LABORATORY FOR
CONCURRENT COMPUTING SYSTEMS
COMPUTER SYSTEMS ENGINEERING
School of Electrical Engineering
Swinburne Institute of Technology
John Street, Hawthorn 3122, Victoria, Australia.

The CSIRAC II Dataflow Computer

Token and Node Definitions

Technical Report 31-001

G.K. Egan

School of Electrical Engineering
Swinburne Institute of Technology
John Street
Hawthorn 3122
Australia

Original Document October 1979 Victoria University of Manchester
Revision 2.7 May 1990

Abstract:

This document the architecture of CSIRAC II developed originally at the Victoria University of Manchester. The architecture falls outside the accepted of taxonomy of static and dynamic architectures as it contains features of both.

1. INTRODUCTION

This document describes the Token and Node Set of CSIRAC II. The original forms of which are described in [1] and were implemented on a pilot multi-processor in 1978. The system does not conform to the normally accepted two-class (static [2] and dynamic [3]) taxonomy of data-flow systems.

1.1 General Characteristics

Although it is not the intention of the document to provide an overview of the architecture, which may be found in [4], the following is presented for context:

1) The system hardware consists of a number of processing and structure-store elements, communicating over a multi-stage packet switched network.

2) Tokens are strongly typed and of variable length. The token-types are not constrained to simple objects but may be complex e.g. the node descriptions which comprise the graph to be evaluated.

3) Node-functions are weakly typed and accept a set of argument types; this increases graph generality while reducing the size of the node-function set. Type coercion is performed where *sensible*. The node functions are of fine to medium granularity e.g. arithmetic and vector inner product respectively.

4) A variety of structure manipulating mechanisms are provided including both stored and transmitted lists, vectors and records. These are supported in sufficient generality to allow for example lists of lists of vectors.

5) Input-output is accomplished using system nodes. The names of these nodes are associated with particular input or output devices, which in turn are associated with particular processing- elements. As these nodes already *exist* within the system, they must be linked into the graph at load or evaluation time; this is done by sending response-destination tokens to the appropriate nodes.

6) The architecture supports re-entrant sub-graphs in sufficient generality to allow multiple concurrent mutual recursion. Tokens involved in concurrent invocations of a shared sub-graph are separated by means of a colour. Temporal ordering of tokens is preserved by strict queuing of tokens of the same colour on any given arc.
Tokens of differing colours may overtake allowing full unravelling of re-entrant graphs.

1.2 Emulation Facility

A multi-processor facility has been constructed to support message passing MIMD architecture studies [7]. Considerations of emulation efficiency have had some impact on the representations of objects described in this document and therefore object field definitions should be regarded as mutable.

2. TOKENS

A BNF like notation is used for definitions. All tokens carry name, data fields and as required a colour tag distinguishing it from other invocations of the target node. The data fields contain type, length and data. Tokens are of variable length and the data may consist of none to several datum of simple to complex type. In the current implementation all objects in the system are described by a sequence of one or more 16 bit words.

<token>::=

< name ><colour>< data fields>

2.1 Name

<name> ::=

< process >	< element >	< element object >	< input point >	< monadic>
8	8	22	1	1

<process>::= distinguishes separate graphs running concurrently in the system. This field is set by the loader.

<element>::= the physical processing or structure storage element to which the token is directed.

<element object>::= the name of the object within the partition of the graph or stored objects assigned to the target element.

<input point>::= specifies to which input point of the <element object> the token is directed.

<monadic>::= when **true** indicates that the matching process may be bypassed.

2.2 Colour

<colour>::= used as a qualifier for the statically allocated <name> fields linking <nodes> in shared invokations.The <colour> is an internally generated unique 38 bit (See Colour Functions).

2.3 Data Fields

<data fields>::=

< type >	< data >|
8	

< type >	< >	< data >|
	8	

< type >	< length >	< data >|
	24	

< type >< length >< low bound>< data >
16

<type>::=	the type of the data (See Token Types).
<length>::=	where appropriate the length of the vector in units of datum length.
<low bound>::=	two's complement vector low bound for vector types.
<data>::=	the token's data.

3. TOKEN TYPES

The specific forms of <data fields> are described in the following sections. In general vectors are transmitted least significant element first (element 0).

Some variants of the generic types are not currently implemented in the simulators and emulators (See Implementation Restrictions).

3.1 Simple Types

3.1.1 Real

A reference to **real** implies any one of the following <types>.

< **real32** >< >< IEEE single precision>
8 32

< **real64** >< >< IEEE single precision (ms words first)>
64

< **real32_vector** >< length ><low bound>{< IEEE single precision>}
32

< **real64_vector** >< length ><low bound>{< IEEE double precision>}
64

3.1.2 Integer

A reference to **int** implies any one of the following <types>.

< **int8** >< two's complement integer>
8

< **int16** >< >< two's complement integer>
8 16

< **int32** >< >< two's complement integer>
8 32

< **int8_vector** >< length ><low bound>{< two's complement integer>}
8

< **int16_vector** >< length ><low bound>{< two's complement integer>}
16

< **int32_vector** >< length ><low bound>{< two's complement integer>}
32

3.1.3 Character

A reference to **chars** implies any of the following types.

< **char** >< ASCII char >
8

< **char_vector** >< length ><low bound>{< ASCII char >}
8

Note: character vectors are ASCII **nul** padded to least significant byte of least significant i.e. last transmitted <word> boundary.

3.1.4 Bit

A reference to **bits** implies any of the following types.

< **bit** >< >{< **true** | **false** >}
7 1

< **bit _vector** >< length ><low bound>{< **true** | **false** (ls bit first>}
1

< **true** >::= single bit set.

< **false** >::= single bit clear.

