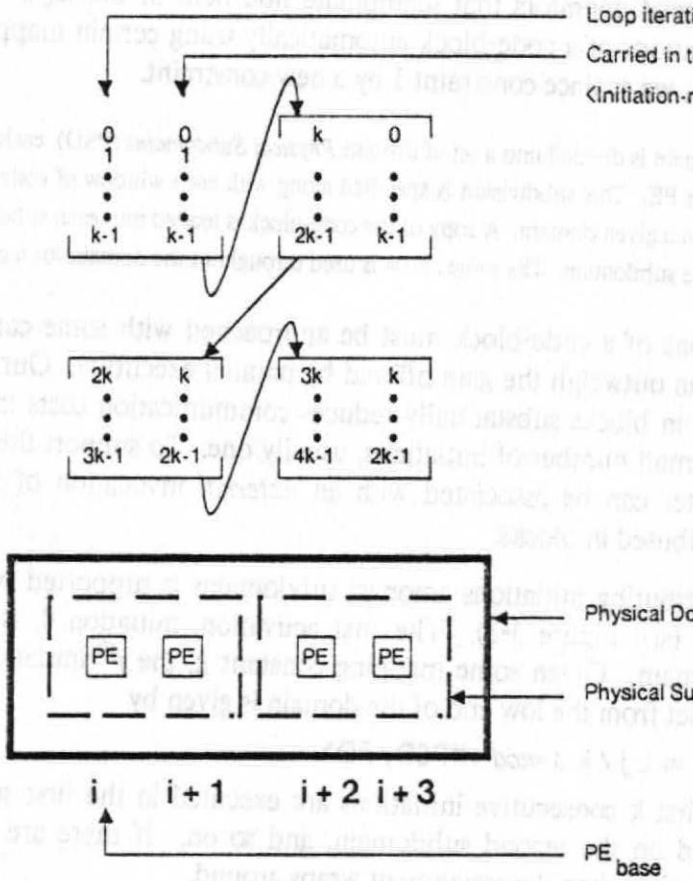

Figure 3-3: Iterations

Upon receiving the invocation request, the System Manager determines which is the best domain of the set of domains that have sufficient resources available to support the activation. Program memory, code-block registers and colors are allocated for the activation. The System Manager also allocates two I-structures: one for arguments and one for results. The manager must now make a number of requests to change the local states of each of the PE in the domain: it must request each PE to update its base and map registers, it must transfer blocks of code to various PE's (if the code-block is not already loaded), and it must request certain regions of I-structure storage to be initialized. While it waits for the acknowledgment of these tasks, the System Manager proceeds with other invocation requests. Once the initialization process is complete, the two I-structure descriptors are sent into the newly initiated (called) activation and into the (calling) activation that requested the invocation. The calling activation may store arguments and issue requests for results at its discretion: this communication link is totally asynchronous. The called activation immediately issues requests for arguments. Once enough requests are satisfied, it begins whatever operation it is to perform, eventually storing results. The scenario is depicted in Figure 3-4.

An important point to note is neither the hardware, nor the manager can determine when these various resources (program memory, colors, and I-structure storage) may be recycled. The onus of

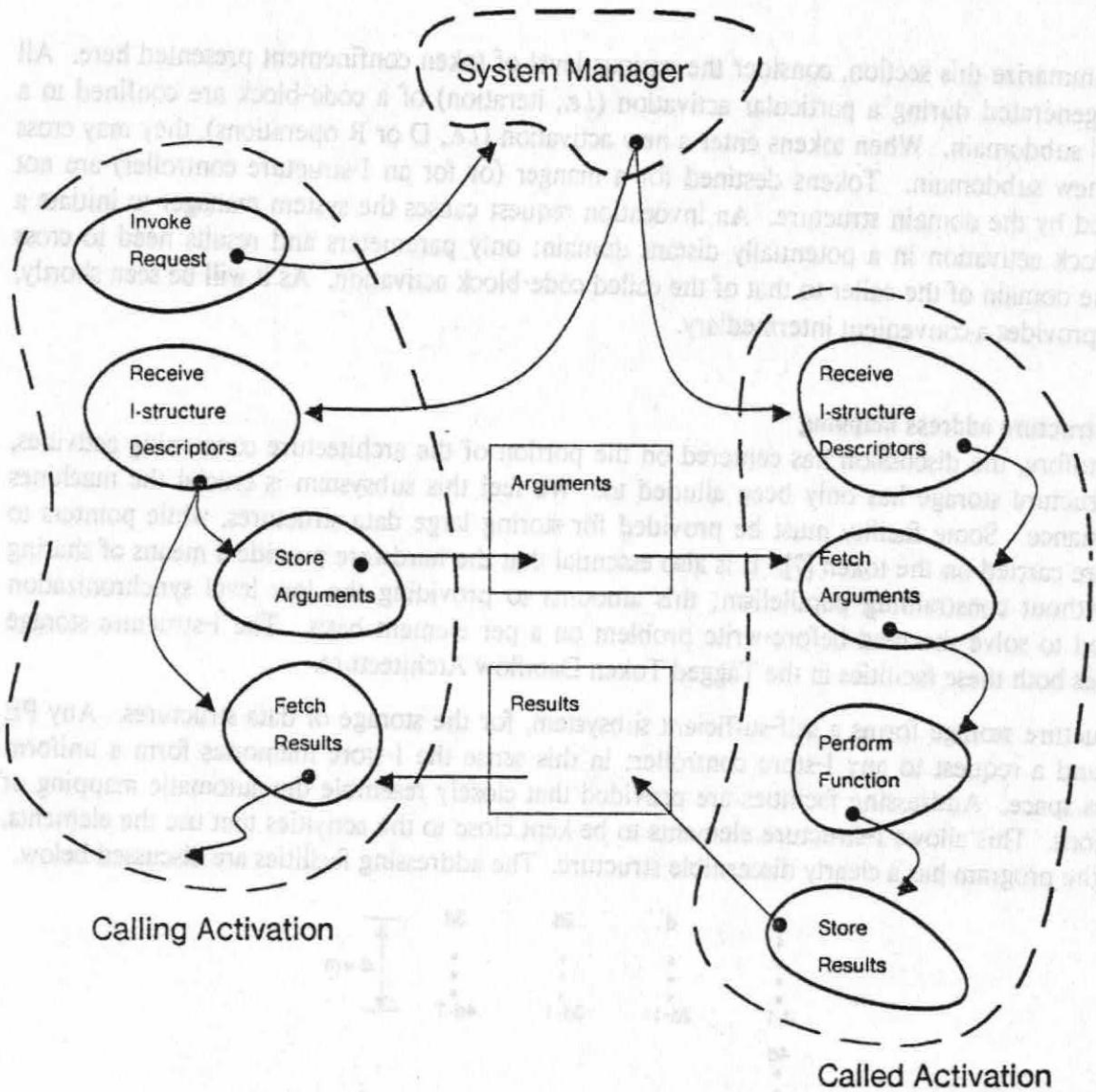

Figure 3-4: Scenario for Code-block Invocation

determining when resources can be deallocated is on the compiler. Thus, not just any graph is to be run on the machine. The graphs must be well behaved enough so that they are self cleaning. Also, the compiler must detect when resources (*e.g.*, color, I-structure) are no longer needed, and add instructions to release resources at appropriate points in the graph The graphs generated from Id by the compiler developed by the Functional Languages and Architectures group have these properties.

The tag generation mechanism requires a fairly small number of parameters and yet offers two degrees of freedom in distributing the work involved in a code-block activation: spreading a code-block over a number of PE allows the parallelism within a code-block to be exploited, distributing

activations over a set of groups of PE allows the parallelism offered by concurrent activations to be exploited, and in the case of internal activations, without intervention of the resource management system.

To summarize this section, consider the various level of token confinement presented here. All tokens generated during a particular activation (*i.e.*, iteration) of a code-block are confined to a physical subdomain. When tokens enter a new activation (*i.e.*, D or R operations), they may cross into a new subdomain. Tokens destined for a manger (or for an I-structure controller) are not restricted by the domain structure. An invocation request causes the system manager to initiate a code-block activation in a potentially distant domain; only parameters and results need to cross from the domain of the caller to that of the called code-block activation. As it will be seen shortly, I-store provides a convenient intermediary.

3.4. I-Structure address mapping

Heretofore, the discussion has centered on the portion of the architecture concerning activities, data structure storage has only been alluded to. We feel this subsystem is crucial the machines performance. Some facility must be provided for storing large data structures, while pointers to them are carried on the token [7]. It is also essential that the hardware provide a means of sharing data without constraining parallelism; this amounts to providing the low level synchronization required to solve the read-before-write problem on a per element basis. The I-structure storage provides both these facilities in the Tagged Token Dataflow Architecture.

I-structure storage forms a self-sufficient subsystem, for the storage of data structures. Any PE may send a request to any I-store controller; in this sense the I-store memories form a uniform address space. Addressing facilities are provided that closely resemble the automatic mapping of initiations. This allows I-structure elements to be kept close to the activities that use the elements, when the program has a clearly discernible structure. The addressing facilities are discussed below.

