SWARD

SRI-CSL-77-002n

SWARD ARCHITECTURE

PRINCIPLES OF OPERATION

G. J. Myers

August 31, 1979 (Current Version Date)

September 16, 1977 (Initial Version Date)

IBM Systems Research Institute

CONTENTS

1. OVERVIEW

The SWARD (software-reliability-directed) architecture has two primary objectives: 1) enhancing software reliability by detecting or preventing common semantic errors and certain logic errors, limiting the consequences of errors, encouraging the use of good software design and programming practices, and supporting testing and debugging packages, and 2) enhancing system performance by substantially reducing the number of bits that must be processed by the CPU to execute a given program. These goals are discussed in more detail elsewhere; hence they are not discussed in this document.

This document defines the computer architecture of the processor (i.e., the abstraction of the processor as seen by a machine-language programmer or a compiler writer). The first chapter (this one) contains a brief overview of many of the concepts employed in the architecture. Only chapters 2-7 form the official architecture definition. Also, occasionally notes will be seen in rectangular boxes. These notes are not part of the architectural specification; in most cases they are notes about implementation and are included for clarity.

The major deviations in this architecture from conventional architectures are in the concept of storage and addressing. Rather than representing storage as a single linear address space, storage is represented as a set of uniquely named storage objects. Also, rather than treating "secondary" storage (e.g., disk files) differently from "main" storage, the view of storage has been unified into a single representation.

Furthermore, all data in storage is self-identifying (tagged). The machine recognizes composite data types (e.g., arrays, structures) as well as primitive data types. Rather than employing a fixed-size word concept, data and addresses are variable in size. The machine provides a facility for defining supplemental instruction sets and data types.

A major concept in the architecture is that the machine should severely restrict the address space available to an individual module (e.g., FORTRAN subroutine or function subprogram, PL/I external procedure or function, COBOL subprogram). That is, a module's address space should be reduced to only those data to which the module needs access: its parameters, locally defined variables, and constants (i.e., only those data named in the source-language version of the module). The implication of this is that the machine must manage storage at a high level, much higher than the von Neumann view of a single linear sequential memory.

Related to the first concept is a second concept: traditional machine addresses should be discarded. There are four types of storage objects that must be uniquely addressable: a module, an activation record (the collection of data allocated for an activation or invocation of a module), a data-storage object (explicitly allocated area of storage), and a port (an interprogram communication device). When one of these objects is created (i.e., a module is defined to the machine, a module is activated (called), a program explicitly allocates some storage, or a port is created), the machine assigns it a unique name (called a logical address and stored in an area called a pointer). Hence the machine employs capability-based addressing. The machine prohibits programs from creating logical addresses on their own and from altering the value of a pointer. When one of these objects is freed, its unique logical address is never reused.

The instructions within a module can only address data defined within the module or data within any storage that is dynamically created by the module. Since a module cannot, on its own, create or alter a pointer, the only other storage that it can reference is storage whose pointer is passed to the module from another module. Not only does this concept facilitate the detection of addressing errors (e.g., the dangling-reference problem), but it also serves as a storage protection mechanism. In addition, it introduces a fine granularity of storage protection and sharing, even down to the level of a single variable or word, and eliminates the need for privileged states (e.g., "supervisor state").

One can cause pointers or capabilities to refer through other pointers, thus establishing transparent indirect addressing to any level between machine instructions and the data or objects upon which they operate.

The third necessary concept in the architecture is that all data must be self-identifying. This means that descriptive information will be stored with each item of data, describing such attributes as its size and type. This self-identification allows the machine to detect incompatible operands of an operation and allows it to enforce other rules (e.g., the rule above prohibiting the creation and manipulation of pointers). Two rules concerning self-identification, or tags, must be enforced: 1) the tag always describes the programmer's intended properties of the data (e.g., the attributes in the DECLARE statement), and 2) the value and representation of the data always agree with the tag.

In most other non-von Neumann machines, the concepts of tags and descriptors are treated distinctly. However the concepts have much in common. In the SWARD machine the two

concepts have been generalized into one concept called a tag.

To close the semantic gap between language data types and machine data representations, most data types known to the machine are variable in size. Not only does this prevent certain types of semantic errors that arise when variable-size language data types are mapped into fixed-size machine data types, but it leads to more efficient use of storage.

Employing both tagged storage and capability-based addressing gives the architecture an added level of security. Even if a program were to obtain, from another program, a pointer (capability) to an object that it should not have, the program would be able to reference the object only if it knew its precise attributes (i.e., the representations of the data within the object).

The architecture also contains important data-independence concepts, allowing one to write programs that are highly insensitive to the data being processed, yet without compromising the reliability and security goals of the architecture. These concepts, for instance, allow one to write a program to sort (order) the elements in any array (i.e., without being dependent on the attributes of the array elements), or to write a computational subroutine that is independent of the attributes of its arguments (e.g., binary, decimal, floating-point). These concepts, called D-typed, D-sized, and D-bounded cells, will lead to innovative programming-language extensions not achievable in conventional machines.

One consideration that is influenced by many of the previous points is the method used by instructions to address their operands. Many forms have been proposed, but a basic underlying consideration is whether the architecture should contain general-purpose registers or evaluation stacks (or both or neither). In studying various addressing mechanisms, no apparent relationships to software reliability were found. However the architecture contains neither registers nor evaluation stacks (but it does use stacks for subroutine linkages). An instruction addresses an operand by specifying the relative location of that operand within the address space of the module. Registers and evaluation stacks were not used because, contrary to popular belief, storage to storage addressing appears to be more efficient. The addressing method used in the architecture results in small address fields in instructions, thus negating the usually cited advantage of register and stack-oriented instruction sets.

Another necessary concept in the architecture is the ability to distinguish between defined and undefined data

values. In addition to its valid values, each data item can have an additional value called "undefined." Any attempt to use a data item's "undefined" value will be detected by the machine. All addressable data that is not initialized to some value is automatically initialized to "undefined." In addition, instructions are present to explicitly test a data item for an undefined value and to explicitly mark a data item as undefined (e.g., for a language in which the value of a loop iteration variable is supposed to be undefined when the loop terminates).

For collections of data in which individual items can be referenced, the individual items can be defined or undefined. Thus, in an array, it is possible that some elements will have defined values and that some will be undefined. In a character string, it is possible that some of the character positions are "undefined" values.

Given that the machine is aware of the concepts of modules and activation records, and given that the machine must check arguments and parameters for consistent number and attributes, a logical deduction is that the architecture should provide a call/return mechanism that is semantically close to, or equivalent to, the CALL/RETURN statements in programming languages. That is, the call mechanism allocates an activation record for the called module and adds it to the stack of current activation records, initializes variables in the activation record, initializes parameters, suspends execution of the current module, and begins execution of the called module. Since all data in the system is tagged, the call mechanism needs a "die" for variables in the activation record, describing how each data item should be tagged when the activation record is created.

Since the architecture is supposed to detect such errors as exceeding an array dimension bound and inconsistent definitions of records (e.g., PL/I structures) among modules, the architecture must be aware of these data types. Hence the architecture contains the "less primitive" data types of arrays, structures (ordered sets of heterogeneous data items), and strings. Supporting such data types is more involved than it first appears. For instance a language such as PL/I provides for arrays of structures, structures of arrays, arrays of strings, structures of structures, structures of arrays of structures, and so on.

Note that the self-identification property mentioned earlier applies to these data types. For instance every array, structure, and string is tagged. The machine instructions are generic; for instance there is only one ADD instruction, and its two operands can be any meaningful data types that pass certain consistency tests. For instance an operand to an ADD can be a simple numeric variable, an array

element, an entire array, or a numeric field in a structure.

Another consideration in the architecture is a mechanism to handle exceptional conditions. The mechanism uniformly applies to any type of "fault" or interrupt, be it a machine detection of an error, detection of some explicitly identified event (i.e., for ON-units), or a machine detection of some debugging action such as the execution of a particular instruction. Each module is capable of describing what types of faults it desires to handle. When a fault occurs, the machine searches through the activation-record stack looking for the first module that wants to handle that type of fault. When one is found, the machine "calls" the module (entering it at a particular point and making it a subroutine of the module initiating the fault) and passes it arguments describing the fault. A fault-handling entry point has the ability to resume execution at the point beyond the fault, repeat execution of the instruction causing the fault, or to decide to buck the fault up to a higher module. It is assumed that the highest module (the first one invoked in executing a program) is part of the operating system or a debugging tool, and this module will specify that it can handle all types of faults.

In summary, the key attributes of the architecture are:

- Self-identifying, or tagged, storage
- Nested, or recursive tags, for describing less-primitive data types
- Capability-based addressing
- Indirect addressing with capabilities
- Send/receive machanism for interprogram communication and source/sink I/O
- One address space per program module
- Variable-size addresses
- Hierarchical fault-handling mechanism
- Domain addressing (addressing columns in tables)
- One-level store
- Automatic subroutine management via activation stacks
- Fixed-point decimal data representations
- Powerful instruction repertoire, including array operations, a table-search instruction, field/string operations, automatic data conversions
- Generic instructions
- Ability to write highly data-independent programs
- Program-tracing facilities
- Ability to add supplemental instruction sets and data types
- Frequency-based-encoded operation codes

2. DATA TYPES

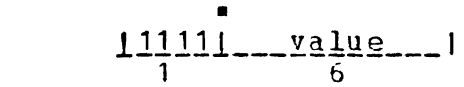
Before discussing the data types, a few basic storage concepts must be introduced. The basic unit of storage allocation is a token, a four-bit quantity. The basic unit of storage addressing is a cell, a variable-number of contiguous tokens. A cell corresponds to a variable or data item in a source program and has two major components: a tag which describes the attributes of the cell, and a content which describes the value of the cell.

The machine recognizes 15 data or cell types of which 10 are considered primitive data types, 1 is a structure data type, and 4 are nested data types. The basic difference among the 3 categories are that primitive cells have single values, structure cells describe collections of other cells, and nested cells have tags which in turn contain tags.

PRIMITIVE CELL TYPES

The primitive cell types are integer, decimal fixed-point, decimal floating-point, boolean field, character field, token field, boolean string, character string, token string, and pointer. The tag of each cell describes its type and size, and the contents component describes its value.

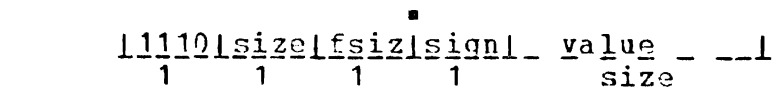
An integer (i) cell has the following format:



The first field (one token) indicates that this cell is an integer cell. The second field contains the value of the cell in base-two two's-complement representation. The value can range from -8,388,607 to +8,388,607. If the second field has the value 800000 (in hexadecimal), the cell has the value "undefined."

The • mark is used in this specification to indicate the boundary between the tag and content components.

A decimal fixed-point (dfx) cell has the following format:

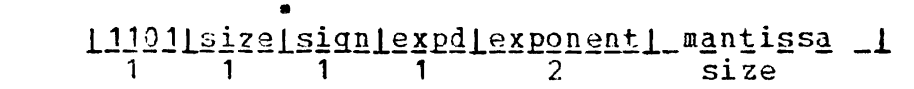


The size field defines the number of digits in the number. The fsiz field indicates the number of digits to the right of an imaginary decimal point and must be less

than or equal to size. The sign field specifies the sign of the value. If it is set to 0000, the value is positive; if it is set to 0001, the value is negative; if it is set to 1111, the cell has the value "undefined." The last field contains the absolute value of the number times 10 to the power fsiz. The value is expressed in the base-10 binary-coded decimal representation.

As an illustration, a variable with attributes FIXED DECIMAL(5,2) and having the value 7.9 would be represented as E52000790.

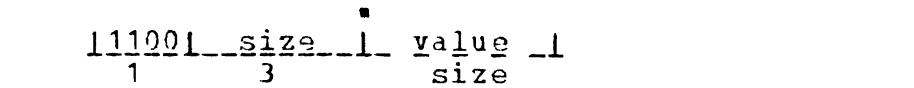
A decimal floating-point (dfl) cell has the following format:



The second field defines the length of the mantissa, the sign field is the same as that described for the previous cell, and the fourth field describes the sign of the exponent. The fifth field contains the absolute decimal value of the exponent (0 to 99). The last field contains the decimal mantissa. Operations on floating-point cells always normalize the mantissa (shift it so that no leading zeros occur unless the cell's value is zero). The exponent and mantissa are expressed in the base-10 binary-coded decimal representation.

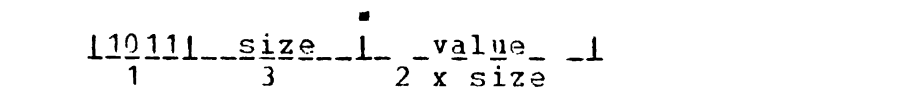
Note that the decimal fixed-point and floating-point cells allow two representations of zero (+0 and -0). Only +0 is a valid representation of zero; -0 is treated as an unknown data format.

The boolean field (bf) cell has the format:



It represents a fixed-length field of boolean (true or false) values. The second field indicates the number of elements (1 to 4094). The third field, whose length is specified by the second field, contains the string elements (one per token). The only element values are 0000 (false), 0001 (true), and 1111 (undefined).

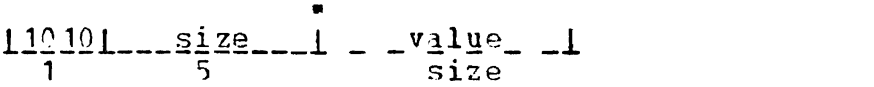
A character field (cf) cell has the format:



It represents a fixed-length field of EBCDIC characters. The second field specifies the number of elements (1 to 4094 characters). The third field, whose

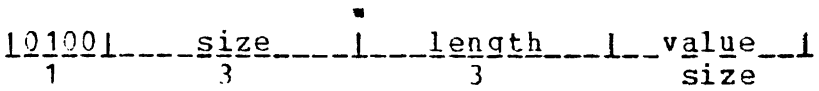
length (in tokens) is two times the value of the second field, contains the string elements. The element value 11111111 indicates an undefined element.

A token field (tf) cell has the format:



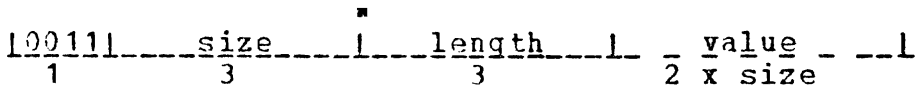
It represents a fixed-length field of four-bit quantities. The second field indicates the number of elements (1 to 1,048,574). The third field, whose length is specified by the second field, contains the elements (one per token). This is the only cell type that cannot have the "undefined" value.

