Instruction Set Definition for a Tagged-Token Data Flow Machine

Computation Structures Group Memo 212
10 December 1981

Arvind

Robert A. Iannucci

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

Abstract

This report focuses on the definition of the instruction set for a tagged-token data flow machine. This definition will serve to establish the interface between the compiler (in this case, ID) and the machine itself.

The definition is broken down into three parts:

1. **Tokens:** A detailed description of the tokens is given wherein cases are enumerated, and sub-fields are defined.
2. **Instructions:** A clear delineation is made between instructions and the operations they denote. Rules for forming an instruction are given.
3. **Operations:** The set of operations is defined. For each operation, we define the domain of valid inputs and describe the range of output values (including the production of error tokens where appropriate).

Key words and phrases: data flow, instruction set, multiple processor systems

Table of Contents

1. Introduction 1
2. Tokens 3
    2.1. Tokens Corresponding to Values in Data Flow Graphs 3
    2.2. System-Generated Tokens 6
3. Instructions 6
    3.1. Format 9
    3.2. Examples 10
    3.3. Instruction Processing Paradigms 11
        3.3.1. Paradigm 1: <d=0> to <d=0> 11
        3.3.2. Paradigm 2: <d=0> to <d=1> 13
        3.3.3. Paradigm 3: <d=0> to (<d=1> to <d=0>) 13
        3.3.4. Paradigm 4: <d=0> to (<d=1> to <d=1,*Entry>) 13
4. Operations 14
    4.1. Data Types 14
    4.2. The Arithmetic Operations 18
        4.2.1. Addition and Subtraction ($+_c, +_{nc}, -_c, -_{nc}$) 18
        4.2.2. Multiplication and Division ($\times_c, \times_{nc}, \div_c, \div_{nc}$) 18
        4.2.3. Explicit Coercions 19
    4.3. The Boolean Operations 19
        4.3.1. Monadic ($\neg$) 19
        4.3.2. Dyadic ($\wedge, \vee, \oplus$) 19
    4.4. The Bit String Operations 20
        4.4.1. Monadic ($\neg_1, \neg_2$) 20
        4.4.2. Dyadic ($\wedge, \vee, \oplus$) 20
        4.4.3. Shift 20
        4.4.4. Concatenate 20
        4.4.5. Adjust-length 21
    4.5. The Extract/Construct Operations 21
        4.5.1. Extract-type, Extract-value 21
        4.5.2. Construct-data 21
    4.6. The Relational Operations 22
        4.6.1. Arithmetic ($\leq, <, =, \neq, >, \geq$) 22
        4.6.2. Non-arithmetic ($\leq, <, =, \neq, >, \geq$) 22
        4.6.3. Non-Arithmetic ($\neq_{lax}$) 22
    4.7. The I-Structure Operations 23
        4.7.1. I-structure Implementation 23
        4.7.2. Form-Address 23
        4.7.3. I-Fetch 23

4.7.4. *I-Fetch-ME; a d=1 Operation 24
4.7.5. *I-Fetch-UE; a d=1 Operation 24
4.7.6. Form-Address-I-Fetch 24
4.7.7. I-store 25
4.7.8. *I-Store-ME; a d=1 Operation 25
4.7.9. *I-Store-UE; a d=1 Operation 25
4.7.10. *I-Store-Ack; a d=1 Operation 26
4.7.11. Form-Address-I-Store 26
4.8. Control Operations 26
4.8.1. Compress 27
4.8.2. Expand 27
4.8.3. *Entry; a d=1 Operation 28
4.8.4. Exit 29
4.8.5. Reset-I-Structure-Memory 29
4.8.6. *I-reset, a d=1 Operation 31
4.8.7. Write-Color-Register 31
4.8.8. *Write-color-register; a d=1 Operation 31
4.8.9. Iteration - D and D^{-1} 31
4.8.10. Identity 32
4.8.11. Switch 33
4.9. The USE Operation 33
5. Conclusion 35

Instruction Set Definition for a Tagged-Token Data Flow Machine

1. Introduction

This paper defines the instruction set for the tagged-token data flow machine being constructed by the Functional Languages and Architectures group at MIT. This machine is of the form suggested by Arvind and Kathail [3]. Management and scheduling of enabled activities is a realization of the mechanism for the U-interpreter [2] and [1]. A simulation study of a similar machine was done by Gostelow and Thomas [4]. Shimada [7] has also designed a simulator and has done preliminary work toward defining the instruction set presented herein.

It is assumed that the reader already has some familiarity with the machine described in [3]. The block diagram of this machine is shown in Figure 1. The *waiting-matching* section receives tokens from the communication network, and attempts to pair them. Having accomplished this, the *instruction fetch* section retrieves information from the *program memory* and passes this (along with token data) to the *service section.* Arithmetic operations are performed by the ALU (contained in the service section), while all I-structure processing is handled by a separate section. Result tokens are generated and propagated by the *output section.* In the following discussion, we will use these terms:

- **Enabled Activity:** Refers to the output of the waiting-matching section; a single token (nt=0), or token pair that has been successfully matched.

- **Instruction:** Refers to that information in the implementation corresponding to an actor in the data flow program graph. The instructions may be viewed as the fundamental pieces of the "stored program".

- **Operation Packet:** Refers to that information in the implementation that is built from an enabled activity and the corresponding instruction. The operation packet is the vehicle for communicating a complete *activity* (an instance of an instruction) to the service section.

- **Operation:** Informally referred to as the *opcode.* It is the part of the operation