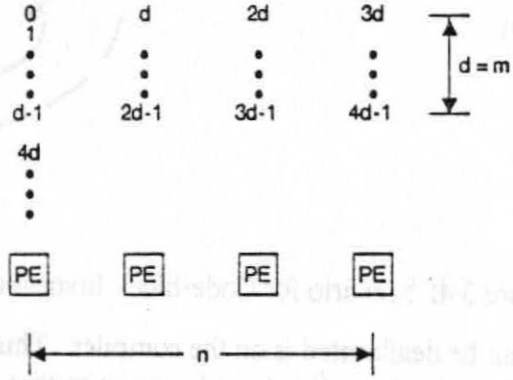

Figure 3-5: I-Structure Address Mapping

The compiler views an I-structure as an array of N elements. The index values for accessing elements of an I-structure of N words range over 0 to N-1. When an I-structure is created, as shown in Figure 3-5, a rectangular window of addresses in the physical address space is allocated to it. Given the constants d and n, the offset of an index value j from the upper left corner of the window can be calculated as follows.

$$\langle \text{PE-offset} \rangle = \lfloor j / d \rfloor \bmod n$$
$$\langle \text{Address-offset} \rangle = \lfloor \lfloor j / d \rfloor / n \rfloor * d + (j \bmod d)$$
$$= \lfloor j / d * n \rfloor * d + (j \bmod n)$$

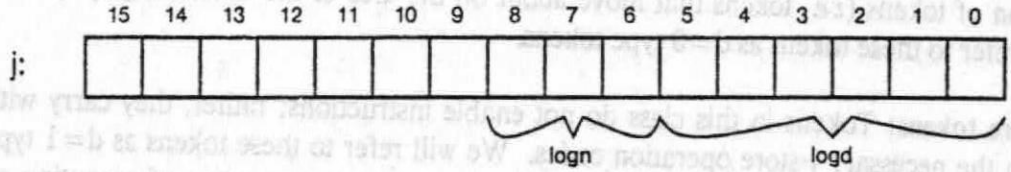

Figure 3-6: Address Mapping Example

If we assume that d and n are powers of 2, the above calculations are quite trivial as the following example illustrates. Let j be a 16 bit number, $d = 2^{logd}$ where logd = 6 and $n = 2^{logn}$ where logn = 3. Refer to Figure 3-6. Then,

<PE-offset> = number given by bits 8 through 6
<Address-offset> = number given by bits 15 through 9 and 5 through 0

The translation of an index into a physical address is done by the ALU when it executes the `Form-Address` instruction. The `Form-Address` instruction requires a *descriptor* for the I-structure in addition to the index value; the descriptor must contain the following information:

1. The physical address of the upper left hand corner of the window (*i.e.*, (<PE-base>,<Base-local-address>)).

2. The values of constants n and d.

It is not required that an I-structure mapping be identical to that of the mapping of initiations (*i.e.*, d does not have to be equal to any k, and n does not have to be equal to the size of any physical domain). The addressing facility allows locality to be exploited when data structures and initiations interact in a fairly simple manner; this is the case with most numerical programming.

I-structures have another vital role in the Tagged Token Architecture; each I-structure is a small address space, independent of tags. This allows them to be used to interface disjoint tag spaces; the utility of this arises in invoking a new code-block (see Section 3.3)

4. Processing Element

In this section the implementation of the Processing Element is discussed in detail, except the I-structure memory controller which is discussed later. The PE has 8 asynchronously functioning subsystems; these are connected by finite size buffers and communicate with each other using a send-acknowledge protocol. A block diagram of the PE is shown in Figure 4-1. The important differences from Figure 1-1 are the introduction of an input stage and output stage to interface with the network and a special service processor. The special service processor, marked *PE control*, has

access to the memory of all subsystems via a common bus, and is used for Input/Output, diagnostics and special memory functions.

The tokens entering a PE can be divided into the following three groups:

1. **Tokens corresponding to values in dataflow graphs:** These are associated with the usual notion of tokens (*i.e.*, tokens that move about on the arcs of the dataflow graph). We will refer to these tokens as d=0 type tokens.

2. **I-store tokens:** Tokens in this class do not enable instructions; rather, they carry with them the necessary I-store operation codes. We will refer to these tokens as d=1 type tokens. A d = 1 type of token is always generated as a consequence of executing an instruction in the ALU.

3. **Tokens for PE control and diagnostic purposes:** These types of tokens are similar to d=1 type tokens except they carry commands for the PE control subsystem including entry to managers. Tokens of this type will be referred to as d=2 type tokens.

Each type of token has a different format and follows a different path inside the PE.

4.1. Program representation

An instruction specifies an opcode, constant operands (if any) and the information about the destination instructions needed to construct the output tokens. An operator may require as many as three operands, but in the case of ternary operators, one operand is required to be a constant. One of the operands can be stored in the instruction itself if it is a constant for all possible executions of the instruction. For example, the increment-by-1 operator can be implemented using a + and a constant 1. Also, a constant area can be associated with a given code-block invocation; this allows operands which remain constant during a loop invocation to be treated as constants rather that circulated. An example of such an operand is the upper bound in the loop predicate of a FOR loop. Also, I-structure descriptors are invariably constant throughout a loop invocation. However, the instruction format does not allow more than one constant to be stored in any instruction. To increase flexibility in arranging inputs for an operator a 2 bit *disposition* field is associated with each of the 2 possible input values and the constant value. The disposition field specifies if the corresponding value is to become the 1st, 2nd or the 3rd operand or ignored entirely. Ignoring a constant means that there is *no* constant operand. Ignoring an input token means it serves only as a trigger for the activity. The general format for instructions is given below where the number to the right of a field name specifies the number of bits needed to represent that field.

```
<Header>
   <Opcode>                        (8)
   <Token-1-disposition>           (2)
   <Token-2-disposition>           (2)
   <Constant-disposition>          (2)
   <Constant-source>               (1)
   <Destination-list-flag>         (1)
<Constant-specification>
   <Data-type>
      <Data-length>                (4)
      <Data-class>                 (4)
```

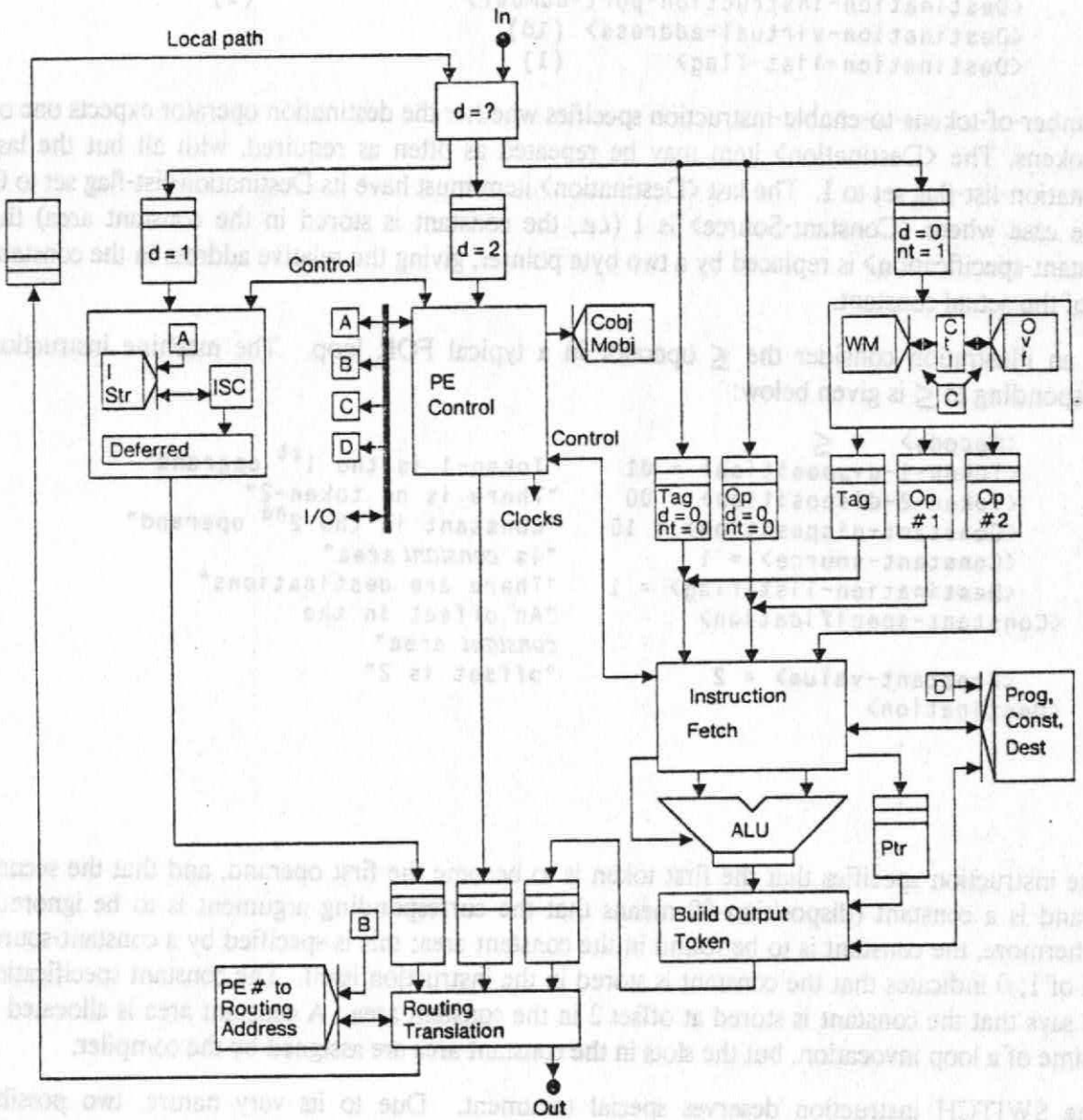

Figure 4-1: A Block Diagram of the PE

```
   <Data-value>                    (0-120)
<Destination>
   <Number-of-tokens-to-enable-instruction>          (1)
   <Destination-instruction-port-number>             (1)
   <Destination-virtual-address> (16)
   <Destination-list-flag>       (1)
```

Number-of-tokens-to-enable-instruction specifies whether the destination operator expects one or two tokens. The <Destination> item may be repeated as often as required, with all but the last Destination-list-flag set to 1. The last <Destination> item must have its Destination-list-flag set to 0. In the case where <Constant-Source> is 1 (*i.e.*, the constant is stored in the constant area) the <constant-specification> is replaced by a two byte pointer, giving the relative address in the constant area of the actual constant.