A boolean string (bst) has the following format:



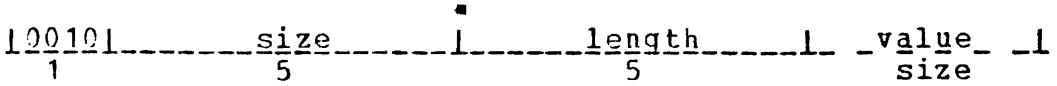
The second field indicates the maximum number of booleans in the string (1 to 4094). The third field indicates the current number of booleans in the string (0 to 4094), thus allowing the string to shrink and grow dynamically. The fourth field contains the actual string where each element is represented as 0000 (false) or 0001 (true). If the length is FFF, the entire string is interpreted to have the undefined value.

A character string (cst) cell has the format:



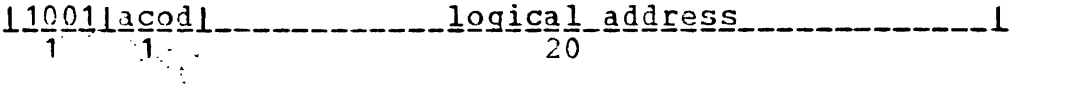
The fields have the meaning described above. The fourth field contains the actual string where each element is represented as 8 bits.

A token string (tst) cell has the format:



The meaning of the fields is the same as described above, but a token string has a maximum size and length of 1,048,574, and each element in the string is a four-bit quantity. If the length field contains FFFFF, the string has the undefined value. The token string is intended for use only by compilers and debugging tools.

The last primitive cell is a pointer (p); it has the following format:



A pointer is a cell that can hold the unique logical address of an object (module, activation record, data-storage object, or port) or an entity within an object (e.g., a cell within an object). Logical addresses are always assigned by the machine and can never be altered by a program. However, a program is allowed to copy the value of one pointer cell into another pointer cell.

The acod field in the pointer cell represents an access or authority code to the object. Its definition is

Bit	Authority if 0	Authority if 1
1	read	no read
2	write	no write
3	destroy	no destroy
4	copy	no copy

The value 1111 is the undefined value. Copy authority is the ability to make a copy of the pointer itself. If a pointer does not have copy authority, it cannot be used as the source operand of a MOVE or SEND instruction.

An instruction is available to allow a program to alter the access code, but the instruction allows one to only lower (further restrict) the access.

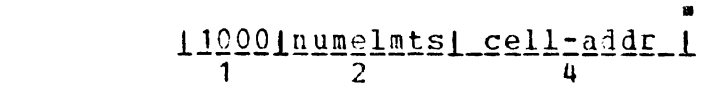
****
* Note: Although the bit content of a logical address is *
* not architected, its interpretation in one implementation*
* of this architecture may be enlightening. The logical *
* address contains a 8-token unique system-object name and *
* two 6-token offsets into the object, offsets of the *
* addressed item's tag and content components. If *
* the logical address refers to an entire object, the last *
* two fields are unused and set to zero. If a logical *
* address refers to a cell within an object, the first *
* field contains the object name and the last two fields *
* are used to locate the cell within the object. Assuming *
* that object names are assigned on the average of one per *
* 10 milliseconds, there is a 10-year supply of unique *
* names. The length of the offset fields imply that the *
* maximum object size is 16 million tokens. *
****

STRUCTURE CELL TYPES

The only data type in this category is a structure cell. A structure describes a heterogeneous collection of

other cells. The properties that distinguish a structure from an array are that, in addition to the entire collection of elements having a name, in a structure each element also has a name, and the elements in a structure can be different data types.

A structure (st) cell has a tag, but no content, component and its format is:



The second field, a binary number from 1 to 255, specifies the number of cells (elements) included in this structure. The last field is the cell address of the first cell in the structure. The element cells must reside in contiguous locations in the address space and must be in the same storage die (see Chapter 3) as the structure cell. Permissible element types are all primitive cells and array cells. Where the structure is an array (see next section), the only permissible element types are domains of primitive cells. Where a structure is a parameter or relocatable (see next section), the only permissible element types are relocatable primitive and array cells.

The concept of a cell address will be defined in a later section, but it will be summarized just briefly here. Each module (e.g., a PL/I external procedure) has an associated address space in which all cells reside that are accessible by the module. A cell address is simply the location of a cell within an address space. As an example the following PL/I structure

```
DECLARE 1 PERSON,
          2 SALARY FIXED DECIMAL (7,2),
          2 NAME,
            3 LASTNAME  CHARACTER (20),
            3 FIRSTNAME CHARACTER (12);
```

would result in five cells. A fixed-point cell and two character-field cells would reside in contiguous storage locations. One structure cell (representing PERSON) would specify three elements and contain the cell address of the fixed-point cell. Another structure cell (representing NAME) would specify two elements and contain the cell address of the first character-field cell.

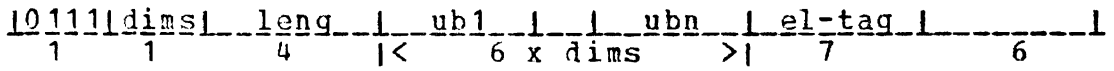
A structure is no more than a collective name for a sequence of other cells and hence is more general than the concept of the same name in such languages as PL/I and COBOL. Machine instructions can operate on structure operands as well as primitive and other cell types. For instance a structure can be passed as an argument, or a structure can be moved into another structure (which causes

the machine to locate and move the physical elements within the structure).

NESTED CELL TYPES

The remaining four cell types are array, parameter, relocatable, and domain.

An array (a) cell has the following format:



The second field is a binary number which specifies the number of dimensions (1 to 15). The third field is a binary number (1 to 65535) which indicates the length of the content component of the array element. When an array tag appears as a nested tag in a parameter or relocatable cell, the length field is not used and can contain any value. The next fields (six tokens in length, one field per dimension) define the upper bound of the array in the corresponding dimension. The product of these fields times the third field is the total number of tokens occupied by the array elements. All dimensions have an implicit lower bound of one.

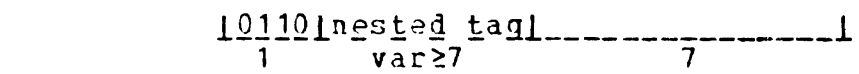
The next field is a nested tag; it is a tag describing the array element. Its length is always seven, but not all seven tokens are always relevant. For instance, for an array of decimal fixed-point values, the nested tag would be three tokens long and would contain 1110 followed by the integer size and fraction size and padded with four "don't care" tokens. Valid element types in an array are all primitive cell types and a structure. When an array element is a structure, the nested tag is a tag for a structure cell. The allowable elements of the structure element are primitive cells. They must be defined as domain cells (discussed below).

Conceptually, the last field, the content component of the array, is viewed as containing the space for the array elements.

```
**************************************************************
* Note: The content component of an array obviously does not *
* include enough space for the array elements.  Since the    *
* machine performs all subscripting operations, the program  *
* need not know the physical location of the elements.  The  *
* last unarchitected field is used by the machine to identify*
* the physical location of the elements.  When an array is   *
* created (at the time the program is loaded, for "static    *
* arrays," at the time an activation record is created, for  *
* "automatic arrays," or at the time an array is explicitly  *
* dynamically allocated by the program), the machine         *
* allocates storage for the elements and places some         *
* internal address in this last field.                       *
**************************************************************
```

All subscripting is done by the machine, and many machine instructions function with entire arrays as well as array elements as operands. As an illustration of an array cell, a one-dimensional, 12 element, array of boolean strings of size 10 would be represented as 71000D00000C400A000XXXXXX.

A parameter (pm) cell has the following format:



Any variable in a module that is received as a parameter is defined by a parameter cell. The second field is a nested tag; it is a tag describing the attributes of the parameter and is used by the machine to check the correspondence between arguments and parameters. Valid tags are tags for all primitive cell types, structures, and arrays. The nested-tag field must be seven tokens in length, unless the nested tag is for an array.

If the nested tag in a parameter cell consists of seven zero tokens, or if the nested tag in a parameter cell is an array tag and the nested tag in the array tag is zero, the parameter is a dynamically typed (D-typed) parameter. A D-typed parameter dynamically takes on the attributes of its corresponding argument. If a parameter cell contains a nested tag having a size field (i.e., decimal fixed-point or floating point, boolean, character, or token string or field) and the size field has the value zero, or if such is the case for a size field in the nested tag of an array tag within a parameter, the parameter is a dynamically sized (D-sized) parameter, meaning that it dynamically acquires the size of the corresponding argument. If the nested tag within a parameter tag is an array tag, and one or more upper-bound fields in the array tag have the value zero, the parameter is a dynamically bounded (D-bounded) parameter, meaning that it dynamically acquires the corresponding extent (upper bound) of the corresponding argument array.

See the last section of this chapter for more details.

The last field is not architected, as was the case for the last field in the array cell. However, if the last field has the value FFFFFFF, the parameter has the value "undefined." Regardless of what is placed in the last field, parameter cells are always initialized by the machine to the undefined value.

```
**************************************************
* The last field contains an access code and an internal    *
* address to the corresponding argument. The internal       *
* address points to the content component corresponding to  *
* the argument.                                             *
**************************************************
```

A program uses a parameter cell as if it were a cell described by the nested tag. The only difference (which is transparent to the program) is that a reference to a parameter causes the machine to indirectly locate the storage via the last field.

A relocatable (r) cell has the following format:

10101	cell-addr	RCA	nested tag
1	4	4	var

A relocatable cell represents a cell whose content component is located elsewhere (i.e., indirectly located). Cells that would be represented as relocatable cells include based variables and element variables in a structure, where the structure is based or a parameter.

The second field is a cell address of the cell from which this cell is relocatable. Such cells are referred to here as locator cells. A locator cell can only be a pointer, parameter pointer, or parameter structure cell.

The third field (relative cell address), in the case of a relocatable cell that is not an element of a structure, must be zero. Where the relocatable cell is an element of a structure (the locator is a pointer to a structure or is a parameter structure), the field represents the cell address of this cell relative to the first cell in the indirectly located structure. For example, RCA for the first element is 1, RCA for the second element is 1 plus the size (tag and content) of the first cell in the indirectly located structure.

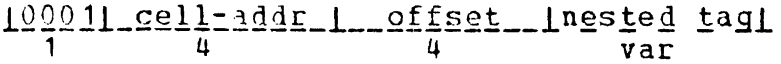
If the relocatable cell represents a structure, but the series of cells named by the structure does not begin at the cell addressed by the locator (i.e., the structure is a "substructure"), RCA is the relative cell address of the first cell in the substructure.

The fourth field is a nested tag defining the type of relocatable cell. Valid nested tags are tags for all primitive cell types, structures, and arrays. In general, if the nested tag of the relocatable cell is not the same as that of the cell indirectly located by the locator cell, the machine will detect it as an error when the relocatable cell is referenced as an instruction operand. However, relocatable cells, in an identical fashion as parameter cells, can be D-typed, D-sized, or D-bounded. See the section at the end of this chapter.

As was the case for a parameter, a program uses a relocatable cell as if it were not one, that is, it uses the cell as if its tag were the nested tag. The machine uses the locator cell to locate the appropriate storage location.

The rules governing the compatibility requirements between the attributes of the indirectly located cell and the attributes of the relocatable cell (i.e., the information in its nested tag) are the same as the rules governing argument/parameter compatibility discussed under the ACTIVATE instruction in Chapter 7 and the section at the end of this chapter discussing D-typed, D-sized, and D-bounded cells. If the compatibility rules are violated, an incompatible-operands fault occurs.

A domain (d) cell has the following format:



It is similar in concept to a relocatable cell, but it represents a structure element in an array of structures. To visualize the concept, think of a one-dimensional array of structures (i.e., a table, where entry in the table contains multiple data items such as a part name and quantity). Array element I corresponds to the Ith row in the table, a domain corresponds to a column in the table, and a domain element corresponds to the Ith value in a particular column.

The second field is a cell address of the array cell. The array must be an array of structures (nested tag is that of a structure). The array cell can be a parameter or relocatable.

The third field defines the offset of the content component of this domain within the array element. That is, the first domain contains the offset zero, the second contains the size of the content component of the first, and so on.

The fourth field is a nested tag defining the type of domain. Valid nested tags are tags for all primitive cells.

Programs address domains as if they were arrays having this nested tag. The array properties of the domain (dimensions, upper bounds) are those in the corresponding array cell. Unless otherwise noted in this specification, discussions of arrays include domains, and discussions of array elements include domain elements.

Hence the machine has 15 cell types. When one accounts for the nested or recursively defined tags, however, the possible cell types are:

Primitive	Parameter array of primitives
Structure	Parameter array of structures
Array of primitives	Relocatable primitive
Array of structures	Relocatable structure
Parameter primitive	Relocatable array of primitives
Parameter structure	Relocatable array of structures
	Domain of primitives

where "primitive" denotes any of the 10 primitive cell types. Also, most cases of parameter and relocatable cells can have the D-typed, D-sized, and/or D-bounded attribute. Uses of many of the cell types are illustrated in examples in Chapter 8.

Note that 15 out of a possible 16 cell types have been defined, implying that only one more cell type could be added if the architecture is extended. This is not necessarily true; if the first four bits of a cell are 0000, this is intended to represent an "escape" code, meaning that the next four bits identify the cell type, thus allowing the machine to potentially have an unlimited number of cell types. A later section describes a feature of the architecture that allows it to have supplemental instruction sets; this escape code allows the supplemental instruction sets to define new cell types. For instance if a FORTRAN-oriented supplemental instruction set is active, a cell beginning with the bits 00001111 might represent a FORTRAN complex number (e.g., a numerical value with a real and an imaginary part).

AUXILIARY DATA TYPES

In addition to the 15 cell types, the architecture also provides several auxiliary data types, which are discussed below.

Indirect Pointers

Indirect addressing is provided in the architecture by the use of indirect pointers. An indirect pointer

physically points to another pointer (which is not an indirect pointer), but logically points to where the latter pointer points. Any reference through an pointer to that which it addresses, in the case of an indirect pointer, is identical to performing the same operation with the latter pointer, except that the access code in the indirect pointer is used. Any operation directly on a indirect pointer (e.g., move, comparison) has the same effect as that on a direct pointer. For instance, if an indirect pointer A points to pointer B, any use of A to reference storage in an instruction has the same effect as using B, although the access code in pointer A, not pointer B, is used. Any operation directly on A itself refers to only A and not B. Indirect addressing occurs whenever pointer resolution occurs (e.g., reference to a relocatable cell, CALL, SEND to a port).

An indirect pointer is not a new data type. It is a pointer cell that has been given a value via the COMPUTE-INDIRECT-POINTER instruction. The pointer is marked in the unarchitected logical-address field as an indirect pointer.

The indirect pointer has many uses. One is security, where program A wishes to give program B access to some data, but program A wishes to retain the right to withdraw this access at any time. By giving B an indirect pointer to a pointer to the data, A, at any time, can modify the latter pointer to withdraw B's access to the data. Another use is dynamic object or module replacement, without having to rebind programs. If module X calls module Y through an indirect pointer, module Y can be replaced with a new version by changing the direct pointer to it and not having to change module X itself. A third use is by object-access-control mechanisms, such as in an operating system. If an operating system contains a mechanism allowing programs to ask for objects with different types of exclusivity (e.g., shared access, exclusive access), it can guarantee this integrity by giving programs indirect, rather than direct, pointers.