Note: bit vectors are **false** padded to least significant bit of least significant i.e. last transmitted <word> boundary.

3.1.5 Byte

A reference to **bytes** implies any of the following types.

< **byte** >< bit field >
8

< **byte_vector** >< length ><low bound>{< bit field >}
8

3.1.6 Word

A reference to **words** implies any of the following types.

< **word** >< bit field >
16

< **word_vector** >< length ><low bound>{< bit field >}
16

3.1.7 Null

< **null** >< >
8

3.2 Compound Type

Compound tokens differ from vector tokens in that they may be used to carry several datum of differing types. Each datum therefore carries type and where necessary length fields.

< **compound** >< length >< data fields >

length length in < word> units of the data fields.

3.3 List Markers

List markers are used to delimit streams. Nested lists are supported. A reference to **list_markers** means any of the following types.

3.3.1 Start of List

< **start_of_list** >< >
8

3.3.2 End of List

< **end_of_list** >< >
8

3.3.2 Inter List

Inter list tokens are used to mark the outermost end-of-list on lists or nested lists.

< **inter_list** >< >
8

3.4 Tag Types

The following types are associated with token context tags.

3.4.1 Type

< **type** >< type >|

< **type** >< type >< length >|

< **type** >< type>< length ><low bound>

3.4.2 Name

< **name** >< >< name >
8

< **name_vector** >< length >< low bound >{< name >}

3.4.3 Colour

< **colour** >< >< colour >
8

< **colour_vector** >< length >< low bound >{< colour >}

3.4.4 Environment

The environment token carries the calling environment context (See Colour Functions).

< **environment** >< >< name ><colour>
8

3.5 Exception Types

3.5.1 Don't Know (?)

Sent to successor nodes on the occurrence of an exception. The <environment> in which the exception occurred is preserved by successor nodes.

A fatal exception such as the use of a token not of type **bit** where required for path control (See Path Control Functions) causes the **?** token to be sent to the system exception node (See System Functions).

< **?** >< length >< reason ><environment>>
16

< **?** >< length >< reason >< environment>< token >|

< **?** >< length >< reason >< environment >< token >< data fields >

length	in <word> units.	
<reason>::=	**noobject**	name object does not exist
	emptyobject	empty object
	notname	**name** type expected
	notcolour	**colour** type expected
	notenv	**environment** type expected
	notbits	**bits** type expected
	argtype	unexpected type
	notvector	vector type expected
	indexrange	index out of range
	toolarge	magnitude of datum too large
	toosmall	magnitude of datum too small
	toopos	positive overflow
	tooneg	negative underflow
	divzero	divide by zero
	nocoerce	type coercion not possible
	msover	matching store overflow
	osover	object store overflow
	nsover	node store overflow
	The reason mnemonic is given in boldface.	
name	the name of the destination object at which the exception occurred.	

token	where appropriate the exception causing token.
data fields	where appropriate the matching token's data fields.

3.5.2 Trace Type

Sent to system trace node when the trace bit of a < node > or object in the object store is accessed.

<**trace**>< length><environment>|

<**trace**>< length><environment>< function fields | allocation_desc >< arguments >< result >

length	in <word> units.
environment	<name> and <colour> of the object being accessed.
function fields	if the destination object is a <node> its <function fields> or
allocation_desc	if the destination is an object in the object store its <allocation_desc>.
<arguments>::=	argument <data fields>.
<result>::=	result <data fields>.

3.6 Node Type

< **node** >< length >< name >< node >

length	in <word> units.
node	node function description (See Nodes).

3.7 Internal System Types

There are a number of system token types used for internal system communication which are beyond the scope of this document.

3.8 Structure Descriptors

The structure descriptors are in fact **compound** tokens although their defined structures allows them to be viewed as 'psuedo' types.

3.8.1 Read Descriptors for Transmitted Objects

Vector or **compound** transmitted object fields may be accessed using the following descriptor:

<**null**>|
< **int16** ><index >|
< **int16_vector** >< indices >

The descriptor forms are used as follows:

<null> Return the entire object

<int16>|<int16_vector> The index or index vector is used to access recursively target data, datum or datum field. If a datum is reached and the indices are not exhausted then the remaining indices are used to access datum fields.

3.8.2 Write Descriptors for Transmitted Objects

The write descriptor is similar to the read descriptor the <data fields> to be written following the indexing fields.

< **compound** >< length >
[<**null**>|
< **int16** ><index0 >|
< **int16_vector** >< indices >]
< data fields >

<null> Overwrite the entire object

<int16>|<int16_vector> The index or index vector is used to access recursively target data, datum or datum field. If the target is a vector element and the descriptors <data fields> contain a vector the last indice specifies the starting element and the <length> of the descriptors <data fields> determines the number of elements to be overwritten.

If the object is a stored list then the element indexed by index0 is overwritten.

3.8.3 Read and Write Descriptors for Stored Objects

Stored **Vector** or **compound** object fields may be accessed using the following descriptor:

< **int16** ><index >|
< **int16_vector** >< indices >

The descriptor forms are used as follows:

<int16>|<int16_vector> The index or index vector is used to access recursively target data, datum or datum field. If a datum is reached and the indices are not exhausted then the remaining indices are used to access datum fields.

3.8.4 Copy Descriptor for Stored Objects

Blocks of stored **objects** may be copied using the following descriptor:

< **int16_vector** >< old base index ><length>< new base index >

3.8.5 Fill Descriptor for Stored Objects

Blocks of stored **objects** may be initialised using the following descriptor:

< **int16_vector** >< base index >< length >

The data for block fills is provided as the other argument of the structure block fill function.