As an illustration consider the $\leq$ operator in a typical FOR loop. The machine instruction corresponding to $\leq$ is given below:

```
   <Opcode>  = ≤
   <Token-1-disposition> = 01     "Token-1 is the 1st operand"
   <Token-2-disposition> = 00     "There is no token-2"
   <Constant-disposition> = 10    "Constant is the 2nd operand"
   <Constant-source> = 1          "is constant area"
   <Destination-list-flag> = 1    "There are destinations"
<Constant-specification>          "An offset in the
                                  constant area"
   <Constant-value> = 2           "offset is 2"
<Destination>
          .
          .
          .
```

The instruction specifies that the first token is to become the first operand, and that the second operand is a constant (disposition 00 means that the corresponding argument is to be ignored). Furthermore, the constant is to be found in the constant area; this is specified by a constant-source field of 1; 0 indicates that the constant is stored in the instruction itself. The constant specification then says that the constant is stored at offset 2 in the constant area. A constant area is allocated at the time of a loop invocation, but the slots in the constant area are assigned by the compiler.

The SWITCH instruction deserves special treatment. Due to its very nature, two possible destination lists must be kept (one for the TRUE branch, and one for the FALSE branch). The TRUE list begins immediately after the <Data-Value> and is terminated by a zero <Destination-list-flag>. The FALSE list follows the TRUE list, and the relative address of the start of the FALSE list is given by the two byte offset in the <constant-specification> field. If the <Constant-disposition> is 0, there is no FALSE destination list.

4.2. Processing of standard activities

This section describes the execution of standard operators such as arithmetic operators. It also serves to introduce the salient features of the sections of the PE. This type of operator produces tokens which correspond to the values in the data flow graph (d=0 type tokens). The format of the type 0 token is given below:

```
<Token-type> = 0                                    (2)
<PE-number>                                         (8)
<Tag>
   <Color>                                          (4)
   <Instruction-address>                            (24)
   <Initiation-number>                              (8)
<Number-of-tokens-to-enable-instruction>                    (1)
<Port-number>                                       (1)
<Data>
   <Data-type>
      <Data-length>                                 (4)
      <Data-class>                                  (4)
   <Data-value>                                     (0-120)
```

Most of the field names are self evident. The Number-of-tokens-to-enable-instructions field allows for bypassing the *waiting-matching* section if the token is destined for a monadic operator or if its partner is a constant stored in the program memory. The port indicates whether the token is the first or the second operand for an operator. Recall, the color allows tokens from two different activations of a loop or procedure to be separated. The data values in the Tagged Token machine carry the type and length information with them. The data value may be 0 to 15 bytes long excluding one byte used for type and length information.

4.2.1. Waiting-matching section

When a d=0 type token enters a PE, it follows the path indicated by d = 0 (see Figure 4-1). Any token which needs a partner is passed to the *waiting-matching* section where its tag is associatively matched against the tags of the tokens already stored in the waiting-matching section. If a match is not found then the token is placed in the associative store. It may happen that there is no room for the token in the associative store in which case the token is stored in the overflow memory of the *waiting-matching* section. Management of overflow storage is an inherently slow process because, as long as the overflow storage has tokens, an incoming token that does not find its partner in the associative store must be sequentially matched against all the tokens in the overflow storage. Refusing the incoming token is not an option because that is guaranteed to deadlock the machine because token pairs can leave the *waiting-matching* section only when a new token arrives. Since an overflow in the *waiting-matching* section may cause severe performance degradation, the probability of overflow must be minimized by building a sufficiently large associative store. It should be noted that crowding in the *waiting-matching* section depends on both hardware and software considerations. The greater the time difference between the production of two tokens for an activity, the larger the associative store will have to be. Program decomposition and scheduling of activities have direct bearing on the time difference between the generation of two tokens for an activity.

4.2.2. Instruction fetch section

The *Instruction fetch* section has four responsibilities: fetch the instruction indicated by the INSTRUCTION-ADDRESS portion of the tag, determine the location of the destination list, fetch any constant operand, and align operands as per their disposition.

Fetching the current instruction is straightforward since the INSTRUCTION-ADDRESS field gives the absolute address in the program memory. However, in order to locate the constant area pointers, the base address of the code-block must also be determined. Also, the *Build output token*

section requires this information to translate relative addresses into physical addresses. To this end, the instruction fetch section includes a sequence of register pairs of the form <code-base-address, map-register-number>. These are kept packed contiguously and ordered by code-base-address. An associative $\leq$ comparison is made between each code-base-address and the INSTRUCTION-ADDRESS value, the lowest register with a successful comparison specifies the base address and map register number (map registers are discussed below) for the code-block activation this activity is part of.

If a constant area is required by the code-block, a list of constant area pointers is stored at addresses immediately preceding the code-base-address. They are indexed by COLOR with the lowest address corresponding to color 0. Each pointer is a 24 bit address giving the base address of the constant area for the particular activation. The constant-specification is added to this constant area base address to get the first byte of the constant. Figure 4-2 depicts the relationship of base registers, map registers and program memory.

Once the constants are fetched, the operands are aligned and combined with the opcode to be sent on to the ALU. The map register index and the destination list pointer is forwarded to the *Build output token* section.

4.2.3. ALU

The *ALU* section basically receives operation packets and performs the operation on the accompanying operands. However, the set of operations is quite large and some of the operations are quite sophisticated. For example, I-structure operations involve an address translation in I-structure address space analogous to that performed on destination addresses. A complete description of the ALU operations is given in the Instruction Set Definition [4]. The operations that generate other than standard tokens are discussed in Sections 4.3, 4.5.

4.2.4. Build output token section

For each destination in the instruction, the build-token section generates an output token. The data value to be included in the token is provided by the ALU. Computation of the output token's PE number, physical instruction address, and initiation number is as explained in Section 3. The mapping parameters required for tag generation are kept in *map registers* local to the build-token section. The map register index provided by the Instruction-fetch section specifies which set of mapping parameters to use. Each map register is a table containing:

```
<Domain-Base-PE>                    (8)
<Code-Base-Addr>                    (24)
<code-per-PE>                       (24)
<Initiations-per-PE>                (8)
<color-continuation-flags>          (16)
<Subdomain-Base-PE>                 (8)
<SubDomain-Size>                    (8)
<Last-SubDomain-flag>               (1)
```

The <color-continuation-flags> is a bit-list indexed by color. It allows for pre-allocation of colors, effectively extending the INITIATION-NUMBER field as discussed in Section 4.4

The *Build output token* section reads the destination list from program memory and computes the new destination address based on the incoming tag and the information in the map register. A token is formed in this way for each destination and sent to the output section.