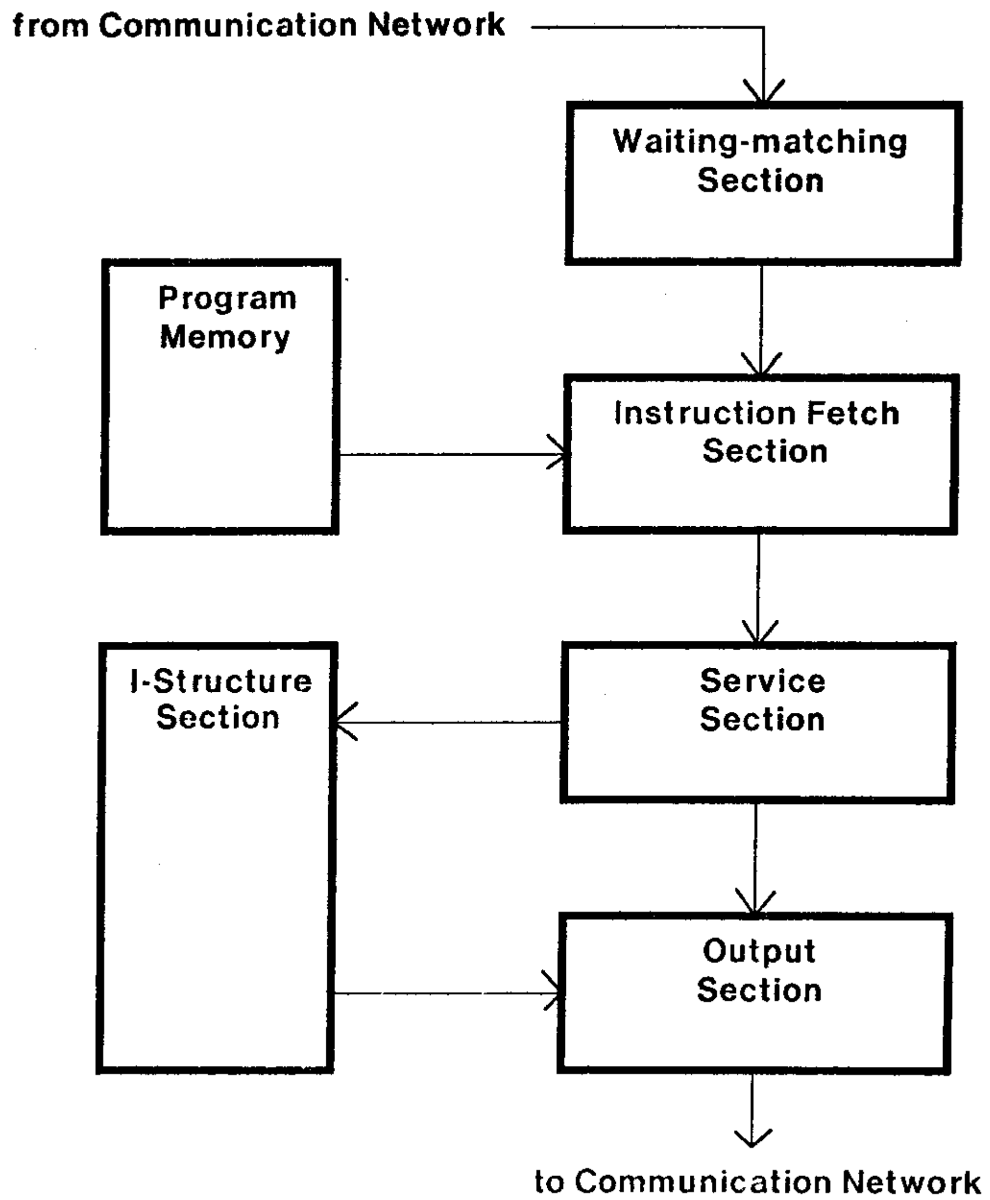

Figure 1: PE Block Diagram

packet which defines the action to be taken by the service section. The result will be a new token to be distributed as per a destination instruction list. The produced token will contain a valid result or a special datum indicating that an error has occurred.

The next three sections describe tokens, instructions, and operations (respectively).

2. Tokens

Arvind and Kathail identify two token types in their definition of the processor's architecture [3]:

1. **Tokens corresponding to values in data flow graphs:** These are associated with the conceptual notion of tokens (those that are viewed as moving about on the arcs of the data flow graph). They are to be distinguished from

2. **System-generated tokens:** This class of tokens is closely tied to the specific implementation under investigation and has no direct interpretation in terms of the data flow graph. Tokens in this class do not "denote" instructions to be enabled; rather, they *carry with them* the necessary operation codes.

We will describe each of these token classes in turn.

2.1. Tokens Corresponding to Values in Data Flow Graphs

The bit-patterns for tokens of this class can be broken down as follows:

```
<Token-type>                                        (1)
<PE-number>                                         (8)
<Tag>
   <Color>                                          (4)
   <Local-instruction-address>                      (16)
   <Iteration-number>                               (8)
<Number-of-tokens-to-enable-instruction>            (1)
<Port-number>                                       (1)
<Data>
   <Data-type>
      <Data-length>                                 (4)
      <Data-class>                                  (4)
   <Data-value>                                     (0-64)
```

The numbers to the right of the elemental entries indicate the number of bits reserved for expressing the corresponding field. Let us examine each of these fields.

1. **Token-type:** This bit (called *d* in [3]) is used to distinguish the tokens corresponding to values in data flow graphs from the system-generated tokens. This description pertains to **d=0** type tokens.

2. **PE-number:** This field is used to uniquely identify the Processing Element which is the target for any given token. The communications subsystem will perform the token-switching function based on this field[1]. Note that once a token has arrived at the selected PE, the *PE-number* is of no further use (save for checking of proper delivery) and can thus be discarded.

3. **Tag:** When combined with the PE number, the *tag* constitutes a unique activity name, and is derived from the U-interpreter token labeling rules. The tag is further broken down:

   a. **Color:** Each PE in the machine is capable of concurrently handling tokens from a fixed number of disjoint procedure/loop invocations. To keep the tokens separate, they are assigned identifiers that are common within the procedure/loop but unique across procedures/loops. The field that implements this is called the token's *color.*

   b. **Local-instruction-address:** Tokens of the d=0 class denote instructions corresponding to actors in the data flow program graph. This field serves as a pointer to the instruction so denoted.

   c. **Iteration-number:** This field is copied directly from the U-interpreter's *<u.c.s.i>* token label. It is a (finite) implementation of the iteration counter[2,3].

4. **Number-of-tokens-to-enable-instruction:** Each actor in the data flow graph will

---

[1] Further encoding of this field may be necessary to convert it into a proper routing address.

[2] However, this does not imply that the *i* field in *<u.c.s.i>* can not be larger than 2^8. A scheme described in [3] essentially calls for allocating a new color when the limit is reached.

[3] For procedure activations, the *iteration-number* field does not change; and, hence, 8 tag bits are wasted. If the same color and different iteration numbers are assigned to several activations of a procedure, then it is possible to make very efficient use of tag bits in case of recursion.

have at least one and at most two input tokens[4]. In the case that two tokens are needed to enable an instruction, the *waiting-matching* section of the targeted PE will have to perform the pairing function. However, when only one token is expected (this is known at compile time), the waiting-matching section can be bypassed. By carrying such information on the token itself, it is a simple matter to conditionally bypass the waiting-matching hardware. This field performs exactly that function.

5. **Port-number:** As mentioned, an actor (instruction) will have one or two inputs. In the case that two inputs are expected, the token itself must know for which of the two it is destined.

6. **Data:** This field is made up of the *Data-type* and the actual *Data-value* to be used by the target instruction.

   a. **Data-type:** Data objects may represent different items (e.g, booleans, I-structure descriptors). The *Data-type* field describes the datum by defining its *Data-class* which serves to make this distinction. Further, the implementation makes use of the fact that not all data objects require the same amount of storage space. As such, the type must also define the length of the actual data field. We have chosen an implementation architecture which supports addressing to the byte (8 bits) level; furthermore, we wish to permit a maximum length of 16 bytes (one byte for the <Data-type> and 15 bytes for the <Data-value>). Hence, four bits are required to express the length ("0000" denotes a <Data> object with no <Data-value> field; "0001" denotes a one byte <Data-value>; "1111" denotes a 15 byte <Data-value>). This information is carried in the *Data-length* field. It is important to observe that the *Data-length* field describes the number of implementation bytes that are used to store the object. The length of the actual datum may be further restricted (i.e., some bits may be ignored) by the *Data-class* field. This is seen most clearly in the case of boolean data. The *Data-length* indicates "0001" (one byte); but knowledge that this byte is used to represent a boolean value would cause seven of these eight bits to be ignored. The *Data-type* composite is further discussed in section 4.1 on page 14.

---

[4]More precisely, actors in the graph may have from one to three inputs; these inputs may come from a maximum of two input tokens along with an optional constant. This is explained in greater detail in the Instruction section.

b. **Data-value:** This field (whose length is defined as noted) contains the actual data object.

2.2. System-Generated Tokens

System-generated tokens are somewhat simpler in (generic) structure but tend to have varied formats in specific contexts. The basic format is

```
<Token-type>                                    (1)
<PE-number>                                     (8)
<Ack-expected>                                  (1)
<Data>                                          (*)
```

Again, we examine each field.

1. **Token-type:** System-generated tokens are of type **d=1**. Note that, unlike d=0 tokens, d=1 tokens never require pairing (i.e., each one stands alone). As such, all d=1 tokens bypass the waiting-matching and instruction fetch sections of the machine (refer to Figure 2).

2. **PE-number:** This field is identical to the corresponding field previously described.

3. **Ack-expected:** The operation performed by this token may generate another token (e.g., an *acknowledgement*). Such a situation is indicated by this bit. The process of "generating another token" is described in section 3.3 on page 11.

4. **Data:** The meaning assigned to *Data* for a d=1 token varies as a function of the operation code that it carries. In some cases, this field will contain a datum that is similar to that for a d=0 token. In other cases, this field may actually contain the encodings of several discrete items in a way that would defy some "standardized" description. Since d=1 tokens are an implementation-level device, we can see that the *Data* field contained therein is also implementation related. The field may be as large as the remiander of the token (i.e., 128-total of all other bits).

