LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Addendum to the Instruction Set Definition for a Tagged-Token Data Flow Machine

Computation Structures Group Memo 212-3-1
12 October 1983

Vinod Kathail

Research support provided by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0661.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

1. Changes to the Architecture

The following is a brief description of the architectural changes to the machine.

1. The instruction address part of the tag on a token is now a tuple of the form <base-register-number, relative address> where base-register-number is 8 bits and relative address is 16 bits. A base register has two parts: a code base register (CBR) which contains base address of the code-block, and a data base register (DBR) which points to a memory area where constant area pointers and other information is stored (see 2 below). The instruction-fetch section calculates the physical address by adding the relative address to the code-block base address. The build-output-token section calculates the relative address (and not the physical address) in addition to the base-register number.

2. The color-continuation flags are no longer part of the map registers in the build-output-token section. All the information associated with a color (*i.e.*, constant area pointer, primary color, primary base register number, and next color) is stored in the memory area pointed to by DBR (see Figure 1-1).

   The primary base-register number and the primary color refer to the base register number and the very first color allocated to a loop at the time of invocation. Since a loop may use several colors and base-registers during its execution, D^{-1} uses this information to get back to the original context. The next color field specifies the next color to be used in case the initiation-number part of the tag overflows. It can be maintained in several different ways, *e.g.*, color-continuation flag, circular list of colors (see section 2). We may also want to allow only one constant area per base register. All these variations can be simulated using the general structure shown in Figure 1-1.

3. The build-output-token section is split into two sections: one which computes the tag part of the token (Compute-tag section), and one which puts together the tag and the value part (Construct-token section). The relevant part of the PE diagram is shown in Figure 1-2.

   For each instruction, the construct-token section receives two streams of values—one from ALU and one from the compute-tag section, and a stream of commands. Associated with each value is a *end-of-stream* marker which is true if it is the last value in the stream. A command specifies what type of value is coming from the construct-token section and how the values coming from ALU and the construct-token section are to be combined. We can view a command as consisting of three separate fields; their functions are described below.

   - **Field 1:** It specifies the type of value coming from the construct-token section. The type can be one of the following:

     a. d = 0 token skeleton,

     b. chain = 0,

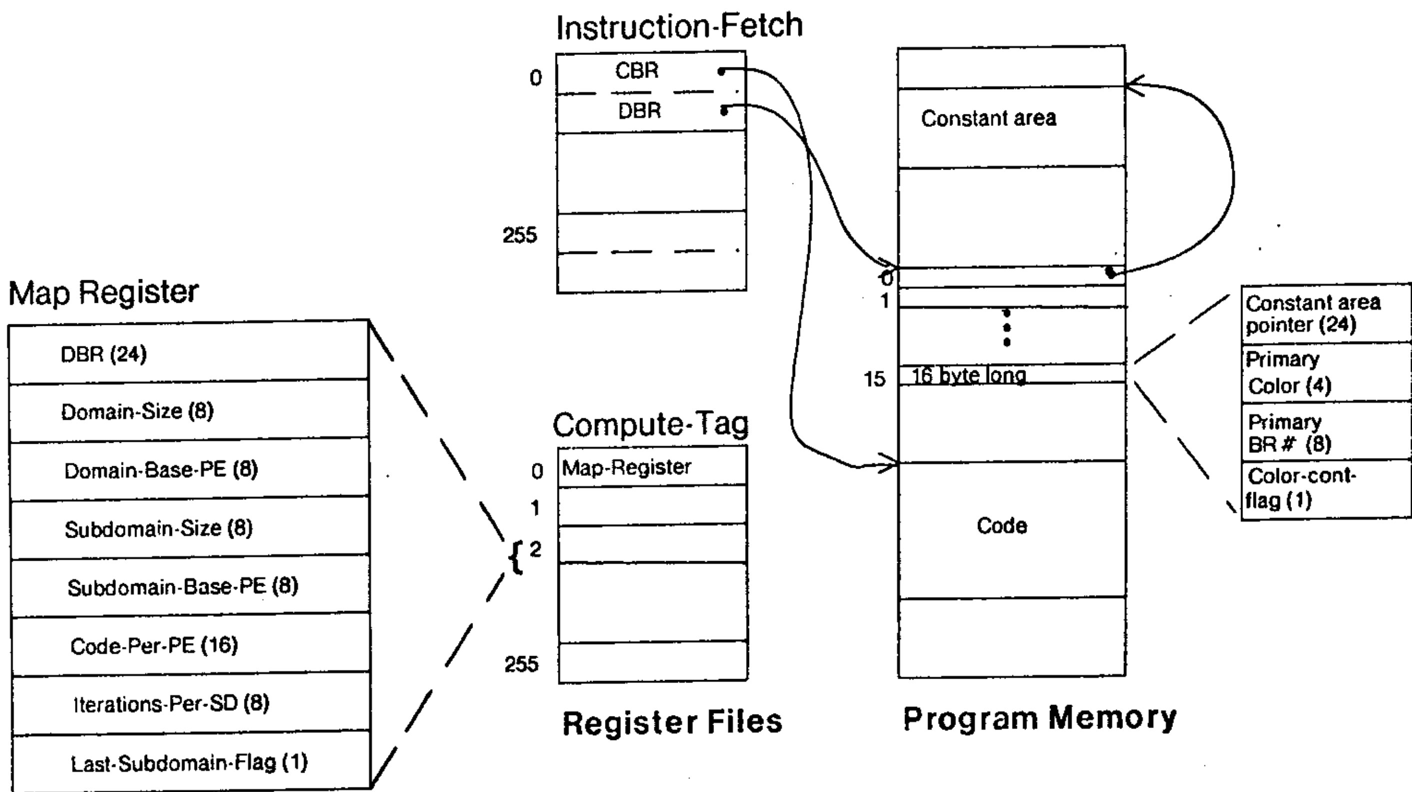

Figure 1-1: Base Registers, Map Registers, and Constant Areas

c. PE, chain = 0,

d. chain = 1, d = 0 tag,

e. PE, chain = 1, d = 0 tag,

f. empty, *i.e.*, there is no value.

Note that if the value coming from the construct-token section is a d = 0 type token skeleton and a value from ALU is needed, the value coming from ALU must be a valid data type; in all other cases the value coming from ALU must be a d = 1/2 type token skeleton.

- **Field 2:** This group specifies how the values coming from ALU and the construct-token section are combined together.

  a. Combine the value supplied by the ALU and the value supplied by the

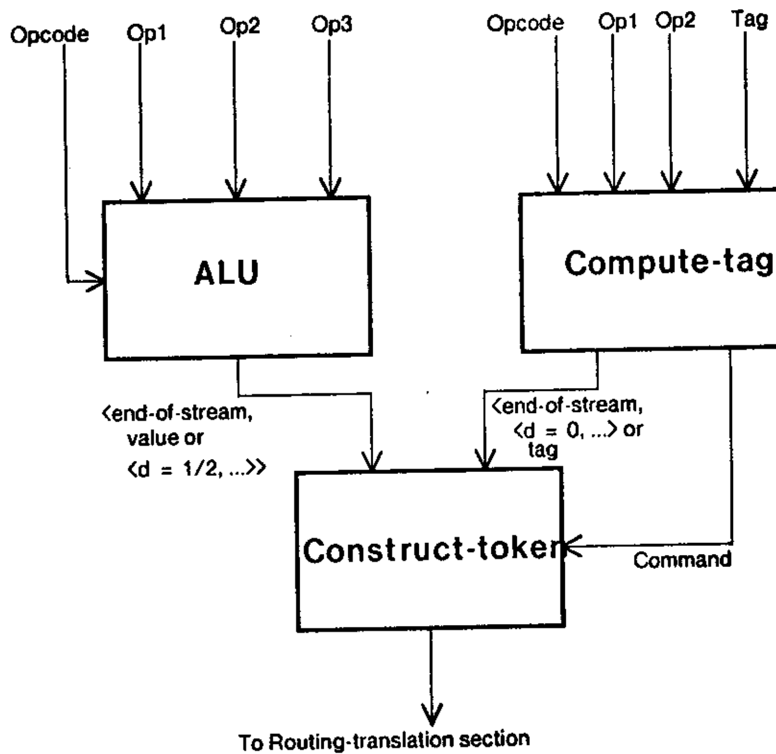

Figure 1-2: Compute-tag and Construct-token Sections

construct-token section, *i.e.*, either put the value coming from ALU on the $d = 0$ token skeleton supplied by the compute-tag section or put the value coming from the compute-tag section on the $d = 1/2$ token skeleton coming from ALU.