D-Typed, D-Sized, and D-Bounded Cells

Parameter and relocatable cells can have the properties of being dynamically typed (D-typed), dynamically sized (D-sized), and/or dynamically bounded (D-bounded). These properties allow one to write generic programs, that is, programs that are significantly independent from the data they are processing.

Parameter fields and strings (character, boolean, and token) and parameter fixed-point and floating-point cells can be specified as D-sized by specifying, in the nested tag of the parameter cell, a zero-valued size. For instance,

6B003000FFFFFF is a parameter character field of size three, but 6B000000FFFFFF is a D-sized parameter character field. Likewise, parameter arrays of fields, strings, and fixed-point and floating-point values can be specified as D-sized by specifying, in the nested tag in the array nested tag, a zero-valued size.

The above is similar to "asterisk notation" in PL/I, but the full concept, as expanded later, is considerably more powerful and efficient. A few examples of D-sized parameters, along with their corresponding representation in a PL/I-like syntax, are

```
                                Q: PROCEDURE(A,B);
6E0XXXXXFFFFFF                  DCL A FIXED DECIMAL(*);
671XXXX000009B000XXXFFFFFF      DCL B(9) CHARACTER(*);
```

(The element-length field in an array nested in a parameter or relocatable cell is never used and can be set to any value.) As usual, an X represents a don't-care value.

If a parameter is D-sized, it dynamically acquires the size attribute of the corresponding argument. See the definition of the ACTIVATE instruction for the type-consistency rules between arguments and parameters.

Relocatable fields, strings, and fixed-point and floating-point values can also be D-sized, providing that the RCA field is zero and that the locator cell is a pointer or parameter pointer. Any relocatable array can be D-sized. D-sized relocatable cells are specified in the same manner as D-sized parameters. A few examples are

```
5YYYY00003000                   DCL A CHAR(*) VARYING BASED(P);
5YYYY000071XXXX000009E0X        DCL B(9) FIXED DECIMAL(*) BASED(P);
```

YYYY represents the cell address of the locator cell P. The PL/I-like examples are hypothetical, since PL/I does not allow such data types.

If a relocatable cell is D-sized, it dynamically acquires, upon each reference, the size of the indirect cell. Consistency requirements between a relocatable cell and the indirect cell are the same as those for arguments and parameters.

D-bounded parameter arrays can be specified by specifying, in one or more of the upper-bound fields in the nested array tag, a zero value. A few examples are

```
                                      Q: PROCEDURE(A,B);
672XXXX0000000000002E42XXXXFFFFFFF    DCL A(*,2) FIXED DEC(4,2);
671XXXX000000B000XXXFFFFFFF           DCL B(*) CHAR(*);
```

As shown in the second case, the D-bounded and D-sized properties are independent; that is, a parameter array can be both D-bounded and D-sized.

If a parameter array is D-bounded, for each zero-valued bound it acquires the corresponding bound of the argument array.

Any relocatable array can be D-bounded; this is achieved in the manner described above. An example is

```
5YYYY000071XXXX000000B000XXX  DCL A(*) CHAR(*) BASED(P);
```

Again, this example is both D-bounded and D-sized. Again, the PL/I notation is hypothetical, since PL/I does not provide this capability.

If a relocatable array is D-bounded, it dynamically acquires upon each reference, for each zero-valued bound, the corresponding bound of the indirect array.

A parameter is specified as being D-typed by having a nested tag of seven zero tokens. A parameter array is specified as being D-typed by encoding zeros in the nested tag (element attributes) within the array nested tag. Examples are

```
                                  Q: PROCEDURE(A,B);
60000000FFFFFFF                   DCL A D-TYPED;
671XXXX0000000000000FFFFFFF       DCL B(*) D-TYPED;
```

Again, the PL/I-like illustrations are hypothetical. The second example is both D-typed and D-bounded.

If a parameter scalar is D-typed (the first example), it dynamically acquires the full attributes of the corresponding argument. However, the argument cannot be a structure or array. If a parameter array is D-typed (the second example), it dynamically acquires the full element attributes of the corresponding array argument. However, the argument array cannot be an array of structures.

A relocatable cell is specified as being D-typed by having a nested tag of six zero tokens. A relocatable array is specified as being D-typed by encoding zeros in the nested tag within the array nested tag. For a relocatable scalar to be D-typed, its RCA field must be zero and the locator cell must be a pointer or parameter pointer. Any relocatable array can be D-typed. Examples are

```
5YYYY0000000000                 DCL A D-TYPED BASED(P);
5YYYY000071XXXX0000000000000    DCL B(*) D-TYPED BASED(P);

                                T: PROCEDURE(TA);
6803QQQQFFFFFFF                 DCL 1 TA,
5RRRR0001B003                         2 TB CHAR(3),
5RRRR000B71XXXX0000000000000          2 TC(*) D-TYPED,
5RRRR0024E42                          2 TD FIXED DEC(4,2);
```

Again, the PL/I-like illustrations are hypothetical. The second example is both D-typed and D-bounded. The third case shows a D-typed and D-bounded relocatable array in a parameter structure. QQQQ is the cell address of the second cell and RRRR is the cell address of the first cell.

If a relocatable scalar is D-typed (the first example), it dynamically acquires, upon each reference, the full attributes of the indirect cell. The indirect cell cannot be a structure or array. If a relocatable array is typed, it dynamically acquires, upon each reference, the full element attributes of the indirect array. The indirect array cannot be an array of structures.

The D-typed, D-bounded, and D-sized properties do not compromise the reliability and security properties of the architecture. They, given the concepts of tagged storage and generic instructions in the architecture, allowing one to write highly data-independent programs. Where there is an mismatch of data types (e.g., one is trying to perform arithmetic on a character field), the D properties still cause the error to be detected, but not perhaps as early as it might have been if the properties were not used. For instance, if a parameter is specified as being a one-dimensional array of 10 character-field elements of size 6, the machine would signal an error (when the procedure or module is invoked, see the ACTIVATE instruction) if the corresponding argument did not have identical attributes. However, for instance, if this parameter was both D-typed and D-bounded, the parameter checking would test for only a one-dimensional array argument. If, during the execution of instructions in the procedure, an incorrect assumption was made about the argument array (e.g., referencing a nonexistent element, using it as an arithmeitc value when it is not, referencing beyond the end of a field/string array element), the error would be detected during the execution of the instruction.

3. STORAGE OBJECTS

The machine contains four types of storage objects: modules, activation records, data-storage objects, and ports.

THE MODULE

The principal storage object in the machine is the module. A module contains a sequence of machine instructions and a definition of the address space for those machine instructions.

A module object corresponds to such programming language constructs as PL/I external procedures and functions, Cobol subprograms, and Fortran subroutine subprograms. A module object is created with a LOAD-MODULE instruction, which takes the external form of a module (shown in Figure 3.1), represented in a token string, and uses it to form a module object. Hence the form of the module object is not architected; it is defined only in terms of the external module. A module object can be destroyed by the DESTROY instruction or, optionally, at the time of program termination.

Figure 3.1 and the subsequent sections define the external module, the principal interface to the machine since it represents the output of a compiler. As shown, an external module consists of three variable-length components: the header, the address space, and the instruction space.

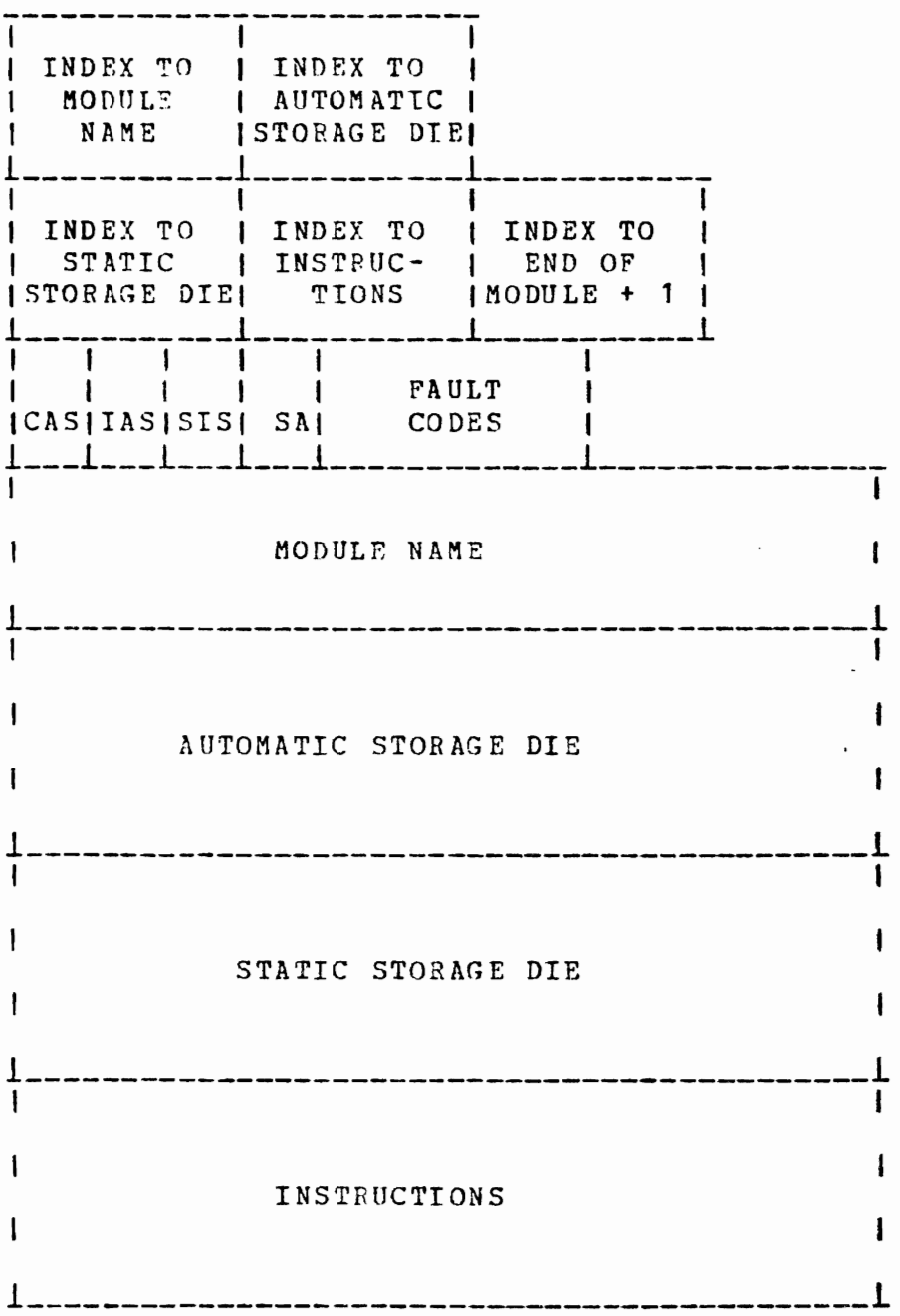


Figure 3.1 Format of an external module.

The Header

The module header defines certain attributes of the module and defines sections of the other two components. The first five fields in the header are five-token fields containing the binary value of the index within the module of the beginning of a particular section of information (except for the fifth field, which indicates the end of the module). Since the index of a section is also used to indicate the end of the previous section, the sections must be contiguous. If a section is not present, its index field points to the start of the next section. For instance, if there is no automatic storage die section, its index field and the index field for the static storage die have the same value.

The next two one-token fields (CAS and IAS) define the lengths of cell addresses and instruction addresses within instructions in this module. Each field can contain a binary value from two to five, indicating two-token addresses to five-token addresses. Cell and instruction addresses are described in the later section on instruction formats. (Note that cell addresses within cells, that is, in structure, relocatable, and domain cells, have a fixed length: four tokens.)

Since the addressing space of a module is limited to only those cells defined in the module, it is desirable to limit the address-field sizes to the smallest size needed. That is, rather than defining fixed-length address fields within instructions, the size of an address field can vary from module to module. Cell addresses need only be large enough to address the cells within the module (the module's address space). Instruction addresses need only be large enough to address instructions within the instruction space. In other words a module with only a few small cells (a small address space) needs only a tiny cell-address field; a module with more and bigger cells needs a larger cell address. Use of variable-size addresses is worthwhile because 1) the physical size of the module can be reduced, 2) the number of bits transmitted between the memory and the processor can be reduced, thus increasing the memory bandwidth, and 3) arbitrary compromises concerning the upper bound of an address space can be avoided.

The next one-token field (SIS - supplemental instruction set) in the header defines the language in which this module was written. The motivation for this field was the thought that the basic instruction set of the machine might be extended to provide additional instructions that are specialized toward a particular language. For instance if this field is zero, operation code '0007' might be invalid. If the field is one, operation '0007' might be a COBOL-oriented table search instruction; if the field is

two, operation '0007' might be a PL/I-oriented PICTURE editing instruction. If the field is three, operation '0007' might be an instruction intended only for the operating system. This points out another motivation for such a feature: there is no need (nor desire) to bother a COBOL compiler writer with information about instructions intended for the operating system. In fact it is desirable to hide such instructions from those people and programs that have no direct use for them.

This "language" or supplemental instruction set field gives the machine the ability to vary part of its instruction set dynamically and gives system designer the ability to specialize and tailor the instruction set in a way that is transparent to existing programs.

```
**************************************************
* The current implementation of the architecture contains  *
* one supplemental instruction set (SIS=1111) for the      *
* benefit of the operating system.  It contains a few      *
* instructions, which are not described in this document.  *
* These instructions, like all others, are not privileged. *
**************************************************
```

The next one-token field (SA) currently has no purpose.

The next six-token field specifies the faults (machine-detected conditions) that this module wishes to handle. The meaning of this field is described in a later section on fault handling.

The next field is variable in length and contains the name of the module, using two tokens to represent each character. No machine instructions currently access this field, so it need not be present.

The Address Space

The second component of a module is its address space. The address space contains a series of cells defining the data that is accessible by the module. The index of the first token of a cell within the address space is known as its cell address. That is, the cell address of the first cell is one; the cell address of the second cell is one plus the total length of the first cell, and so on.

Although the address space looks like one entity to the program, it is subdivided into two sections as shown in Figure 3.1. These two sections are used by the machine for storage management and allocation purposes.

The automatic storage die holds all cells that are to be dynamically allocated space whenever the module is invoked. When the module is invoked, the machine allocates

an activation record and copies the automatic storage die into the activation record. When the module's code refers to a cell in the automatic storage die, the machine automatically translates its cell address to a location within the activation record.