3.9 Type Mnemonics

R32	**real32**
R64	**real64**
R32V	**real32_vector**
R64V	**real64_vector**
I8	**int8**
I16	**int16**
I32	**int32**
I8V	**int8_vector**
I16V	**int16_vector**
I32V	**int32_vector**
C	**char**
CV	**char_vector**
B	**bit**
BV	**bit_vector**
BY	**byte**
BYV	**byte_vector**
WD	**word**
WDV	**word_vector**
N	**null**
CM	**compound**
S	**start_of_list**
E	**end_of_list**
EE	**inter_list**
TY	**type**
NA	**name**
NAV	**name_vector**
CL	**colour**
CLV	**colour_vector**
EN	**environment**
Q	**?**

4. NODES

<node>::=

< function fields >|
32

< function fields >{<name>}|

< function fields >{<data fields >}|

< function fields >{< data fields }{<name>}

4.1 Function Fields

<function fields>::=

< literal present >< trace >< match class >< function >< no. of dests.>
1 1 6 8 8

<literal present>::= when **true** indicates that the <node > has a literal argument in <data fields>*.

<trace eval>::= when **true** causes a **trace** token to be sent to the system trace node (See System Nodes).

<function>::= the name of the node function (See Node Functions).

<match class>::= the matching class to be used with node operands (See Match Classes).

<no. of dests.>::= number of <name> fields or successor objects.

*Literals are confined to what may be carried in a single token including vectors and compound tokens. list literals are not permitted.

5. NODE FUNCTIONS

Node functions are either monadic or diadic. In principal any <match class> that produces one argument (<arg.0> or <arg.1>) may be used with a monadic function and any <match class> that produces two arguments (<arg.0> and <arg.1>) may be used with a diadic function; the combination may not however always be sensible.

The following pseudo types are defined to reduce the complexity of the descriptive material:

< real > ::= real32 | real64 | real32_vector | real64_vector

< int > ::= int8 | int16 | int32 | int8_vector | int16_vector | int32_vector

< chars > ::= char | char_vector

< arith > ::= real | int | colour | colour_vector

< bits > ::= bit | bit_vector

< logical > ::= bit | bit_vector | byte | word

< list_markers > ::= start_of_list | end_of_list | inter_list

5.1 Computational Functions

The set of computational functions will be extended to include additional functions such as hyperbolics. The intent is to increase the general function granularity.

Vector operands of diadic functions must be of equal length.

Unless specified otherwise the type of arg.1 must be compatable with arg.0. Type coercion is performed where sensible. e.g. for multiply where one operand is **int** and the other **real**, the **int** operand is coerced to **real** and the result of the function is real.

5.1.1 Arithmetic

Diadic

ADD	arg.0:**arith** + arg.1 -> out:**arith**
SUB	arg.0:**arith** - arg.1 -> out:**arith**
MUL	arg.0:**arith** * arg.1 -> out:**arith**
INR	inner product::= *inner product of* arg.0:**arith** *and* arg.1 -> out: **arith**
DVD	arg.0:**arith** / arg.1 -> out:**arith**
DIV	arg.0:**int** div arg.1:**int** -> out:**int**
MOD	arg.0:**int** mod arg.1:**int** -> out:**int**
PWR	arg.0:**arith** ^ arg.1:**arith** -> out:**arith**

Monadic

All of the diadic functions above with one literal argument.

NEG	- arg.0:**arith** -> out:**arith**
ABS	absolute(arg.0:**arith**) -> out:**arith**
EXP	e ^ arg.1:**arith** -> out:**real**
LNE	$\log_e$(arg.0:**arith**) -> out:**arith**
LN2	$\log_2$(arg.0:**arith**) -> out:**arith**
LOG	$\log_{10}$(arg.0:**arith**) -> out:**arith**
SQT	square_root(arg.0:**arith**) -> out:**arith**
SQR	square(arg.0:**arith**) -> out:**arith**
SIN	sine(arg.0:**arith**) -> out:**arith**

COS cosine(arg.0:**arith**) -> out:**arith**

TAN tangent(arg.0:**arith**) -> out:**arith**

Arguments for the above trigonometric functions are in radians.

ATN arc_tangent(arg.0:**arith**) -> out:**arith**

ASN arc_sine(arg.0:**arith**) -> out:**arith**

ACS arc_cosine(arg.0:**arith**) -> out:**arith**

5.1.2 Logical and Set

Diadic

AND arg.0:**logical** and arg.1 -> out:**logical**

OR arg.0:**logical** or arg.1 -> out:**logical**

IMP arg.0:**logical** implies arg.1 -> out:**logical**

EQV arg.0:**logical** equivalent arg.1 -> out:**logical**

NQV ~(arg.0:**logical** equivalent arg.1) -> out:**logical**

TSB test_bit(arg.0:**logical**, arg.1:**int**) -> out:**bit**

STB set_bit(arg.0:**logical**, arg.1:**int**) -> out:**logical**

CLB clear_bit(arg.0:**logical**, arg.1:**int**) -> out:**logical**

SHD shift_down(arg.0:**logical**, arg.1:**int**) -> out:**logical**

SHU shift_up(arg.0:**logical**, arg.1:**int**) -> out:**logical**

SHU is **false** filled. The <length> of the result is adjusted appropriately for both the SHD and SHU functions.

Monadic

All of the diadic functions above with one literal argument.

MSB most_significant_bit(arg.0:**logical**) -> out:**int16**

NOT ~arg.0:**logical** -> out:**logical**

5.1.3 Relational

Relational operations on **char_vector** are over the entire vector i.e. they are string comparisons. Relational operations over **arith** vectors are element by element yielding a **bit_vector** result.