b. Discard the value supplied by ALU, and pass the token coming from the compute-tag section.

c. Pass the token coming from ALU, and discard anything coming from the compute-tag section.

d. Discard both of the values, and don't generate any output token.

- **Field 3:** It specifies when a new value from the ALU stream is consumed.

  a. Get a new value or token skeleton from ALU, and also store it for future use.

  b. Use the stored value as the value coming from ALU.

While processing instructions that require the two streams to be combined element by element, *e.g.*, **Expand**, **In-Fetch**, we may run out of the elements in either of the stream. If the values in the stream coming from ALU are exhausted and a new value is required, an error value (**too_many_destinations**) is used. This error value is either put on the d = 0 token skeleton or pass (d = tag) part of the value coming from the compute-tag section. On the other hand, if there are extra values in the stream coming from ALU, they are simply absorbed.

2. Color Management

Compiler assumes that the assignment of colors and initiation-numbers to various invocations is done in the following manner:

1. A recursive procedure is assigned a color, and can use the initiation numbers for the recursive calls (via R and R^{-1} operators). If initiation-number overflows during the execution of an R operator, the normal procedure call mechanism is used.[1]

2. Any other procedure is assigned a color or an initiation number (if a copy already exists) depending on manager's discretion.

3. An invocation of a loop may require more than one color to complete its execution. One possible way to get a new color is to call the system manager via the normal procedure call mechanism to start a new invocation of the loop. We have ruled out this possibility because of its inherent overhead. Instead, the manager is called to get a new color belonging to either the same copy of the code-block or a new copy that is distributed identically to the original copy. Since the new copy uses the same mapping parameters as the original copy, we can avoid using I-structures to pass the values. The copying can be done by allocating a new base register and making its CBR part point to the original copy. We hope that allocating a small number of colors and reusing them would be sufficient most of the time.

   A code-block can get a new color in several ways; they differ from each other in when the manager is called and how the manager returns the new color.

   a. If the number of colors needed during the execution can be determined at the time of the invocation then this information is passed to the manager, and the manager is responsible for supplying that many colors. Note that the manager need not allocate all the colors at the time of the invocation; it may allocate only a few colors and reuse them. The colors are made available to the code-block either by setting the next color field associated with the allocated colors, or by passing tokens carrying the new color to a specified instruction in the code-block where

---

[1] A scheme similar to the scheme used for loops can be used to take care of overflow in the case of R operator; however, reusing the colors, which make sense in the case of a loop, doesn't make much sense in the case of recursive procedures. A normal call mechanism may have to be used sometime or other.

they wait until consumed. At present, the next color field is one bit (also called *color-continuation* flag), and the manager allocates a contiguous set of colors.

b. If the numbers of colors needed during the execution cannot be determined at the time of invocation, the manager is asked to allocate a new color when the initiation-number field overflows. Note that the manager is free to guess the number of colors to be allocated. If it decides to allocate more than one color, it must set the next color fields associated with the allocated colors and not pass the colors on tokens.

To allow implementation of the above mentioned schemes and variations thereof without changing the compiler, the compiler generates code shown in Figure 2-1. The C-switch instruction is a dummy instruction. The loader is responsible for modifying the code to bypass this instruction, *i.e.*, the destination address in the D operator which points to C-Switch instruction is replaced by the destination address on either the true side of the C-switch or the false side of the C-switch depending on the scheme we want to use.

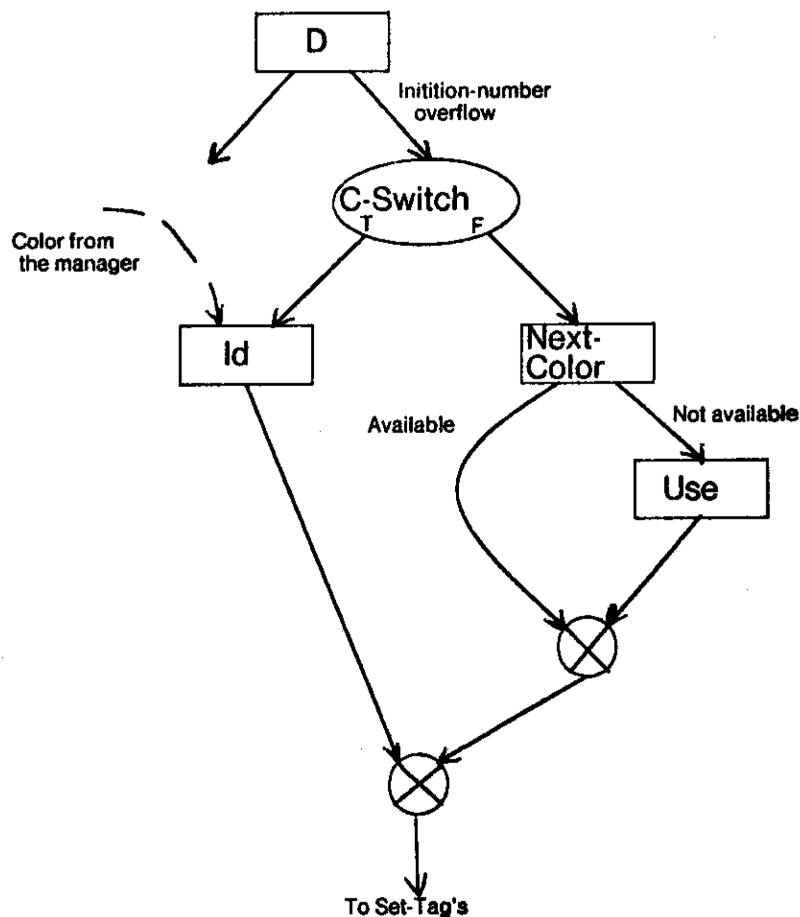

Figure 2-1: Compiler Generated Code to Get a New Color

3. Managers and PE Controllers

If managers were implemented using streams as suggested in [1], a use of a manager would involve sending a d = 0 type token to the `entry` operator. However, streams have not been implemented yet, and a manager use involves sending a d = 2 token with `*entry` opcode to the PE controller. We need to distinguish these d = 2 type of tokens from those (d = 2 type of tokens) which are used purposes specific to the implementation. Thus, to explicate the above mentioned difference, the generalized paradigm of instruction processing may be described as follows:

d = 0, {d = 1/2}, [d = 0]

where { } means 0 or more occurrences and [ ] means 0 or 1 occurrence.