Note that the machine does a bit-by-bit copy of the automatic storage die into the activation record. This implies that the compiler can cause an automatic variable to have an initial value simply by putting the value in the variable's cell in the automatic storage die. If an automatic variable (or any other variable) has no defined initial value, the compiler is responsible for setting the cell to the undefined value. An exception to this discussion is pointer cells; for reasons of security, the machine always initializes them with the undefined value when the module object is created. All cell types may appear in the automatic storage die. If an array cell resides in the automatic storage die, space for the array elements is created in the activation record and the elements are initialized to the undefined value. Parameter cells must reside in this storage die and are always set to the undefined value when the module object is created.

The static storage die holds all cells that are to be allocated once prior to execution (that is, at the time of the LOAD-MODULE instruction). If a static variable is to have an initial value, the value should be placed in its cell in the die. If not, the cell should be set to the "undefined" value. (All pointer cells are always initialized by the machine to the "undefined" value.) All cell types except parameter may appear in the static storage die. Array elements are initialized to the undefined value.

```
**************************************************
* Since relocatable, domain, and structure cells do not  *
* directly change during execution (only the cells to    *
* which they refer do), it is recommended, for reasons of *
* performance, that these cell types be placed in the    *
* static storage die.                                    *
**************************************************
```

The Instruction Space

The last component of a module is its instruction space. The instruction space contains a series of machine instructions. Most machine instructions are represented in a variable number of tokens. The index of the first token of an instruction within the instruction space is known as its instruction address. The instruction address of the first instruction is one; the instruction address of the second instruction is one plus the length of the first instruction, and so on.

```
**********************************************************
* Programming Note: Since array elements receive no space *
* in the dies, it is not immediately obvious how a compiler*
* would initialize an array. The following suggestion is  *
* offered. If the array is automatic, the compiler must   *
* generate code (one or more MOVE instructions) at each   *
* entry point to initialize the array. To initialize a    *
* static array, the compiler can give the module an extra *
* entry point and generate code at this entry point to    *
* initialize the array. After the LOAD-MODULE instruction *
* has been executed, this special entry point can be      *
* called to initialize the static array.                  *
**********************************************************
```

THE ACTIVATION RECORD

An activation-record object contains space for the cells in a module's automatic storage die. It is created whenever a module is invoked (by a CALL instruction) and destroyed whenever a module returns to its caller or the program terminates. Since a program does not directly "see" an activation record, but addresses it through the automatic storage die, no further information about the activation record is architected.

THE DATA-STORAGE OBJECT

A data-storage object is created by a program that wishes to dynamically allocate space for a relocatable cell. It is created by an ALLOCATE instruction and can be destroyed by the DESTROY instruction or, optionally, at program termination. Since a program does not directly "see" a data-storage object, but addresses it through a relocatable cell, no further information about the data-storage object is architected.

THE PORT

A port is an abstract object that is used to connect two or more programs together for purposes of interprogram communication. A port is created by a CREATE-PORT instruction and is destroyed by the DESTROY instruction or at program termination. Since a port is defined only by the semantics of the two instructions that can operate on it, SEND and RECEIVE, no further information about the port is architected.

4. INSTRUCTION FORMATS AND ADDRESSING

A machine instruction consists of an operation code followed by one or more address fields. Some instructions have just one address field, others have two, others have three, and certain instructions have a variable number of address fields.

OPERATION CODES

The first field of each instruction is the operation code. Rather than use a single-length field for operation codes, a frequency-based encoding was done. That is, the operation-code field for the fifteen most-frequent instructions is one token long, the field is two tokens long for the second most-frequent set of fifteen instructions, and so on. The motivation for doing a frequency-based encoding, the rationale for choosing this particular encoding, and the selection of the operation codes is discussed in other documentation available from the author.

ADDRESS FIELDS

There are seven types of address fields which are grouped into three categories: operand addresses, instruction addresses, and immediate fields.

An operand address references an operand in the address space. There are four types of operand addresses.

1. Cell address. A cell address is an N-token binary field that refers to a cell in the address space (N is the value of the cell-address-size field in the module header). For instance, if N (CAS) has the value 2, the operand address 1A refers to the cell beginning at the 26th token in the module's address space. Cell addresses cannot be used to address array or domain cells.
2. Literal. A literal field consists of N tokens of zeros followed by one token having the value zero, one, ..., or nine. A literal field is assumed to be a one-digit positive integer. As an example, if N (CAS) has the value 2, the operand address 004 is a literal of value +4.
3. Array element address. An array element address consists of D+1 subfields. The first subfield is a cell address of an array (or domain) having D dimensions. The next D subfields are cell addresses, literals, or array element addresses specifying the values of the subscripts (the values must be integers). For example if array

cell A has the index (in hexadecimal) of 20 in the address space, if a variable I has the index 3C, and if N is 2, then the operand address for A(4,I) is 200043C. If N was 3, the operand address would be 020000403C.

4. Array address. An array address refers to an entire array or domain. Array addressing is identical to array element addressing, except that all of the subscript subfields are specified as "*". The "*" is represented by a literal field with the value F (1111). Hence array A is addressed by 2000F00F.

```
**********************************************************
* Note: This addressing scheme allows for the possibility *
* of addressing array cross-sections (e.g., the PL/I      *
* expression A(*,I) would produce the address 2000F3C),   *
* although this is currently not part of the architecture. *
**********************************************************
```

Unless otherwise mentioned, any of these four forms can be used as operand addresses in instructions. One general exception is that a literal cannot be used as a target operand. An operand is that data referred to by an operand address (possibly indirectly through a relocatable, structure, domain, or parameter cell); a target operand is an operand in which an instruction stores a result.

The second category of address fields is an instruction address. An instruction address is an M-token field that refers to an instruction in the instruction space (M is the value of the instruction-address-size field in the module header).

The last category of address fields is an immediate field. An immediate field is a one or two-token field containing not an address but some value that is used directly by the instruction. Since immediate fields have specialized purposes and are only used in a few of the instructions, definition of the immediate fields is deferred to the definitions of these instructions.

5. FAULT HANDLING

Since the major objective of this machine is to prevent and/or detect certain classes of programming errors, the methods by which the machine detects and reports errors are of special importance. This section defines the conditions (called faults) detected by the machine, the information that the machine presents to the program when a fault occurs, and how the program and machine can interact to handle faults.

FAULT DESCRIPTIONS

The following descriptions define the types of faults detected by the machine and the situations under which they arise. If multiple fault situations occur during the execution of an instruction, the first type of fault detected by the machine, or the order of the faults detected, is not architected.

An invalid operation (type 1) fault occurs when the machine fetches an instruction but its operation code is invalid, or when the end of the instruction space is encountered during the fetching of an instruction.

An addressing (type 2) fault occurs when (1) a cell address is being used but it falls beyond the module's address space or resides in an incorrect storage die, (2) when an array subscript is not an integer, (3) when an array cross-section address is specified, (4) when a reference within a module from outside (e.g., via the LINK or COMPUTE-ENTRY-POINTER instruction) does not obey the addressing rules of the instruction, (5) when an error is detected while processing cell addresses (e.g., the rules concerning relocatable cells are not obeyed), or (6) when a loop is detected when resolving indirect pointers (e.g., an indirect pointer refers to itself).

An unknown data format (type 3) fault occurs when the machine references a cell that has an unrecognizable format or value.

A protection (type 4) fault occurs when (1) the program attempts to destroy, write to, or read from a cell that is located through a pointer cell, but the pointer does not have the appropriate access code, (2) the program attempts to explicitly destroy storage that resides within an activation record or module, (3) the program attempts to alter a parameter that was transmitted as read-only, or (4) the program attempts to move the value of a pointer cell which does not have copy authority.

An invalid pointer (type 5) fault occurs when the program uses a pointer cell but the logical address in the pointer is unknown to the machine (implying that the storage referred to by the pointer has been previously freed).

A bounds-exceeded (type 6) fault occurs when the program refers to an array element using a subscript that is beyond the bounds of the corresponding dimension, or when a program refers to a string element that is beyond the size or current length of the string.

An invalid operand type (type 7) fault occurs when the type of an operand does not match the valid operand type(s) in the instruction specification, or when the category of storage object being referenced by an instruction does not match the categories of storage objects that can be referenced by the instruction.

An undefined operand (type 8) fault occurs when the machine attempts to use the value of an operand, but 1) the operand, or 2) a pointer or 3) parameter cell used to locate the operand, has the value "undefined." This fault does not occur for condition 1 in the DEFINED instruction, which is an explicit test to determine if an operand has an undefined value.

An incompatible operands (type 9) fault occurs when two or more operands of an instruction are incompatible. The conditions of operand compatibility are defined in the specifications of the instructions. This fault can also occur in an ACTIVATE or LOCAL-ACTIVATE instruction when the type of a parameter cell is incompatible with the type of the corresponding argument cell, or in a RECEIVE or SEND instruction when the type of a receiver operand is incompatible with the type of corresponding argument. The fault also occurs when the attributes of a relocatable cell do not match the attributes of the indirectly located data.

An overflow (type 10) fault occurs when the target operand in an instruction is too small to hold the value produced by the instruction. For arithmetic operands this occurs when loss of high-order non-zero digits would occur or when the exponent of a floating-point result is greater than 99. For string operands this occurs when the size of the target string is too small to hold the value produced by the instruction.

An underflow (type 11) fault occurs when the floating-point result of an instruction has an exponent of less than -99.

A divide (type 12) fault occurs when division by zero is attempted.

An invalid module (type 13) fault occurs during a LOAD-MODULE instruction when the machine discovers a format error in the module being loaded.

An invalid transfer (type 14) fault occurs for various reasons in an instruction that transfers control flow. The most common situation is attempting to branch beyond the instruction space of the module.

An invalid transmission count (type 15) fault occurs in an ACTIVATE or LOCAL-ACTIVATE instruction when the number of parameters specified does not equal the number of arguments transmitted, or in a RECEIVE and SEND instruction when the number of receiver operands does not equal the number of arguments in the corresponding SEND instruction.

A conversion (type 16) fault occurs during the CONVERT instruction when the operands do not match the conversion rules listed in the specification of the CONVERT instruction.

A yes-branch-trace (type 17) fault occurs during any instruction in the comparison-and-branch group (except ITERATE) if 1) the instruction results in a branch being taken and 2) yes-branch tracing is enabled for the module containing the instruction.

A no-branch-trace (type 18) fault occurs during any instruction in the compariosn-and-branch group (except ITERATE) is 1) the instruction results in the branch not being taken and 2) no-branch tracing is enabled for the module containing the instruction.

A call-trace (type 19) fault occurs during the execution of a CALL or LCALL instruction if call tracing is enabled for the module containing the instruction.

An insufficient-storage (type 20) fault occurs when an instruction requires the machine to dynamically acquire storage for a storage object, but sufficient storage is not available.

A fault-handling (type 21) fault occurs when 1) one attempts to CONTINUE beyond a fault for which continuing is prohibited, 2) one attempts to execute a RAISE-FAULT instruction with an invalid fault type, or 3) one attempts to execute a CONTINUE or TRANSFER-FAULT instruction while not in a fault handler.

ENTRY-POINT ZERO

Each fault type has an associated number as given in the previous section. These numbers also correspond to a bit position in the fault-code field in the module header. For example fault type 1 (invalid operation) corresponds to the bit 1 in the fault-code field. If a bit is set to one in the fault-code field in a module, this indicates that this module desires to handle the associated fault. Bit 0 (the first bit) in the fault-code field indicates whether the module desires to handle faults of type 28-255 (program-defined faults - see the RAISE-FAULT instruction).

When a fault occurs, the machine attempts to call entry-point zero of the current module. (Entry-point zero is the first instruction in the module.) Entry-point zero will be called if the fault is enabled (the corresponding bit in the fault-code field is one). If not, the machine attempts to call another entry-point zero by searching backwards through the stack of active modules until a module is found with this fault enabled. If none are found, the program is terminated.

```
**************************************************
* Programming Note: It is anticipated that the first    *
* module invoked in each program is a special module    *
* generated by the compiler or operating system that has *
* all faults enabled.                                   *
**************************************************
```

When an entry-point zero (hereafter called a fault handler) is invoked, it is called by the machine as an internal procedure. Therefore the fault handler has addressability to the address space in the module in which the fault handler resides. The machine also passes the following five arguments to the fault handler:

1. An integer containing the fault type.
2. A pointer to the module object in which the fault arose. The pointer has read and copy authority.
3. A pointer to the entry point at which the faulting module was entered. The pointer has read and copy authority.
4. A token-field cell of size 5 containing the instruction address of the instruction causing the fault.
5. A token-field cell of size 6.

The fault handler is given read-only access to the arguments.

For the invalid-module fault, argument 5 is an error code describing the error in the module (provided that the fault was not raised by the RAISE-FAULT instruction). For

all other faults, argument will be taken, 5 contains the first six tokens of the faulting instruction.

Machine instructions are available to allow a module to dynamically enable or disable specific faults, to allow a program to explicitly generate faults, and to allow a fault handler to resume execution at the instruction following the faulting instruction, retry the faulting instruction, or to transfer the fault to a higher fault handler.

A fault in a fault handler is treated like any other fault situation. The only difference is that, to prevent a faulting fault handler from entering endless recursion, the search for an applicable fault handler starts with the module that called the module containing the faulting fault handler.

Faults are nested, meaning that if a fault occurs in fault handler A and is handled by fault handler B, which returns or continues to fault handler A, A is back in its original state (i.e., the fault arguments available to A still describe the initial fault).

Note that the fault handler is assumed to start at the first instruction in the module. Either the fault-handling code, (beginning with a LACT instruction) or a branch to the fault-handling code, is placed here.

PROGRAM STATE AFTER A FAULT

A key consideration in fault handling is the state in which the machine leaves the program when a fault occurs. In most cases the faulting instruction does not affect the state of the program. A fault handler terminates with one of four instructions: LOCAL-RETURN, which terminates the fault handler and begins execution again of the faulting instruction, CONTINUE, which terminates the fault handler and resumes execution at the instruction that would have been executed next, had the fault not occurred, RETURN, which deletes the activation record for this module and all later modules and returns control to the module that previously called this module, and TRANSFER-FAULT, which terminates the fault handler and causes the machine to search for and call a higher fault-handler. Exceptions to these general rules are discussed below.

1. Issuing the LRETURN instruction to return from a fault generated by a RAISE-FAULT instruction causes execution to resume at the instruction following the RAISE-FAULT instruction.
2. Faults that occur during the processing of a field, structure, or array result in the elements processed before the fault taking on their new

values, but all remaining elements remain unchanged.

3. If an overflow or underflow fault occurs, the target operand is given the undefined value.
4. Issuing the CALL instruction in a fault handler causes any and all activations beneath that of the current activation to disappear (to avoid turning the activation stack into a tree).

6. INSTRUCTION SUMMARY

This section summarizes the instructions of the machine. Chapter 7 describes each instruction in greater detail.