Diadic

EQ arg.0 = arg.1 -> out:**bits**

NE arg.0 <> arg.1 -> out:**bits**

GE arg.0: **arith,chars,bits** >= arg.1 -> out:**bits**

GT	arg.0: **arith,chars,bits** > arg.1 -> out:**bits**
LE	arg.0: **arith,chars,bits** <= arg.1 -> out:**bits**
LT	arg.0: **arith,chars,bits** < arg.1 -> out:**bits**
RA	range ::= (arg.0: **arith,chars** >= arg.1[0]) and (arg.0 <= arg.1[1]) -> out:**bits**

Monadic

All of the diadic functions above with one literal argument.

5.1.4 Sorting

Diadic

GTS

```
greater than swap ::=
if arg.0:arith,chars,env > arg.1 then
        arg.0 -> out.0;
        arg.1 -> out.1
else
        arg.1 -> out.0;
        arg.0 -> out.1;
```

LTS

```
less than swap ::=
if arg.0: arith,chars,env  < arg.1 then
        arg.0 -> out.0;
        arg.1 -> out.1
else
        arg.1 -> out.0;
        arg.0 -> out.1;
```

MAX

```
if arg.0:arith,chars,env  >= arg.1 then
        arg.0 -> out
else
        arg.1 -> out
```

MIN

```
if arg.0: arith,chars,env  <= arg.1 then
        arg.0 -> out
else
        arg.1 -> out
```

WDW

```
window ::=
if arg.0: arith, chars < arg.1[0] then
        arg.1[0] -> out
else
        if arg.0 > arg.1[1] then
            arg.1[1] -> out
        else
            arg.0 -> out
```

OFF

```
offset ::=
arg0[0]-arg0[1]+arg0[2]: int->out.0
(arg0[0]>= arg0[1]) and (arg0[0]<=(arg0[0]+arg0[1]-1)->out.1:bit
```

Monadic

All of the diadic functions above with a literal argument.

5.1.5 Sequence

Monadic

SUC	arg.0:**bits,chars,int,colour** + 1 -> out:**bits,chars,int,colour**
PRE	arg.0:**bits,chars,int,colour** - 1 -> out:**bits,chars,int,colour**

The result of these functions is bounded by the type value range.

5.2 Type Functions

5.2.1 Coercion

Diadic

COE	*coerce* <type> *of* arg.0 *to* arg.1:**type** -> out

COE is an important constructor function which makes the best attempt at conversion *by example* to the target type. Note that because the top 8 bits of the <length> field in the **type** token is used to specify the target <type> the target length is limited to 2^{15}-1.

Monadic

The COE *function with a literal argument.*

ORD	ordinal(arg.0:**bit,int,chars**) -> out:**int**
CHR	character(arg.0:**int,chars**) -> out:**char**
RND	round(arg.0:**arith**) -> out:**int**
TRN	truncate(arg.0:**arith**) -> out:**int**
FLT	float(arg.0:**arith**) -> out:**real32** | **real32_vector**
DBL	double_precision(arg.0:**arith**) -> out: **real64** | **real64_vector**

Explicit conversion functions may have some evaluation time advantage over the more general COE function.

5.2.2 Type

Diadic

CPT	compare type::= <type> of arg.0 = <type> of arg.1 -> out:**bit**
ERR	is error ::= <type> of arg.0 is ? and <reason> = arg.1:int -> out:**bit**

Monadic

The CPT *function with a literal argument.*

TOF	<type> of arg.0 -> out: **type**

5.2.3 Compound Token Constructors

Diadic

FMC form compound token::=
flatten and append arg.1 *to* arg.0 -> out: **compound**

CCM concatenate compound token::=
concatenate arg.1 *to* arg.0 -> out: **compound**

CCN cons compound token::=
cons arg.1 *to* arg.0 -> out: **compound**

CAP append compound token::=
append arg.1 *to* arg.0 -> out: **compound**

Monadic

CGT get compound token::=
first datum of arg.0 -> out.0
rest -> out.1

An important use of the compound token constructors function is the generation of *descriptors* for accessing the fields of vector and compound types (See Structure Descriptors).

5.3 Object Manipulation Functions

5.3.1 Transmitted Objects

5.3.1.1 Transmitted Token Lists

Many of the operations on transmitted lists are provided by ta node's <match class> (See Match Classes). Those that are associated withe the node's <func> are detailed below. Indexing functions may be used to access < data fields > of defined complex types such as ?; complex types may be regarded as compact representations of commonly used **compound** types.

Restrictions to prevent the synthesis of < names > may apply. In particular the manipulation of < process > is not permitted.

Diadic

LFL list fill ::=
create list of length arg.1:**int16** *with element values* arg.0 -> out

Monadic

LCL collapse transmitted list::=
arg.0 *absorbing* < **list markers** > -> out

5.3.1.2 Transmitted Vector and Compound Tokens

Access to as yet undefined fields of vectors return:

< **reals** >	IEEE "not a number"
< **ints** >	-("maxint"+1)
< **chars** >	ASCII NUL
< **bits** >	false
< **words** >	0
< **bytes** >	0

Resizing of compound tokens other than by the use of compound token constructors is not permitted. In particular types or lengths of fields of compound tokens cannot be changed using TWR (Transmitted Write). Some implementations may restrict the maximum length of transmitted objects.

Diadic

TRD — transmitted field read :: =
arg.0 *accessed using* arg.1:***transmitted_read_desc*** -> out

TWR — transmitted field write ::=
update arg.0 *accessed using* arg.1: ***transmitted_write_desc*** -> out

TFL — transmitted vector fill ::=
fill vector with low bound arg.1[1] *and length* arg.1[0]:**int16**
using arg.0->out

TVC — transmitted vector concatenate ::=
vector arg.0 *concatenated with* arg.1 ->out

TSC — transmitted vector scatter ::=
elements of vector arg.0 -> out.0;
indices -> out.1;

TDS — transmitted vector distribute ::=
elements of vector arg.0 -> out[index];

TLB — set transmitted vector low bound ::=
set vector low bound of arg.0 with arg.1:**int16** -> out

Monadic

The above diadic functions with one literal argument .