This generalized paradigm does include the paradigm 4 defined in [2] because d = 0 and d = 2 `*entry` tokens are of the same nature. However, paradigm 4 of instruction processing is nothing but an instruction followed by the `Use` or `Decrement-CObj` instruction. Such a paradigm may be useful in the future, but it is really not necessary. In fact, the three instructions using this paradigm are too restricted; they require that a manager or counter object always be the one to receive the acknowledgment. Thus, the tag (on a d = 1/2 type token) where the result or acknowledgment is to be sent is always a d = 0 type tag.

An Id manager may call a PE controller, an I-structure controller, or some other manager to perform some action, and may have to wait for acknowledgment. Such waiting would be achieved by sending the tag (d = 0 type) of the destination instruction on the d = 1/2 type of tokens.[2] On the other hand, PE controller and I-structure controller instead of being implemented as Id managers are implemented in hardware (or equivalently as sequential program). Therefore, they should be structured in a way so that when they call other controllers or managers in the course of processing a request, they don't wait for an acknowledgment.

4. PE controller, I/O Devices and Master Copy of Code-blocks

We envision that some of the PE controllers in the machine will have I/O devices (at least disks) connected to them, and that a PE controller will be responsible for managing (buffering etc.) the I/O devices connected to it.

Devices in the system are identified by a 16 bit integer (referred to as *device specification*); the most significant 8 bits represent the PE number, and the least significant 8 bits (referred to as *device*) represent device group and device number inside the group. The intention is that devices can be logically grouped together; for example, disks, printers. A manager may want to deal with only the device group in which case selection of the device number is left to the PE controller. For now, both device group and device number are 4 bits in length. A device number of 0 means that

---

[2] If a manager is implemented as a sequential program, waiting for an acknowledgment (*i.e.*, an asynchronous event) has to be simulated using some sort of dummy tokens. Exactly how this is done is left to the implementor (*i.e.*, simulator/emulator people).

no device number is specified. If a device is addressable then a 32 bit integer is used to specify the address. A manager may or may not specify the address when dealing with a device.

A copy of the code-blocks available during the execution of a program is maintained on disks. Before the compiler-generated code can be executed, a mapping between code-block names (either user-defined or compiler-generated) and their disk addresses has to be established. The steps involved in this process are as follows. In the compiled code references to code-block definitions are represented using the `proc` type whose 32 bit value part is left unspecified. The code-block header specifies the symbolic names for all the code-blocks referenced inside this code-block and the places where they are referenced. These symbolic names must be converted to 32 bit values (called *code-block identifier*), and these values must be put in the code at the appropriate places. A code-block environment, which specifies the mapping between code-block identifiers and their disk addresses, must be created and made available to the system manager. At present, a code-block identifier is an integer offset in the code-block environment maintained as a linear array.

We assume that code-blocks are already resident on the disk, and that the environment is available to the manager. How this is done depends to some extant on the user interface.

5. Data Types

Data types marked with * cannot occur as constants in the compiler generated code.

Data Class	Names		Data Length	
01	Bool		1	
02	Char		1	
03	CObj		4	*
04	Err		1	*
05	FP			
		- FP-32	4	
		- FP-64	8	
06	Int			
		- Int-8	1	
		- Int-16	2	
		- Int-24	3	
		- Int-32	4	
07	Isa[3]			*
		- Isa-T-Var	0	
		- Isa-T-Fix	1	
		- Isa-U-Fix	2	
08	Isd's			*
		- Isd-T-Var-Short	0	
		- Isd-T-Fix-Short	1	

[3]Note that the data length field is an encoding of the type of I-structure address, and not the real length in bytes.

	- Isd-U-Fix-short	2	
	- Isd-T-Var	4	
	- Isd-T-Fix	5	
	- Isd-U-Fix	6	
11	Mdef	4	
12	MObj	4	*
13	Proc	4	
14	Smash	0-15	*
15	Bits	0-15	

5.1. Counter objects

A counter object consists of two parts—a 32 bit integer and an address where the acknowledgment is to be sent once the count goes to zero. The address can be a bit string of the form <PE, tag, nt, port>, a manager object, or a counter object (see Figure 5-1). Both the integer value as well as the return address are stored along with their data type.

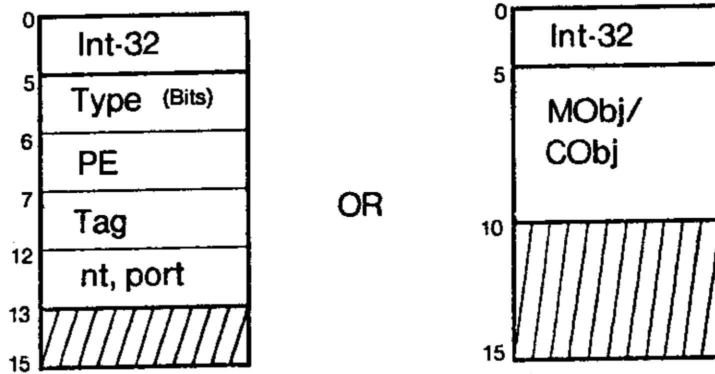

Figure 5-1: Counter Object

Since the system manager is responsible for managing the memory, allocation and deallocation of counter objects is left to the manager (*i.e.*, there are no explicit `Allocate-CObj` and `Deallocate-CObj` instructions). Instead, an instruction to initialize the counter object is provided.

5.2. I-structures

Three new type of I-structure descriptors (Isd-T-Var-Short, Isd-T-Fix-Short, and Isd-U-Fix-Short) have been added. Note that the length field in the I-structure descriptor is no longer the actual length of the descriptor, but simply an indication of its type. Given the type, length can be derived. Formats of I-structure descriptors have changed; the new formats are given below.

ISD-T-Var:

<Base-location>
<Base-PE-number> (8)

Field	Bits
<Base-address>	(24)
<Total-structure-elements>	(16)
<Mapping-information>	
<elements per group>	(16)
<Number of PE's>	(8)

ISD-T-FIX:

Field	Bits
<Base-location>	
<Base-PE-number>	(8)
<Base-address>	(24)
<Total-structure-elements>	(16)
<Mapping-information>	
<elements per group>	(16)
<Number of PE's>	(8)
<Element-length>	(4)

ISD-U-Fix:

Field	Bits
<Base-location>	
<Base-PE-number>	(8)
<Base-address>	(24)
<Total-structure-elements>	(16)
<Mapping-information>	
<elements per group>	(16)
<Number of PE's>	(8)
<Element-type>	
<Data-length>	(4)
<Data-class>	(4)

Short I-structure descriptors are always stored in the memory of one PE, and therefore don't need mapping information. Other than the lack of mapping information they are similar to their corresponding larger versions. The operations on I-structure descriptors (*e.g.*, **Form-Address**, **I-Fetch**) should be suitably extended to deal with the short descriptors.

Associated with an I-structure is a reference count which is kept at addresses immediately below the base address. Though the reference count is stored only on one PE, the locations are reserved on all the PE's over which the I-structure is distributed to simplify the memory management. We can use the base PE of the I-structure as the PE where reference count is stored; however, to distribute the work of maintaining the reference counts, we use the following simple scheme to determine the PE number.

$$PE_{\text{reference count}} = \text{Base-PE} + \text{Base-address}_{8\text{-}15} \; mod \; \text{Number-of-PE's}$$

where $\text{Base-address}_{8\text{-}15}$ is the number represented by bits 8 to 15 of the base address.

The information necessary to maintain the reference count for an I-structure is as follows: a 32-bit integer, a return address (manager object, counter object, or bit-string of the form <PE, tag, nt, port>), and the descriptor of the I-structure. We will need 24 bytes to store this information.