General Instructions

The three general instructions are MOVE, CONVERT, and UNDEFINE. Operands of the three instructions may be single scalar cells, arrays, strings, domains, fields, and structures. MOVE is used to transfer the value of one operand to another. CONVERT performs the same function as MOVE, but it also performs an explicit data conversion. For instance if one used a MOVE instruction to move a character value into an integer, the operation would fail and an incompatible-operands fault would occur. If one used a CONVERT instruction, the operation would succeed; the character value would be converted into an integer according to a set of predefined rules.

The UNDEFINE instruction is used to set the value of an operand to undefined.

Arithmetic Instructions

The arithmetic instructions include ADD, SUBTRACT, MULTIPLY, DIVIDE, REMAINDER, ABSOLUTE, COMPLEMENT (unary minus), and POWER (compute X to the Yth power). The ADD, SUBTRACT, MULTIPLY, DIVIDE, REMAINDER, and POWER instructions have two operands; the result is stored in the first operand. ABSOLUTE and COMPLEMENT have one operand. The operands must be arithmetic scalars or arrays.

Comparison-and-Branch Instructions

The EQUAL-BRANCH-FALSE, NOT-EQUAL-BRANCH-FALSE, LESS-THAN-BRANCH-FALSE, GREATER-THAN-BRANCH-FALSE, LESS-THAN-OR-EQUAL-BRANCH-FALSE, and GREATER-THAN-OR-EQUAL-BRANCH-FALSE instructions have two operands and an instruction address. The values of the operands are compared; if the condition is false, control is transferred to the specified instruction address. In general the two comparison operands may be any cell types (e.g., pointer, character string, array, structure).

The remaining two instructions are DEFINED-BRANCH-FALSE and ITERATE. The first tests an operand to determine if its value is defined. ITERATE is provided for loop control in iterative DO loops.

Boolean Instructions

The boolean instructions are AND and OR, which have two operands, and NOT, which has one operand. The operands may be boolean, boolean strings, or arrays of booleans or boolean strings.

String and Search Instructions

Although many of the machine instructions can have string operands, the string instructions work exclusively with string operands. The operands may be boolean, character, or token fields or strings.

The CONCATENATE instruction appends the value of one operand to the end of the other operand. The MOVESUBSTRING instruction overlays a substring in one operand onto a substring in the other operand. The INDEX instruction searches a string for a designated substring. The LENGTH instruction returns the current length of a string.

The remaining instruction in this group is SEARCH. Given an array or domain and a search value, it returns the subscript value of the element whose value is equal to the search value.

Control Instructions

The control instructions are associated with unconditional transfers of execution flow. The CALL, ACTIVATE, and RETURN instructions are associated with calls to modules, the LOCAL-CALL, LOCAL-ACTIVATE, and LOCAL-RETURN instructions are associated with calls to local subroutines within a module, and the BRANCH instruction alters execution flow within a module.

The CALL instruction specifies the entity being called (entry-point within a module) and a list of arguments. A subset of these arguments may be designated as being read-only, implying that the called module may not alter nor free them. CALL allocates the storage specified in the automatic storage die of the called module and branches to the specified entry point. If parameters are to be received by a called entry point, an ACTIVATE instruction must be executed in the called module before the parameters are referenced. The ACTIVATE instruction specifies a list of parameters. The instruction checks the compatibility of the arguments and parameters and initializes the parameters (the transmission method is by-reference). The RETURN instruction frees the automatic storage and transfers control to the module that called this module.

The LCALL instruction specifies an instruction address of a local procedure and a list of arguments. The first

instruction of a local procedure must be LACT (LOCAL-ACTIVATE). LACT specifies a list of parameters and causes the compatibility of the arguments and parameters to be checked and the parameters to be initialized. LCALL does not allocate any automatic storage, which means that the machine provides only minimal support of local procedures. If storage allocation and scope-of-name rules are necessary, they are the compilers' responsibility. The LRETURN instruction transfers control back to the instruction following the last LCALL instruction.

The GUARD and UNGUARD instructions are provided to protect critical sections of instructions from simultaneous execution, allowing one to use the program-design concept of monitors.

Addressing Instructions

This group of instructions is associated with the manipulation of pointers and storage objects. The COMPUTE-POINTER instruction produces a pointer to a specified operand. COMPUTE-INDIRECT-POINTER creates an indirect pointer to a pointer. The CHANGE-ACCESS instruction is provided to lower (further restrict) the access code in a pointer. The ALLOCATE instruction is used to dynamically allocate storage space, and the FREE instruction is used to dynamically free an object (i.e., a module or a dynamically allocated storage space). CHANGE-LOGICAL-ADDRESS allows one to rename (cause the machine to assign a new logical address to) an existing object.

The LOAD-MODULE instruction defines a module to the machine and returns a pointer to it. The COMPUTE-ENTRY-POINTER instruction is used to compute the logical address of an entry point or cell in a designated module. The LINK instruction is used to assign a value to a pointer cell in a loaded module. (COMPUTE-ENTRY-POINTER and LINK are used to bind modules; that is, they are used by "linkage-editing" functions.)

The DESCRIBE instruction, given a pointer as an operand, returns certain descriptive information about the pointer and that to which it points.

The CREATE-PORT, SEND, and RECEIVE instructions are used for interprogram communication. SEND transmits a message to a port, and RECEIVE accepts a message from a port.

Debugging Instructions

The last set of instructions are associated with debugging and fault-handling functions. The ENABLE and DISABLE instructions provide the program with a way to dynamically enable or disable faults designated for the module's fault handler. The RAISE-FAULT instruction is used to explicitly trigger a fault and enter a fault handler.

The CONTINUE instruction provides a fault handler with the ability to resume execution of the faulting module at the instruction following the faulting instruction. (LRETURN is used to resume execution at the faulting instruction.) If a fault handler determines that a fault should be transferred to a "higher" fault handler, the TRANSFER-FAULT instruction is used.

The DISPLAY-TAG and DISPLAY-CONTENTS instructions are intended for debugging operations. Given a cell address and a pointer to a module, the instructions will place either the tag or the content of the referenced cell into a token string.

The TRACE and NOTRACE instructions are used for monitoring execution flow. The TRACE instruction enables a trace of branch instructions, call instructions, or both in a specified module, and the NOTRACE instruction disables the same. If a branch trace is enabled for a module, all comparison-and-branch instructions, except ITERATE, generate a branch-trace fault. Branch tracing can be specified for situations where the branch is taken, the branch is not taken, or both. If a call trace is enabled for a module, all CALL and LCALL instructions generate a call-trace fault.

7. INSTRUCTION SPECIFICATIONS

This chapter defines the basic instruction set of the machine. General notes that are applicable to many of the instructions are:

1. Where an instruction permits two operands to be arrays, the arrays must be conformable. That is, they must have the same number of dimensions and the same number of elements in each dimension. The same applies to domains.
2. Where an instruction permits two operands to be structures, the structures must be identical. That is, each structure must contain the same number of elements. Corresponding elements in each structure must have identical attributes (tags).
3. Where an instruction specifies a particular cell type as a valid operand, the operand can also be a nested cell, unless otherwise noted. For instance, if an operand should be an integer, the operand address can point to an integer cell, an element in an array of integers, a relocatable integer, an integer parameter, an integer domain element, etc.
4. Most of the instructions can generate a common set of faults. For brevity, the set of fault types named the general set is defined as including the following faults: addressing, unknown data format, protection, invalid pointer, bounds-exceeded, invalid operand type, undefined operand, and incompatible operands.
5. "Arithmetic operands" are defined as the set - integer, literal, fixed-point, and floating-point. "String/field operands" are defined as the set - boolean string and field, character string and field, and token string and field. "Character operands" are the set - character field and character string.
6. In the specifications of instruction formats, the first field is the operation code, which consists of one to four tokens depending on the instruction. The abbreviation "OA" designates an operand address; "IA" designates an instruction address.
7. Literals are permitted as operand addresses, except where the instruction alters the operand's value or where the operand cannot be arithmetic.
8. The length of a boolean, character, or token field is the value of its fixed-size field in the tag. The length of a boolean, character, or token string is the value of the length field in the content component.

GENERAL INSTRUCTIONS

Instruction: MOVE
Function: The value of the second operand is moved into the first operand.
Format: 1,OA,OA
Operands: Both operands must be compatible, that is, both must be arithmetic, character, boolean, token, pointers, or structures. Both operands can be arrays or domains, implying that an element-by-element move is done, or the first operand can be an array or domain and the second not, meaning that the value of the second operand is moved into each element.

If the operands are arithmetic but have different types or sizes, the result is first converted to agree with the first operand. No rounding ever occurs in the MOVE instruction. When a string or field is moved into a string, the length of the first operand is set equal to the length of the second operand. On a move into a character or boolean field where the second operand is shorter than the first, the first operand is padded on the right with blanks (if character) or zeros (if boolean). On a move into a token field where the second operand is shorter than the first, the first operand is padded on the left with zeros.

The operand combinations (first/second)

token field	character field
token field	character string
token string	character field
token string	character string

are valid, and the combinations

character field	token field
character field	token string
character string	token field
character string	token string

are valid. A straight move is done (no conversion of values, other than the length).

A move of a structure into a structure requires that both structures have the same number of elements, and that the elements have identical attributes. A structure move is semantically identical to specifying a move of each individual element.
Faults: General set (excluding invalid operand type) plus overflow.

Instruction: CONVERT
Function: The value of the second operand is moved into the first operand. A limited number of conversions may be done if the types of the two operands differ.
Format: 09,0A,0A
Operands: The rules of the MOVE instruction apply, but the rules concerning operand compatibility are somewhat relaxed. Table 7.1 describes the valid conversions. A blank in the matrix indicates that no conversion will be performed and the incompatible-operands fault will occur. If a conversion is attempted but the value of the second operand does not meet the conversion rules, a conversion fault will occur. The operands cannot be entire arrays or structures.
Faults: General set plus conversion and overflow.

Operand2 type

Operand1 type	i	dfx	dfl	bf	cf	tf	bst	cst	tst
i	1	1	1		4	2		4	2
dfx	1	1	1		5			5	
dfl	1	1	1		6			6	
bf				1	8	9	1	8	9
cf	15	16	17	11	1	12	11	1	12
tf	7			13	14	1	13	14	1
bst				1	8	9	1	8	9
cst	15	16	17	11	1	12	11	1	12
tst	7			13	14	1	13	14	1

1- Acts identical to a MOVE instruction.
2- Converts it from binary to a positive integer value.
4- All characters must be numeric ("0" - "9") except for the first, which can optionally be a "+" or "-".
5- The string must be numeric optionally preceded by a "+" or "-", or an optional "+" or "-" followed by zero or more numerics followed by a "." followed by zero or more numerics.
6- The string must be 1) numeric optionally preceded by a "+" or "-", or 2) an optional "+" or "-" followed by zero or more numerics followed by a "." followed by zero or more numerics, or 3) a number of form 2 followed by "E", followed by an optional "+" or "-", followed by one or two numerics.
8- Character(s) must be "0" or "1", or "F" or "T".
9- Token(s) must be 0000 or 0001.
11- Produces the character(s) "F" or "T".
12- Produces the character(s) "0"-"9" and "A"-"F".
13- Produces the token(s) 0000 or 0001.
14- Character(s) must be "0"-"9" and "A"-"F".
15- Produces a string of numerics, preceded by a "-" if the number is negative.
16- Produces a string of the form "numerics.numerics", preceded by a "-" if the number is negative or a blank if positive.
17- Produces a string of the form "0.numericsEnumerics". If the number is negative, a "-" precedes the string; if positive, a blank precedes the string. If the exponent is negative, a "-" follows the "E".

Table 7.1 Conversion rules.

Instruction: UNDEFINE (UNDEF)
Function: The value of the operand is set to undefined.
Format: 001,0A
Operands: The operand can be of any type. If it is a token field, the instruction has no effect. If the operand specifies a collection of cells (array, domain, or structure), each element receives the undefined value.
Faults: General set (excluding incompatible operands).

ARITHMETIC INSTRUCTIONS

Instruction: ADD
Function: The values of the two operands are added and the result is placed in the first operand.
Format: 2,0A,0A
Operands: Both operands must be arithmetic. Both operands can be arrays or domains, implying that an element-by-element addition is performed. If the first operand is an array or domain and the second operand is a scalar, then the second operand is added to each element of the first operand.

If the operands have different types or sizes, the value of the second operand is temporarily converted or adjusted to agree with the first operand before the addition is performed. The rules of arithmetic are identical to those in the PL/I language. Floating-point results are always normalized.
Faults: General set plus overflow and underflow.

Instruction: SUBTRACT (SUB)
Function: The value of the second operand is subtracted from the value of the first operand and the result is placed in the first operand.
Format: 3,0A,0A
Operands: See ADD instruction.
Faults: General set plus overflow and underflow.

Instruction: MULTIPLY (MULT)
Function: The values of the two operands are multiplied and the result is placed in the first operand.
Format: 4,0A,0A
Operands: See ADD instruction.

Faults: General set plus overflow and underflow.
Notes: In the case of array operands, an element-by-element multiplication is done, not a "matrix multiplication."

Instruction: DIVIDE
Function: The value of the first operand is divided by the value of the second operand and the result (quotient) is placed in the first operand. If the first operand is an integer, it becomes the integer whose magnitude is the largest integer that does not exceed the mathematical quotient and whose sign is the same as the sign of the mathematical quotient.
Format: 02,0A,0A
Operands: See ADD instruction.
Faults: General set plus overflow, underflow, and divide.

Instruction: REMAINDER
Function: The value of the first operand is divided by the value of the second operand and the remainder is placed in the first operand.
Format: 002,0A,0A
Operands: Both operands must be integers. Both can be arrays or scalars, or the first can be an array and the second a scalar.
Faults: General set plus overflow and divide.

Instruction: ABSOLUTE (ABS)
Function: The sign of the operand is set to positive.
Format: 01,0A
Operands: The operand must be arithmetic. If the operand is an array, the operation is performed on each element.
Faults: General set (excluding incompatible operands).

Instruction: COMPLEMENT (COMP)
Function: The sign of the operand is reversed.
Format: F,0A
Operands: The operand must be arithmetic. If the operand is an array, the operation is performed on each element.
Faults: General set (excluding incompatible operands).

Instruction: POWER
Function: The value of the first operand is raised to the power given by the value of the second operand and the result is placed in the first operand.
Format: 008,OA,OA
Operands: Both operands must be arithmetic. If the first operand is an integer, then the second operand must be an integer.
The first operand can be an array, implying that the operation is performed on each element. The result is always rounded if least-significant digits will be lost. Floating-point results are always normalized.
Faults: General set plus overflow and underflow.

COMPARISON-AND-BRANCH INSTRUCTIONS

Instruction: EQUAL-BRANCH-FALSE (EQBF)
Function: If the values of the operands are equal, the instruction has no effect; otherwise, control is transferred to the specified instruction address.
Format: 7,OA,OA,IA
Operands: The operands must be compatible (both arithmetic, character, boolean, pointer, or token). If they are arithmetic but have different types or sizes, the value of the second operand is temporarily converted to agree with the first operand before the comparison is made. (Overflow faults never occur. If an overflow condition is encountered, the two operands are defined as unequal.)