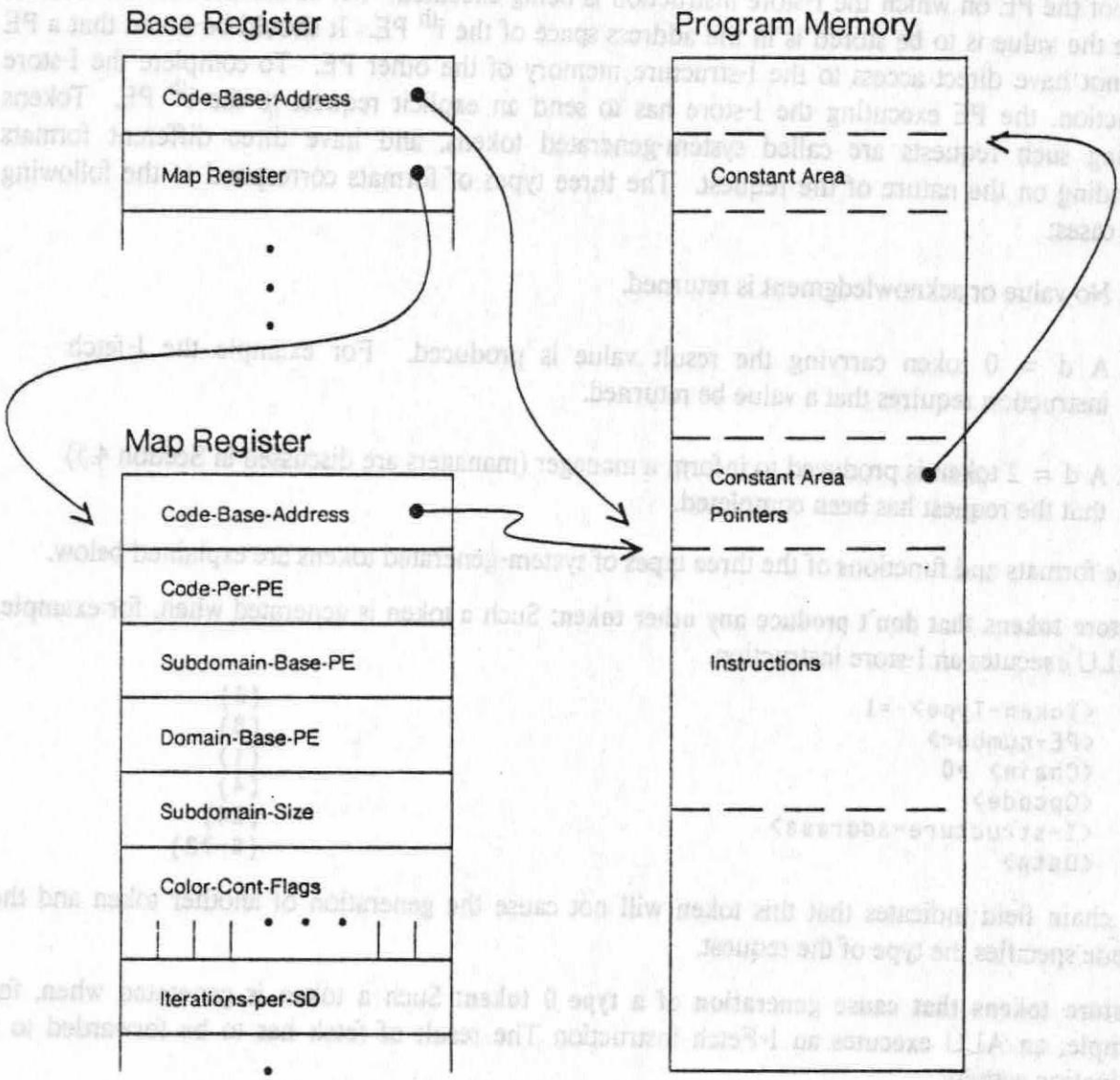

Figure 4-2: Code Block Registers and Constant Areas

4.2.5. Output section

The *output* section translates the PE# field into a network address and routes the token into the communications network. The communications network is a packet switched network, of a robust topology. The use of routing tables and very general routers allows the exact network topology to be modified easily. The nature of the translation to routing addresses is discussed by Arvind and Iannucci [6]. The design of the basic network element is described by Iannucci [9].

4.3. I-structure instructions: operators that produce d = 1 tokens

The various operations on I-structures produce d = 1 type of tokens which are sent to an I-store controller. Consider the operator I-store which takes an I-structure memory address and a value as arguments and stores the value at the specified address. Because an I-structure will usually be distributed over several PE, the address where the value is to be stored may not be in the memory space of the PE on which the I-store instruction is being executed. Let us assume that the address where the value is to be stored is in the address space of the i^{th} PE. It should be noted that a PE does not have direct access to the I-structure memory of the other PE. To complete the I-store instruction, the PE executing the I-store has to send an explicit request to the i^{th} PE. Tokens carrying such requests are called system-generated tokens, and have three different formats depending on the nature of the request. The three types of formats correspond to the following three cases:

1. No value or acknowledgment is returned.

2. A d = 0 token carrying the result value is produced. For example the I-fetch instruction requires that a value be returned.

3. A d = 2 token is produced to inform a manager (managers are discussed in Section 4.5) that the request has been completed.

The formats and functions of the three types of system-generated tokens are explained below.

I-store tokens that don't produce any other token: Such a token is generated when, for example, an ALU executes an I-store instruction.

```
<Token-Type> =1                                  (2)
<PE-number>                                      (8)
<Chain> =0                                       (1)
<Opcode>                                         (4)
<I-structure-address>                            (24)
<Data>                                           (8-72)
```

The chain field indicates that this token will not cause the generation of another token and the opcode specifies the type of the request.

I-store tokens that cause generation of a type 0 token: Such a token is generated when, for example, an ALU executes an I-Fetch instruction The result of fetch has to be forwarded to a destination activity.

```
<Token-Type> =1                                  (2)
<PE-number>                                      (8)
<Chain> =1                                       (1)
<Opcode>                                         (4)

<Destination>
    <new-Token-type> =0                          (2)
    <new-PE-number>                              (8)
    <new-Tag>                                    (36)
    <new-Number-of-tokens-to-enable-instruction> (1)
    <new-Port-number>                            (1)
<I-structure-address>                            (24)
```

```
<Data>                                          (8-72)
```

The information contained in the destination field is copied into the various fields of the newly generated token.

I-store tokens that cause generation of a type 2 token: Many instructions such as reset I-structure must generate a token for a manager so that the system can ascertain that the instruction has been completed (*e.g.*, the I-structure has been reset).

```
<Token-type> = 1                                (2)
<PE-number>                                     (8)
<Chain> = 1                                     (1)
<Opcode>                                        (4)

<Destination>
     <new-Token-type> = 2                       (2)
     <new-PE-number>                            (8)
     <new-Chain> =0                             (1)
     <new-Opcode> = *entry                      (4)
     <Manager-address>                          (24)
<I-structure-address>                           (24)
<Data>                                          (8-72)
```

The new d=2 type token is passed to the manager specified by the manager-address a receipt identifying the acknowledgment.

When a d=1 type token enters a PE, it is forwarded to the I-structure controller which is responsible for reading, writing and managing the I-structure storage.

The I-structure controller is especially suited for a multiprocessor environment in which the synchronization involved in producing and consuming data structures must be efficient and fine-grained. We associate with each memory cell in the I-store special flags (called *presence* bits) which indicate the memory cell's status - written or unwritten. This offers 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.[4]

When a write request for that location arrives at a later time, 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 4-3) 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 [8].

---

[4]The idea of associating a status bit with each memory cell is not new - the Denelcor HEP multiprocessor [10] 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.

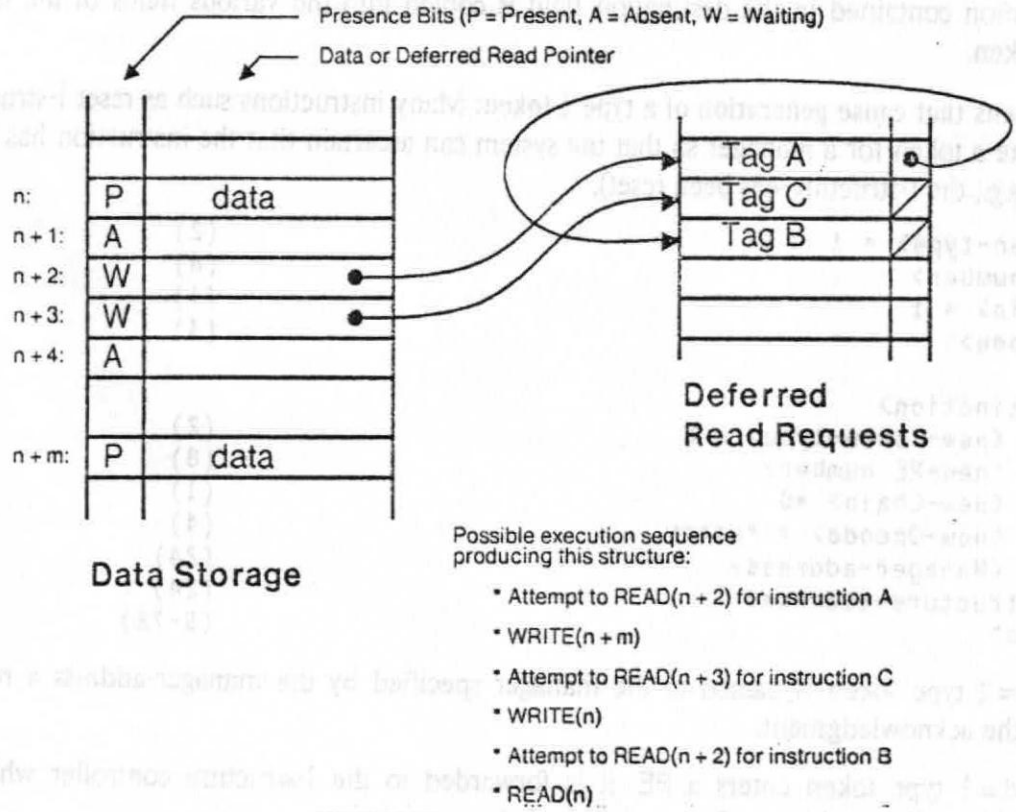

Figure 4-3: I-Structure Storage

4.4. Operators that distribute initiations

In Section 3.2 the need for operators that manipulate portions of the tag was discussed; these are the D and R operators in the Tagged Token Dataflow Architecture[5] Details are presented here for the D operator. The D operator increments the INITIATION field of the tag. This automatically distributes work based on the mapping parameters in the build-token section. It should be noted that the INITIATION-NUMBER field is of finite size (8 bits) and hence may overflow during the execution of the D operator. Overflowing of initiation number field is handled by allocating a new color. For some loop expressions the number of iterations, and hence the number of colors it will need during the execution can be predetermined. In such cases, rather than a single color, a group of colors can be allocated. When such a code-block is invoked, color continuation flags are set up in the map register for the activation. In case the INITIATION-NUMBER field overflows, the color continuation flag is used to determine if the next color can safely be used. However, if all color/initiation values for the code-block have been exhausted then the system manager must be informed to get a new color. This essentially involves invoking the loop in a tail-recursive fashion.