3. Instructions

Instances of "firings" of actors in the data flow program graph are referred to as *activities.* An activity is described in terms of the data which invokes it and the "skeleton" that defines the actor. In this paper we have used the term *instruction* to describe that

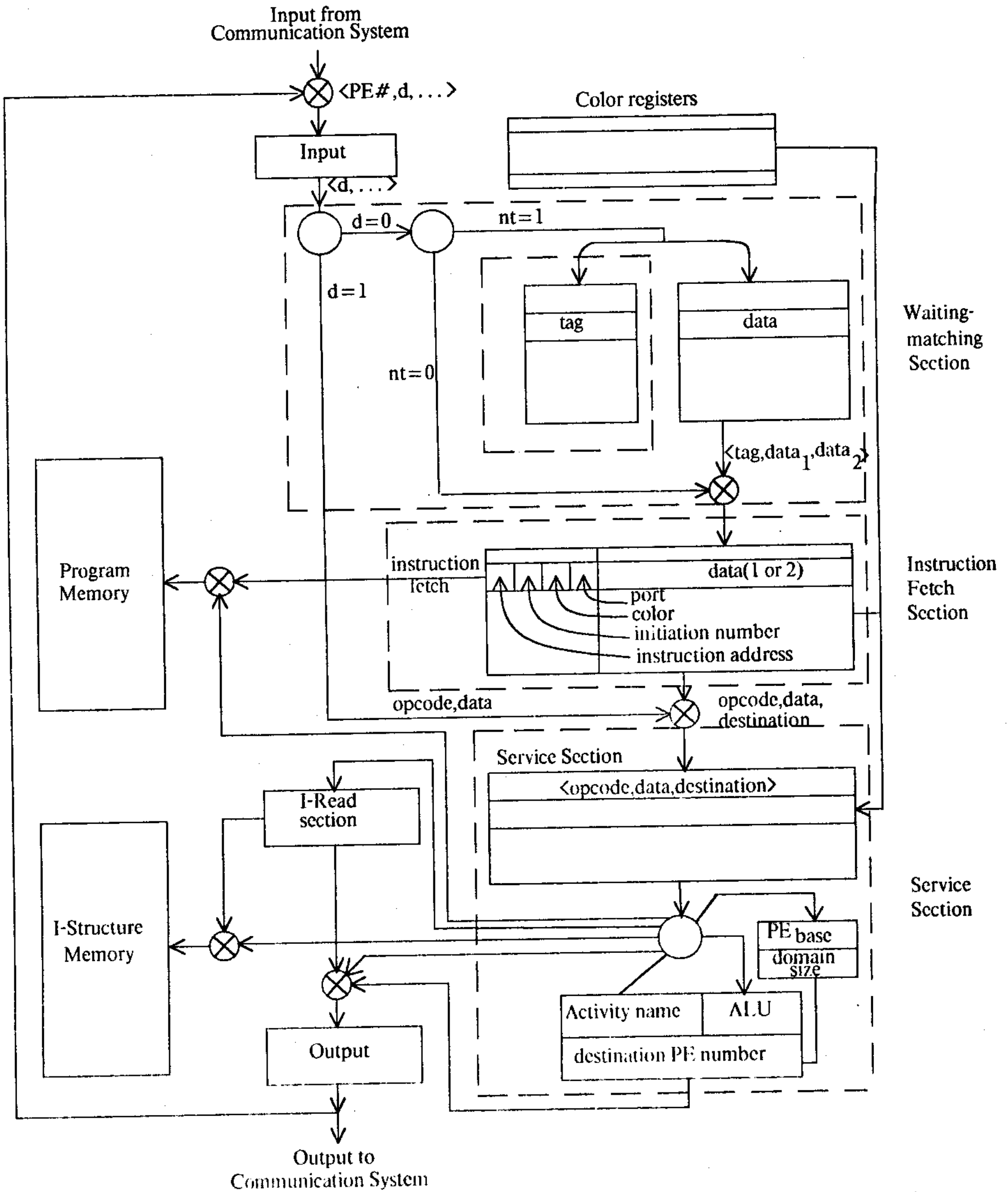

Figure 2: Processing Element

information in the machine's storage that represents this "skeleton". The exact form of an instruction will be given in this section. It is important to differentiate the concept of an instruction from that of an operation; an operation can be derived from several different instructions containing the same generic operator (e.g., +). Input values may be from any of the following:

- Tokens of the d=0 type, or

- Data from the activiation record, or

- Constants associated with the instruction itself.

Input values (along with the instruction) are used to form an operation packet for the service section. By this mechanism, we can define a single *operator* (again, the example +) which takes exactly two inputs, and by correct construction of the *instruction*, create the effect of single-input actors (<Data> + <constant> or <constant> + <Data>) as well as the more obvious double-input actors. It is even possible to build activity templates that will ignore the <Data> field of an input token altogether (i.e., using it only as a trigger).

We can view an instruction as having at least one, and at most two, input "ports" through which d=0 tokens may pass. Information must be contained in the instruction to specify what, if anything, is to be done with the data contained therein.

To clarify this, consider the actions performed by the *instruction fetch* hardware described in [3]:

- Tokens corresponding to an activity which is ready for execution (an enabled activity) are taken from the waiting-matching section's output queue.

- The local instruction address (found in the token's <Tag> field) is used to fetch the correct instruction from the program memory. Building an operation packet to be fed to the service section can begin at this point - the opcode and the destination instructions (i.e., the "downstream" actors in the data flow graph) are now known.

- Completion of the packet building process is performed by "filling in" the operand values. While an instruction must have exactly one or two inputs, the

denoted operation may have from one to three inputs. We make the association as follows: the instruction contains information concerning the handling of data carried on input tokens as well as information regarding an optional "constant" field in the instruction itself. For each input token, two bits in the instruction (called the *disposition*) specify one of the following actions:

* **Disposition="00":** The corresponding data value is to become activity operand 1.

* **Disposition="01":** The corresponding data value is to become activity operand 2.

* **Disposition="10":** The corresponding data value is to become activity operand 3.

* **Disposition="11":** The corresponding data value is to be discarded. This encoding will provide, for example, the capability of triggering any dyadic operation in which one of the operands is constant.

For the (optional) constant field, one bit of the instruction specifies that the data value contained in the template represents a program constant, or represents an index into the activation record wherein will be found the actual data. This determines the *value* of the constant. The *disposition* is then handled exactly as for token data (i.e., two more bits).

Note that not all combinations of encodings are legal. Specifically, each operand (1, 2, 3) may only be specified once in a valid instruction (i.e., it is illegal to code two disposition fields as "00" in one instruction).

3.1. Format

With this in mind, we define the following format for instructions:

```
<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)
```

```
      <Data-value>                                         (0-64)
   <Destination>
      <Number-of-tokens>                                   (1)
      <Destination-instruction-port-number>                (1)
      <Destination-relative-address>                       (16)
      <Destination-list-flag>                              (1)
```

Note that the <Constant-specification> is optional and will not be included whenever <Constant-disposition>="11" (unused). Further, the destination list is of arbitrary length (i.e., the <Destination> item may be repeated *ad infinitum*).

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 scheme used here will encode the optional <Constant-specification> field with a pointer to the FALSE destination list; the TRUE destination list will be part of the instruction. In the case of a SWITCH with no FALSE destinations, the pointer will not appear at all. This fact will be reflected through correct coding (i.e., "11") of the <Constant-disposition> field.

3.2. Examples

Below are shown some sample instruction codings; the important fields for each example are given.