Comparisons between arithmetic values of dissimilar attributes are consistent with the rules of PL/I.

If the operands are strings and/or fields of unequal length, the shorter is temporarily padded with blanks (for character) or zeros (for boolean or token) before the comparison is made. Character and boolean strings/fields are padded on the right and token strings/fields are padded on the left.

The first operand may be an array or domain, or both operands may be arrays or domains, in which case an element-by-element comparison is done. If the first operand is a structure, the second operand must be an identical structure and an

element-by-element comparison is done. The result is true only if the relation holds between all corresponding elements. If the operands are pointers, only the logical addresses (not the access codes) are compared.

Faults: General set (excluding invalid operand type) plus invalid transfer, yes-branch trace, and no-branch trace.

Instruction: NOT-EQUAL-BRANCH-FALSE (NEBF)

Function: If the values of the operands are unequal, the instruction has no effect; otherwise, control is transferred to the specified instruction address.

Format: 6,OA,OA,IA

Operands: See EQUAL-BRANCH-FALSE instruction.

Faults: See EQUAL-BRANCH-FALSE instruction.

Instruction: LESS-THAN-BRANCH-FALSE (LTBF)

Function: If the value of the first operand is less than the value of the second operand, the instruction has no effect; otherwise, control is transferred to the specified instruction address.

Format: 8,OA,OA,IA

Operands: The operands must both be arithmetic, character, or token. If they are arithmetic but have different types or sizes, the value of the second operand is temporarily converted to agree with the first operand before the comparison occurs. (Overflow faults never occur. If an overflow condition is encountered, the first operand is taken as being less than the second.)

Comparisons between arithmetic values of dissimilar attributes are consistent with the rules of PL/I.

Character strings/fields are compared based on the collating sequence of characters (EBCDIC representation). Token strings/fields are compared by viewing them as positive hexadecimal numbers. Unequal-length strings or fields are padded as described in the EQUAL-BRANCH-FALSE instruction.

The first operand may be an array or domain, or both operands may be arrays or domains, in which case an element-by-element comparison is done. The result is true only if the relation holds between all corresponding elements.

Faults: General set plus invalid transfer, yes-branch

trace, and no-branch trace.

Instruction: GREATER-THAN-BRANCH-FALSE (GTBF)
Function: If the value of the first operand is greater than the value of the second operand, the instruction has no effect; otherwise, control is transferred to the specified instruction address.
Format: 9,OA,OA,IA
Operands: See LESS-THAN-BRANCH-FALSE instruction.
Faults: See LESS-THAN-BRANCH-FALSE instruction.

Instruction: LESS-THAN-OR-EQUAL-BRANCH-FALSE (LEBF)
Function: If the value of the first operand is less than or equal to the value of the second operand, the instruction has no effect; otherwise, control is transferred to the specified instruction address.
Format: A,OA,OA,IA
Operands: See LESS-THAN-BRANCH-FALSE instruction.
Faults: See LESS-THAN-BRANCH-FALSE instruction.

Instruction: GREATER-THAN-OR-EQUAL-BRANCH-FALSE (GEBF)
Function: If the value of the first operand is greater than or equal to the value of the second operand, the instruction has no effect; otherwise control is transferred to the specified instruction address.
Format: B,OA,OA,IA
Operands: See LESS-THAN-BRANCH-FALSE instruction.
Faults: See LESS-THAN-BRANCH-FALSE instruction.

Instruction: DEFINED-BRANCH-FALSE (DEFBF)
Function: If the value of the operand is defined, the instruction has no effect; otherwise, control is transferred to the specified instruction address.
Format: 004,OA,IA
Operands: The operand can be of any type. If the operand is a token field, the instruction has no effect. If the operand specifies a collection of data (array or structure), the condition is true only if every element has a defined value. If the operand is a character or boolean field, the condition is true only if every element in the field is defined.
Faults: General set (excluding incompatible operands and invalid operand type) plus invalid transfer, yes-branch trace, and no-branch trace. The undefined-operand fault will

not occur unless the values of other cells are needed to address the operand (e.g., parameter, pointer, array subscript) and one of these cells has the undefined value.

Instruction: ITERATE
Function: An addition is performed between two operands and relationships among three operands are tested. If true, control is transferred to the specified instruction; otherwise, control is transferred to the next instruction.
Format: 5,OA,OA,OA,IA
Operands: All three operands must be arithmetic and cannot be entire arrays or domains. The instruction first performs the operation OP1=OP1+OP3 following the semantics of the ADD instruction. Then a branch is taken if either or both or the following expressions are true

(OP1 > OP2) & (OP3 >= 0)
(OP1 < OP2) & (OP3 < 0)

The comparison between OP1 and OP2 follows the semantics of the LTBF instruction.
Faults: General set plus invalid transfer.
Notes: ITERATE is intended to be used at the bottom of iterative loops.

BOOLEAN INSTRUCTIONS

Instruction: AND
Function: The values of the two operands are "anded" and the result is placed in the first operand.
Format: 05,OA,OA
Operands: The operands must both be equal-length boolean strings or fields. The first operand may be an array or domain, or both operands may be arrays or domains.
Faults: General set.

Instruction: OR
Function: The values of the two operands are "or-ed" and the result is placed in the first operand.
Format: 06,OA,OA
Operands: See AND instruction.
Faults: General set.

Instruction: NOT
Function: The value of the boolean operand is inverted.
Format: 005,0A
Operands: The operand must be a boolean string or field, or an array or domain of boolean strings or fields.
Faults: General set (excluding incompatible operands).

STRING AND SEARCH INSTRUCTIONS

Instruction: CONCATENATE (CONCAT)
Function: The second operand is concatenated to the first operand.
Format: 03,0A,0A
Operands: The first operand must be a string. The second operand must be a string or field of the same type. The length of the first operand is incremented by the length of the second operand, and the value of the second operand is appended to the end of the first operand.
Faults: General set plus overflow.

Instruction: MOVE-SUBSTRING (MOVESS)
Function: The substring (part of a string or field) designated by the second set of operands is moved into the substring designated by the first set of operands.
Format: 04,0A,0A,0A,0A,0A
Operands: Operands 2 and 4 designate the two strings or fields. The two operands must be compatible (both character, boolean, or token).
Operands 1, 3, and 5 must be integers. Operand 1 specifies the length of the substring to be moved. Operand 3 specifies the index of the start of the substring in the target string/field, and operand 5 specifies the index of the start of the substring in the source string/field.
Faults: General set.
Notes: MOVESS performs an overlay rather than an insertion. That is, the length of the target string is unchanged.

Instruction: INDEX
Function: A string or field is searched for a specified substring, starting at a specified position. If the substring is found, the first operand contains the index of the start of the matching substring in the string/field. If the substring is not found, the first operand is set to zero.
Format: 07,0A,0A,0A
Operands: The first operand must be an integer; it initially specifies the index of the point in the string at which the search should begin. The second operand is the string or field to be searched. The third operand must be a string or field having the same type as the second operand; it represents the substring to be located.
Faults: General set.

Instruction: LENGTH
Function: The length of the second operand (a string or field) is placed in the first operand.
Format: 08,0A,0A
Operands: The first operand must be an integer and the second operand must be a string or field.
Faults: General set (excluding incompatible operands and undefined operand).

Instruction: SEARCH
Function: The instruction specifies an array or domain cell, a subscript value, and a key value. Each element of the array or domain is searched for the key value, starting with the element specified in the subscript operand. If an element is found, the subscript operand is set equal to the element's subscript. If not, the subscript operand is set to zero.
Format: 003,0A,0A,0A
Operands: The first operand must be an integer; it initially contains the starting element number and is filled with the matching element number (or zero). The second operand must be a one-dimension array or domain (the operand address must be an array or domain address). The third operand is the key (the value to be searched for). The elements of the array are compared to the key value according to the rules specified in the EQBF instruction.
Faults: General set.

CONTROL INSTRUCTIONS

Instruction: CALL
Function: Execution of the current module is suspended and execution of another module begins at the specified entry point. Allocation and initialization of automatic storage is performed for the called module.
Format: D,OA,X,A1,OA1,...,Ax,OAx
Operands: The first operand is a pointer to a module entry point (must have read access). The pointer must have been created by a CREATE-ENTRY-POINTER instruction. The first immediate field is a two-token hexadecimal number (X) specifying the number of arguments to be passed. The subsequent X pairs of fields specify the arguments. Ai is a one-token immediate field indicating whether the argument is passed with read/write access (value=0011) or read-only access (value=0111). OAi is the operand address of the argument. Arguments cannot be literals or domains. Arguments cannot be relocatable cells where the locator is a pointer or parameter pointer.
Faults: General set plus call-trace, invalid transfer, and insufficient storage.

If CALL is executed in a fault handler, any and all activations beneath (after) the current activation disappear.
Notes: The CALL instruction does not actually transfer arguments to the corresponding parameters in the called module. This must be done via an ACTIVATE instruction at the called entry point.
CALL creates an activation record and places it on the top of the stack of the activation records for the program.

Instruction: ACTIVATE (ACT)
Function: Argument/parameter compatibility is checked and the specified parameters are initialized.
Format: C,X,CA1,...,CAx
Operands: The immediate field (X) is a two-token hexadecimal number specifying the number of parameters. The following X fields must be cell addresses of parameter cells. The parameters are initialized to the arguments transmitted by the last CALL instruction executed in the program.
Faults: Addressing, unknown data format, invalid operand type, invalid transmission count, incompatible operands.

Notes: The rules for argument-parameter compatibility are defined in Table 7.2. An ACTIVATE instruction need not be the first instruction at an entry point to a module, but an ACTIVATE instruction must be executed before any reference is made to a parameter cell. (If not, the parameter would have the undefined value.)

If the parameter is	Then the argument must be
D-typed	Any primitive cell
Integer, pointer	Identical type
Fixed-point, floating-point, boolean, character, or token field/string	Identical type, and identical size unless the parameter is D-sized.
Structure	A structure having the same number elements. The type and size of each element must be identical to the type and size of each element in the parameter structure. Arrays in the structure must have the same number of dimensions as the arrays in the parameter structure. They must also have the same upper-bound values and element attributes, unless the arrays in the parameter structure are D-typed, D-sized, or D-bounded.
Array	Array of identical dimensions. Unless the parameter array is D-bounded, the bounds must be equal. Unless the parameter array is D-typed, the element type must be identical. Unless the parameter array is D-sized, the element size must be identical.

Table 7.2 Rules for argument-parameter compatibility.

```
**************************************************
* Note: In those situations in Table 7.2 where the       *
* parameter is D-typed, D-sized, or D-bounded, ACT       *
* adjusts the proper fields in the parameter cell in the *
* activation record to match those of the argument.      *
* Hence, there is no need, during execution of the module,*
* for the machine to refer to the tag of an argument.  It *
* need only reference its content component, which is    *
* located through the parameter cell.                    *
**************************************************
```

Instruction: RETURN
Function: Execution of the current module is terminated and execution is resumed after the CALL instruction that called this module.
Format: 0A
Operands: None.
Faults: None.

Notes: RETURN "undoes" the effect of the previous CALL and ACTIVATE instructions. That is, the current activation record is destroyed. If the current activation record is the only one, the program is terminated.

Instruction: LOCAL-CALL(LCALL)
Function: Execution is suspended and control is transferred to an instruction within the module.
Format: 0B,IA,X,A1,0A1,...,Ax,0Ax
Operands: The first address field specifies an instruction address to which control is transferred. The remaining fields are identical to those of the CALL instruction. Arguments cannot be literals, domains, or relocatable cells.
Faults: General set plus invalid transfer, call trace, and insufficient storage.
Notes: LCALL, unlike CALL, does not create an activation record, which means that internal procedures cannot be recursive (unless the compiler uses an ALLOCATE instruction to simulate the effect of an activation record), and that any scope-of-name rules are the compilers' responsibility.

Instruction: LOCAL-ACTIVATE (LACT)
Function: Argument/parameter compatibility is checked and the specified parameters are initialized.
Format: 0C,X,CA1,...,CAx
Operands: The immediate field (X) is a two-token hexadecimal number specifying the number of parameters. The following X fields must be the cell addresses of parameter cells. The parameters are initialized to the arguments transmitted by the last LCALL instruction executed in the program, or by the machine in the case of a LACT beginning a fault handler.
Faults: Addressing, unknown data format, invalid operand type, invalid transmission count, incompatible operands.
Notes: The rules for argument-parameter compatibility are the same as those for the ACTIVATE instruction.

Instruction: LOCAL-RETURN (LRETURN)
Function: Execution is transferred to the instruction following the last LCALL instruction executed in the current module.
Format: 0D
Operands: None.
Faults: Invalid transfer (if there was no previous LCALL instruction).
Notes: If LRETURN is executed in a fault-handler, and if there is no outstanding LCALL instruction in this module, execution of the fault-handler is terminated and execution begins at the faulting instruction.

Instruction: BRANCH (B)
Function: Control is transferred to the designated instruction address.
Format: E,IA
Faults: Invalid transfer.

Instruction: GUARD
Function: If the current module is not in the guarded state, it enters the guarded state and control is transferred to the next instruction. If the current module is in the guarded state, program execution is suspended until it leaves the guarded state.
Format: 000C
Faults: None.
Notes: If a program executes a GUARD instruction after the same program has placed the module in a guarded state, the GUARD instruction has no effect.
The only exit from the guarded state is by the execution of an UNGUARD instruction. Executing a RETURN instruction, or an abnormal termination of a module activation (e.g., as a result of a return of a higher fault handler) does not affect the guarded state of a module.

Instruction: UNGUARD
Function: The state of the current module is set to unguarded.
Format: 000D
Faults: None.

ADDRESSING INSTRUCTIONS

Instruction: COMPUTE-POINTER (CPTR)
Function: The first operand is assigned the logical address of the second operand.
Format: 0E,0A,0A
Operands: The first operand must be a pointer. The second operand may be any operand except a parameter, literal, entire domain, or a relocatable or domain based on a nonpointer parameter. The access code in the pointer is set to copy and no-destroy; its read/write access is set to the class of access that the module currently has to the second operand.

The logical address is the address of the cell represented by the second operand. If the second operand is a relocatable cell, the address of the cell addressed by the relocatable cell is computed and assigned to the first operand. If the second operand is a relocatable cell, its associated pointer must have copy authority.

