LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

A Multiple Processor Dataflow Machine that Supports Generalized Procedures

Computation Structures Group Memo 205
February 1981
Revised June 1981

Arvind
Vinod Kathail

This paper was presented at the 8th Annual Architecture Conference
in Minneapolis, MN, May 1981.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

A Multiple Processor Dataflow Machine that Supports Generalized Procedures

Arvind
Vinod Kathail

Laboratory for Computer Science
Massachusetts Institute of Technology
545 Technology Square
Cambridge, Mass. 02139.

Abstract

Programs for data flow machines are written in functional languages some of which require efficient support for dynamic procedure invocation to achieve high performance and programming flexibility. Among the proposed data flow machines, few support procedures in any generality. Our machine, which is a hardware realization of the U-interpreter for data flow languages, provides support for a variety of procedure calling conventions. Because the U-interpreter assigns a unique activity name to each instance of a computation (activity), an activity name may become arbitrarily large in the case of nested or recursive procedure calls. Hardware considerations, however, require that an activity name be represented by a fixed-size tag. We describe a mechanism that uses fixed-size, reusable tags in hardware. Like processor and memory resources, a group of tags is allocated and deallocated for each procedure activation. The proposed mechanism passes procedure arguments and results efficiently given the distributed environment of our machine.

Key Words : Data Flow, Multiple Processor Machines, Parallelism, Functional Languages.

This research was supported in part by NSF grant No. MCS 7902782, and in part by the Advanced Research Project Agency of the Department of Defense, Office of Naval Research contract No. N00014-75-C-0661.

1 Introduction

The data flow model of computation is one basis for exploiting massive amounts of parallelism present in many important computer applications. Prototype computers based on data flow principles are in various stages of development at several laboratories in the United States [5, 7, 8, 10, 11] in Europe [13, 14] and in Japan [3, 12]. Architectures of these machines uses "arrival of operands" as the main mechanism for controlling instruction execution and rely on functional languages (such as Lisp, Id [4], and Val [2]) to express programs for these machines. These projects also show a remarkable diversity in actual hardware structures that are used (or proposed). The diversity stems primarily from three essential features of data flow machines :

1. The mechanism for detecting enabled instructions
2. The mechanism for scheduling instructions on processors
3. The mechanisms for handling data structures

The machine that we are designing[1] consists of N identical processing elements (PE's) communicating via a bit-serial packet communication network (Figure 1). Each PE is essentially a complete computer that includes a private program and data storage of up to 128K bytes, and a floating point arithmetic and logic unit (ALU). A prototype of this machine that will have a minimum of 64 processing elements is being designed; however two important design goals are (1) to incorporate up to several thousand PE's simply by "plugging in" a group of PE's and (2) to show improvements in performance proportional to the number of PE's in the machine.[2]

2 The U-interpreter, and Activity Names

Our machine is a hardware implementation of the U-interpreter for a graphical data flow base language [4]. Programs written in any functional language which can be compiled into this base language may be executed on our machine. Such a compiler for the high-level data flow language Id [4] is already in use. The U-interpreter uncovers parallelism in programs during execution by uniquely labeling independent activities as they are generated. Each instance of execution of an operator is called an *activity*, and is given a unique *activity name*. Activities that have all their input values available can execute provided a processor is available. An activity name contains four fields :

---

[1] The reader should not confuse our machine with the data flow machine of J. Dennis, which is also being designed and constructed at the M.I.T. Laboratory for Computer Science.

[2] The performance goal is meaningful only for those applications that have an *inherent parallelism* greater than the number of PE's in the machine. Many applications in the areas of partial differential equation simulation, scene and vision analysis, and simulation of VLSI designs satisfy this criterion.

- **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 of u 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.

Within a U-interpreter machine, each value v is combined with its destination activity name into a packet called a *token*, denoted by $\langle u.c.s.i, v \rangle_{port}$ in which the term *port* identifies the input port to an operator. The basic operation of the machine is to bring together tokens with identical names, to execute the desired operation, and to generate output token(s) holding the result value(s) along with their destination activity names. For example, two data values x and y with destination activity name u.c.s.i are intended for the i^{th} execution of instruction s of code block c which was invoked in some context u. When these two tokens get together, the code (i.e. instruction s of code block c) is fetched, the operation indicated by the instruction is performed, and result data values with new destination activity names are generated. In most cases, these new names are derived simply by changing the statement number part of an input token's activity name. In the previous example if c.s refers to +, and the output of s is connected to an input of t then the result data token will have the value x+y and the destination activity name u.c.t.i. Base language operators for implementing procedures (A, A^{-1}, BEGIN, and END) and loops (L, L^{-1}, D, and D^{-1}), however, manipulate activity names in such a way that these names may become arbitrarily large. The operators related to procedure application are briefly described below.

According to the semantics of Id language, all arguments for a procedure are combined together into a structure, and the structure is passed as a single argument to the procedure. Similarly, all results are returned within a single structure. The first instruction of every procedure is the operator BEGIN, which is given the statement number *begin*; the last instruction is always operator END, with the statement number being *end*. A, the operator to activate a procedure, expects two inputs : Q, a procedure definition, and x, a structure of arguments (Figure 2). A passes x to the BEGIN operator of Q, and BEGIN distributes x within a new activation of Q. The END operator of Q ultimately returns the structure containing results to A^{-1}, the procedure terminate operator. A^{-1} distributes these results within the environment that invoked Q.