TLO — transmitted vector low bound ::=
low bound of vector arg.0 -> out:**int16**

TLE — transmitted object length ::=
length of arg.0 -> out:**int16**

TVL — transmitted vector to list::=
convert vector arg.0 *to list* -> out

5.3.2 Stored Objects (I)

For stored objects:

1) access to stored objects is not qualified by <colour> but arguments to accessing functions may carry a <colour> which is preserved on result tokens,

2) access is qualified by <process>,

3) objects in the <process> space are not persistent i.e. they are lost when the <process> terminates.

It is intended that objects in process 0 space will be persistent.

With deferred access to as yet uninstantiated elements of stored objects, accesses from any given Object Store node may not be honoured in request order. Strict ordering may be obtained where necessary by encapsulating the accessing node in a PRT (Protect) construct such that further requests operands are denied until the last transaction is complete. The mechanisms described in this section have aspects in common with those described in [5] and [6].

5.3.2.1 Stored Vector and Compound Tokens

For stored compound and vectors non-defered access to as yet undefined datum are not treated as errors. Vectors and compound objects with non-deferred access have the following initial values:

< reals >	IEEE "not a number"
< ints >	-("maxint" +1)
< colour >	0
< chars >	ASCII null
< bits >	false
< words >	0
< bytes >	0

The **compound** *descriptor* may be constructed initially using the compound token constructors.

If the *descriptor's* indexing fields are **null** then the entire object is written or returned.

Diadic ,

ORE object field read :: =
[arg.0: **name**] *accessed using* arg.1:*read_desc* -> out

OWR object field write ::=
update [arg.0: **name**] *accessed using* arg.1: *write_desc*
acknowledge: **bit** -> out

ORW object field read before write ::=
[arg.0: **name**] *accessed using* arg.1: *write_desc* -> out
update [arg.0: **name**] *accessed using* arg.1: *write_desc*

OFL stored vector fill ::=
fill [arg.0: **name**] *accessed using* arg.1: *write_desc*
acknowledge:**bit** -> out

OSC scatter stored vector ::=
elements of [arg.0: **name**] *accessed using* arg.1: *read_desc* ->out.0
indices of [arg.0: **name**] *accessed using* arg.1: *read_desc* ->out.1

OLB set stored vector low bound ::=
set low bound of [arg.0:**name**] *with* arg.1:**int16**
arg.1 -> out

Monadic

OLO stored vector low bound::=
low bound of vector [arg.0:**name**] -> out:**int16**

OLE stored object length::=
length of vector [arg.0:**name**] -> out:**int16**

Field accessing functions are defined for single tokens with multiple fields and elements of stored lists.

The OAC function adds the value field of the write_desc to the stored counter value. The accumualtor may be a simple object or contained within a vector or compound token; partial indexing is not permitted i.e. the indexed field must be a datum.

5.3.2.2 Stored Token Lists

For stored lists the <name> of a list is provided as the argument and returned as a result where appropriate. Stored token lists allow shared access and manipulation; copying and distributing the <name> of the list should be done with care.

Diadic

OIF insert front ::=
insert arg.1 *as new first element of* [arg.0:**name**]
acknowledge -> out

OIL insert last::=
insert arg.0 *as new last element* [arg.0: **name**]
acknowledge -> out

OIF and OIL do not guarantee the integrity of stored lists. Some caution should be exercised when using these functions to manipulate a stored list.

Monadic

All of the above diadic functions functions with one literal argument.

ORF return first ::=
return first element of [arg.0: **name**] -> out

The following functions act upon a stored list.

OHD object head::=
head atom or first list of nested lists [arg.0:**name**] -> out

ORS object rest::=
absorb first atom or list of nested lists [arg.0:**name**], [*rest* -> out

OGT object get::=
first atom or list of nested lists [arg.0:**name**] -> out.0
[*rest* -> out.1

OEM object empty ::=
empty [arg.0:**name**] -> out: **bit**

OCL object collapse::=
[arg.0:**name**] *absorbing* **list_markers** -> out

ORL object return list ::=
[arg.0: **name**] -> out

ORL is constrained to lists contained entirely within an Object Store <element> partition. <name> fields in elements within the list are not followed by ORL.

5.3.2.3 Reference Count

All objects in the object store have an associated <reference count> which is initially set to 1. Objects are de-allocated when the <reference count> becomes < 1.

ORC object reference count ::=
[arg.1: **name**].<ref_count> := [arg.1: **name**].<ref_count> + arg.0: **int16**
if [arg.1: **name**].<ref_count> = 0 **then**
de-allocate object;
[arg.1:**name**].<ref_count> -> out

5.3.3 Stored Objects (II)

These functions are based on the original non deferred structure store mechanisms which have been extended to provide I-Structure semantics at an object level.

The functions access a single vector of objects mapped across the processing elements (modulo (maxpe+1)).