- Addition of two token-carried numbers:

```
expect 2 tokens
<Opcode>="+"
<Token-1-disposition>="00"
<Token-2-disposition>="01"
<Constant-disposition>="11"
```

- Addition of a constant to a token-carried number:

```
expect 1 token
<Opcode>="+"
<Token-1-disposition>="00"
<Token-2-disposition>="11"
<Constant-disposition>="01"
<Constant-source>=0/1
```

- Addition using a trigger token:

```
expect 2 tokens
<Opcode>="+"
<Token-1-disposition>="00"     (value)
<Token-2-disposition>="11"     (trigger)
<Constant-disposition>="01"
<Constant-source>=0/1
```

- A three-input operation (Form-address-I-store(s,i,v)):

```
expect 2 tokens
<Opcode>="Append"
<Token-1-disposition>="01"     (index)
<Token-2-disposition>="10"     (value)
<Constant-disposition>="00"    (struct.)
<Constant-source>=0/1
```

- The SWITCH operation:

```
expect 2 tokens
<Opcode>="Switch"
<Token-1-disposition>="00"
<Token-2-disposition>="01"
<Constant-disposition>="10"
<Data-value>=ptr. to FALSE dest. list
```

- Illegal use of disposition (non-disjoint):

```
<Token-1-disposition>="00"
<Token-2-disposition>="00"
```

3.3. Instruction Processing Paradigms

Four distinct cases of instruction processing can be enumerated. Each is intended for one or more specific situations. All instruction sequences begin with a d=0 token. Some sequences result in further d=0 tokens (self perpetuating sequences); others do not ("dead end" sequences). This section will attempt to describe the cases in detail.

3.3.1. Paradigm 1: <d=0> to <d=0>

This case may be thought of as the "default" situation; it represents the passing of tokens directly across the arcs in a data flow graph. We examine the production of the output token as a function of the input token. Below are listed all the fields in the d=0 token type along with a description of how each is transformed during the processing of the token.

- **Token-type:** The output token will be d=0.

- **PE-number:** See below.

- **Tag:** Color remains unchanged. Local instruction address and iteration number are described below (note that the iteration number field will only change due to D or D^{-1} operators in this paradigm).

- **Number-of-tokens-to-enable-instruction:** Taken directly from the <Destination> field.

- **Port-number:** Taken directly from the <Destination> field.

- **Data:** Semantics for each operation are given in the Operations section (4 on page 14).

Computation of the output token's PE number, local instruction address, and iteration number is done using the following algorithm. We are assuming that the necessary information (e.g., for any one code block, the physical domain size, number of PEs in a physical subdomain, and associated destination relative addesses) are resolved prior to run time. This reduces the overhead of destination determination and eliminates the need to modify any pointers during loading of the code block. Further, physical domains may only be constructed from a number of processors which is a power of 2 (i.e., 1, 2, 4, 8, ...).

Four variables are associated with each code block; the number of PEs in a physical subdomain (#PE/PSD), the base PE number (PE_{base}), the mapping constant (k), and a code block base address. Further, a status bit must be provided in each color register which is asserted only in the last physical subdomain (this information is used by the D operator which implements a different algorithm; see section 4.8.9 on page 31).

The normal tag algorithm takes, as arguments, the input tag, the destination relative address (from the instruction), #PE/PSD, PE_{base}, code block base address, and physical subdomain number. The algorithm is as follows:

1. Examine the first LOG_2(#PE/PSD) bits in the relative address. This will specify the PE offset within the *current physical subdomain*, thus uniquely determining the PE number.

2. The remainder of the bits in the relative address are to be added to the code block base address to form the local instruction address.

We abbrevite the format of the output token as follows:

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

3.3.2. Paradigm 2: <d=0> to <d=1>

This is the most straightforward use of d=1 tokens. It corresponds to the case where an operation is to be performed on a PE that is not necessarily the one on which the d=0 operation (which generated the d=1 token) was performed, yet no response or acknowledgement is needed. An example of such a situation would be the *I-store operations (acknowledgement, if required, is handled by a separate mechanism). The output d=1 token is of the format

- <d=1,PE,ack=0,<opcode,data=<address,value>>>

The PE number, opcode, address, and value are derived from the <d=0> activity.

3.3.3. Paradigm 3: <d=0> to (<d=1> to <d=0>)

This situation represents the return of some data object from a "remotely executed" d=1 operation to some arbitrary activity. The PE/tag information in the imbedded d=0 token is derived using the normal tag algorithm. The two tokens produced in this sequence may execute on different PEs; their format is as follows:

- <d=1,PE_1,ack=1,<d=0,PE_2,tag,nt,port>,<opcode,$data_1$>>

- <d=0,PE_2,tag,nt,port,$data_2$>

3.3.4. Paradigm 4: <d=0> to (<d=1> to <d=1,*Entry>)

The final case may be characterized as the execution of a d=1 operation whose explicit result is to be returned to a manager which acts as a transaction processor. Below are shown the two d=1 tokens (in sequence). In the second token, PE_1 and $opcode_1$ are automatically included to help the manager identify the origin of the token received.

- <d=1,PE_1,ack=1,<d=1,MOBJ=<PE_2,$addr_2$>>,<$opcode_1$,$data_1$>>

- $\langle d=1, PE_2, ack=0, \langle opcode=*Entry, data=\langle addr_2, PE_1, opcode_1, ...\rangle\rangle\rangle$

4. Operations

We begin this section by describing the basic data types. This is followed by a description of the operations themselves (logically grouped). Operations are described on the basis of opcode, input and output types, error checking, and error propagation. It is to be understood that the following definition is incomplete with respect to streams, managers, and input/output.

Note that in the definiton of the operation codes, several conventions will be used. Operations are represented as functions, and follow this basic format:

OP1 {× OP2 {× OP3}} ⇒ RSLT {+ ERR}

OP1, OP2, and OP3 represent the *data-types* for operands 1, 2, and 3. RSLT represents the type of the result. Items enclosed in braces {...} are optional and may be omitted from some definitions. The + nontation should be read as "or".

It should be understood that the equations given represent the only valid semantic forms. A *Type−mismatch* error token will be generated for any application of an operation that does not match one of the given equations.

The reader is to assume that the token processing follows paradigm 1 (from 3.3 on page 11) unless otherwise indicated.

4.1. Data Types

All data typing considerations in this machine are deferred until run time. As such, each datum has associated with it a *data-type* identifier. The following is a list of the legal types (the data-type mnemonic is shown in **boldface**) with a description of each.

- **BITS:** Bit string (1 to 8 bytes in length).

- **BOOL:** Boolean value (i.e., either TRUE or FALSE). Represented as a 1 bit binary number. Padding of the representation with dummy bits will be done as

necessary to maintain boundary alignment.

- **CHAR:** ASCII character (7 bits).

- **ERR:** Error. Signifies that a processing error has occurred. Error handling is discussed at length in [6]. The machine being described here can generate the following types of error tokens:

1. Type – mismatch: An application of an operator has been attempted that does not match one of the valid semantic forms.

2. Positive – overflow: The result of an arithmetic computation is more positive than the largest possible positive number that can be expressed in the target type's format.

3. Negative – overflow: The result of an arithmetic computation is more negative than the largest possible negative number that can be expressed in the target type's format.

4. Positive – underflow: The result of a floating-point computation is smaller than the smallest possible positive number that can be expressed in the target type's format.

5. Negative – underflow: The result of a floating-point computation is smaller than the smallest possible negative number that can be expressed in the target type's format.

6. Division – by – zero: An attempt has been made to divide by zero.

7. Illegal – address: An attempt has been made to create or use an address which exceeds the limits of the object being manipulated (e.g., an I-structure).

8. Result – size: The result of an operation is too large to fit in the target type (used by the COMPRESS operation).

- **FP-32:** Short floating point. Defined in accordance with the IEEE floating point standard; 32 bits in length [5]. This format is referred to as *basic single.*

```
<Sign>                                   (1)
<Biased-exponent-of-2>                   (8)
<Fraction>                               (23)
```

- **FP-64:** Long floating point. Defined in accordance with the IEEE floating point standard; 64 bits in length [5]. This format is referred to as *basic double.*

```
<Sign>                                        (1)
<Biased-exponent-of-2>                        (11)
<Fraction>                                    (52)
```

- **INT-8:** Signed, two's complement integer (8 bits in length).