Suppose A receives[3] tokens $\langle u.c.s_A.i, Q \rangle_{proc}$ and $\langle u.c.s_A.i, x \rangle_{arg}$. The output tokens of A will refer to c_Q, the code block corresponding to procedure Q. Since data flow languages are purely applicative languages, Q is

[3] $c.s_A$ refers to an A operator; $c.s_T$ refers to the companion A^{-1}.

like a mathematical function : it has no internal memory and it cannot affect or be affected by any other procedure in the machine. Hence, we create a new context to execute Q and execute all activities belonging to this invocation of Q independently of all other activities. The A operator does this by sending x to the first operator of Q by assigning it the activity name u'.c_Q.begin.1 where u' equals u.c.s_T.i. When the END operator of Q is ready with the result structure, it sends the result structure to A^{-1} in the old context u. The proper activity name for A^{-1} is u.c.s_T.i, which is easily extracted from u'. c_Q.end.1.

For a thorough understanding of the U-interpreter, the reader is encouraged to read section 3 of Arvind et al [4]. The behavior of L and L^{-1} is very similar to A and END. The D operator is used to increment the i field which also may become arbitrarily large. In our machine, we use the same basic mechanism for implementing both procedures and loops. Before this mechanism is explained, however, (1) the way programs are mapped onto the machine and (2) the special type of storage used for passing arguments and results are reviewed in the following two sections, respectively.

3 Schemes for Distributing Activities on PE's

For the purpose of distributing activities on the machine, assume that operators that produce activity names with a new context part (for example, A, END, L and L^{-1}) are separated from the ones that do not. A group of PE's, known as a *physical domain*, is allocated whenever a procedure or a loop is invoked (A or L is executed). All activities of the invoked procedure (or loop) take place within the physical domain, except those activities which are caused by an operator that changes the context part. The activities of a procedure (loop) can be distributed within a physical domain on the basis of the instruction number or the iteration number of an activity name. Suppose a physical domain includes PE's with numbers from PE_{base} to PE_{base} + dom-size − 1 in which dom-size is the size of the physical domain. If the activities are being distributed on the basis of the s part of activity names then the destination PE number for an activity can be either PE_{base} + s *mod* dom-size or PE_{base} + (s/j) in which j is greater than or equal to the number of instructions in the code body divided by dom-size. Either scheme can distribute the code uniformly over the physical domain. Parallelism, however, could be severely limited because all initiations of an instruction will take place on one PE. If a similar mapping is done based on the i part of activity names, each PE in the domain will need a complete copy of the code, but as many as dom-size number of initiations can execute in parallel. It is also possible to combine the two schemes.

Once a physical domain and a mapping scheme have been selected the code needed by a PE is fixed and can be preloaded in the program memory of the PE. Further, if we assume that program graphs are stored using forward link pointers then static relocation will eliminate the need for evaluating the mapping function dynamically. Our implementation minimizes the number of addresses to be relocated statically by providing

base registers for relocating code. If only one procedure or loop is permitted to be active in a physical domain, there is no need for the tokens to carry the u and c parts of activity names. In fact, if the code is being preloaded, the c part is absorbed by the forward link pointers. In case an implementation permits more than one invocations (but a fixed maximum number of invocations) in a physical domain, the tokens will have to carry extra bits to signify which invocation they belong.

The scheme for mapping programs discussed in this section needs a scheduler to allocate physical domains. The scheduler is called when a procedure (loop) is invoked, and it selects a domain by taking into account such factors as how many activities are expected to be generated, the size of the code block, if the code block is already present in some other physical domain, and most important how much data has to be moved between the invoking physical domain and the new physical domain. It appears at this stage that the scheduler would need hints from the user to perform its task optimally. The scheduler could be either a program (executed by predesignated PE or PE's) or a special-purpose processor.

4 I-Structures

In functional languages, modification of even one element of a data structure results conceptually in a new data structure [1]. Hence the meaning of a statement like x[i] ← v is *new* x ← *append* (x,i,v) where x and *new* x differ only in the value on selector i. The copying semantics for data structures in functional languages has to be preserved, or the advantage of these languages for parallel processing is lost. I-structures have been proposed to avoid excessive copying [6].

An I-structure is an array-like data structure with certain constraints on its creation. An element of an I-structure can be written into (i.e. defined) only once. Besides ensuring functionality, this restriction also makes it possible to do several concurrent writes on a single I-structure. We plan to implement all data structures as I-structures and provide special hardware storage for them in our machine. A cell in the I-structure storage can be read only after the cell has been written, and a cell can be written only if it is empty. A bit is associated with every cell to indicate the empty / full condition.

In our machine, an I-structure may be distributed over several PE's, but all I-structure storage forms a single address space. Thus, an I-structure pointer can be decomposed into two parts : a PE number and a local memory pointer. An attempt to read or write I-structure storage can result in tokens going from one PE to another PE. Consider the select operation x[k] which requires x_0, the address of the first element of x, and integer k.[4] As shown in Figure 3, the processor executing x[k] sends a read request to I-structure memory

[4]The scheme actually used in the machine is more general : x_0 is treated as an array descriptor with enough information to determine which PE has the x[k] element given k.

location $x_0 + k - 1$ along with the activity name where the result should be sent. Note that the PE in which x[k] is executed can be different from the PE that has the memory location $x_0 + k - 1$ as well as the PE that executes the destination activity of x[k].