Monadic

SSR structure store read ::=
SS[arg0:**int16**] -> out

SRD structure store deferred read ::=
if defined **then**
SS[arg0:**int16**] -> out
else
defer read until defined

SCL structure store clear ::=
set SS[arg0[0]..arg[1]:**int16**] *to undefined*

SBC structure store block copy ::=
copy arg0[2] *objects from* SS[arg0[0] *to* SS[arg1[1]

Diadic

SSW structure store write ::=
SS[arg.1:**int16**] := arg0;
honour pending reads
acknowledge -> out:**bit**

SWR structure store single assignment write ::=
if not defined **then**
SS[arg.1:**int16**] := arg0;
honour pending reads
acknowledge -> out:**bit**

SRW structure store write and read once::=
if not defined **then**
honour FIRST pending read
else
SS[arg1] := arg0

SRR structure store read and reset ::=
SS[arg0:**int16**]->out
set to undefined

SAC structure store single assignment write ::=
if not defined **then**
SS[arg.1:**int16**] := arg0;
honour pending reads with arg0
else
SS[arg1:**int16**] := SS[arg1]+arg0;
SS[arg1] -> out

SBF structure store block fill ::=
if not defined **then**
fill arg1[1] *SS objects starting at* arg1[0] *with* arg0;
honour pending reads

SSM structure store read before write ::=
SS[arg.1:**int32**] -> out;
honour pending reads
SS[arg1:**int32**] := arg.0

5.4 Path Control Functions

5.4.1 Replication

Monadic

DUP arg.0 -> out.0, out.1

The DUP (duplicate) function has two destination names and may have some evaluation time advantage over the more general n-output REP (replicate) function.

REP arg.0 -> out

ID arg -> out

5.4.2 Synchronisation

Diadic

```
PRS     presence::=
            true -> out: bit
```

PRS is used to synchronise path control by signalling when both inputs are present. A tree of PRS nodes may be used where synchronisation over more than inputs is required. In principle a tree of any other diadic operators could be used e.g. AND. However PRS may be faster on some implementations as no argument type checking is required.

Monadic

```
CHN     channel ::=
        on arg.0 then
            arg.0 -> out.0
        on arg.1 then
            arg.1 -> out.1
```

The CHN (channel) function has two destination names and is used where the associated match class returns arg.0 and/or arg.1 e.g. GET.

5.4.3 Gating

Diadic

```
PIT     pass if true::=
        if  arg.1:bit  then
            arg.0  ->  out

PIF     pass if false::=
        if  ~arg.1:bit  then
            arg.0  ->  out

PIP     pass if present::=
        on    arg.1   then
            arg.0  ->  out

SWI     switch ::=
        if arg.1: bit then
            arg.0 -> out.1
        else
            arg.0 -> out.0

DST     distribute ::=
        arg.0  ->   out.(ord(arg.1:bit,int,colour,char,typ))
```

RTR and BTR may be used in a doubly recursive graph to generate all values within some range of numbers e.g. indices for all elements of a structure. RTR generates values at each branch point while descending the tree and BTR generates them at the leaves.

```
RTR       recursion tree ::=
          if arg.0:int,colour<= (arg.1:int,colour div 2) then
               true->out.0
               arg.0*2+1->out.1
               arg.1->out.2
               if arg.0<(arg.1 div 2) then
                    true->out.3
                    arg.0*2+2->out.4
                    arg.1->out.5
               else
                    false->out.3
          else
               false->out.0

BTR       bottom tree::=
          if arg.0:int,colour=arg.1:int,colour then
               true->arg.0
               arg.0->out.1
          else
               false->out.0
               arg.0->out.2
               (arg.0+arg.1) div 2->out.3
               (arg.0+arg.1) div 2+1->out.4
               arg.1->out.5
```

The following two functions would usually be used in conjunction with the Protect (PRT) match class to implement lazy or eager gating of arguments or results respectively of selected graph regions.

```
LMO       lazy merge output ::=
          true -> out.(ord(arg.0:bit,int,colour,char))

EMO       eager merge output ::=
          true -> out.(ord(arg.0:bit,int,colour,char))
          false -> all other outputs
```

5.4.4 Name

Monadic

```
YLN       yield name::=
          name of arg.0 -> out.0:name;
          arg.0 -> out.1
```

Diadic

```
STN       set name::=
          arg.0 -> [arg.1:name ]
```

5.4.5 Sequence

Some caution should be exercised with these functions as they are capable of generating long bursts of tokens which may overload processing-element matching stores.

Diaidic

PRO proliferate::=
arg.1:**int** *copies of* arg.0 ->out

Monadic

SEQ sequence::=
while seq value initially arg.0[1]:**int16_vector** *and incremented by* arg.0[2] < arg.0[0]
false -> out.0;
seq value -> out.1
true -> out.0

The sequence descriptor may be have 1 to 3 elements. For one element descriptors the discriptor type is **int16**. The defaults are 1 for step size and for 0 starting value.

SEQ token sequences are:

1) n **int** tokens commencing at a sequence bound and incremented or decremented as appropriate by the step value until the other bound is reached and,

2) n -1 **false** tokens followed by one **true** token.

5.5 Colour Functions

5.5.1 Direct Colour Manipulation

Monadic

CRC create colour::=
on inp.0 **then**
unique colour -> out: **colour**

CCS create colour sequence::=
create colour seqence of length arg1 *in cycles of* arg0 *colours* -> out:**colour**

YLC yield colour::=
the colour of arg.0 -> out:**colour**

RCL remove colour::=
arg.0 *with no colour* -> out.0
old colour ->out.1:**colour**

EVC exchange value and colour::=
exchange the value and colour fields of arg.1:**colour** | **int** -> out:colour

Diadic

STC set colour::=
arg.0 *with colour* arg.1:**colour** | **int** -> out

5.5.2 Context

Diadic

SRL set return link::=
form environment from arg.0:**name** *and colour of* arg.1
-> out.0:**env** *with colour set to* arg.1: **colour**

arg.0 is usually a literal <name> with no <colour>. Set return link is used in combination with the E (exit) function to return results to the invoking context.