- **INT-16:** Signed, two's complement integer (16 bits in length).

- **INT-24:** Signed, two's complement integer (24 bits in length).

- **INT-32:** Signed, two's complement integer (32 bits in length).

- **ISA-ME:** Address of an element in an I-structure of mixed elements. The address is made up of

```
<Structure-element-address>
   <PE-number>                                (8)
   <Local-address>                            (16)
```

- **ISA-UE:** Address of an element in an I-structure of uniform elements. The address is made up of

```
<Structure-element-address>
   <PE-number>                                (8)
   <Local-address>                            (16)
<Element-type>
   <Data-length>                              (4)
   <Data-class>                               (4)
```

- **ISD-ME:** Descriptor for an I-structure which contains elements of mixed length and class (i.e., totally different types). The descriptor is made up of

```
<Structure-base-address>
   <PE-base>                                  (8)
   <Base-address>                             (16)
<Total-structure-bytes>                       (24)
```

- **ISD-UE:** Descriptor for an I-structure which contains uniform elements (all of the same type). The descriptor is made up of

```
<Structure-base-address>
   <PE-base>                                  (8)
   <Base-address>                             (16)
<Total-structure-elements>                    (16)
<Physical-domain-size>                        (8)
```

```
<Mapping-constant>                         (16)
<Element-type>
   <Data-length>                           (4)
   <Data-class>                            (4)
```

- **ISD-UL:** Descriptor for an I-structure which contains elements of uniform length (but not necessarily of the same class). The descriptor is made up of

```
<Structure-base-address>
   <PE-base>                               (8)
   <Base-address>                          (16)
<Total-structure-elements>                 (16)
<Physical-domain-size>                     (8)
<Mapping-constant>                         (16)
<Element-length>                           (4)
```

- **MDEF:** Manager definition (24 bits).

- **MOBJ:** Manager object (24 bits).

- **PROC:** Procedure definition (24 bits).

- **SMASH:** A special type made up of the bit-by-bit concatenation of two to eight <Data> fields (i.e., including <Data-type>). SMASH types are constructed by the COMPRESS operator. Base types are re-extracted by the EXPAND operation. The SMASH type is of variable length.

The pseudo type **COMP** is used to desribe a situation where any one of the following types may appear: BOOL, CHAR, ERR, FP-32, FP-64, INT-8, INT-16, INT-24, INT-32, ISD-ME, ISD-UE, ISD-UL, MDEF, MOBJ, PDT, PROC, or SMASH. The **ARITH** pseudo type combines FP-32, FP-64, INT-8, INT-16, INT-24, and INT-32. **ANY** permits any valid data type. Where ambiguous, pseudo types will be subscripted to indicate where the base types must be the same (e.g., $COMP_1 \times COMP_2 \Rightarrow COMP_1$ specifies that two different types are accepted as input, and the output will be of the same type as the first operand).

The notation for d=1 tokens and operations is slightly different; <...> will be used where appropriate.

4.2. The Arithmetic Operations

Arithemetic operations are further broken down (primarily for the purpose of error checking). Automatic coercion of operands takes place during operation application for operators with a "c" subscript (all arithmetic operations come in two varieties; those which allow coercions and those which allow no coercions). Where coercion applies, the process is as follows:

1. Examine both operands. If either is non-arithmetic, generate an error.

2. Make the two input operands the same type. Where appropriate, convert integers to floating point; convert short forms to long forms.

3. Apply the operator.

4. Attempt to convert this internal representation of the result into the most compact arithmetic form without sacrificing precision. In any event, a longer format is considered preferable to an overflow error.

5. Distribute the result (ARITH type or ERR type depending on the success of the operator application and the compaction).

For the "nc" operators, both input operands must be of the same type (class and length); the result is guaranteed to be of this type (or ERROR).

4.2.1. Addition and Subtraction ($+_c, +_{nc}, -_c, -_{nc}$)

One form, three possible errors:

- $ARITH_1 \times ARITH_2 \Rightarrow ARITH_3 + ERR$

- Errors: ±overflow, type

4.2.2. Multiplication and Division ($\times_c, \times_{nc}, \div_c, \div_{nc}$)

Four forms, six possible errors:

- $ARITH_1 \times ARITH_2 \Rightarrow ARITH_3 + ERR$

- Errors: ±overflow, ±underflow, division by zero, type

4.2.3. Explicit Coercions

It is sometimes necessary to explicitly convert one ARITH operand to another ARITH format. Also, for input/output, it is desirable to be able to convert between INT format and CHAR format. The operations described here implement "coercion by example". The first operand is always the datum to be converted. The second operand is an "example" of the target type (recall that any operand can be constant; to construct a CONVERT TO INT-8 instruction, one only need include a constant INT-8 in the instruction). Any attempt to convert an object into a format which cannot contain it will result in an arithmetic error. Three forms, five possible errors:

- $ARITH_1 \times ARITH_2 \Rightarrow ARITH_2$ + ERR

- CHAR × INT-8 ⇒ INT-8 + ERR

- INT-8 × CHAR ⇒ CHAR + ERR

- Errors: ±overflow, ±underflow, type

4.3. The Boolean Operations

Booleans are broken into the monadic and dyadic varieties.

4.3.1. Monadic (¬)

One form, one possible error:

- BOOL ⇒ BOOL

- Errors: type

4.3.2. Dyadic (∧,∨,⊕)

One form, one possible error:

- BOOL × BOOL ⇒ BOOL

- Errors: type

4.4. The Bit String Operations

Bit string operations are also broken down into monadic and dyadic forms.

4.4.1. Monadic ($\neg_1$,$\neg_2$)

These operations produce the ones- and twos-complement of the input bit string. One form; one possible error:

- BITS ⇒ BITS

- Errors: type

4.4.2. Dyadic (∧,∨,⊕)

These operations accept two bit strings of equal length and perform the indicated function. One form; one possible error:

- BITS × BITS ⇒ BITS

- Errors: type

4.4.3. Shift

The SHIFT function is slightly different than the other dyadic operations in that one of the operands is not a BITS type. SHIFT is implemented with eight different opcodes. These specify shift direction (right/left) and padding function (zero, one, sign, wrap). The second operand specifies the shift amount (in bits). One form, one possible error:

- BITS × INT-8 ⇒ BITS

- Errors: type

4.4.4. Concatenate

This operation takes two BITS operands (which may be of different lengths) and concatenates them such that, in the result, the most significant bit is the most significant bit of operand 1. The resulting BITS string must be eight bytes or less in length. One form, two possible errors:

- BITS × BITS ⇒ BITS

- Errors: size, type

4.4.5. Adjust-length

This operation accepts two BITS operands and produces a result string constructed from the first, but having a length equal to that of the second. The operation is implemented with four opcodes which allow the specification of a fill value (zero, one) for expansion as well as the specification of the side on which the padding/truncation is to be done (left, right). One form, one possible error:

- $BITS_1 \times BITS_2 \Rightarrow BITS_2$

- Errors: type

4.5. The Extract/Construct Operations

The operations described in this section allow direct access to the representations of data objects. They are to be used with care.