An append operation (append (x, k, v)) is similarly broken in two steps : one to form an address from x_0 and k; the other to send value v to the address calculated. An append operation has no destination, because several appends together create one I-structure whose descriptor is forwarded separately to select instructions. I-structures are allocated by a memory manager, which does reclamation of storage by maintaining reference count of I-structure descriptors.

5 Program Representation

Each loop and each procedure of an Id program is compiled as a separate code block. An individual instruction is uniquely identified by a code block number and an offset within the code block. The format for a typical instruction is shown in Figure 4 in which *opcode* is an 8 bit operation code. Each instruction provides space for constants and the destinations where result values are to be sent. The *c* bit is used to chain the destination list; a 0 *flag* indicates an empty destination list. Each destination consists of *s,p,nt,* and *af* fields:

- *s* : The relative address of the destination instruction
- *p* : A one-bit number indicating the port of the destination operator (An operator is allowed to have at most two inputs.)
- *nt* : A one-bit number indicating the number of tokens to enable the destination instruction
- *af*: The mapping scheme to be used in calculating the destination PE number.

Fields *wc* and *pc* specify the addressing mode and the port number for the constant values, respectively. A constant may be stored in the instruction itself or in a block (called an *activation record*) associated with each invocation of a code block. An activation record has space for input arguments and output results and is allocated at the time of procedure invocation.

6 The Processing Element

Activity names are represented by fixed size tags in the machine. The manipulation of tags by the machine is isomorphic to the manipulation of activity names by the U-interpreter; however, the exact correspondence between an activity name and a tag is not easily described, because the correspondence involves the architecture of the machine and the schemes for mapping programs on the machine simultaneously.

We assume that the local program memories of all PEs (like the I- structure storage) constitute a single address space. A complete memory address, therefore, has two parts : a PE number and a local memory address within the PE. A tag must contain the address of the instruction to be executed. A processing element assumes that the instruction indicated by the tag of an incoming token has been loaded in the local memory before its arrival. This permits using physical addresses as parts of tags and, consequently, a part of a tag to determine the destination PE number. Two kinds of tokens that are transmitted between PE's are tokens corresponding to values in data flow graphs and tokens generated by the system.

6.1 Tokens Corresponding to Values in Data Flow Graphs

The first kind of tokens are those tokens that correspond one to one with the tokens in data flow graphs. These tokens are referred to as $d = 0$ type and contain the following fields:

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

in which the tag can be further subdivided into three fields: *color, local instruction address,* and *iteration number.* Note that the PE number is logically a part of the complete tag. It is used by the communication system to route tokens and is deleted when a token enters PE. When a token arrives at the input of PE (Figure 5) the token's *d* and *nt* fields are examined. Tokens with $d = 0$ that need a partner ($nt = 1$) have their tags matched associatively against the tags of tokens in the waiting-matching buffer. In case a match is found both tokens are moved to the instruction-fetch buffer; otherwise, the incoming token is put in the waiting-matching buffer. In case the waiting-matching buffer is full, local memory is used to handle the overflow of tokens. If an incoming token does not need a partner ($nt = 0$, which means a single-operand instruction) it is directly moved to the instruction-fetch buffer. Using the instruction address bits of the tag, an instruction is fetched, and then a packet containing the operation code, operands, and destinations is passed on to the service section. The arithmetic and logic unit in the service section executes the operation code, and new tags are generated as dictated by the U-interpreter and the program mapping scheme being used. The result tokens are sent to the output buffer, which in turn sends them to other PE's through the communication network. If the destination PE on a token is the same as the source PE, a local path from the output buffer to the input register of a PE is used to reduce external communications traffic.[5]

6.2 System-Generated Tokens

There is a class of instructions (for example, a request to read the I-structure memory or to reset the ALU and other registers in a PE) whose behavior does not fit the pattern described previously. The operation codes for these instructions are carried within system-generated tokens ($d = 1$); such tokens are routed

[5]The U-interpreter was developed at the University of California at Irvine, and consequently, the architecture of our PE is related to the PE of the simulated machine of Gostelow and Thomas [9].

directly to the service section because no instruction fetch is needed. A $d = 1$ type of token contains the following fields:

<PE number, d = 1, opcode, data>

To understand how these tokens come about, consider again the high-level operation, *append* (x_0, i, v). The FORM-ADDRESS instruction calculates < PE_{num}, local-pointer>, an I-structure pointer, and forwards it to the I-STORE instruction on a $d = 0$ type token. Upon receiving a value v and <PE_{num}, local-pointer> the I-STORE instruction produces the following token:

<PE number = PE_{num}, d = 1, opcode = I-WRITE, data = <local-pointer, v>>

The PE with number PE_{num} performs the write operation, and no further tokens are generated. The select operation similarly forms an address <PE_{num}, local-pointer> and passes it to an I-FETCH instruction, which produces the following token:

<PE number = PE_{num}, d = 1, opcode = I-READ, data = <local-pointer, destination-info>>

where destination-info is the PE-number, tag, nt, and port of the token to be sent to the destination of the select. The contents of the I-structure location addressed by local-pointer is read by PE_{num}, which produces a d = 0 type of token using destination-info and the value read. Note that a separate section to hold deferred reads (requests to read an absent element of an I-structure) is provided to avoid blocking the service section (Figure 5). A generalization of the I-FETCH instruction is IN-FETCH (*indexed fetch*) instruction. The IN-FETCH instruction has n destinations and generates n (d = 1) type of tokens for reading n successive locations starting with the input I-structure pointer. Each value read is sent to a different destination.