When the reference count associated with an I-structure reaches zero, the stored I-structure descriptor is forwarded to the return address. Note that reference counts of all the elements which happen to be I-structures must be decremented by one before storage used by an I-structure can be deallocated. Since deallocation of storage is done explicitly via the `Deallocate` instruction, we have left the task of decrementing the reference counts of component I-structure to `*Deallocate` family of I-structure controller operations.

5.3. Error values

The following new types of error values have been added.

1. `Illegal_argument`: An argument to the operator has wrong value.
2. `Device_read`: An error occurred while reading from a device and the operation was aborted.
3. `Device_write`: An error occurred while writing on a device and the operation was aborted.
4. `Too_many_destinations`: An instruction that require different values to be sent to different destinations (*e.g.*, `Expand, In-Fetch`) has more destinations than the number of values supplied.
5. `Never_written`: The requested I-structure element is not going to be produced as the I-structure has been deallocated.

6. Instructions as Specified in the Instruction Set

The compiler assumes the following opcodes for the instructions. An instruction marked with * is not used by the compiler at this time. The description of an instruction marked with ! has changed (see Section 7).

NAME	OPCODE	
ARITHMETIC		
+ c	001	
- c	002	
* c	003	
/ c	004	
↑ c	005	
+ nc	006	*
- nc	007	*
* nc	008	*
/ nc	009	*
↑ nc	010	*

arith-to-arith	011	*
char-to-int	012	*
int-to-char	013	*
//	014	

BOOLEAN

not bool	016	
ior bool	017	
and bool	018	
xor bool	019	*

BIT STRING

not 1	020	*
not 2	021	*
ior bits	022	*
and bits	023	*
xor bits	024	*
shift-left 0	032	*
shift-left 1	033	*
shift-left s	034	*
shift-left w	035	*
shift-right 0	036	*
shift-right 1	037	*
shift-right s	038	*
shift-right w	039	*
concatenate	040	*
adjust-lengthl0	041	*
adjust-lengthr0	042	*
adjust-lengthl1	043	*
adjust-lengthr1	044	*

RELATIONAL

<= c	048	
< c	049	
= c	050	*
~= c	051	*
> c	052	
>= c	053	
<= nc	054	*
< nc	055	*
= nc	056	*

Instruction	Code	
~= nc	057	*
> nc	058	*
>= nc	059	*
= any	060	
~= any	061	
EXTRACT/CONSTRUCT		
extract-type	064	*
extract-value	065	*
construct-data	066	*
I-STRUCTURE		
form-address	072	
i-fetch	073	*
form-address-i-fetch	074	
i-store	075	
form-address-i-store	076	
allocate	077	*!
deallocate	078	*!
SIGNALLING		
allocate-cobj	080	*!
deallocate-cobj	081	*!
decrement-cobj	082	*
ITERATION AND RECURSION		
D	088	!
R	089	!
D^{-1}	090	!
R^{-1}	091	
DATA MOVEMENT		
read-byte	096	*!
write-byte	097	*!
transfer	098	*!
INPUT/OUTPUT		
input-block	104	*!
output-block	105	*!
CONTROL		
exit	112	*
write-code-block-register	113	*!

identity	114	
switch	115	
set-supervisor-mobj	116	*!

MISC

use	120	
compress	121	
expand	122	!

In addition, the compiler assumes that the following instructions are available.

fix	130
float	131
sqrt	132
min	133
max	134
atan	135
sin	136
cos	137
remainder	138
abs	139
log	140

7. New Instructions and Changes to the Existing Instructions[4]

7.0.1. Allocate (077)[5]

Allocate no longer takes a manager or counter object. It takes two arguments: the first argument is an I-structure descriptor, and the second argument is an `Int-8` specifying a PE number where the storage is to be allocated. A d = 1 token is sent to the PE specified by the second argument. Note that, to allocate an I-structure that is distributed over n PE's, this instruction will have to be issued n times.

- **ISD × Int-8 ⇒**
  <d=1,PE_1,chain=0,<***Allocate**,<addr,elements-per-PE,element-length>>>
  or

  <d=1,PE_1,chain=1,<d=0,PE_2,tag,nt,port>,<***Allocate**,<addr,elements-per-PE,element-length>>>

- Paradigm: 2 or 3

---

[4]In the data field of a d = 1/2 type of token, a field written in bold is the data along with data class and data length field, otherwise it is just the appropriate number of bytes. Type **ANY** is used to denote any of the valid data types except the ISA's.

[5]The number within ( ) refers to the opcode of the instruction.

- Possible errors: `type_mismatch`

Elements-per-PE is the number of elements on each PE – this is different then the length per group carried on the token. In case the I-structure is of type `ISD-T-Var`(short or long), the element length defaults to one word (4 bytes).

7.0.2. Deallocate (078)

Similar to allocate, deallocate also doesn't take a manager or counter object as one of the arguments. The first argument is an I-structure descriptor, and the second argument is an `Int-8` specifying a PE number. A d = 1 token is sent to the PE specified by the second argument. Note that, just as in the case of `Allocate`, to deallocate space on n PE's, this instruction will have to be issued n times.

- `ISD-T-Var` × `Int-8` ⇒
  <d=1,PE_1,chain=0,<`*Deallocate-T-Var`,<addr,elements-per-PE>>>
  or

  <d=1,PE_1,chain=1,<d=0,PE_2,tag,nt,port>,<`*Deallocate-T-Var`,<addr,elements-per-PE>>>

- `ISD-T-Fix` × `Int-8` ⇒
  <d=1,PE_1,chain=0,<`*Deallocate-T-Fix`,<addr,elements-per-PE, element-length>>>
  or

  <d=1,PE_1,chain=1,<d=0,PE_2,tag,nt,port>,<`*Deallocate-T-Fix`,<addr,elements-per-PE, element-length>>>

- `ISD-U-Fix` × `Int-8` ⇒
  <d=1,PE_1,chain=0,<`*Deallocate-U-Fix`,<addr,elements-per-PE,element-length, element-type>>>
  or

  <d=1,PE_1,chain=1,<d=0,PE_2,tag,nt,port>,<`*Deallocate-U-Fix`,<addr,elements-per-PE, element-length, element-type>>>

- Paradigm: 2 or 3

- Possible errors: `type_mismatch`

The behavior for short descriptors is the same as above.

7.0.3. Allocate-Cobj (080) and Deallocate-CObj (081)

These instructions no longer exist. Instead, the manager is responsible for allocating and deallocating counter objects.

7.0.4. Set-CObj (083)

This instruction is used to initialize a counter object. It takes two arguments: a counter object and a smash type. Smash type itself contains two components: a 32 bit integer value, and a return address (*i.e.*, a manager object, counter object or a bit string of the form <PE, tag, nt, port>).

- `CObj × Smash ⇒`
  <d=2,PE_1,chain=0,<*`Set-CObj`,<CObj-addr,`Int-32`, (`BITS`∨`MObj`∨`CObj`)>>>
  or
  <d=2,PE_1,chain=1,<d=0,PE_2,tag,nt,port>,<*`Set-CObj`,<CObj-addr,`Int-32`, (`BITS`∨`MObj`∨`CObj`)>>>

- Paradigm: 2 or 3

- Possible errors: `type_mismatch`

PE_1 is the PE where the counter object resides, and CObj-addr is the 24 bit local address.

7.0.5. D (088)

It doesn't check the next color field associated with the color field of the incoming tag; that is left to a separate instruction. If the initiation-number overflows, it passes the value without changing its initiation-number to destinations specified in the second destination list.