4.5.1. Extract-type, Extract-value

These monadic operations accept an ANY type token, and produce a BITS type token. The EXTRACT-TYPE operation returns a one byte string which is the <Data-type> field of the input operand. The EXTRACT-VALUE operation returns a string containing the bit representation of the <Data-value> field (which is of variable length). Normal BITS operations may then be used on the extracted information. One form, one possible error:

- $ANY \Rightarrow BITS$

- Errors: type

4.5.2. Construct-data

To complement the EXTRACT operations, the CONSTRUCT-DATA operation allows two bit strings (operand one denoting a type, the other denoting a value) to be combined to produce an arbitrary <Data> value of arbitrary type. Note that, depending on the target type, certain consistency checks will be performed to assure that the result is indeed legal. One form, one possible error:

- BITS × BITS ⇒ <Data>

- Errors: type

4.6. The Relational Operations

Relationals are broken down into two categories: Arithmetic and Non-arithmetic. The sole difference is that Arithmetic relationals will perform coercions (previously described) prior to performing the comparison; Non-arithmetic relationals perform no coercions.

4.6.1. Arithmetic ($\leq$,<,=,$\neq$,>,$\geq$)

Equalization of the types of the two operands is done in accordance with the rules previously given (convert INT to FP, convert short to long). One form, one possible error:

- $ARITH_1 \times ARITH_2 \Rightarrow BOOL$

- Errors: type

4.6.2. Non-arithmetic ($\leq$,<,=,$\neq$,>,$\geq$)

One form (both COMPs must be of the same type), one possible error:

- $COMP_1 \times COMP_1 \Rightarrow BOOL$

- Errors: type

4.6.3. Non-Arithmetic ($\neq_{lax}$)

This special case of $\neq$ permits the two operands to be of different type; To generate a boolean TRUE, the operands must be of the same type (class and length), and must have identical value fields. All other cases return FALSE independent of type inequality. One form, no possible errors:

- $COMP_1 \times COMP_2 \Rightarrow BOOL$

- Errors: none

4.7. The I-Structure Operations

We describe briefly the implementation strategy for I-structures. Then, the actual operations are defined.

4.7.1. I-structure Implementation

This implementation supports three basic I-structure classes:

1. **Uniform-elements:** All elements in the I-structure are of the same type (class and length). Hence, no type information is stored on a per-element basis. Rather, the I-structure descriptor carries the definition of the element type. Element address calculation and checking are done at run time.

2. **Uniform-length:** All elements in the I-structure are of the same length, but may be of different classes. Type information is stored along with each element in the I-structure. This approach is slightly more flexible than the uniform elements case in that some mixing of classes is permitted while still retaining the dynamic nature of element address computation.

3. **Mixed-elements:** Elements may be of any class or length. For this reason, offsets from the base of the I-structure (element addressing) must be known at compile time.

4.7.2. Form-Address

Three forms, two possible errors:

- ISD-ME $\times$ INT-24 $\Rightarrow$ ISA-ME + ERR

- ISD-UE $\times$ INT-24 $\Rightarrow$ ISA-UE + ERR

- ISD-UL $\times$ INT-24 $\Rightarrow$ ISA-ME + ERR

- Errors: address, type

4.7.3. I-Fetch

This operation initiates a paradigm 3 process. The PE address for the token is derived from the I-structure base address and the mapping scheme used. The token also contains the destination to which the result should be forwarded. Two forms, one possible error:

- ISA-ME ⇒
  <d=1,PE_1,ack=1,<d=0,PE_2,tag,nt,port>,<*I-fetch-ME,Addr>>

- ISA-UE ⇒
  <d=1,PE_1,ack=1,<d=0,PE_2,tag,nt,port>,<*I-fetch-UE,Addr,Type>>

- Errors: type

4.7.4. *I-Fetch-ME; a d=1 Operation

The input *address* field for this operation denotes a location in the I-structure memory containing length, class, and value. One form, no possible errors:

- <Addr> ⇒ <Data>
  where <Data> contains length, class, and value fields.

- Errors: none

4.7.5. *I-Fetch-UE; a d=1 Operation

The input *address* field for this operation denotes a location in the I-structure memory containing only the data value. The type information (length and class) is provided in the input d=1 token. One form, no possible errors:

- <Addr,Type> ⇒ <Data>
  where <Data> contains length, class, and value fields.

- Errors: none

4.7.6. Form-Address-I-Fetch

This operation initiates a paradigm 3 process. Three forms, two possible errors:

- ISD-ME × INT-24 ⇒
  <d=1,PE_1,ack=1,<d=0,PE_2,tag,nt,port>,<*I-fetch-ME,Addr>>

- ISD-UE × INT-24 ⇒
  <d=1,PE_1,ack=1,<d=0,PE_2,tag,nt,port>,<*I-fetch-UE,Addr,Type>>

- ISD-UL × INT-24 ⇒
  <d=1,PE_1,ack=1,<d=0,PE_2,tag,nt,port>,<*I-fetch-ME,Addr>>

- Errors: address, type

4.7.7. I-store

This operation initiates at least one paradigm 2 process. The I-store operation may specify at most one destination; if one is specified, an *I-store-ack d=1 token will be generated (thus initiating another paradigm 2 process). Two forms, one possible error:

- ISA-ME × COMP ⇒
  <d=1,PE_1,ack=0,<*I-store-ME,<Addr,COMP>>>
  <d=1,PE_1,ack=1,<d=0,PE_n,tag,nt,port>,<*I-store-ack,Addr>>

- ISA-UE × COMP ⇒
  <d=1,PE_1,ack=0,<*I-store-UE,<Addr,COMP>>>
  <d=1,PE_1,ack=1,<d=0,PE_n,tag,nt,port>,<*I-store-ack,Addr>>

- Errors: type

4.7.8. *I-Store-ME; a d=1 Operation

The type of the datum is written along with the value. One form, no possible errors:

- <Addr,Type> × COMP ⇒ BITS + ∅
  where *BITS* is a trigger and "∅" indicates *no explicit result.*

- Errors: none

4.7.9. *I-Store-UE; a d=1 Operation

Only the value of the datum is written; no type information is preserved. One form, no possible errors:

- <Addr> × COMP ⇒ BITS + ∅
  where *BITS* is a trigger and "∅" indicates *no explicit result.*

- Errors: none

4.7.10. *I-Store-Ack; a d=1 Operation

The intent of this operation is to acknowledge to some activity that a selected element of an I-structure has been written. As such, its behavior is characterized as a type of *I-fetch, except that the value returned (once the corresponding location has been written) is only a trigger token (a valueless BOOL). One form, no possible errors:

- <Addr> ⇒ BOOL

- Errors: none

4.7.11. Form-Address-I-Store

This operation takes advantage of the three operand mechanism and the fact that many I-store operations may be done using a constant I structure descriptor. The behavior is exactly the same as the combination of FORM-ADDRESS and I-STORE. Three forms, one possible error:

- ISD-ME × INT-24 × COMP ⇒
  <d=1,PE_1,ack=0,<*I-store-ME,<Addr,COMP>>>
  <d=1,PE_1,ack=1,<d=0,PE_n,tag,nt,port>,<*I-store-ack,Addr>>

- ISD-UE × INT-24 × COMP ⇒
  <d=1,PE_1,ack=0,<*I-store-UE,<Addr,COMP>>>
  <d=1,PE_1,ack=1,<d=0,PE_n,tag,nt,port>,<*I-store-ack,Addr>>

- ISD-UL × INT-24 × COMP ⇒
  <d=1,PE_1,ack=0,<*I-store-ME,<Addr,COMP>>>
  <d=1,PE_1,ack=1,<d=0,PE_n,tag,nt,port>,<*I-store-ack,Addr>>

- Errors: type

4.8. Control Operations

This section contains those operatins which don't relly fit any of the previous categories, but also do not merit a section of their own.