6.3 Color Registers

We want to include the possibility of activating several code blocks (not necessarily distinct from each other) within a physical domain. This is achieved by assigning a different *color* to each activation.[6] Only a finite number of colors are allowed within a physical domain, and if all colors of a physical domain are being used, no new loop or procedure activation can be scheduled on it. Colors are released when a loop or procedure terminates. Sharing of code blocks within a physical domain is feasible, because all invocations carry different colors.

There are several *color registers* in each PE. Each color register contains two sets of base-and-limit pointers: one set of pointers points to the base and limit of the code block associated with a color; the other set points to the activation record in the local memory. Color registers are not necessary, but provide several benefits :

1. They allow the instruction address in a tag to be shorter, because it is made relative to the program

[6] The *color*, along with a completely logical entity called the *physical domain name*, represents the u part of an activity name.

base pointer. (The program base pointer also reduces the number of forward pointers that have to be statically relocated.).

2. The activation record pointer, together with the special addressing mode, avoids the need to circulate values that remain constant during a loop invocation. This reduces both the token traffic as well as the number of tokens in the waiting-matching buffer.

7 Implementation of Procedures

Whenever a procedure is invoked, memory for the activation record has to be allocated (conventional languages often use stack allocation for this purpose). It is preferable to allocate storage for an activation record in the physical domain of the invoked procedure to take full advantage of color registers. Id semantics require all input arguments to be present before a procedure can be invoked while Id loops, and many other languages relax this restriction [4]. The following scheme allows the compiler to implement either argument passing convention. Assume the compiler designates an input to a procedure as a trigger to call the scheduler which in turn calls a memory manager. Let this trigger be known as t_a. The scheduler will return a newly allocated activation record pointer, which can be used by the invoking domain to store the arguments. It makes sense to distinguish between the allocation of a domain, and the startup of the associated procedure, which is done by having the compiler generate another trigger called t_b to start procedure execution; however, it is not necessary for t_a and t_b to be different from each other or from a procedure input or a code block name. A trigger similar to t_b, called t_r, is also required by the invoked procedure to inform the invoking domain about the availability of results.

The steps involved in a procedure call and return (Figure 6) for the case when the size of the activation record can be determined at compile time are as follows :

1. The INVOKE instruction is executed whenever the code block name and trigger t_a become available. The INVOKE instruction sends the code block name, the size of the activation record, and the destination tag for the following DISTRIBUTE operator to the scheduler. The DISTRIBUTE instruction merely sends copies of its input to various destinations.

2. The scheduler either on its own or with some advice from the programmer/compiler, assigns a physical domain to the invoked procedure and performs four steps:

   a. It asks a memory manager to allocate an I-structure area in the new physical domain for the activation record of the called procedure and sends the resulting I-structure pointer to the DISTRIBUTE operator in the calling domain.

   b. It loads the code block on all PE's of the physical domain. The destination fields within the code block being loaded depend on the mapping scheme to be used for this invocation. If a copy of the code block is already present in the physical domain, this step is not performed. In general, code blocks reside permanently in secondary storage, and loading one of them can take significant time.

c. It assigns a color and stores the pointers for the loaded code block and the allocated local memory area in the appropriate color register of each PE in the physical domain.

d. It sends a token to read the first location of the activation record; this read will be delayed until the first location is actually written, because the activation record is an I-structure. The destination for the result of the read instruction is the IN-FETCH-CR (*indexed fetch through color register*) instruction[7] . By convention trigger t_b is written into the first location of the activation record; therefore, no new tokens in the activated procedure are generated until t_b is written by the calling domain.

3. As soon as the activation record pointer is received by the DISTRIBUTE instruction, it makes n+2 copies of it: one for writing t_b, n for writing n arguments in the activation record, and one for reading t_r from the activation record.

4. When indirectly triggered by t_b, the IN-FETCH-CR instruction reads n values from the activation record and sends them to various instructions as dictated by code block Q.

5. The invoked procedure writes m results in an I-structure area (not necessarily different from a separate part of the activation record) and passes a pointer to this area to the calling domain through the t_r location of the activation record.

6. Since the writing of t_r controls when the results are actually available to the calling procedure, the generation of trigger t_r must be according to some policy. For example, if the results should be returned only after all of them are available, the compiler has to put in code to detect that all m writes have been completed. In general, the condition that all results have been stored has to be detected to release a color. Releasing a color does not deallocate the activation record.