Implementing this is somewhat complicated. The D operator maintains two destination lists--one

[5]The D operator corresponds closely to the D operator in the U-interpreter.

which specifies the *normal* destinations, and one which specifies the operators to be used in case color/initiation values have been exhausted. The compiler is responsible for generating the code to call the manager and the code to transmit the values coming from D operators to the appropriate places. The algorithm to compute the new tag is given below.

1. Calculate the PE-offset within some physical subdomain and the physical-instruction-address as explained in Section 3.1.

2. Compute the new initiation number by adding 1 to the old initiation number. If ⟨initiation number⟩ *mod* k ≠ 0, the new PE number is computed using the PE-offset from step 1 and the subdomain-base-PE from the map register.

3. If ⟨initiation number⟩ *mod* k = 0 and this is not the last physical subdomain, the new PE number will be computed by adding the PE-offset from step 1 to the number of the first PE of the *next* physical subdomain (*i.e.*, the new PE number is (Subdomain-base-PE + Subdomain-Size + PE-offset) The value of k (initiations per PE) is subtracted from the initiation number in the result token.

4. If ⟨initiation number⟩ *mod* k = 0 and this is the last physical subdomain, the new PE number will be computed by adding the PE-offset (from step 1) to Domain-base-PE. No adjustment of the initiation number is necessary.

5. If in any case, the initiation number should overflow (in the result token), a new color must be allocated. The color location corresponding to the current color is examined to determine the value of the *color-continuation-flag*. If the *color-continuation-flag* is TRUE, the next (sequential) color is used in the output token (*i.e.*, new color = old color + 1), and the initiation number is set to zero.

   If the *color-continuation-flag* is FALSE (indicating that all color/initiation values for this code block have been exhausted), the incoming value (without changing the initiation-number field) is sent to the destinations specified in the second destination list. This will result in a new invocation.

4.5. Implementation of managers

Many functional languages provide some means for resource management within the applicative context. In the dataflow language, Id, managers have this role. They are Id programs which can be "called" like a procedure from several different places or unrelated Id code blocks. However, unlike a procedure, all "calls" to a manager are non-deterministically merged together and affect the same manager body. It is more correct to think of a manager as an object which is *used* non-deterministically by several users. The interconnection of the operators involved in a Manager "call", U, U^{-1}, Entry, and Exit is shown in Figure 4-4. The *Entry* operator non-deterministically merges all incoming requests and produces an input stream for the main body of the manager. It also passes the "return activity names" on the incoming tokens to the *Exit* operator so that the Exit operator can form the result token with the appropriate activity name. It should be noted that an execution of the Entry operator requires no matching of tokens but is sensitive to the history of the operator (*i.e.*, it must know how many other tokens have passed through it to generate a correct

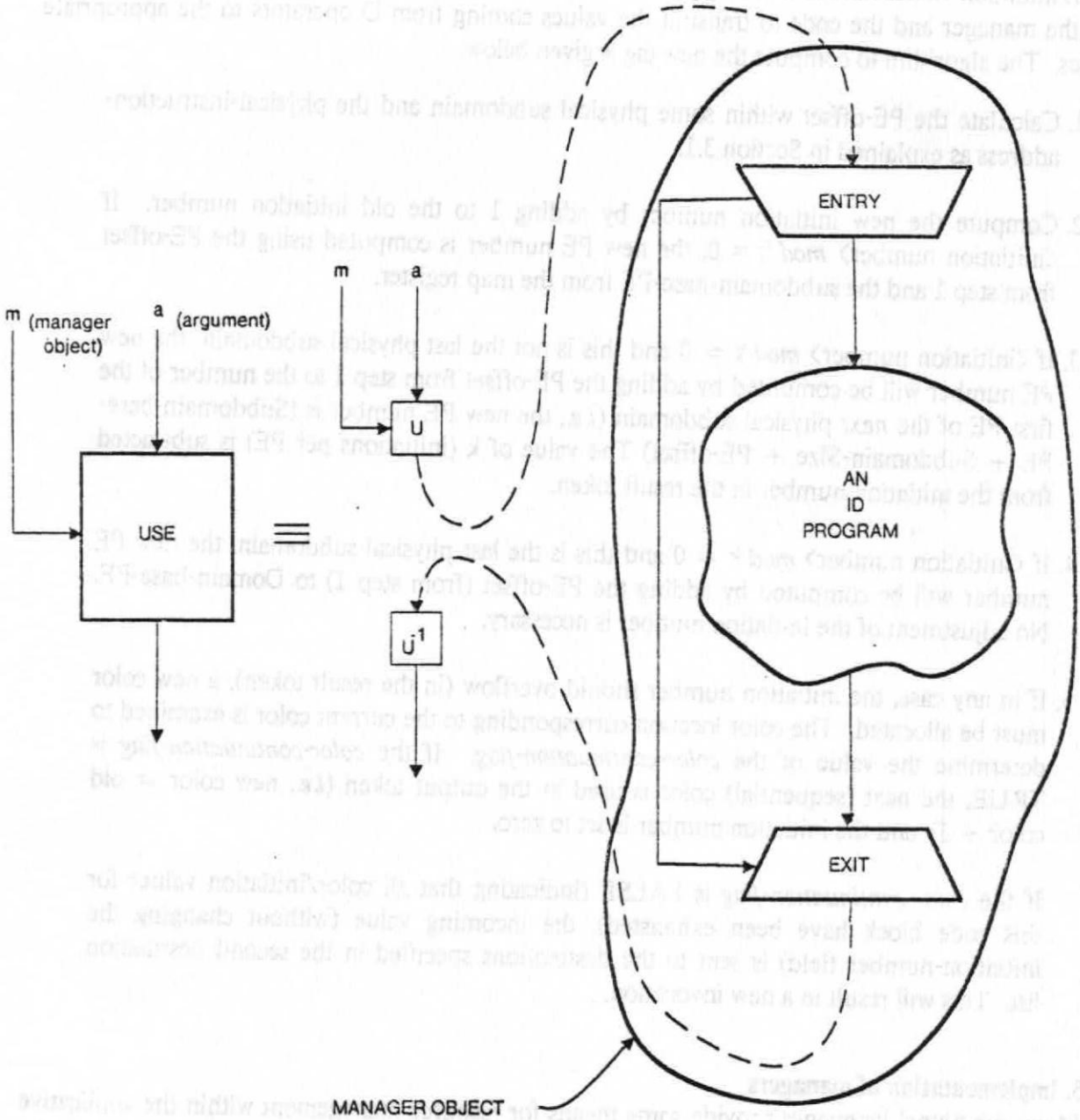

Figure 4-4: Using a Manager

activity name).

In the Tagged Token Dataflow Architecture, we have decided to use storage to implement manager objects. A manager object is represented by a 32 bit physical address (*i.e.*, an 8 bit PE address and a 24 bit local memory address); a block of storage starting at that address is allocated to the manager object (see Figure 4-5). The table has n slots, where n is the maximum number of concurrent users of the manager allowed by the implementation. A slot in the table has a flag to indicate if the slot is in use and space to store a return activity name. A counter j is also associated

with each manager object; the value of this counter is used as the initiation field for the tags on the tokens that enter the manager. The rest of the tag for tokens entering the manager is specified when the manager is created and stored in the manager object table.