If the second operand is a structure, the pointer points to the first element of the structure.
Faults: Addressing, unknown data format, invalid pointer, invalid operand type, and protection.

```
**************************************************************
* Note: Although activation records, being system objects,   *
* have system object names, there is no need to assign       *
* every activation record an object name, since the only     *
* time it would be used is when a CPTR instruction refers    *
* to a cell in the automatic storage die. Hence, for         *
* reasons of performance, activation records should not be   *
* automatically assigned object names during creation.       *
* Rather, CPTR should check to see if the activation         *
* record has a name; if it doesn't, a name should be         *
* assigned at this point.                                    *
**************************************************************
```

Instruction: COMPUTE-INDIRECT-POINTER (CIPTR)
Function: The first operand becomes an indirect pointer to the second operand.
Format: 0006,0A,0A
Operands: Both must be pointers. Operand 1 becomes an indirect pointer to operand 2 and is assigned the access code in operand 2. Operand 2 must have copy authority and cannot be an indirect pointer. Operand 2 cannot be a parameter pointer, or a relocatable or

domain-element pointer that is based on a nonpointer parameter.
Faults: General set (excluding incompatible operands).

Instruction: CHANGE-ACCESS (CACC)
Function: The access code (authority) in the operand is restricted (changed) to the value specified.
Format: 006,X,0A
Operands: The operand must be a pointer. The immediate field (X) is a single token. The immediate field is ORed into the access code of the pointer, thus further restricting the authority of the pointer.
Faults: General set (excluding incompatible operands).

Instruction: ALLOCATE (ALLOC)
Function: A data-storage object is created, containing allocated storage for the operand.
Format: 0F,X,0A
Operands: X is a one-token immediate field. The last bit designates whether the storage area should be automatically destroyed upon program termination. The value xxx0 indicates yes; xxx1 indicates no.

The third bit designates whether the storage should be initialized to the undefined value. xx0x specifies initialization to undefined; xx1x specifies no initialization. (If no initialization is requested and the storage is to contain pointer cells, the request is overriden and the pointers are given the undefined value.)

The operand address must be a cell address, array element address, or array address. The operand must be relocatable and can describe any type of cell, except a domain. The locator cell referenced by the relocatable cell must be a pointer or parameter pointer, and RCA in the relocatable cell must be 0. The relocatable cell cannot be D-sized or D-typed. Any associated relocatable cells (if the relocatable-cell operand is a structure, cannot be D-sized, D-typed, or D-bounded.

If the operand is an array and it is addressed via an array address, the upper-bound fields in the operand are used to determine the size of the allocated array. If the array is addressed via an array-element address, the upper-bound fields in the array tag must be zero

(i.e., it must be D-bounded; in this case the values of the subscripts are taken to represent the upper bounds desired in the allocated array.

The allocated storage is constructed identical to the storage described by the nested tag (i.e., it includes tags). The access code in the pointer is set to 0000 (read/write/destroy/copy) and its logical address refers to the data-storage object. If the storage is not explicitly destroyed, it is destroyed when the program ends (if X=xxx0).

Faults: General set (excluding incompatible operands) plus insufficient storage.

```
**************************************************
* Given the nature of the architecture, it is recommended *
* that programs specify "initialize to undefined" when    *
* executing ALLOCATE.  The option was provided primarily  *
* for compilers, when it is known that the ALLOCATE will  *
* be followed by a programmed initialization.             *
**************************************************
```

Instruction: DESTROY

Function: The storage object specified by the operand is destroyed.

Format: 007,0A

Operands: The operand must be a relocatable, pointer, or parameter pointer cell. If it is relocatable, the locator cell referenced by the relocatable cell must be a pointer or parameter pointer and the RCA field in the relocatable cell must be zero. The object referenced by the pointer is destroyed. If the operand is a pointer, the object referenced by it is destroyed.
In both cases the pointer must have destroy access. The pointer is given the undefined value at the end of the instruction.

If a DESTROY instruction is applied to a module and that module is still active (i.e., activation records still exist), the system-object name of the module object is immediately destroyed (meaning that it can no longer be referenced, such as in a CALL instruction), but the object is not actually destroyed by the system until all of its activations cease to exist. If a port is destroyed on which pending send and/or receive requests exist, the port is destroyed and the pending SEND or RECEIVE instructions terminate with the invalid-pointer fault.

Faults: General set (excluding incompatible operands).
Notes: The pointer must name an entire storage object. For example attempting to destroy a cell within an activation record or a cell <u>within</u> a dynamically allocated storage area would result in a protection fault.

Instruction: CHANGE-LOGICAL-ADDRESS
Function: An object is given a new logical address by the system. Its current logical address is destroyed.
Format: 000A,0A
Operands: The operand must be a relocatable or pointer cell. If it is relocatable, the locator cell referenced by the relocatable cell must be a pointer and the offset field in the relocatable cell must be zero. The pointer must have read/write/destroy/copy authority and must refer to an entire object. The object referenced by the pointer is assigned a new logical address, which is placed in the pointer with read/write/destroy/copy access. Any attempt after this point to use the prior logical address of the object will result in an invalid-pointer fault.
Faults: General set (excluding incompatible operands).

Instruction: LOAD-MODULE (LMODULE)
Function: A module object is created.
Format: 009,X,0A,0A
Operands: X, a one-token immediate field, indicates whether the module object should be automatically destroyed upon program termination. The value xxx0 indicates yes. If it is to be automatically destroyed, the same qualification about destroying a module as described under the DESTROY instruction applies here.

The next operand must be a pointer and the third operand must be a token string or field. Its value must have the form of an external module (see Figure 3.1). The machine checks the validity of the format of the module, copies it into internal storage, and creates a pointer to it. The pointer is assigned read/write/destroy/copy access (but writing into a module, except with the LINK instruction, is prohibited by the machine). All pointer and parameter cells in the module are set to the undefined value. Space is allocated within the module object

for arrays within the static storage die.

Faults: General set (excluding incompatible operands) and invalid module and insufficient storage. If the invalid-module fault occurs, argument 5 transmitted to the fault handler indicates the type of problem. Its value may be:

00001 - Error in indexes in module header
00002 - Unsupported CAS, IAS, or SIS value
00003 - Invalid cell or cell relationship in the module
00004 - An instruction's operand address does not refer to the beginning of a cell

Instruction: COMPUTE-ENTRY-POINTER (CEP)
Function: A pointer value (logical address) is computed for a specified instruction in a specified module.
Format: 00F,0A,0A,0A
Operands: The first operand is the target pointer. The second operand is a pointer to a module object. The third operand is a token field of size 5 that specifies the instruction address of an instruction in the module. The logical address of the instruction in the module object is stored in the first operand. The access code is set to read/copy.
Faults: General set (excluding incompatible operands). The addressing fault will occur if the instruction address does not point within the instruction space. The protection fault will occur if the second operand does not have read/copy authority.

Instruction: LINK
Function: A pointer value is assigned to a specified pointer cell in a loaded module.
Format: 00A,0A,0A,0A
Operands: The first operand is a pointer to a loaded module. The pointer must have write access. The second operand is a token field of size 5 which specifies a cell address in the loaded module. The third operand is a pointer. The value of this pointer is assigned to the pointer cell specified by the first and second operands.
Faults: General set (excluding incompatible operands). The addressing fault will occur if the target cell is not in the address space or not a pointer.

Instruction: CREATE PORT
Function: A port storage object is created.
Format: 0007,X,0A
Operands: X, a one-token immediate field, indicates whether the port object should be automatically destroyed upon program termination. The value xxx0 indicates yes.

The operand must be a pointer cell. The logical address of the port is placed in the pointer cell, with read/write/destroy/copy access.
Faults: General set (excluding incompatible operands) plus insufficient storage.

Instruction: SEND
Function: The specified operands (arguments) are transmitted through a port to another program. Execution of the instruction does not complete until another program receives (via a RECEIVE instruction) the arguments from the port.
Format: 00B,0A,X,0A1,...,0Ax
Operands: The first operand is a pointer to a port (must have write access). X is a two-token immediate field specifying the number of arguments to be passed (0-255). The subsequent X operand addresses specify the arguments. Arguments cannot be literals, relocatables, or domains.
Faults: General set plus invalid transmission count. If the number of arguments is unequal to the number of receiver operands in the corresponding RECEIVE instruction, the invalid-transmission-count fault occurs. If the arguments are incompatible with the types of receiver operands in the corresponding RECEIVE instruction, the incompatible-operands fault occurs. If an argument has an undefined value, the undefined-operand fault occurs.

Any faults that occur after data movement starts (protection, if a pointer argument does not have copy authority, incompatible operands, undefined operand) cause the SEND and corresponding RECEIVE instruction to complete with partial data movement. These faults, as well as the invalid-transmission count fault, cause both the SEND and corresponding RECEIVE instruction to fault.

Instruction: RECEIVE
Function: The values of the first set of arguments (a set of arguments consists of those named in a single SEND instruction) in the specified port are transmitted to the specified operands and removed from the port. If the port does not

contain a set of arguments, execution of the instruction does not complete until a set of arguments has been placed in the port.

Format: 00C,OA,X,OA1,...,OAx

Operands: The first operand is a pointer to a port (must have read access). X is a two-token immediate field specifying the number of receiver operands. The X operand addresses specify the receiver operands. They may be of any type except literal, relocatables or domains. (They need not be parameters, since the arguments in the port are transmitted by value. If a receiver operand is a parameter, the value is transmitted to the associated argument, an argument transmitted to this module by a CALL or LCALL instruction.)

The rules concerning compatibility between SEND arguments and receiver operands are identical to those for the ACTIVATE instruction (i.e., the attributes of the SEND arguments and corresponding receiver operands must be identical).

Faults: General set plus invalid transmission count. If one or more of the arguments have the undefined value, if the number of SEND arguments is unequal to the number of receiver operands, if one or more SEND arguments are incompatible with the corresponding receiver arguments, or if a pointer SEND argument does not have copy authority, the undefined-operand, invalid-transmission-count, incompatible-operands, or protection fault is generated in both the RECEIVE instruction and the corresponding SEND instruction (see description of the SEND instruction).

Instruction: DESCRIBE

Function: Given a pointer, returns information about the pointer and that to which it points.

Format: 000B,OA,OA,OA

Operands: The first operand must be an integer. The second operand must be a one-dimensional array of character fields of size 6. (The operand address must be an array address.) The array must contain at least four elements. The third operand is the pointer.

The instruction stores information about the pointer and its referenced object in operands 1 and 2. Table 7.3 defines the information. OA2(i) represents the ith element of the third operand.

If pointer refers to: Then resultant information is:

	OA1	OA2(1)	OA2(2)	OA2(3)	OA2(4)
Module	1	6	7	8	12
Port	2	6	7	9	11
Data-storage object	1	6	7	10	11
Cell in module (SSD)	3	6	7	11	12
Cell in act. record	3	6	7	11	12
Cell in DSO	3	6	7	11	11
Entry point in module	4	6	7	11	12
Source/sink stream	5	6	7	11	11

Notes:

1 - size of the object (number of tokens)
2 - number of programs currently enqueued on the port
3 - cell type (value = first four bits of tag)
4 - instruction address of entry point
5 - unchanged
6 - authority possessed by the pointer
    char 1 = blank or R (read)
    char 2 = blank or W (write)
    char 3 = blank or D (destroy)
    char 4 = blank or C (copy)
    char 5 = type of pointer - blank (direct) or
             I (indirect)
    char 6 = blank
7 - type of object referenced by the pointer
    char 1-2 = MO (module object)
               PO (port object)
               DO (data storage object)
               MC (cell in module - SSD)
               AC (cell in activation record)
               DC (cell in data storage object)
               ME (entry point in module)
               SS (source/sink stream)
    chars 3-6 = blank
8 - module status
    char 1 = blank or P (to be freed upon program
             termination)
    char 2 = blank or A (module is active)
    char 3 = blank or G (module is in guarded state)
    char 4 = blank or C (call trace is active)
    char 5 = blank or Y (yes-branch trace is active)
    char 6 = blank or N (no-branch trace is active)
9 - port status
    char 1 = blank or P (to be freed upon program
             termination)
    char 2 = blank or S (send outstanding) or R
             (receive outstanding)
    chars 3-6 = blank
10- status
    char 1 = blank or P (to be freed upon program
             termination)

char 2-6 = blank
11- blank
12- first six characters of the name of the module, or associated module, object

Table 7.3 Result from the DESCRIBE instruction.

DEBUGGING INSTRUCTIONS

Instruction: ENABLE
Function: The specified token field is ORed into the fault-code field as defined in the module header. The fault-code field is maintained in the module's activation record, meaning that this instruction affects only the current activation.
Format: 0008,0A
Operands: The operand must be a token field (of size N) whose size is equal to or less than the length of the fault-code field. If the token field is shorter than the fault-code field, only the first N tokens of the fault-code field are changed.
Faults: General set (excluding incompatible operands) and overflow.
Note: The ENABLE and DISABLE instructions do not alter the fault-code field in the module; they affect only the current activation of the module.

Instruction: DISABLE
Function: The inverse (negation) of the specified token field is ANDed into the module's fault-code field in the activation record.
Format: 0009,0A
Operands: See ENABLE instruction.
Faults: General set (excluding incompatible operands) and overflow.

Instruction: RAISE-FAULT
Function: A fault occurs. The two-token immediate field X becomes the fault type (i.e., the value of the first argument to the fault handler).
Format: 00D,X
Faults: Whatever type is indicated by the immediate

field X. X should not be zero or 22-27; if it is, the fault-handling fault occurs. If X does not specify the value of an architected fault type, the fault is program-defined.
For program-defined faults (28-255), even-valued ones allow the fault handler to resume execution after the RAISE-FAULT instruction, while odd-numbered ones do not.

Instruction: CONTINUE (CONT)
Function: Execution of the fault-handler is terminated and execution resumes at the instruction that would have been executed after the faulting instruction, had the faulting instruction not faulted.
Format: 00E
Faults: Fault-handling (if there is no current fault, if continuing beyond the current fault is not permitted, or if CONTINUE is issued from a local subroutine called by a fault handler).
Notes: If a fault-handler wishes to resume execution at the faulting instruction, it should issue the LRETURN instruction. If the fault-handler wishes to resume execution at the instruction following the faulting instruction, it should issue the CONTINUE instruction. The only faults that may be followed by a CONTINUE instruction are call trace, yes-branch trace, no-branch trace, or an even-numbered fault in the range range 28-254 generated by a RAISE-FAULT instruction.

Instruction: TRANSFER-FAULT (TRFAULT)
Function: The current fault-handler is terminated and a higher fault-handler (one lower in the activation stack) is called. If an applicable fault-handler cannot be found, the program is terminated.
Format: 0005
Faults: Fault-handling (same first and third situations as in the CONTINUE instruction).
Notes: TRFAULT is used by a fault-handler that has a particular fault enabled, but after receiving such a fault it decides to send it to a "higher authority."

Instruction: DISPLAY-TAG (DTAG)
Function: The tag of the designated cell is assigned to a token operand.
Format: 0001,OA,OA,OA
Operands: The second operand is a pointer to a loaded module; the pointer must have read access.
The third operand is a token field of size 5 which specifies a cell address in the loaded module. The tag of this cell is moved into the first operand, which must be a token field or string.
The overflow fault is suppressed; if the tag is longer than the first operand, the first operand is filled with the leftmost tokens of the tag.
If the pointer is undefined, it is assumed to designate the current module (i.e., an undefined-operand fault will not occur for the second operand).
Faults: General set (excluding incompatible operands).
Notes: This instruction is intended only for use by debugging functions. For planning purposes, the largest possible tag is 84 tokens (a relocatable array of 15 dimensions).