7.0.6. R (089)

The mapping algorithm for the R operator is slightly different than the mapping algorithm for the D operator. The value of initiation-number on the token is always the calculated value (i.e., a*i + b) regardless of the subdomain to which the token is sent.

7.0.7. D^{-1} (090)

It reads the primary color and base register number associated with the color field of the incoming tag (see Figure 1-1), and uses them to calculate the new tag.

7.0.8. Read-Bytes (096)

This operation replaces the `Read-byte` instruction. It takes two arguments: a 32 bit integer that specifies the starting address, and a 8-bit integer that specifies the number of bytes to read. At most 15 bytes in the address space of a single PE can be read. The result is returned as a bit string.

- `Int-32 × Int-8 ⇒`
  <d=2,PE_1,chain=1,<d=0,PE_2,tag,nt,port>,<*`Read-Bytes`,<addr, length >>

- Paradigm: 3

- Possible errors: `type_mismatch, illegal_argument`

Addr is the 24 bit local address, and length is one byte.

7.0.9. Write-bytes (097)

This operation replaces the `write-byte` instruction. It takes two arguments: a 32-bit integer specifying the starting address and a bit string (1 to 15 bytes in length). The addresses where the bit string is to be stored must be in the address space of a single PE. At most one destination where an acknowledgment is to be sent may be specified.

- `Int-32 X Bits` ⇒
  <d=2,PE_1,chain=0,<`*Store-in-Memory`,<addr,data>>
  or
  <d=2,PE_1,chain=1,<d=0,PE_2,tag,nt,port>,<`*Store-in-Memory`,<addr, length, data>>

- Paradigm: 2 or 3

- Possible errors: `type_mismatch, illegal_argument`

Addr is the 24 bit local address. The number of bytes in the data is specified by the 24 bit length field.

7.0.10. Transfer (098)

A destination may or *may not* be specified.

7.0.11. Expand (122)

The way mask and destination list are interpreted has changed. It takes two arguments: The first argument is of `smash` type, and the second argument is of `Bits` type. The most significant bits of the second argument are interpreted as a mask.

The operation of the instruction can be explained as follows: First, a list of data values contained in the `smash` type is formed. Then a new list is formed by discarding the values for which the corresponding mask bit is zero; the most significant bit, if zero, will cause the first data value to be discarded. The first datum in this new list 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. If there are more <*Destination*>s than data values, error values (`too_many_destinations`) will be sent to the remaining destinations.

- `Smash X Bits` ⇒ {$\langle Data \rangle_i$ ∨ `Error`| $1 \leq i \leq n$ where n is the number of destinations}

- Paradigm: 1

- Possible errors: `type_mismatch, too_many_destinations`

7.0.12. Input-from-Device (104)

This operation replaces the `input-block` instruction. It takes three arguments: the first argument is either a 16 bit device specification or a smash type of 16 bit device specification and 32 bit device address, the second argument is a 32 bit integer specifying the destination address, and

the third argument is a 24 bit integer specifying the block length in bytes. Note that one of the operands must be constant. At most one destination where the acknowledgment is to be sent may be specified.

- **Int-16 × Int-32 × Int-24 ⇒**
  <d=2,PE_1,chain=0,<***Input-from-Device**,<destination-addr,length,device>>
  or
  <d=2,PE_1,chain=1,<d=0,PE_2,tag,nt,port>,<***Input-from-Device**,<destination-addr, length,device>>

- **Smash × Int-32 × Int-24 ⇒**
  <d=2,PE_1,chain=0,<***Input-from-Device**,<destination-addr,length,device,device address>>
  or
  <d=2,PE_1,chain=1,<d=0,PE_2,tag,nt,port>,<***Input-from-Device**,<destination-addr, length,device,device-address>>

- Paradigm: 2 or 3

- Possible errors: **type_mismatch**

PE_1 is the PE number to which the device is connected, and device is the <device group, device number> part of the device specification.

7.0.13. Output-to-device (105)

This operation replaces output-block. It takes three arguments: the first argument is a 32 bit integer specifying the source address, the second argument is either a 16 bit device specification or a smash type of 16 bit device specification and 32 bit device address, and the third argument is a 24 bit integer specifying the block length. Note that one of these operands must be constant. At most one destination may be specified. The result sent to the destination is either an acknowledgment or a smash type of the device specification and the device address where the block was written (if such an address makes sense and was not specified).

- **Int-32 × Int-16 × Int-24 ⇒**
  <d=2,PE_1,chain=0,<***Read-and-Forward**,<addr,length,device-specification>>
  or
  <d=2,PE_1,chain=1,<d=0,PE_2,tag,nt,port>,<***Read-and-Forward**,<addr,length, device-specification>>

- **Int-32 × Smash × Int-24 ⇒**
  <d=2,PE_1,chain=0,<***Read-and-Forward**,<addr,length,device-specification, device-address>>
  or
  <d=2,PE_1,chain=1,<d=0,PE_2,tag,nt,port>,<***Read-and-Forward**,<addr,length, device-specification,device-address>>

- Paradigm: 2 or 3

- Possible errors: **type_mismatch**

PE_1 is the PE number derived from the source address.

7.0.14. Write-Base-and-Map-Register (113)

This operation replaces the operation write-code-block-register. It takes two operands: an Int-16 specifying a PE number and a base register number, and a bit string. Bit string is of the form <code-base-address (3), data-area-pointer (3), domain-size (1), domain-base-PE (1), code-per-PE (2), subdomain-size (1), subdomain-base-PE (1), iterations-per-subdomain (1), last-subdomain-flag (1)> where the number in parentheses is the byte length of the field.

- **Int-16 × BITS ⇒**
  <d=2,PE_1,chain=0,<***Write-Base-and-Map-Register**,<register-number,data>>
  or
  <d=2,PE_1,chain=1,<d=0,PE,tag,nt,port>,<***write-base-and-map-register,**
  <register-number,data>>

- Paradigm: 2 or 3

- Possible errors: **type_mismatch**

7.0.15. Set-Supervisor-MObj (116)

A destination may or *may not* be specified.

7.0.16. Length(071)

This operation returns the number of elements in an I-structure.

- **ISD ⇒ Int-16**

- Paradigm: 0

- Possible errors: **type_mismatch**

7.0.17. Constant-Store (099)

The intended use of this instruction is to store constants in the constant area. It takes two arguments: The first argument is a value, and the second argument is an integer offset. It generates n d = 2 type tokens where n is the number of PE's in the physical domain. The values stored in the constant area are assumed to be of the same length, usually 16 bytes.

- **ANY × Int ⇒**
  {<d = 2,PE_i,chain = 0, <***Constant-Store,** <Base-address,Address-offset, **Any**>> | $1 \leq i \leq n$}

- Paradigm: 2

- Possible errors: `type_mismatch`

Base-address is the base address of the constant area associated with the color, and address-offset is integer offset multiplied by 16.

7.0.18. In-Fetch (79)

This instruction is similar to `form-address-i-fetch` except that it fetches a number of consecutive elements starting at a specified selector. The first argument is an I-structure descriptor, the second argument is an integer specifying the starting value of the selector, and the third argument (n) is an integer specifying the number of elements to read. The actual number of values that are read is given by *min*(n, number of destinations). The first fetched value is forwarded to the first destination, the second to the second destination, etc. If the number of destinations is more than n, error values (`too_many_destinations`) are forwarded to the remaining destinations. Note that one of the arguments must be a constant.