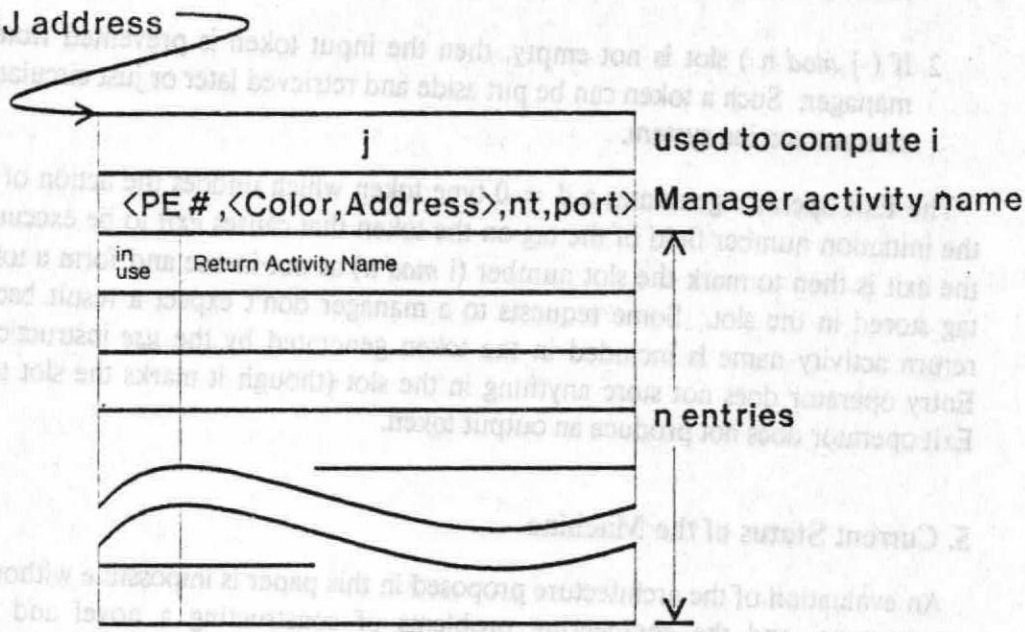

Figure 4-5: Table Associated with a Manager Object

An instruction called *use* (which implements abstract operator U) is provided to communicate with the manager. The following d = 2 type token is generated when an ALU executes the *use* instruction:

```
<Token-type> = 2
<PE-number>                                  "8 higher order bits of the
                                             32 bit Manager Object address"
<Chain> = 1                                  "Contains a return activity
                                             name"
<Opcode> = *entry
<Destination>                                "The return activity name"
     <new-Token-type> = 0
     <new-PE-number>
     <new-Tag>
     <new-Number-of-tokens-to-enable-instruction>
     <new-port-number>
<I-structure-address>                        "24 lower order bits of the
                                             Manager Object address"
<Data>                                       "8-72 bits of any input
                                             data for the manager"
```

The token is routed to the PE-control section of the destination PE where the following steps are

taken:

1. If (j *mod* n) slot is empty, the tag for the result value (*i.e.*, <Destination> field on the input token) is stored in the slot and a new token with data from the input token and a tag constructed by using the j value and the information in the table[6] is produced. The j counter is incremented by 1.

2. If (j *mod* n) slot is not empty, then the input token is prevented from entering the manager. Such a token can be put aside and retrieved later or just circulated back in the communication system.

The Exit operator generates a d = 0 type token which undoes the action of the entry. Suppose the initiation number field of the tag on the token that causes *exit* to be executed is i. The effect of the exit is then to mark the slot number (i *mod* n) as not-in-use and form a token whose tag is the tag stored in the slot. Some requests to a manager don't expect a result back, in which case no return activity name is included in the token generated by the *use* instruction. Accordingly the Entry operator does not store anything in the slot (though it marks the slot to be in-use) and the Exit operator does not produce an output token.

5. Current Status of the Machine

An evaluation of the architecture proposed in this paper is impossible without the construction of a prototype, and the engineering problems of constructing a novel and large multiprocessor machine can not be exaggerated. As discussed in [6], we have decided on two "soft" implementations before considering a direct VLSI implementation. Two 64 processor implementations are in various stages of development : a *simulated* machine on IBM 4341, and an *emulated* machine on 64 Symbolics 3600 (Lisp) machines. Both implementations will accept the code generated by the Id compiler, and model each subsystem of the PE and the communication system explicitly. The simulator will also accept data dependent timing specifications for each subsection while the emulator will "fake" the internal pipelining of PE's and the communication system by the scheduling of tasks where a task will represent one subsystem of the PE. A primitive version of the emulated machine comprising 8 PE's should be operational in 1984 while the full machine is not expected to be available until late 1985.

A specification of each subsystem and the overall simulation program has been written in Pascal. To conduct architectural experiments on the simulator, we are developing a variety of resource managers for I-structure storage, and allocation and deallocation of PE's and colors. The simulation work is being done in cooperation with IBM Research (Yorktown) and we describe its near term goals and very preliminary results briefly.

It should be easy to see that the proposed PE design can indeed sustain one instruction execution per cycle (*i.e.*, pipeline beat) provided there is sufficient parallelism in the application program. Preliminary analysis of several large scientific codes show that parallelism in applications will not be

---

[6] The first location in the table contains the color allocated to the manager and the address of the first instruction in the manager body

the bottle-neck [1]. The kind of parallelism needed to support one instruction execution per cycle in our machine is much more pervasive than the kind of specialized homogeneous parallelism (*i.e.*, vector operations) needed to support that rate in machines like Cray-1. However, dataflow machine rate is not directly comparable to instruction rate of von Neumann machines because dataflow machine language programs appear to have 3 to 4 times as many instructions as a conventional sequential machine[7]. A major factor in the profusion of instructions on a dataflow machine is instructions executed to compute the addresses for store and fetch operations. Index registers and address generation hardware in vector machines eliminates most of these instructions. It should be obvious that even a von Neumann machine will execute a very large number of additional instructions if index registers are not provided. Though we have several ideas about solving this problem, no specific proposal has been incorporated so far. The procedure call mechanism also introduces a lot of instructions but most of these can be eliminated by restricting the generality of parameter passing convention.

It is instructive to consider the execution of a totally sequential code-block on one PE. If each subsection takes one unit of time to process a token then the n instructions will execute in 4*n time units. In general, a program with the same functionality as this code-block will take much less time on a sequential computer (built out of the same technology) because sequential computers can overlap execution of instructions even in totally sequential code. However, execution of two such code-blocks may not take substantially longer on one PE while it will take twice as long on a sequential processor. Hence evaluation of our architecture must take into account both the internal architecture of our PE and the "small grain" parallelism present in code-blocks.

An important parameter to be determined is the size of the Waiting-Matching section and the factors that affect crowding in it. Scheduling (*i.e.*, mapping) of activities, I-structure storage maps, and the relative speed of PE and communication system all affect the number of tokens that wait for their partners in the Waiting-Matching section. However, the qualitative effect of any of these parameters (simply by analysis) has eluded us. Once the resource managers have been implemented, we plan to study these interactions on the simulator. It remains to be seen how much analysis of a program by the compiler is required for its efficient mapping.

Though not as crucial as the size of the Waiting-Matching section, the size of tags and the size of the deferred read section in I-store controller is also of concern. As noted earlier, factors such as scheduling affect these parameters. Besides the issue of size, scheduling will affect the number of fetches that will get deferred in the I-store. Every read that has to be deferred in some sense slows down the machine.

Questions about the scalability and ultimate performance of the Tagged-Token Dataflow machine cannot be answered until some of these inter-relationships have been understood.

---

[7] We are indebted to Dr Ekanadham and Dr Bruer of IBM for this observation.

Appendix A. Instruction graph for Code-Block Invocation

There are three logical phases in the invocation process: allocating resources and establishing the called code-block, initiating activity in the called code-block, and finally expecting results from that invocation. The compiler generates triggers to initiate the various phases.

1. A trigger T_a initiates the first phase. It causes a request to be sent to the System Manager. The called code-block is technically activated when it receives the argument and result descriptors. However, in order to cater to specific semantics of a procedure call in a language and to reduce the load on the Waiting-Matching section,[8] it must successfully fetch a specific argument (T_b) before any other activities may be enabled; it essentially sleeps until the calling activation triggers it into action.

2. Once the descriptors are received from the manager the calling code-block begins storing arguments. When it has stored enough arguments, it stores a distinguished argument T_b. This is the trigger the called code-block is waiting on; it initiates activity in the called code-block.

3. The calling code-block waits for a specific result (T_r) before it starts result fetches in earnest. The role of the trigger T_r is similar to the role of the trigger T_b.

Triggers T_a,T_b, and T_r need not be different from one of the values being passed or returned. Since the manager may take some time in loading the code into the memory of PE, it may be advantageous to generate T_a ahead of time. Depending on how T_b is generated, a strict or nonstrict call mechanism can be implemented; for example, generating T_b when all the values to be passed to the called code-bock have been written in the I-structure will result in a strict call mechanism.

The architecture supports constant areas allocated on a per color basis. The intent is that Id loops have constant areas, while Id procedures do not. The critical issue is that the arguments to be stored in the constant area must in fact be stored before the code-block is put into activity (*i.e.*, before T_b is generated). The Id compiler divides loop constants into two categories: essential constants (those it can determine will be necessarily used in the execution of the loop) and nonessential constants (all others). Essential constants are stored in the constant area; the others are circulated.

One more piece of information needs to be understood before the actual calling sequence is presented. A special data type called *smash* is provided in the machine to facilitate sending information to a manager. The *smash* type is used to put more than one data value (limited only by the size of the token) alongwith their data-length and data-type fields onto one token. A smash type is generated by the *compress* operation which takes two values of any type. The *expand* operation recovers the original values and sends them to their respective destinations; the first value is sent to the first destination and the second to the second destination.

---

[8] This is an example of one of the factor that influences the load on the Waiting-Matching section.