7. In the invoking domain, a read request for t_r is made as soon as the pointer for the activation record becomes available (step 3). This request is satisfied only after t_r is actually produced. The destination for the result of this read request is the IN-FETCH instruction.

8. The IN-FETCH instruction in the invoking domain reads m results and sends them to operators as dictated by the invoking code. After all the results have been read, the activation record is released.

If we want parameters stored in an activation record to be accessed as constants, it is possible that a copy of a part of the activation record may be needed by each PE in the physical domain. This copying will depend upon the mapping scheme employed and can be initiated by the scheduler.

---

[7] IN-FETCH-CR is the same as IN-FETCH, except that it uses the I-structure pointer from the color register.

8 Further Considerations -- Procedures and Loops

We have not discussed two additional problems related with procedure (loop) implementation. Allocation of storage for data structures implemented as I-structures is often coupled with procedure (loop) invocation; deallocation of I-structures is associated with procedure (loop) termination. In general, an I-structure can be distributed across several PE's; hence, an additional memory manager with a global view of the machine is needed for efficient storage management. Because the scheduler must interact with this memory manager to minimize data movement between the invoking and invoked physical domains, the scheduler's response in returning an activation record pointer may be slow; however, the compiler has the flexibility of executing INVOKE early by using the trigger t_a.

The mechanism described works for the case in which there is no mismatch in the number of parameters passed by the calling expression and expected by the called procedure. The Id language permits such a mismatch in case of procedures (but not loops) and gives precise rules for dealing with the situation. The basic rule is to neglect extra parameters and produce *error* values for those required but not supplied. A protocol for implementing such procedures first forms a structure of the arguments in the calling domain. Then, the scheduler in response to the INVOKE operation sends the number of expected parameters along with the activation record pointer. Some compiler-generated code can use this information to copy the appropriate number of arguments from the original argument structure into the activation record and store *error* values for missing arguments. A similar strategy is possible for returning results.

The mechanisms described in this paper are sufficient for implementing procedure and loop invocations correctly without sacrificing any generality. An additional operator called D is used inside loops to increment the i field, which can cause problems if there are more than a fixed maximum number of iterations. Setting i to 0 after a fixed maximum number of iterations can produce duplicate activity names and incorrect results. One way to solve this problem is to allocate a new color if D needs to increment i beyond the maximum. The new color register will have exactly the same setting as the old color register. When no tokens with the old color remain (all tokens with i less than or equal to the fixed maximum number of iterations have been produced and consumed), the old color can be released. This allocation of a new color can be done quickly, because no resources besides a color register are needed.

It is possible to have more than one scheduler in the machine by using some static allocation rule for dividing procedure activations among various schedulers. This has no impact on the hardware structure of our machine. For large numerical applications, however, scheduling to minimize data movement remains a significant research issue.

9 Current Status of the Machine

We are in the process of designing the processing element that will be implemented using one or two custom VLSI MOS chips and commercially available memory boards and ALUs. We are planning to use a 32 token waiting-matching buffer per PE and 36-bit tags : 10 bits for the PE number, 3 bits for color, 16 bits for an instruction address, and 7 bits for the iteration field. The validity of these sizes is to be determined by hardware and software simulation experiments. There are still a number of design details to be worked out, regarding especially the interface with commercial chips. We expect to fabricate a processing element by December 1982.

A compiler to translate Id language into Lisp is currently in use. A compiler to translate Id language to data flow machine language is being written and should be in use by May 1981.

Acknowledgments

The authors thank Keshav Pingali, for his participation in this project, and Bill Ackerman and Bob Thomas, for their valuable suggestions.

References

1. Ackerman, W. B. Data Flow Languages. AFIPS Conference Proceedings, Vol. 48, June, 1979, pp. 1087-1095.

2. Ackerman, W. B. and Dennis, J. B. VAL -- A Value - Oriented Algorithmic Language: Preliminary Reference Manual. TR -218, Laboratory for Computer Science, MIT, Cambridge, Mass., December, 1978.

3. Amamiya, Makoto et. al. Data Flow Machine Architecture. Internal Notes 1 and 2, Musashino Electrical Communication Laboratory, Nippon Telephone and Telegraph, May, 1980.

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

5. Arvind, Kathail, V., and Pingali, K. A Dataflow Architecture with Tagged Tokens. TM -174, Laboratory for Computer Science, MIT, Cambridge, Mass., September, 1980.

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

7. Davis, A. L. The Architecture and System Methodology of DDM1 : A Recursively Structured Data Driven Machine. Proceedings of the 5th Annual Symposium on Computer Architecture, April, 1978, pp. 210-215.

8. Dennis, J. B., Boughton, G. A., and Leung, C. K. C. Building Blocks for Data Flow Prototypes. Proceedings of the 7th Annual Symposium on Computer Architecture, May, 1980, pp. 1 - 8.

9. Gostelow, K. P., and Thomas, R.E. Performance of a Simulated Dataflow Computer. *IEEE Transactions on Computers C-29*, 10 (October 1980), 905-919.

10. Johnson. D., et. al. Automatic Partitioning of Programs in Multiprocessor Systems. COMPCON Spring 80, February, 1980, pp. 175-178.

11. Keller, R.M., Lindstorm, G., and Patil, S. A Loosely-Coupled Applicative Multi-Processing System. AFIPS Conference Proceedings, Vol. 48, June, 1979, pp. 613-622.