- `ISD-T-Var` × `Int` × `Int` ⇒
  {<d=1,PE_i,chain=1,<d=0,PE_2,tag,nt,port>,<*`I-Fetch-T`,$addr_i$>> | $1 \leq i \leq n$}

- `ISD-T-Fix` × `Int` × `Int` ⇒
  {<d=1,PE_i,chain=1,<d=0,PE_2,tag,nt,port>,<*`I-Fetch-T`,$addr_i$>> | $1 \leq i \leq n$}

- `ISD-U-Fix` × `Int` × `Int`⇒
  {<d=1,PE_i,chain=1,<d=0,PE_2,tag,nt,port>,<*`I-Fetch-U`,$addr_i$, type>> | $1 \leq i \leq n$}

- Paradigm: 3

- Possible errors: `type_mismatch, illegal_index, too_many_destinations`

The behavior of the instruction for short descriptors is the same except that the PE number is always the base PE.

7.0.19. Set-RC (123)

This instruction is used to initialize the reference count associated with an I-structure. It takes two arguments: an I-structure descriptor and a smash type. The smash type itself contains two components: a 32 bit positive integer, and a return address (*i.e.*, a manager object, counter object, or a bit string of the form <PE, tag, nt, port>.

- `ISD` × `Smash` ⇒
  <d=1,PE_1,chain=0,<*`Set-RC`,<Base-address,`Int-32`,(`BITS`∨`MObj`∨`CObj`),`ISD`>>>
  or
  <d=1,PE_1,chain=1,<d=0,PE_2,tag,nt,port>,<*`Set-RC`,<Base-address,`Int-32`,
  (`BITS`∨`MObj`∨`CObj`), `ISD`>>>

- Paradigm: 2 or 3

- Possible errors: `type_mismatch, illegal_argument`

PE_1 is the PE where the reference count is stored; it is derived following the hashing scheme described in Section 5.2.

7.0.20. Increment-RC (124)

It increments the reference count associated with an I-structure. The first argument is an I-structure descriptor and the second argument is a positive integer. At most one destination where an acknowledgment is to be sent may be specified.

- `ISD X Int` ⇒
  <d = 1,PE_1, chain = 0 <`*Update`, <Base-address,`Int`>>>
  or
  <d = 1,PE_1,chain = 1, <d = 0,PE_2,tag,nt,port>, <`*Update`, <Base-address, `Int`>>>

- Paradigm: 2 or 3

- Possible errors: `type_mismatch, illegal_argument`

PE_1 is the PE where the reference count is stored; it is derived following the hashing scheme described in Section 5.2.

7.0.21. Decrement-RC (125)

It decrements the reference count associated with an I-structure. The first argument is an I-structure descriptor, and the second argument is a negative integer specifying the value to be added to the reference count. The acknowledgment, if needed, is generated by ALU and not by the I-structure controller.

- `ISD X Int` ⇒
  <d = 1,PE_1, chain = 0 <`*Update`, <Base-address, `Int`>>>
  and
  ∅ ∨ <d = 0,PE_2,tag,nt,port, <data>>

- Paradigm: 2

- Possible errors: `type_mismatch, illegal_argument`

PE_1 is the PE where the reference count is stored; it is derived following the hashing scheme described in Section 5.2.

7.0.22. Extract-Tag (067)

This operation returns the tag on the token as a bit-string of the form <PE, base-register number, color, initiation-number>. The PE number included in the bit string is the subdomain-base-PE.

- `ANY ⇒ BIT-STRING`

- Paradigm: 0

- Possible errors: none

7.0.23. Set-Tag (068)

This operation takes two arguments: a value, and a bit string of the form <PE, base-register number, color, initiation-number> which is used to compute the tag on the output token. The PE-OFFSET and ADDR-OFFSET are calculated as usual. The tag on the output token is set to <PE + PE-OFFSET, base-register number, color, initiation-number, ADDR-OFFSET>.

- ANY × BITS ⇒ ANY

- Paradigm: 0

- Possible errors: type_mismatch

7.0.24. Construct-Token (069)

This operation takes two arguments: a value of any type, and a bit string of the form <PE, base-register number, color, initiation-number, relative address, nt, port>. A token with tag equal to the bit string and value equal to the value specified by the first argument is constructed. Acknowledgment tokens are sent to all destinations.

- ANY × BITS ⇒ ANY

- Paradigm: 0

- Possible errors: type_mismatch

7.0.25. Next-Color(126)

This instruction has two sets of destinations. It checks the next color field associated with the color field of the incoming tag. If a color is available via the next color field, it generates a bit-string of the form <PE, base-register number, new color, 0> and sends it to destinations in the first list; otherwise, it passes the token to destinations in the second list. The PE number included in the bit-string is base PE of the *current* subdomain if ((255 *mod* k) + 1) *mod* k is not zero; otherwise, it is the base PE of the *next* subdomain. If k is a power of 2 then ((255 *mod* k) + 1) *mod* k is always 0.

At present, the next color field is actually a bit (color-continuation flag) specifying whether the next sequential color can be used or not, and the new color, if available, is always color + 1.

- ANY ⇒ BITS
  or
  ANY ⇒ ANY

- Paradigm: 0

- Possible errors: none

7.0.26. Use-Immediate (126)

This is similar to the USE instruction except that the acknowledgment, if needed, is generated by ALU.

- `MObj` × **ANY** ⇒
  <d=2,PE_1,chain=0,<`*Entry`,<MObj-addr,**ANY**>>>
  and
  ∅ ∨ <d = 0, PE, tag. nt, port, <acknowledgment>>

- Paradigm: 2

- Possible errors: `type_mismatch`

8. I-structure Controller Operations

These operation may return an explicit value with data length and class fields, an acknowledgment (written **ACK** and probably encoded as boolean true), or nothing at all (∅).

8.0.1. *Allocate (1)

This operation is used to set the status bits of the storage allocated to an I-structure. Note that there is only one `*Allocate` operation.

- Addr × Elements-per-PE × Element-length ⇒ **Ack** ∨ ∅

- Possible errors: none

8.0.2. *Deallocate-T-Var (2), *Deallocate-T-Fix (3), *Deallocate-U-Fix (4)

These instructions deallocate the storage assigned to an I-structure, *i.e.*, set the status bits appropriately. They are also responsible for decrementing the reference counts of all the elements that happen to be I-structure, and for sending error values (`never_written`) to any pending read requests.

1. `*Deallocate-T-Var`:

   - Addr × Elements-per-PE ⇒ (**Ack** ∨ ∅)
     and
     {<d=1,PE_i,chain=0,<`*update`,<ISD_i,-1>>>|ISD_i is an element of the I-structure}
     and
     {<d=0,PE,tag,nt,port, <`never_written`>>| for all pending read requests}

2. `*Deallocate-T-Fix`:

   - Addr × Elements-per-PE × Element-length ⇒ (**Ack** ∨ ∅)
     and
     {<d=1,PE_i,chain=0,<`*update`,<ISD_i,-1>>>|ISD_i is an element of the I-structure}
     and

{<d=0,PE,tag,nt,port, <never_written>>| for all pending read requests}

3. *Deallocate-U-Fix:

   - Addr × Elements-per-PE × Element-length × Element-type ⇒ (Ack ∨ Ø)
     and
     {<d=1,PE_i,chain=0,<*update,<ISD_i,-1>>>|ISD_i is an element of the I-structure}
     and
     {<d=0,PE,tag,nt,port, <never_written>>| for all pending read requests}

- Possible errors: never_written

8.0.3. *Set-RC (5)

This operation initializes the reference count associated with an I-structure. It receives the base-address of the I-structure. Given the base-address, it can calculate the address where reference count is stored by subtracting a fixed offset (24 bytes).