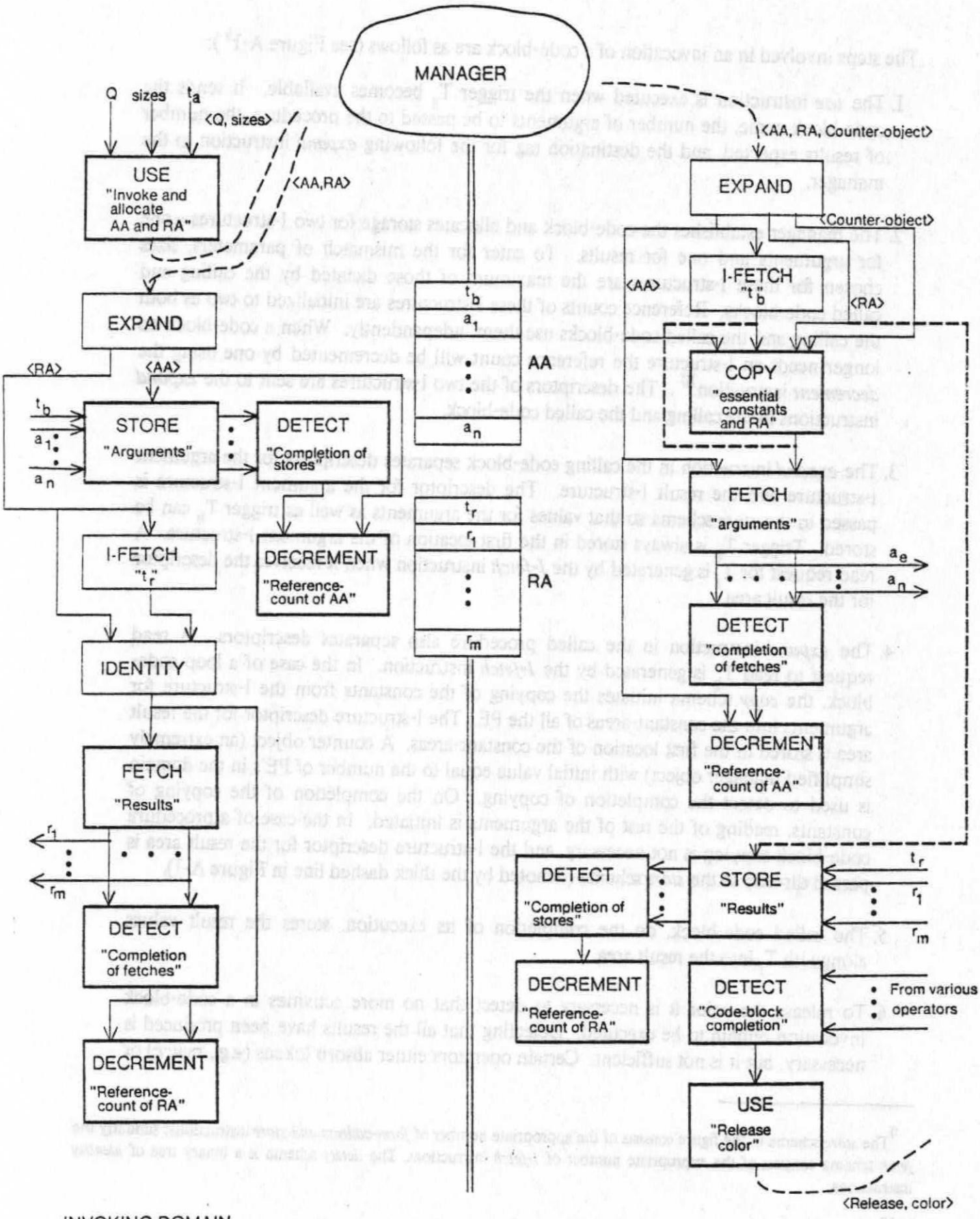

Figure A-1: Graph for a Code-block Invocation

The steps involved in an invocation of a code-block are as follows (see Figure A-1[9]):

1. The *use* instruction is executed when the trigger T_a becomes available. It sends the code-block name, the number of arguments to be passed to the procedure, the number of results expected, and the destination tag for the following *expand* instruction to the manager.

2. The manager establishes the code-block and allocates storage for two I-structures – one for arguments and one for results. To cater for the mismatch of parameters, sizes chosen for these I-structures are the maximum of those dictated by the calling and called code-blocks. Reference counts of these I-structures are initialized to two as both the calling and the called code-blocks use them independently. When a code-block no longer needs an I-structure the reference count will be decremented by one using the *decrement* instruction[10]. The descriptors of the two I-structures are sent to the *expand* instructions in the calling and the called code-block.

3. The *expand* instruction in the calling code-block separates descriptors for the argument I-structure and the result I-structure. The descriptor for the argument I-structure is passed to the *store* schema so that values for the arguments as well as trigger T_b can be stored. Trigger T_b is always stored in the first location of the argument I-structure. A read request for T_r is generated by the *I-fetch* instruction when it receives the descriptor for the result area.

4. The *expand* instruction in the called procedure also separates descriptors. A read request to read T_b is generated by the *I-fetch* instruction. In the case of a loop code-block, the *copy* schema initiates the copying of the constants from the I-structure for arguments into the constant-areas of all the PE. The I-structure descriptor for the result area is stored in the first location of the constant-areas. A counter object (an extremely simplified manager object) with initial value equal to the number of PE's in the domain is used to detect the completion of copying. On the completion of the copying of constants, reading of the rest of the arguments is initiated. In the case of a procedure code-block copying is not necessary, and the I-structure descriptor for the result area is passed directly to the *store* schema (denoted by the thick dashed line in Figure A-1).

5. The called code-block, on the completion of its execution, stores the result values alongwith T_r into the result area.

6. To release the color it is necessary to detect that no more activities in a code-block invocation remain to be executed. Detecting that all the results have been produced is necessary, but it is not sufficient. Certain operators either absorb tokens (*e.g.*, *switch*) or

---

[9]The *store* schema in the figure consists of the appropriate number of *form-address-and-store* instructions; similarly the *fetch* schema consists of the appropriate number of *I-fetch* instructions. The *detect* schema is a binary tree of *identity* instructions.

[10]An I-structure gets deallocated when its reference count goes to zero.

do not affect the result values directly (*e.g., I-store, decrement*). The outputs of all such operators are coalesced into a single signal. In the case of a loop the signal generated by the operators in the loop body needs to be circulated so that signals for all the iterations can be merged together. This signal is used to trigger the *use* instruction which requests the manager to release the color.

7. When T_r becomes available, the calling procedure reads the rest of the result values.

Appendix B. Instruction Set Summary

The resulting instruction set design defines the hardware-recognized data objects and the operations that may be performed on them. Since we view the machine as being completely self-sufficient (*i.e.*, all operations from the level of interpreting compiled Id graphs all the way down to the lowest level I/O operations), the instruction set was designed accordingly. This was done with an eye toward the future - Id self-compilers written in Id, managers performing operating system like services written in Id, etc.

To support this wide range of functions, several different token types were defined (tokens corresponding to values in data flow graphs and several flavors of system-generated tokens) as well as a number of scenarios (we call them *paradigms*) for token processing. All token processing begins with executing an instruction form a compiled data flow graph. Based on the instruction type and the corresponding paradigm, one or more additional tokens may be created which trigger additional graph instructions. Alternatively, system-generated tokens may be created which effect a number of implementation-specific operations in the machine.

All of the tokens carry data objects of self-identifying type. Where appropriate, operation codes are type-independent and will perform the necessary format conversions on both input and output. There are also the facilities for *preventing* such conversions, and also instructions for performing *explicit* conversions.

The hardware implements low-level primitives for procedure invocation and address mapping. These primitives are necessary to efficiently support resource allocation and the partitioning of code blocks across multiple processing elements. For the most part, these issues are not visible from the standpoint of compilation. However, since the facilities are partially static, the instruction set also defines the necessary characteristics of the mapping process so that code generators / loaders will be able to partition code blocks and to allocate instructions amongst these partitions. A summary of the instruction set, excerpted from Arvind and Iannucci [4], is given below.

Token Processing

Paradigm 1: <d=0> to <d=0>
Paradigm 2: <d=0> to <d=1 or 2>
Paradigm 3: <d=0> to (<d=1 or 2> to <d=0>)
Paradigm 4: <d=0> to (<d=1 or 2> to <d=2,*Entry>)

Data Types

Any: Includes all types[11]
Arith: Includes FP-32,FP-64,Int
Bits: Bit string, 0 to 15 bytes in length
Bool: Boolean
Char: ASCII character, 7 bit code
Cobj: Counter object
Comp: Includes Bool,Char,Cobj,Err,FP-32/64,Int,ISD-T-Fix,ISD-T-Var,ISD-U-Fix,MDef,MObj,Proc,Smash
Err: Error value

---

[11] Any, Arith, Comp, and Int are pseudo-types.