12. Suzuki, T., and Moto-oka. Control Structure of TOPSTAR and Its Evaluation. Proceedings of Annual Conference of Japan Information Processing Society, May, 1980, pp. 85-86.

13. Syre, J.C., Comte, D., and Hifdi, N. Pipelining, parallelism, and Asynchronism in the LAU System. Proceedings of the 1977 International Conference on Parallel Processing, August, 1977, pp. 87-92.

14. Watson, I., and Gurd, J. A Prototype Data Flow Computer with Token Labeling. AFIPS Conference Proceedings, Vol. 48, June, 1979, pp. 623-628.

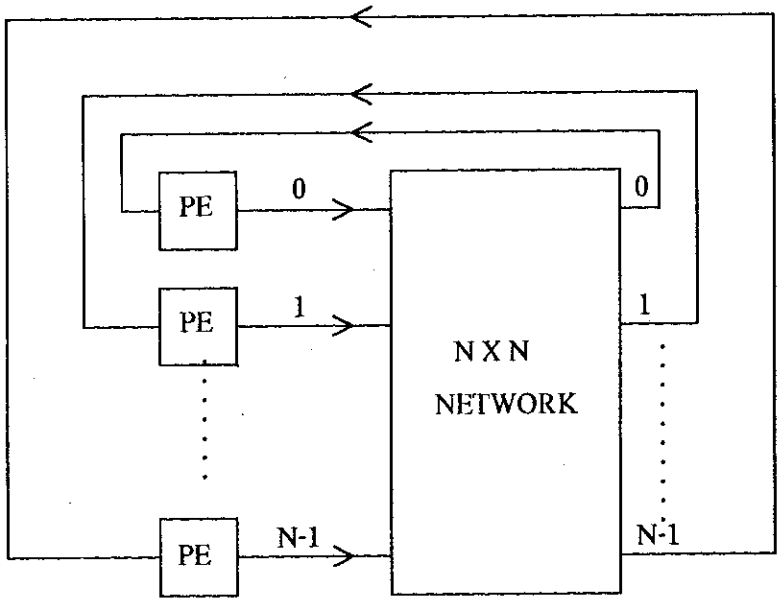


Figure 1: Block Diagram of Machine

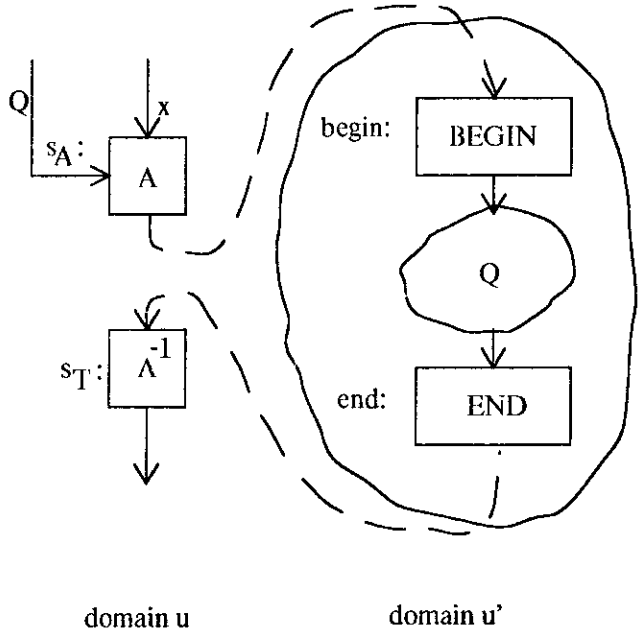


Figure 2: Procedure Application

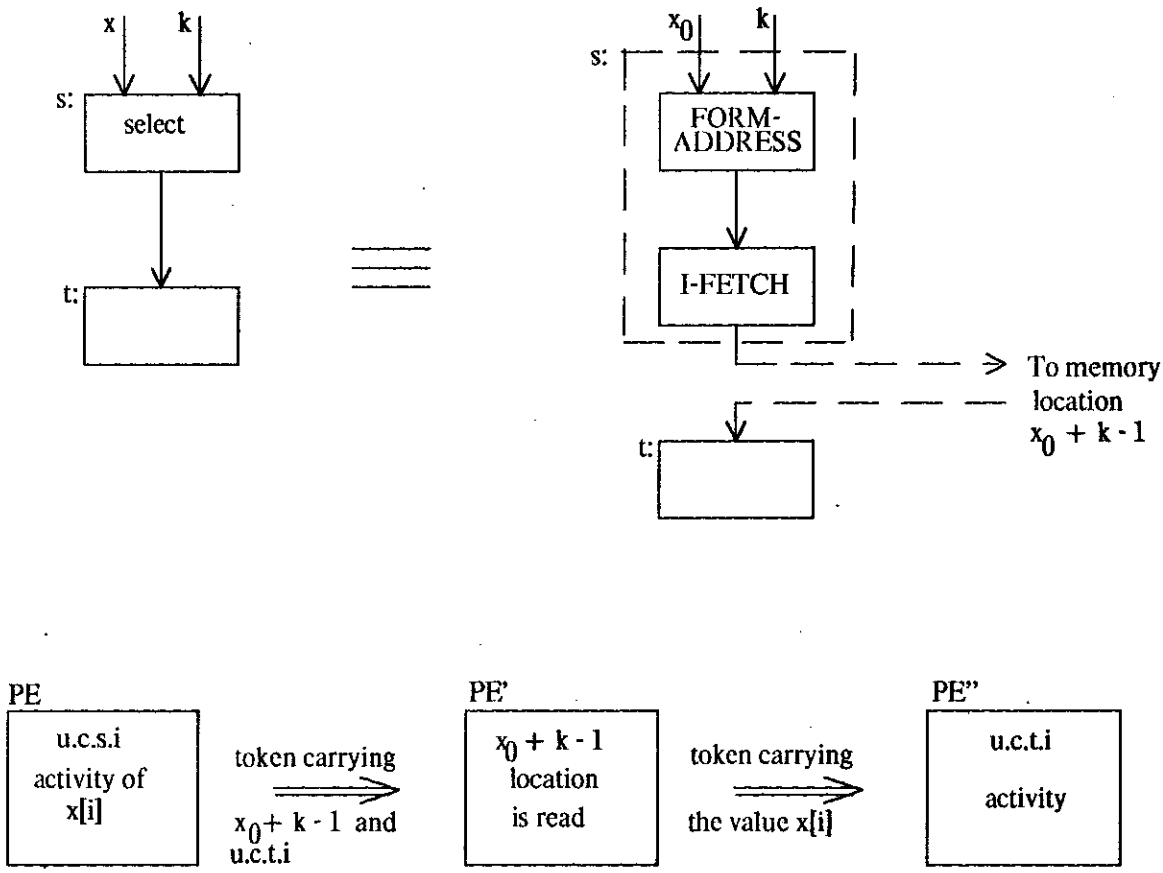
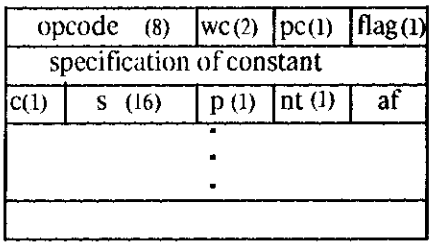