- Base-address × Int-32 ×(MObj ∨ CObj ∨Bits)× ISD ⇒ Ack ∨ Ø

- possible errors: none

8.0.4. *Update (6)

This operation is used to increment or decrement the reference count. It receives the base-address of the I-structure and a 32 bit integer value. Given the base-address, it can calculate the address where reference count is stored by subtracting a fixed offset (24 bytes). The integer value, which may be positive or negative, is added to the reference count.

- Base-address × Int-32 ⇒ ACK ∨ Ø

- Possible errors: none

If the reference count goes to zero then a token of the following type is generated. Note that the I-structure descriptor included in the token is the descriptor stored with the reference count.

- <d=0,PE,tag,nt,port, <ISD>>
  or
  <d=2, PE_1, <*Entry <MObj-address, <Zero-RC,ISD>>>>
    where Zero-RC is the name of manager request (see Section 10).
  or
  <d=2, PE_1, <*Decrement-CObj, <CObj-address>>>

8.0.5. *I-Fetch-T (7), *I-Fetch-U (8), *I-Store-T-Fix (9), *I-Store-T-var (10), *I-Store-U-Fix (11)

See [2].

9. PE Controller Operations

The data coming on a d = 2 token is treated as a sequence of bytes; the exact interpretation of the data is based on the opcode. These operation may return an explicit value with data length and class fields, an acknowledgment (written **ACK** and probably encoded as boolean true), or nothing at all (Ø). If an error occurs during the processing an error token is sent to the destination specified on the token. If no destination is specified, a token of the following form is sent to the manager. Note that the manager address is accessible through a special register.

- <d=2,PE_1,chain=0,<*Entry,<MObj-addr,<Handle-error,ERROR,PE-controller-number, opcode-which-caused-error>>>>
  where **Handle-error** is the name of manager request.

9.0.1. *Entry (1) and *Exit (2)

See [2].

9.0.2. *Constant-Store (3)

This operation takes the base-address of the constant area, an address-offset and a value. The value is stored at base-address + address-offset. The mechanism to detect when all the constant have been written uses two level of counter objects. Counter objects at the first level are stored in the first location of constant areas. There is only one counter object at the second level, and it is stored in the second location of the constant area in domain-base-PE. The second location in all other constant areas is left unused to simplify the allocation of memory.

Once the value is stored, the counter object in the first location of the constant area is decremented. If the count goes to zero, action similar to ***Decrement-CObj** is taken.

- Base-address × Address-offset × **ANY** ⇒ Ø

- Possible errors: none

9.0.3. *Set-CObj (4)

This operation initializes the counter object stored at the address specified by CObj-addr.

- CObj-addr × `Int-32` × `(MObj ∨ CObj ∨ BITS)` ⇒ **Ack** ∨ Ø

- Possible errors: none

9.0.4. *Decrement-CObj (5)

The value of the counter object at the specified address is decremented. If the count goes to zero, an appropriate token is sent to the return address stored in the counter object.

- CObj-addr ⇒
  <d=0,PE,tag,nt,port, <ACK>>
  or

<d=2, PE_1, <*Entry <MObj-address, <Zero-CObj,CObj>>>>
where Zero-CObj is the name of manager request.
or
<d=2, PE_1, <*Decrement-CObj, <CObj-address>>>

- Possible errors: none

9.0.5. *Read-Bytes (6)

This operation reads 1 to 15 number of bytes starting at the address specified by the address field, and returns them as a bit string. Note that when reading from the I-structure memory, invisible pointers are not traced.

- Address × Length ⇒ <d=0,PE,tag,nt,port, <Bits>>

- Possible errors: none

9.0.6. *Store-in-Memory (7)

This operation stores the data at specified addresses. The starting address is given by the address field. The number of bytes in the data is specified by the length field.

- Address × Length × Data ⇒ ACK ∨ Ø

- Possible errors: None

9.0.7. *Input-from-Device (8)

This operation is used to read data from a device and store it at a specified address. The device specification is of the form <device group, device number>. If the device number is 0, any device in the group is selected. The length of data to read in number of bytes is given by the length field. A device address may or may not be specified.

The destination-address is of the form <PE_2, addr>. If PE_2 is the same as the PE on which this instruction is executing, the data is simply written in the local memory; otherwise, a d = 2 token is sent to PE_2.

- Destination-address × Length × Device ⇒
  Ack ∨ Ø
  or
  <d=2,PE_2,chain = 0, <<*Store-in-Memory,<Addr, length,data>>>
  or
  <d=2,PE_2,chain = 1,<d=0,tag,nt,port>,<<*Store-in-Memory,<Addr, length, data>>>

- Destination-address × Length × Device × Device-address ⇒
  Ack ∨ Ø
  or
  <d=2,PE_2,chain=0,<*Store-in-Memory,<Addr, length, data>>>
  or

<d=2,PE$_2$,chain=1,<d=0,PE,tag,nt,port>,<*Store-in-Memory,<Addr,length, data>>>

- Possible errors: **device_read**

Chain and destination address (if any) are copied from the incoming token.

9.0.8. *Read-and-Forward (9)

This operation reads specified number of bytes from the local memory, and writes them on the specified device. The device-specification is of the form <PE$_2$, device>. If PE$_2$ is the same as the PE on which this instruction is executing, the data is simply written on the specified device (see the description of ***output-to-device** instruction for this case); otherwise, it is sent to to the PE controller responsible for managing the device. Note that when reading from the I-structure memory, invisible pointers are not traced.

- Address × Length × Device-specification ⇒
  Ack ∨ ∅ ∨ <d=0,PE,tag,nt,port, <**Smash**>>
  or
  <d=2,PE$_2$,chain=0,<***Output-to-Device**,<device,length,data>>>
  or
  <d=2,PE$_2$,chain=1,<PE,tag,nt,port>,<***Output-to-Device**,<device,length,data>>>

- Address × Length × Device-specification × Device-address ⇒
  Ack ∨ ∅ ∨ <d=0,PE,tag,nt,port, <**Smash**>>
  or
  <d=2,PE$_2$,chain=0,<***Output-to-Device**,<device,length,data,device-address>>>
  or
  <d=2,PE$_2$,chain=1,<PE,tag,nt,port>,<***Output-to-Device**,<device,length,data, device-address>>>

- Possible errors: **device_write**

Chain and destination tag are copied from the incoming token.

9.0.9. *Output-to-Device (10)

This operation writes the data on a device. The device is specified as a tuple of the form <device group, device number>. If the device number is zero, any device in the group is selected. If a device address is specified, the data is written at the specified address; otherwise, it is written at an address chosen by the PE controller. An acknowledgment or a smash type of the device specification (**Int-16**) and the device address (**Int-32**) is returned as the result.

- Device × Length × Data ⇒ ∅ ∨ **Ack** ∨ <d=0,PE,tag,nt,port, <**Smash**>>

- Device × Length × Data × Device-address ⇒ ∅ ∨ **Ack** ∨ <d=0,PE,tag,nt,port, <**Smash**>>

- Possible errors: **device_write**

9.0.10. *Transfer (11)

This operation reads specified number of bytes from the local memory, and is responsible for storing them at the address specified by <dest-PE, dest-addr> pair. If the dest-PE is the same as the PE on which this instruction is executing, the data is simply stored in the local memory; otherwise, a $d = 2$ token is sent to the dest-PE. Note that when reading from the I-structure memory, invisible pointers are not traced.