FP-32:	Floating point, 32 bit representation
FP-64:	Floating point, 64 bit representation
Int:	Includes Int-8,Int-16,Int-24,Int-32
Int-8:	Integer, 8 bit two's complement representation
Int-16:	Integer, 16 bit two's complement representation
Int-24:	Integer, 24 bit two's complement representation
Int-32:	Integer, 32 bit two's complement representation
ISA-T-Fix:	I-Structure address (typed elements of fixed max. size)
ISA-T-Var:	I-Structure address (typed elements of variable size)
ISA-U-Fix:	I-Structure address (untyped elements of fixed max. size)
ISD-T-Fix:	I-Structure descriptor (typed elements of fixed max. size)
ISD-T-Var:	I-Structure descriptor (typed elements of variable size)
ISD-U-Fix:	I-Structure descriptor (untyped elements of fixed max. size)
MDef:	Manager definition
MObj:	Manager object
Proc:	Procedure definition
Smash:	Composite of two or more objects, *e.g.*, <<Int-8>,<MDef>,<Bool>>

Operations (d=0)

{Par}	Operation	Functionality
{1}	**Arithmetic:**	$Arith_1 \times Arith_2 \Rightarrow Arith_3$
{1}	**Conversion:**	$Arith_1 \times Arith_2 \Rightarrow Arith_2$ $Char \times Int\text{-}8 \Rightarrow Int\text{-}8$ $Int\text{-}8 \times Char \Rightarrow Char$
{1}	**Booleans:**	$Bool \Rightarrow Bool$ $Bool \times Bool \Rightarrow Bool$
{1}	**Bit String Logicals:**	$Bits \Rightarrow Bits$ $Bits \times Bits \Rightarrow Bits$
{1}	**Shift:**	$Bits \times Int\text{-}8 \Rightarrow Bits$
{1}	**Concatenate:**	$Bits \times Bits \Rightarrow Bits$
{1}	**Adjust-Length:**	$Bits_1 \times Bits_2 \Rightarrow Bits_2$
{1}	**Extract-Type/Value:**	$Any \Rightarrow Bits$
{1}	**Construct-Data:**	$Bits \times Bits \Rightarrow \langle Data \rangle$
{1}	**Compress:**	$Comp_1 \times Comp_2 \Rightarrow Smash$
{1}	**Expand:**	$Smash \times Bits \Rightarrow \langle Data \rangle_1 \vee \emptyset, \langle Data \rangle_2 \vee \emptyset, \ldots$
{1}	**Arithmetic Relationals:**	$Arith_1 \times Arith_2 \Rightarrow Bool$
{1}	**Non-Arithmetic Relationals:**	$Comp_1 \times Comp_1 \Rightarrow Bool$ $Any_1 \times Any_2 \Rightarrow Bool$
{1}	**Form-Address:**	$ISD \times Int \Rightarrow ISA$[12]
{3}	**I-Fetch:**	$ISA \Rightarrow$ $\langle d=1,PE_1,chain=1,\langle d=0,PE_2,\ldots\rangle,\langle$*I-Fetch...$\rangle\rangle$
{3}	**Form-Address-I-Fetch:**	$ISD \times Int \Rightarrow$ $\langle d=1,PE_1,chain=1,\langle d=0,PE_2,\ldots\rangle,\langle$*I-Fetch...$\rangle\rangle$
{2} or {3}	**I-Store:**	$ISA \times Comp \Rightarrow$ $\langle d=1,PE_1,chain=0,\langle$*I-Store...,$\langle\ldots,Comp\rangle\rangle\rangle$ or $\langle d=1,PE_1,chain=1,\langle d=0,PE_2,\ldots\rangle,\langle$*I-Store...,$\langle\ldots,Comp\rangle\rangle\rangle$
{2} or {3}	**Form-Address-I-Store:**	$ISD \times Int \times Comp \Rightarrow$ $\langle d=1,PE_1,chain=0,\langle$*I-Store...,$\langle\ldots,Comp\rangle\rangle\rangle$ or $\langle d=1,PE_1,chain=1,\langle d=0,PE_2,\ldots\rangle,\langle$*I-Store...,$\langle\ldots,Comp\rangle\rangle\rangle$
{4}	**Allocate:**	$ISD \times Int\text{-}8 \times CObj/MObj \Rightarrow$ $\langle d=1,PE_1,chain=1,\langle d=2,CObj/MObj\rangle,\langle$*Allocate...,$\langle\ldots\rangle\rangle\rangle$
{4}	**Deallocate:**	$ISD \times Int\text{-}8 \times CObj/MObj \Rightarrow$ $\langle d=1,PE_1,chain=1,\langle d=2,CObj/MObj\rangle,\langle$*Deallocate...,$\langle\ldots\rangle\rangle\rangle$
{3}	**Allocate-Cobj:**	$Int\text{-}32 \times MObj \times Int\text{-}8 \Rightarrow$ $\langle d=2,PE_1,chain=1,\langle d=0,PE_2,\ldots\rangle,\langle$*Allocate-CObj,$\langle MObj,counter\rangle\rangle\rangle$
{3}	**Deallocate-Cobj:**	$CObj \Rightarrow$ $\langle d=2,PE_1,chain=1,\langle d=0,PE_2,\ldots\rangle,\langle$*Deallocate-CObj,$\langle CObj\text{-}addr\rangle\rangle\rangle$

---

[12] ISD and ISA are used generically in these semantic forms. ISA represents the types ISA-T-Fix, ISA-T-Var and ISA-U-Fix. ISD represents ISD-T-Fix, ISD-T-Var and ISD-U-Fix.

{2}	**Decrement-Cobj:**	$CObj \Rightarrow$ $\langle d=2, PE_1, chain=0, \langle$*Decrement-CObj,$\langle$CObj-addr$\rangle\rangle\rangle$
{1}	**D and D^{-1}:**	$Comp_1 \Rightarrow Comp_1$
{1}	**R:**	$Comp_1 \times Smash \Rightarrow Comp_1$
{1}	**R^{-1}:**	$Comp_1 \times Int\text{-}8 \Rightarrow Comp_1$
{3}	**Read-Byte:**	$Int\text{-}32 \Rightarrow$ $\langle d=2, PE_1, chain=1, \langle d=0, PE_2, ...\rangle, \langle$*Read-Byte,$\langle$addr$\rangle\rangle$
{3}	**Write-Byte:**	$Int\text{-}32 \times Bits \Rightarrow$ $\langle d=2, PE_1, chain=1, \langle d=0, PE_2, ...\rangle, \langle$*Write-Byte,$\langle$addr,data$\rangle\rangle$
{3}	**Transfer:**	$Int\text{-}32 \times Int\text{-}32 \times Int\text{-}24 \Rightarrow$ $\langle d=2, PE_1, chain=1, \langle d=0, PE_2, ...\rangle, \langle$*Transfer,$\langle$src,dest,PE,length$\rangle\rangle\rangle$
{3}	**Input-Block:**	$Int\text{-}32 \times Int\text{-}24 \times Int\text{-}8 \Rightarrow$ $\langle d=2, PE_1, chain=1, \langle d=0, PE_2, ...\rangle, \langle$*Input-Block,$\langle$addr,length,device$\rangle\rangle$
{3}	**Output-Block:**	$Int\text{-}32 \times Int\text{-}24 \times Int\text{-}8 \Rightarrow$ $\langle d=2, PE_1, chain=1, \langle d=0, PE_2, ...\rangle, \langle$*Output-Block,$\langle$addr,length,device$\rangle\rangle$
{3}	**Exit:**	$Comp_1 \times MObj \Rightarrow$ $\langle d=1, PE_1, chain=0, \langle$*Exit,$\langle$MObj-addr,init-number,Comp$\rangle\rangle\rangle$
{4}	**Write-Code-Block-Register:**	$Int\text{-}8 \times Smash \times MObj/Cobj \Rightarrow$ $\langle d=2, PE_1, chain=1, \langle d=2, MObj/Cobj\rangle, \langle$*Write-C-B-Reg,Smash$\rangle\rangle$
{1}	**Identity:**	$Any_1 \Rightarrow Any_1$
{1}	**Switch:**	$Any_1 \times Bool \Rightarrow Any_1 \vee \emptyset$
{3}	**Set-Supervisor-Mobj:**	$Int\text{-}8 \times MObj \Rightarrow$ $\langle d=2, PE_1, chain=1, \langle d=0, PE_2, ...\rangle, \langle$*Set-supervisor-MObj,$\langle$MObj$\rangle\rangle\rangle$
{2} or {3}	**Use:**	$MObj \times Comp \Rightarrow$ $\langle d=2, PE_1, chain=0, \langle$*Entry,$\langle$MObj,Comp$\rangle\rangle\rangle$ or $\langle d=2, PE_1, chain=1, \langle d=0, PE_2, ...\rangle, \langle$*Entry,$\langle$MObj,Comp$\rangle\rangle\rangle$

References

1. Arvind, and R. E. Bryant. Design Considerations for a Partial Differential Equation Machine. Scientific Computer Information Exchange Meeting, September, 1979, pp. 94-102.

2. Arvind. Decomposing a Program for Multiple Processor System. Proceedings of the 1980 International Conference on Parallel Processing, August, 1980, pp. 7-14.

3. Arvind, and D. E. Culler. Tagged Token Dataflow Architecture. Tech. Rep. 229, Computation Structures Group, Laboratory for Computer Science, MIT, Cambridge, Mass., July, 1983.

4. Arvind, and R. A. Iannucci. Instruction Set definition for a Tagged-token Dataflow Machine. Tech. Rep. 212-3, Computation Structures Group, Laboratory for Computer Science, MIT, Cambridge, Mass., February, 1983.

5. 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.

6. Arvind, and R. A. Iannucci. A Critique of Multiprocessing von Neumann Style. Proc. of the 10th International Symposium on Computer Architecture, June, 1983.

7. Dennis, J. B. First Version of a Dataflow Procedure Language. Memo 93, Computation Structures Group, Laboratory for Computer Science, MIT, Cambridge, Mass., November, 1973. revised MAC TM61, May, 1975

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

9. 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.

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