Figure 3: Implementation of Select Operation



opcode - operation code

wc - addressing mode for constant operand

pc - port number for constant operand

flag - 0 flag indicates empty destination list

c - chain bit for destination list

s - relative address of destination instruction

p - port number

nt - number of tokens expected by destination instruction

af - mapping scheme to be used

Figure 4: Instruction Format

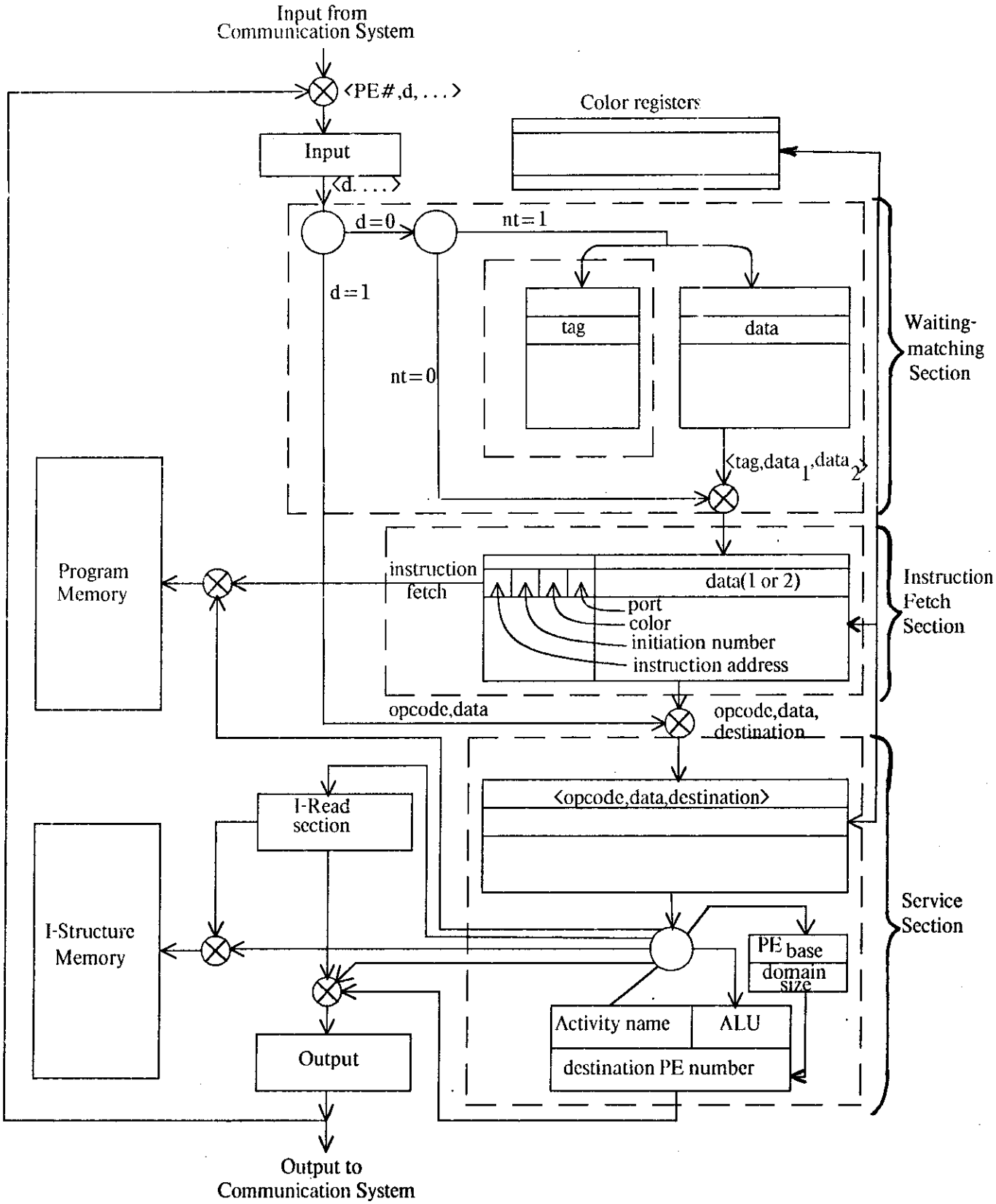


Figure 5: Processing Element

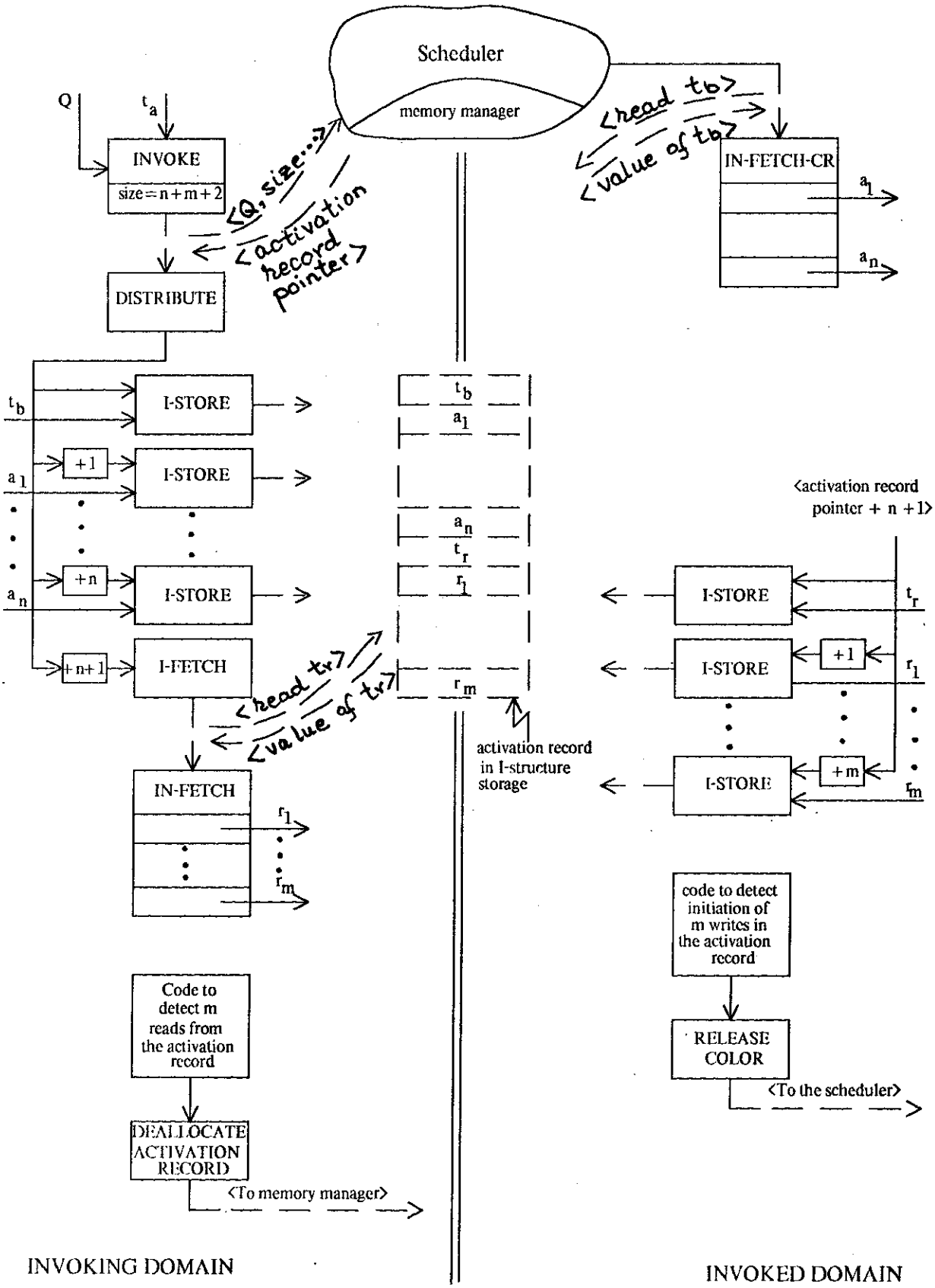


Figure 6: Code for Procedure Call