- Source-addr × Dest-PE × Dest-addr × Length ⇒
  `Ack` ∨ ∅
  or
  <d=2,dest-PE,chain=0,<`*Store-in-Memory`,<dest-addr,length,data>>>
  or
  <d=2,dest-PE,chain=1,<d=0,PE,tag,nt,port>,<`*Store-in-Memory`,<dest-addr,
  length,data>>>

- Possible errors: none

Chain and destination tag are copied from the incoming token.

9.0.11. *Write-Base-and-Map-Register (12)

This operation sets the specified base register and the corresponding map register.

- Register-number × Data ⇒ `ACK` ∨ ∅

- Possible errors: none

Data is of the form <code-base-address, data-area-pointer, domain-size, domain-base-PE, code-per-PE, subdomain-size, subdomain-base-PE, iterations-per-subdomain, last-subdomain-flag>. Code-base-address and data-area-pointer are stored in the base register, and the rest of the information along with the data-area-pointer is stored in the corresponding map register in the compute-tag section.

9.0.12. *Set-Supervisor-MObj (13)

Specified manager object is stored in the special register in the PE controller.

- `MObj` ⇒ `ACK` ∨ ∅

- Possible errors: none

10. Requests to the System Manager

The system manager may be asked to satisfy the following request; note that this list is by no means complete. The argument to the manager is usually a smash type; the first field of the smash type is an `Int-8` request identifier. If a return address is specified but there is no explicit result to return, an acknowledgment should be returned.

1. **Invoke a new code-block:**

   - Argument supplied: <1, code-block-identifier, number of arguments to be passed, (number of iterations)>[6]

   - Result expected: A smash type of the form <argument I-structure descriptor, result I-structure descriptor>

   In the case of a loop, the manager is also responsible to do the following things.

   a. To allocate a constant area of appropriate size. The number of constants that would be stored is specified in the code-block header.

   b. To initialize the data area associated with all the colors allocated to the loop.

   c. To allocate and initialize counter objects used by the `constant-store` instructions. The first level counter objects should be stored in the first location of the constant areas. The count of these counter objects should be initialized to the number of constants to be stored in the constant area, and the return address should point to the second level counter object. The second level counter object, with an initial count of n where n is the number of PE's in the domain, should be stored in the second location of the constant area on the domain-base-PE. This counter object sends the acknowledgment to an operator in the code-block, thus its return address is a proper tag. The address of the operator is specified in the code-block header, and the manager is responsible for generating the proper tag.

2. **`Allocate a new color`:** This is a request to allocate a new color for an already executing loop.

   - Argument supplied: <5, tag as a bit string>

   - Result Expected: A bit string of the form <PE number, base-register number, color, 0>

   If the manager decides to create a new copy of the code-block, it must be loaded in the same domain with the same mapping parameters as the old code-block. Base-address is the base address of the code-block – either new or old. PE number is calculated using the scheme followed by the `Next-color` operator. The manager is also responsible for initializing the data area associated with the color.

3. **`Release color`:**

   - Argument supplied: <2, tag as a bit-string>

   - Result expected: none

[6] Fields in () may be omitted with all following fields also omitted

The manager will receive two requests to release the primary color assigned to a loop. The primary color should be released only when both the requests have been received.

4. `Allocate a new I-structure:`

   - Argument supplied: <3, number of elements, (data class), (data length), (n) (d)> where n and d are mapping parameters.

   - Result expected: I-structure descriptor

5. `Zero-RC:` This request is made when the reference count associated with an I-structure goes to zero. The I-structure should be deallocated.

   - Argument supplied: <4, I-structure descriptor>

   - Result expected: none

6. `Allocate a Counter Object:` It is a request to allocate a counter object. The PE where the counter object is to be allocated may or may not be specified.

   - Argument supplied: <6, (PE number)>

   - Result expected: Counter object

7. `Deallocate the Counter Object:`

   - Argument supplied: <7, Counter Object>

   - Result expected: none

8. `Zero-CObj:` This request is made when the value in a counter object reaches zero.

   - Argument supplied: <8, Counter object>

   - Result expected: none

9. `Handle-error:` This request is made when an error occurs during the processing of a d = 2 type token.

   - Argument supplied: <7, error value, PE controller number, opcode which caused the error>

   - Result expected: none

References

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

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

Table of Contents

1. Changes to the Architecture 1
2. Color Management 4
3. Managers and PE Controllers 6
4. PE controller, I/O Devices and Master Copy of Code-blocks 6
5. Data Types 7
   5.1. Counter objects 8
   5.2. I-structures 8
   5.3. Error values 10
6. Instructions as Specified in the Instruction Set 10
7. New Instructions and Changes to the Existing Instructions[7]
   7.0.1. Allocate (077)[8]
   7.0.2. Deallocate (078) 14
   7.0.3. Allocate-Cobj (080) and Deallocate-CObj (081) 14
   7.0.4. Set-CObj (083) 15
   7.0.5. D (088) 15
   7.0.6. R (089) 15
   7.0.7. D^{-1} (090) 15
   7.0.8. Read-Bytes (096) 15
   7.0.9. Write-bytes (097) 16
   7.0.10. Transfer (098) 16
   7.0.11. Expand (122) 16
   7.0.12. Input-from-Device (104) 16
   7.0.13. Output-to-device (105) 17
   7.0.14. Write-Base-and-Map-Register (113) 18
   7.0.15. Set-Supervisor-MObj (116) 18
   7.0.16. Length(071) 18
   7.0.17. Constant-Store (099) 18
   7.0.18. In-Fetch (79) 19
   7.0.19. Set-RC (123) 19
   7.0.20. Increment-RC (124) 20
   7.0.21. Decrement-RC (125) 20
   7.0.22. Extract-Tag (067) 20
   7.0.23. Set-Tag (068) 21
   7.0.24. Construct-Token (069) 21
   7.0.25. Next-Color(126) 21
   7.0.26. Use-Immediate (126) 22
8. I-structure Controller Operations 22
   8.0.1. *Allocate (1) 22

---

[7]In the data field of a d = 1/2 type of token, a field written in bold is the data along with data class and data length field, otherwise it is just the appropriate number of bytes. Type **ANY** is used to denote any of the valid data types except the ISA's.

[8]The number within ( ) refers to the opcode of the instruction.

8.0.2. *Deallocate-T-Var (2), *Deallocate-T-Fix (3), *Deallocate-U-Fix (4) 22
8.0.3. *Set-RC (5) 23
8.0.4. *Update (6) 23
8.0.5. *I-Fetch-T (7), *I-Fetch-U (8), *I-Store-T-Fix (9), *I-Store-T-var (10), *I-Store-U-Fix (11) 23
9. PE Controller Operations 24
9.0.1. *Entry (1) and *Exit (2) 24
9.0.2. *Constant-Store (3) 24
9.0.3. *Set-CObj (4) 24
9.0.4. *Decrement-CObj (5) 24
9.0.5. *Read-Bytes (6) 25
9.0.6. *Store-in-Memory (7) 25
9.0.7. *Input-from-Device (8) 25
9.0.8. *Read-and-Forward (9) 26
9.0.9. *Output-to-Device (10) 26
9.0.10. *Transfer (11) 27
9.0.11. *Write-Base-and-Map-Register (12) 27
9.0.12. *Set-Supervisor-MObj (13) 27
10. Requests to the System Manager 27

List of Figures

Figure 1-1: Base Registers, Map Registers, and Constant Areas 2
Figure 1-2: Compute-tag and Construct-token Sections 3
Figure 2-1: Compiler Generated Code to Get a New Color 5
Figure 5-1: Counter Object 8