Monadic

E exit::=
arg.0 -> [arg.1:**env**]

These operators may be used in conjunction with YLC (yield colour) STC (set colour) function to form re-entrant sub-graph and iteration constructs.

5.6 Priming Functions

This function is always used in conjunction with the Prime (PRM) match class.

Monadic

```
PRI        prime ::=
           on first  arg.0
                literal -> out;
                arg.0 -> out
           else
                arg.1 -> out
```

A system token is used to signal the initial token to the PRI (prime) function.

6. MATCHING CLASSES

In the following all tokens in the match consideration and directed at the same <name> must have the same <colour>.

6.1 Bypass

All incoming tokens are forwarded with <input point> preserved. Literals are permitted with the input point of the literal being determined by the incoming tokens input point.

BYP   bypass ::=
       inp -> arg

6.2 Normal

Tokens of a particular <colour> and <input point> are queued until a a token with a matching <colour> and complementary <input point> arrives. When this occurs the token at the head of the queue is removed and forwarded along with the arriving token.

NRM   normal ::=
       on (inp.0,inp.1) **then**
           inp.0 -> arg.0; inp.1 -> arg.1

If a token list is directed at one input and an atom at the matching input then the atom is matched with all list elements including the outermost **inter-list** and is then removed. Atoms and lists should not be intermixed on the same arc.

6.3 Empty

EMP   empty list ::=
       list is empty -> arg.0:**bit**

6.4 Start

STA   start of list ::=
       is start of list -> out:**bit**

6.5 Finish

FIN   finish of list ::=
       is finish of list -> out:**bit**

6.6 Cons

The atom arriving on inp.0 is forwarded as the new head of list followed by the list arriving on inp.1. *No literal operands permitted.*

CNS   cons ::=
       atom on inp.0 inserted as new head of list on inp.1 -> out

6.7 List

Forward the new list formed by appending the list on inp.1 to the end of the list on inp.0. List preserves the enclosing list markers of the lists on inp.0 and inp.1 i.e. it creates nested lists. *No literal operands permitted.*

LST list ::=
[*list or atom on inp.0 then list or atom on inp1*] -> out

6.8 Concatenate

Forward the new list formed by concatenating the list on inp.0 to the list on inp.1. The outermost list markers of the lists on inp.0 and inp.1 are removed in this process. *No literal operands permitted.*

CON concatenate ::=
[*unbracket* list or atom on inp.0 then *unbracket* list or atom on inp.1] -> out

6.9 Bracket

Forward the incoming list or atom on inp.0 as arg.0 with additional [ ].

BRA bracket ::=
[list or atom on inp.0] -> arg.0

6.10 Unbracket

Forward the incoming list on inp.0 as arg.0 absorbing the outermost [ ].

UNB unbracket ::=
remove first level of [ ] *from* list on inp.0 -> arg.0

6.11 Head

Forward the head of the list on inp.0 to arg.0 and absorb the rest of the list.

HED head ::=
remove first level of [ ] *from list on* inp.0
first list or atom *of nested lists on* inp.0 -> arg.0;
absorb rest of list

6.12 Rest

Absorb the head of the list on inp.0 and forward the rest of the list as arg.0.

RES rest ::=
absorb first atom or list of nested lists , [*rest* -> arg.0

6.13 Get

The head list of the list on inp.0 is forwarded as arg.0 and the tail as arg.1.

GET get ::=
first atom or list of nested lists -> arg.0
[*rest* -> arg.1

6.14 Store

Store the token arriving at inp.0 overwriting any previous token. On receiving a token on inp.1 forward a copy of the stored token or an **null** token if nothing has been stored. Matching state is reset unconditionally by an **inter_list** token on inp.1; no acknowledgement is issued on reset. *No literal operands permitted.*

```
STO      store ::=
         on inp.1 then
                copy of latest token inp.0 -> arg.0
```

6.15 Store and Reset

Store the token arriving at inp.0 overwriting any previous token. On receiving a token on inp.1 forward a copy of the stored token or an empty token if nothing has been stored.

```
STR      store and reset ::=
         on inp.1 then
                copy of latest token inp.0 -> arg.0;
                reset to empty
```

If no token has been written to the storage or reset nodes then an empty token is returned; there is no deferred access. *No literal operands permitted.*

6.16 Store Deferred

Store the token arriving at inp.0 overwriting any previous token. On receiving a token on inp.1 forward a copy of the stored token otherwise defer the read access until a token has been stored. All outstanding reads are honoured after a write. Matching state is unconditionally reset by an **inter_list** token on inp.1; no acknowledgement is issued on reset. *No literal operands permitted.*

```
STD      store read deferred ::=
         on inp.1 then
                if not empty then copy of latest token inp.0 -> arg.0;
```

6.17 Store Update

Emit the previously written token and update with incoming token. If empty emit <?> token and update. Matching state is unconditionally reset by an **inter_list** token on inp.1; no acknowledgement is issued on reset. *No literal operands permitted.*

```
STU      store update ::=
         on inp.1 then
                if not empty then
                    copy of previous token on inp.0 -> arg.0
                else
                    null->arg.0
         on inp.0 then
                if not empty then
                    copy of last token on inp.0 ->arg.0;
                else
                    null->arg.0
```

6.18 First

The first list or atom of any given <colour> is passed and all subsequent tokens are absorbed. A token on inp.1 resets the matching function.

FIR first ::=
if *first token on* inp.0 **then**
inp.0 -> arg.0

6.19 Prime

The first list or atom arriving on inp.0 causes the associated PRI function to emit a priming literal followed by the triggering list or atom. All other tokens are forwarded as for <monadic>. A token arriving on inp.1 resets the matching function.