4.8.1. Compress

This operation accepts two operands, and produces an output token containing the concatenation of the <Data> values of the two input tokens. Note that this implies the inclusion of <Data-type> fields for each of the inputs. The sum of the lengths of the two input operands (including type fields) must be less than or equal to eight bytes. This will guarantee that the values can be faithfully reconstructed by an EXPAND operation. A *Result–size* error token is generated if the result length would be greater than eight. The SMASH type carries with it the correct (variable) length. It is possible to use a SMASH type as input, providing the length restriction is observed[5,6]. One form, two possible errors:

- $\mathrm{COMP}_1 \times \mathrm{COMP}_2 \Rightarrow \mathrm{SMASH} + \mathrm{ERR}$

- Errors: size, type

4.8.2. Expand

This operation logically un-does the action of a series of COMPRESS operations. In that the <Data-value> field of a token is eight bytes wide, and that the smallest datum is a valueless BITS (one byte wide), it would be possible to have a SMASH type containing as many as eight discrete values. The EXPAND operation can, therefore, have as many as eight distinct destinations (each destination would receive one piece of the SMASH data). The <Destination> list is interpreted as follows: the first datum in the SMASH type is sent to the first <Destination>, the second to the second, etc. Note that if there are more data values than <Destination> entries, they will be discarded. <Destination>s with no corresponding data will also be ignored.

---

[5]Using a SMASH type as input to a COMPRESS operation preserves only the base information in the original SMASH. To illustrate this, if COMPRESS is given <INT$_1$>,<INT$_2$>, it will produce <<INT$_1$>,<INT$_2$>>. If COMPRESS is then given <<INT$_1$>,<INT$_2$>>,<INT$_3$> (i.e., re-COMPRESSing a SMASH), it will produce <<INT$_1$>,<INT$_2$>,<INT$_3$>> and not <<<INT$_1$>,<INT$_2$>>,<INT$_3$>>.

[6]Although it has not been analyzed in detail, it should also be possible to use an I-structure of SMASH types to implement lists (each SMASH item is like a "CONS" cell and can contain pointers to other objects or atomic values).

Further, to allow selective discarding of data (other than those at the "end"), we introduce a second operand for EXPAND; this operand will be a BITS (bit string) type, and the most significant byte will be interpreted as a mask. The most significant bit, if zero, will cause the first data value in the SMASH to be ignored (no token will be generated for the first <Destination>). Subsequent bits perform the same function with subsequent objects in the SMASH type. Note that this mask may be token carried or constant by appropriate construction of the instruction. One form, no possible errors:

- SMASH × BITS ⇒ $\langle Data \rangle_1 + \emptyset, \langle Data \rangle_2 + \emptyset, ...$

- Errors: none

4.8.3. *Entry; a d=1 Operation

This operator is used to nondeterministically merge tokens at the input to a manager. It operates as the combination of ENTRY and E (described in [2]) due to the fact that its output is a sequence of individual tokens rather than a stream. The algorithm is

1. Each manager object has associated with it a counter that can be read and incremented atomically. This is done for each activation of *Entry for a manager object. Whenever the counter overflows, a USE operation is substituted (just as in the case of a D operator).

2. The value read (call it *j*) is hashed (a simple modulo-n technique, where n = maximum number of concurrent uses of the manager).

3. This hashed value is used to index a table containing a flag which indicates whether or not the value of j (*modulo* n) is in use as a tag or not. This table also contains space for storing the input token's optional *return activity name.*

4. If the selected table entry indicates that it is "in use", the d=1 token is recirculated through the communications network.

5. If the selected table entry is not in use, the return activity name (if present) from the token is written[7]. An activity name for the manager object activation is constructed using j as the <Iteration-number>. The remainder of the activity

[7] The activity name would be imbedded in the input token if ack = 1.

name is predefined (see Figure 3), and is retrieved and used as is.

One form, no possible errors:

- <COMP> ⇒ COMP

- Errors: none

4.8.4. Exit

This operation undoes the action of *Entry. The <Iteration-number> field is hashed (*modulo* n) to read the manager object entry table (and to mark the table entry as "not in use"). If a return activity name was saved, the token is forwarded. One form, one possible error:

- $COMP_1 \Rightarrow COMP_1 + \varnothing$
  where "ø" indicates *no explicit result.*

- Errors: type

4.8.5. Reset-I-Structure-Memory

This operation will generate n d=1 tokens (n=number of PEs that contain pieces of the I-structure). The SMASH operand will specify the bounds (byte offsets for start and end) for the reset action. The third operand specifies the manager object that is to receive acknowledgement. Paradigm 4 processing is initiated. Three forms, one possible error:

- ISD-ME × SMASH × MOBJ ⇒
  <d=1,PE_1,ack=1,<d=1,MOBJ>,<*I-reset,<Addr,SMASH>>>

- ISD-UE × SMASH × MOBJ ⇒
  <d=1,PE_1,ack=1,<d=1,MOBJ>,<*I-reset,<Addr,SMASH>>>

- ISD-UL × SMASH × MOBJ ⇒
  <d=1,PE_1,ack=1,<d=1,MOBJ>,<*I-reset,<Addr,SMASH>>>

- Errors: type

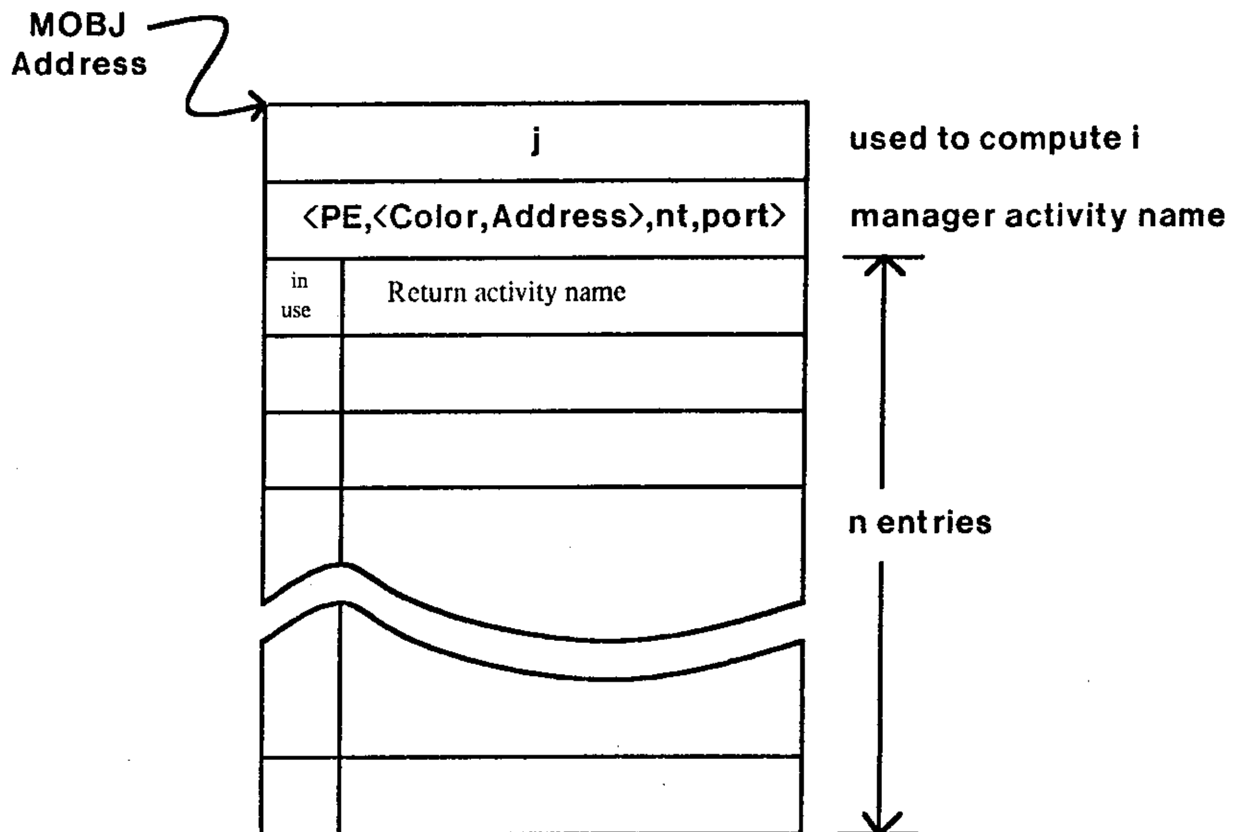

Figure 3: Manager Object Entry Table

4.8.6. *I-reset, a d=1 Operation

Certain information from the input SMASH operand is returned in the output d=1 token. One form, no possible errors:

- $\langle \text{Addr,SMASH} \rangle \Rightarrow$
  $\langle d=1, PE_2, ack=0, \langle \text{*Entry}, \langle addr_1, PE_1, \text{I-reset}, \ldots \rangle\rangle\rangle$

- Errors: none

4.8.7. Write-Color-Register

Initiates a paradigm 4 sequence. The integer operand specifies a PE number; the SMASH operand contains the data to be written; MOBJ is the manager object which will receive the acknowledgement once the color register has been written. One form, one possible error:

- $\text{INT-8} \times \text{SMASH} \times \text{MOBJ} \Rightarrow$
  $\langle d=1, PE_1, ack=1, \langle d=1, \text{MOBJ} \rangle, \langle \text{*Write-color-register,SMASH} \rangle\rangle$

- Errors: type

4.8.8. *Write-color-register; a d=1 Operation

This operation actually writes the values from the input token to the selected color register. An acknowledgement d=1 token is generated. As in the case of *I-reset, certain informatoin from the input SMASH is also returned. One form; no possible errors:

- $\text{SMASH} \Rightarrow$
  $\langle d=1, PE_2, ack=0, \langle \text{*Entry}, \langle addr_1, PE_1, \text{Write-color-register}, \ldots \rangle\rangle\rangle$

- Errors: none

4.8.9. Iteration - D and D^{-1}

The D operators implement a different PE/Local instruction address mapping algorithm by virtue of the meaning of D (as described in [1]). Alteration of the <Iteration-number> field in the input token requires recomputation of physical subdomain using the mapping parameters defined in section 3.3.1.

The intent of the mapping constant (k) is to confine k "adjacent" passes through the loop to one physical subdomain. The value of k may be one (implying no confinement - each pass through the loop represents a change of physical subdomain). As k becomes large, the effect will be to restrict operation to a single physical subdomain. The algorithm is

1. Calculate the PE *offset* within some physical subdomain and the local instruction address as before.

2. Add 1 to the iteration number. If (iteration number) *modulo* k $\neq 0$, the new PE number is computed as for the normal case.

3. If (iteration number) *modulo* k $= 0$ and this is not the last physical subdomain, the new PE number will be computed by adding the offset from step 1 to the number of the first PE of the *next* physical subdomain (current PSD number + #PE/PSD + offset within PSD). The value of k is subtracted from the iteration number in the result token.

4. If (iteration number) *modulo* k $= 0$ and this is the last physical subdomain, the new PE number will be computed by adding the offset (from step 1) to PE_{base}. No adjustment of the iteration number is necessary.

5. In any case, if the iteration number should overflow (in the result token), a USE operation will be substituted.

One form, one possible error:

- $COMP_1 \Rightarrow COMP_1$

- Errors: type

4.8.10. Identity

This operation passes, unmodified, its input. Note that it can be packaged in various instruction formats to perform the function of a constant-generating actor, a "pass-1-when-2" actor, or the basic identity actor. One form, no possible errors:

- $ANY_1 \Rightarrow ANY_1$

- Errors: none

4.8.11. Switch

The SWITCH operation has been discussed already. Note that the only constraint on its application is that the second operand must be a BOOL. If there is no destination list corresponding to the output to which the data token is directed, the token will be absorbed. Four forms, one possible error:

- $ANY_1 \times BOOL \Rightarrow ANY_1 + \varnothing$

- Errors: type

4.9. The USE Operation

[Due to the fact that the complete implemetation of managers relies on streams, and that the definition of streams is unsettled, this section should be considered as preliminary. We recognize, however, the need to define an interface to managers. The following should be interpreted in that light.]

The USE operation is implemented to facilitate communication with managers. Many manager types may exist. They can be categorized as *centralized, semi-centralized, or decentralized.* A centralized manager is the only one of its generic type in the machine. It exists to serve all users; such a manager might be used to coordinate the creation and destruction of physical domains in the system. A semi-centralized manager performs functions associated with a single physical domain. A manager of this type could be used to allocate and deallocate I-structure storage space within the physical domain. A decentralized manager is not associated with any particular boundaries.

It is desirable to have as much flexibility in the creation and use of managers as possible while relying as little as possible on a wide variety of machine operations to perform the necessary functions. This is the intent of the USE operation. Let us explore the desired features.

First and foremost, managers are central to many research questions related to data flow machines; and, as such, should be as changeable as possible. It would be ideal to provide a mechanism whereby all (or nearly all) managers are written and maintained as ID

programs. We can envision a kind of hierarchy of managers in the physical machine:

- **Hypervisors** are the *centralized* managers. These will be used to coordinate global system resources. One of their primary functions would be the creation of subordinate managers called

- **Supervisors.** These are the *semi-centralized* managers that are associated with a physical domain. These managers might in turn be used to create other subordinate managers.

There is probably a useful limit to this scheme (perhaps as few as two levels), but the point to be made is that all such creation and destruction of managers can be derived from

- The existence at the "beginning of time" of the hypervisors, and

- A fundamental operation to communicate with a manager of any type (e.g., USE).

We can then argue by induction that all subordinate managers can be created, used, and destroyed (where destruction is possible) by appropriate application of USE.

Such an "appropriate application" requires the establishment of a methodology for USE. One possibility is to assume the existence of a "bootstrap" hypervisor that can perform only two functions:

1. Allocation of an I-structure of fixed size, and

2. Initiation of another manager.

I-structure allocation (in some simple form) must be a fundamental operation of any proposed bootstrapping manager since all other uses of managers will requre an I-structure for parameter passing.

USE initiates paradigm 2 or 3 processing. The choice depends on the absence (paradigm 2) or presence (paradigm 3) of a <Destination> for the USE. While no explicit acknowledgement occurs from the *Entry operation, the manager itself can generate a "result" token. The manager object which is to receive the token has associated with it a table as shown in Figure 3 (and described under *Entry). Two forms, one possible error:

- $MOBJ \times COMP \Rightarrow$
  $\langle d=1, PE_1, ack=0, \langle *Entry, \langle MOBJ, COMP \rangle\rangle\rangle$

- $MOBJ \times COMP \Rightarrow$
  $\langle d=1, PE_1, ack=1, \langle d=0, PE_2, tag, nt, port\rangle, \langle *Entry, \langle MOBJ, COMP \rangle\rangle\rangle$

- Errors: type

5. Conclusion

We have attempted to address the important details of the compiler-emulator interface. This included a definition of tokens, a description of the instruction format, and an enumeration of the supported operations.

This document should be used as a basis for refinement of the subject topics, and will undoubtedly undergo changes as the project moves closer to a realization. Comments and recommendations should be directed to the authors.

References

1. Arvind, and Gostelow, K. P. The U-interpreter. *COMPUTER* (December 1981). To appear.

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

3. Arvind, and V. Kathail. A Multiple Processor Dataflow Machine That Supports Generalized Procedures. Memo 205, Computation Structures Group, Laboratory for Computer Science, MIT, Cambridge, Mass., February, 1981.

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

5. Stevenson, D. A Proposed Standard for Binary Floating-Point Arithmetic. *COMPUTER 14*, 3 (March 1981), 51-62.

6. Plouffe, W.E. Exception Handling and Recovery in a Dataflow System. Ph.D. Th., Department of Information and Computer Science, University of Californiav, Irvine, California, 1979.

7. Shimada,T. The Structure and Instruction Set of a Simulated Tagged-Token Dataflow Machine. Tech. Rep. (unpublished), Massachusetts Institute of Technology, Cambridge, Mass., August, 1981.