PRM prime ::=
if *first token on* inp.0 **then**
inp.0 -> arg.0
else
inp.0 -> arg.1

6.20 Protect

The first list or atom arriving on inp.0 is forwarded. The <input point> is then protected until reset by a token arriving on inp.1. *No literal operands permitted.*

PRT protect ::=
if *first token on* inp.0 **then**
inp.0 -> arg.0
else
on inp.1 **then**
inp.0 -> arg.0

6.21 Arbitrate

First list or atom arriving is forwarded to arg.0 next on complementary input to be forwarded to arg.1 and then matching function is reset.

ARB arbitrate ::=
first atom or list *arriving* -> arg.0;
next atom or list *arriving* -> arg.1 *then reset*

7. SYSTEM NODES

A number of system nodes are *defined* in every processing-element. In addition input and output nodes are also *defined* but are associated with specific devices which are in turn associated with specific processing-elements; this association varies from installation to installation. The effective <match class> for these nodes is BYP (Bypass).

7.1 System

All system node-names are reserved with their node-descriptions existing in all elements; *e* is the <element >.

Although a particular system-node may be referred to at a number of places in the graph, it represents a single-resource. Multiple referencing therefore, implies non-deterministic merging on the node's input-points. Unless this is intentional, the node should be referenced once within an encapsulating resource manager.

e.-1 inp.0:**node** -> Node-Store

e.-2 inp.0:**?** -> [*last* inp.1:**name**]

e.-3 inp.0:**trace** -> [*last* inp.1:**name**]

e.-4 *every* inp.0:**int** *ticks**, **true** -> [*last* inp.1:**name**]

e.-8 *kill process* inp.0:**int8**

e.-16 *Occupancy** of Input Queue* :**int32** -> [inp.0:**name**]
e.-17 *Occupancy** of Matching Store* :**int32** -> [inp.0:**name**]
e.-18 *Occupancy** of Object Store* :**int32** -> [inp.0:**name**]
e.-19 *Size of Structure Store* :**int32** -> [inp.0:**name**]

*for the timer *tick* interval refer to current implementation notes
**$2^{16}-1$ implies 100% occupancy - occupancy of Input Queue is suggested as the best measure of system workload.

7.2 Input and Output

As input and output nodes have physical devices associated with their <name>, there will be restrictions on the type of tokens produced or accepted by these nodes. Type and length information is preserved in all input/output operations. *i* is the input <element object> and *o* is the output <element object>. Input/output accesses are not qualified by <colour> although <colour> is preserved in these transactions; because the effective match class is BYP (Bypass) the <colour> of the link <name> on inp.1 need not match any token arriving on inp.0.

Monadic

e.-(32+*i*) **on** inp.0 **then**
device.token: **dev.dep** -> [*last* inp.1[0]:**name**]
on inp.1:**name then**
acknowledge:**bit** -> [inp.1[1]:**name**]

e.-(48+*o*) inp.0: **dev.dep** -> device;
inp.0 -> [*last* inp.1[0]:**name**]
on inp.1:**name then**
acknowledge:**bit** -> [inp.1[1]:**name**]

In the case of output nodes provision of an acknowledgment destination name on inp.1 is optional but desirable.

REFERENCES

[1] G.K. Egan, "Data-flow: Its Application to Decentralised Control", Ph.D. Thesis, Department of Computer Science, University of Manchester, England, 1979.

[2] J.B. Dennis and D.P. Misunas, "A Preliminary Architecture for a Basic Data-flow Processor", Proceedings of 2nd. Annual Symposium on Computer Architecture", New York, May 1975.

[3] Arvind and K.P. Gostelow, "The U-Interpreter", Computer, Vol. 15, No. 2, Feb 1983,pp 42-50.

[4] Egan G.K., Webb N.J. and Bohm A.P.W., 'Some Architectural Features of the CSIRAC II Dataflow Computer', Technical Report 31-007, Laboratory for Concurrent Computing Systems, School of Electrical Engineering, Swinburne Institute of Technology, 1990.

[5] Arvind and R.E. Thomas, "I-Structures: An Efficient Data Structure for Functional Languages", MIT/LCS/TM-178, MIT, 1981.

[6] C.C. Kirkham and J. Sargeant, "Stored Data Structures on the Manchester Dataflow Machine", Internal Report, Department of Computer Science, University of Manchester, England.

[7] Young A.J. 'Implementation of a Multistage Network for Interconnecting a Dataflow Multiprocessor', TR 112 077 R, Department of Communication and Electrical Engineering, Royal Melbourne Institute of Technology, Nov. 1988.

APPENDIX - Well Known Names

The following names are associated with input and output nodes:

NA [0 -32 x]	standard input
NA [0 -33..46 x]	"input channels"
NA [0 -48 x]	standard output
NA [0 -49..62 x]	"output channels"

Input channels are linked to the "host" file system with <compound> token datum of the following form:

< **compound** >< link name >{<ack name>}< file name >{< mode >}

< link name >	name to which input nodes direct data and output nodes direct acknowledgments	
< ack name >	name to which link name changes are acknowledged	
< file name >	name of file on "host" system to be read or written	
< mode >	0	token access i.e. files contain tokens in normal (currently textual) form e.g R32 3.14159
	1	character access
	2	binary short integer
	3	binary real

e.g. CM { NA [0 111 0] CV 9 0 'text.data' I16 1 }

The input or output channel node receiving the above would then be linked to the "host" file 'text.data'. Data or acknowledgment tokens would be directed to NA [0 111 0] and transactions would be character.

If the control input subsequently receives another compound token then the previously opened file will be closed and another opened for access; if a <name> token is sent to the control then the output link name only is changed. "Host" files are currently opened for sequential access.

Standard input and output require only a <name> link token; Access mode is character. Currently tokens other than <char> directed to standard output are decoded to normal form.