Instruction: DISPLAY-CONTENTS (DCON)
Function: The contents component of the designated cell is assigned to a token operand.
Format: 0002,OA,OA,OA
Operands: See DTAG instruction. Overflow faults are similarly suppressed. If the cell is in the automatic storage die, its value for the most recent, currently active, activation of the module is displayed. If the module is not active, the cell's initial value in the die in the module is displayed.
Faults: General set (excluding incompatible operands).
Notes: This instruction returns the contents of a cell, which is not always identical to its value. For example the contents of a character string is a three-token length field and a variable-size value; the contents of a pointer is a one-token access code and a 20-token logical address.

If the cell is an array, the element contents are returned as a contiguous stream of tokens. They are returned in "row-major" order (all the elements in the first dimension, then the second, and so on).

The size of a contents component can be

determined by first using a DTAG instruction.

Instruction: TRACE
Function: Designated traces is enabled for a specified module.
Format: 0003,X,0A
Operands: The one-token immediate field (X) specifies the type of trace. The value 01xx specifies a a yes-branch trace, 0x1x specifies a no-branch trace, and 0xx1 specifies a call trace. The second operand must be a pointer to a module and must have write access. The specified traces are enabled for all subsequent activations of the module.
Faults: General set (excluding undefined operand and incompatible operands).
Notes: TRACE and NOTRACE do not affect any existing activations of the specified modules. They take effect when such modules are subsequently called.

Instruction: NOTRACE
Function: Designated traces are disabled for a specified module.
Format: 0004,X,0A
Operands: See TRACE instruction. If a trace was not previously enabled in a module, disabling it has no effect.
Faults: General set (excluding undefined operand and incompatible operands).

8. OBJECT-CODE EXAMPLES

Figures 8.1 and 8.2 are PL/I procedures that will be used as examples. The intent of the examples is to illustrate how a PL/I program would be represented in this architecture.

Figures 8.3 and 8.4 represent the object modules that the compiler would present to the machine. Rather than illustrating the modules as a continuous token string, items of interest (e.g., individual cells and instructions) are illustrated on separate lines. The first and second columns are not part of the module; they indicate, respectively, the index in the module of the first token on the line, and the index in the address space or instruction space of the first token on the line. Each line is also supplemented with a comment. The comments on the instructions take the form of an assembly language. The meaning of the assembly-language statements should be obvious. For instance

```
MOVE  A.B(J),1
```

means move the literal 1 into the Jth element of array B in the structure A. Names beginning with "%" are instruction labels (targets of branch instructions).

```
MATCHES: PROCEDURE(BODY,UNRESNAME,MATCHCODE,SIZE);
DECLARE 1 BODY(*),
            2 NAME CHAR(8),
            2 TYPE CHAR(2),
            2 ADDRESS POINTER;
DECLARE
       MODULE  CHAR(2) STATIC INIT('MD'),
       ENTRYPT CHAR(2) STATIC INIT('EP'),
       EXTREF  CHAR(2) STATIC INIT('ER');
DECLARE NULL BUILTIN;
DECLARE MATCHCODE FIXED DECIMAL(1);
DECLARE UNRESNAME CHAR(8);
DECLARE SIZE FIXED DECIMAL(4);
DECLARE
       I FIXED BINARY(15);
       J FIXED BINARY(15);
MATCHCODE=2;
IF((SIZE>0)&(SIZE¬>2000))
THEN
 DO;
   MATCHCODE=0;
   DO I=1 TO SIZE WHILE(MATCHCODE=0);
     IF(BODY(I).ADDRESS=NULL)
       THEN DO;
              MATCHCODE=1;
              DO J=1 TO SIZE WHILE(MATCHCODE=1);
                    IF((BODY(I).NAME=BODY(J).NAME)&
                       ((BODY(J).TYPE=MODULE)|
                        (BODY(J).TYPE=ENTRYPT)))
                      THEN DO;
                             MATCHCODE=0;
                             BODY(I).ADDRESS=BODY(J).ADDRESS;
                           END;
                      ELSE;
              END;
              IF(MATCHCODE=1) THEN UNRESNAME=BODY(I).NAME;
                              ELSE;
            END;
       ELSE;
   END;
 END;
ELSE;
END;
```

Figure 8.1. Source Module MATCHES

```
Offsets                              Comments

001       000240002600 0A0000C000173  Header
01A       2200000000                   CAS/IAS/SIS/SA/Faults
024       00                          Module name (omitted)
026   01  671000000000080 3001C       BODY (parameter array of
          FFFFFFF                           structures)
      1C  100010000B008               NAME (domain character field)
      29  100010010B002               TYPE (domain character field)
      36  1000100149                  ADDRESS (domain pointer)
      40  6E100000FFFFFFF             MATCHCODE (fixed-pt. param.)
      4F  6B008000FFFFFFF             UNRESNAME (param. character)
      5E  6E400000FFFFFFF             SIZE      (fixed-pt. param.)
      6D  F800000                     I
      74  F800000                     J
0A0   7B  B002D4C4                    MODULE (character field)
      83  B002C5D7                    ENTRYPT
      8B  B002C5D9                    EXTREF
      93  E4002000                    '2000'

0c0   01  C04014F405E       ACT     4,BODY,UNRESNAME,MATCHCODE,SIZE
          140002            MOVE    MATCHCODE,2
          95E000B2          GTBF    SIZE,0,%H
          A5E93B2           LEBF    SIZE,2000,%H
          140000            MOVE    MATCHCODE,0
          16D001            MOVE    I,1
          A6D5EB2           LEBF    I,SIZE,%H
      34  740000B2    %A:   EQBF    MATCHCODE,0,%H
          004366D48         DEFBF   BODY.ADDRESS(I),%B
          EA8               B       %G
      48  140001      %B:   MOVE    MATCHCODE,1
          174001            MOVE    J,1
          A745E99           LEBF    J,SIZE,%F
      5B  74000199    %C:   EQBF    MATCHCODE,1,%F
          71C6D1C748F       EQBF    BODY.NAME(I),BODY.NAME(J),%E
          629747B80         NEBF    BODY.TYPE(J),MODULE,%D
          72974838F         EQBF    BODY.TYPE(J),ENTRYPT,%E
      80  140000      %D:   MOVE    MATCHCODE,0
          1366D3674         MOVE    BODY.ADDRESS(I),BODY.ADDRESS(J)
      8F  5745E0015B  %E:   ITERATE J,SIZE,1,%C
      99  740001A8    %F:   EQBF    MATCHCODE,1,%G
          14F1C6D           MOVE    UNRESNAME,BODY.NAME(I)
      A8  56D5E00134  %G:   ITERATE I,SIZE,1,%A
      B2  0A          %H:   RETURN
```

Figure 8.2. Object Module MATCHES

```
TESTEST: PROCEDURE OPTIONS(MAIN);
DECLARE SIZE FIXED DECIMAL(4);
DECLARE 1 B(7),
            2 N CHAR(8),
            2 T CHAR(2),
            2 A POINTER;
DECLARE UNNAME CHAR(8) INIT('XXXXXXXX'),
        CODE   FIXED DECIMAL(1) INIT(9);
DECLARE NULL BUILTIN;
B(1).N='ABCDEFGH';
B(1).T='ER';
B(1).A=NULL;
B(2).N='ABCDEFGH';
B(2).T='MD';
B(2).A=ADDR(UNNAME);
SIZE=2;
CALL MATCHES(B,UNNAME,CODE,SIZE);
END;
```

Figure 8.3. Source Module TESTEST

```
Offsets                              Comments

 01       .0002400026000840000BE0010E  Header
 1A        2200000000                   CAS/IAS/SIS/SA/FC
 24        00                          No module name used
 26   01   E40F0000                    SIZE
      09   7100290000078030022         B (array of structures)
           000000
      22   100090000B008               N (domain character field)
      2F   100090010B002               T (domain character field)
      3C   1000900149                  A (domain pointer)
      46   B008E7E7E7E7E7E7E7          UNNAME (character field)
      5A   E1009                       CODE
 84   5F   9F000000000000000           MATCHES (pointer)
           000000
      75   B008C1C2C3C4                'ABCDEFGH'
           C5C6C7C8
      89   B002C5D9                    'ER'
      91   B002D4C4                    'MD'
 BE   01   C00                  ACT    0
           12200175             MOVE   B.N(1),'ABCDEFGH'
           12F00189             MOVE   B.T(1),'ER'
           0013C001             UNDEF  B.A(1)
           12200275             MOVE   B.N(2),'ABCDEFGH'
           12F00191             MOVE   B.T(2),'MD'
           0E3C00246            CPTR   B.A(2),UNNAME
           109002               MOVE   SIZE,2
           D5F0430900F          CALL   MATCHES,4,R/W(B),R/W(UNNAME),
           34635A301                   R/W(CODE),R/W(SIZE)
           0A                   RETURN
```

Figure 8.4. Object Module TESTEST

CALCULATION OF THE ADDRESS-FIELD SIZE

The use of variable-size address fields places a burden on the compiler in the form of determining the appropriate size of the address field for the module being compiled. Of course a simple-minded compiler need not face up to this burden; it could simply use a fixed size address field that is large enough for the largest module that can be compiled, but such a solution does not exploit the advantages of variable-size addresses.

The address field size is a function of the size of the address space. The formula for calculating the smallest address field is

N = CEIL(log(1 + address space size - size of last cell))

where CEIL rounds a number to the next-higher integer. All logarithms are base 16.

The other type of variable-size address is the instruction address. The formula for calculating the smallest instruction address needed is:

M = CEIL(log(B + MI))

where

B - number of tokens in the instruction space excluding all instruction address fields and excluding the last instruction that is the target of a branch or LCALL, and all subsequent instructions

I - number of instruction address fields

Since M appears on both sides of the equation, it can be solved by substituting the values 2, 3, ... for M until both sides are equal.

Compiler Considerations

In producing a compiler for this architecture the following approaches are available:

1. Use fixed large values for N and M. This is the simplest approach but it does not take advantage of the use of shorter addresses.
2. Use the formulae for N and M to find the optimal sizes. This approach takes full advantage of the encoding but it complicates the compilers.
3. Rather than using the formulae, use a few simple heuristics to guess at the optimal N and M. If, during code generation, the compiler finds that N or M is too small, increment it by one and begin the code generation again.
4. Choose constant values for N and M. For instance

N=4 seems to be a reasonable upper bound, for it defines an address space of a maximum of 65535 tokens (which seems even more reasonable considering the fact that space for array elements does not appear in the address space). A separate optimization or "module-compression" program can then be written that is compiler and language independent. Its function is to take a module with a possibly over-sized address field and produce an equivalent module with a minimal address field.

9. THE ONE-LEVEL STORE

The machine has no I/O instructions; instead the architecture is based on the notion that "storage is storage is storage" and that "storage management is storage management is storage management." That is, why represent secondary storage with an interface that is different from that of main storage?

If one employs the one-level-store concept, then the architecture is seen to already have memory I/O (as distinct from source/sink I/O) with no changes to the definition of the architecture. That is, one can think of a file as a one-dimensional array of structures. Each array element corresponds to a record in the file. The nested tag in the array cell would likely be a structure, where the structure defines the fields in each record. Since the existing machine instructions are generic and apply to arrays and array elements, the "I/O instructions" are the existing instructions.

Storage areas are created with the ALLOCATE instruction. The immediate field in the instruction indicates whether the object is to be destroyed at program termination. To create a permanent "file," a program issues the ALLOCATE instruction, indicating with the immediate field that the created data-storage object should not be destroyed upon program termination. The ALLOCATE instruction points to a relocatable array which in turn points to a pointer cell. The logical address (capability) that is returned serves to uniquely identify the file until it is deleted (with a DESTROY instruction). The file is constructed by executing MOVE instructions to move data into the array elements.

Operating-system directory services will likely exist to allow programs to say such things as "associate the following logical address with the following symbolic name and remember the association," "given the following symbolic name, give me the associated logical address if I am so authorized," and "authorize the following user to do the above with this particular symbolic name."

Given the removal, at the architectural level, of the distinction between main-memory operations and secondary-storage I/O, a natural extension is to carry this notion into programming languages, that is, the removal of file I/O statements from programming languages.

One problem associated with a one-level store as described above that deserves more research is the mechanism with which a program searches a file (represented as an array) to locate a particular record. (array element). If

hash addressing can be used, representing files as arrays is natural. However, if hash addressing is inapplicable for a particular file, the only other alternative appears to be an iterative sequential search (unless the file is ordered by the search field, in which case a binary search could be used), which is unacceptable for large files. Hence the possibility of storing one or more indexes with array cells comes to mind. Another possibility is allowing designated arrays to be content-addressable. In short, the relationships between the concepts of one-level stores and data base processing need further investigation.

10. THE CONCEPTS OF A "PROGRAM" AND "I/O"

It should be apparent that the architecture contains nothing representing the concept of source/sink I/O (e.g., terminals, card readers, magnetic tapes). The intent is that a processor having this architecture be coupled to an external system (e.g., host system, intelligent I/O channel), that the external system perform such functions, and that the SEND/RECEIVE instructions serve as the I/O mechanism by communicating with the external system. The interface with the external world is not described here, as it is defined elsewhere. Also, it is not yet clear whether this interface will be architected or whether it will be left as "implementation dependent."

To summarize the SEND/RECEIVE mechanism when used for this purpose, when a program executes a SEND or RECEIVE instruction and the object being referenced is not a port or any other recognizable type of storage object, the information in the instruction is converted to an appropriate form and transmitted to the external system. If the logical address represents something meaningful to the external system (e.g., the name of a "source/sink stream"), it performs the designated I/O operation, using the SEND arguments or receiver operands. Currently, a SEND or RECEIVE instruction naming an I/O port can specify only a single operand, and its type must be a character or token field or string.

The SWARD architecture has been specified as a "single-program" architecture, although it does contain a few indications of multiple programs (e.g., the concept of a port). In particular, the architecture (purposely) contains no concept of interrupts nor any way to switch control among programs. The intent is that the concept of multiple programs (or processes), if needed, be created by the external system. The interface to the external system also contains provisions to allow it to support and control the execution of parallel processes on the SWARD machine. The basis of the mechanism is the provision for multiple stacks of activation records, each headed by an internal object called an activation-stack header, and signals to direct the SWARD machine to quickly switch from one stack (process) to another.

11. INSTRUCTION-FORMAT SUMMARY

SWARD INSTRUCTION SET SORTED BY OPERATION CODE

NAME	ABBREV.	FORMAT
MOVE	MOVE	1,OA,OA
ADD	ADD	2,OA,OA
SUBTRACT	SUB	3,OA,OA
MULTIPLY	MULT	4,OA,OA
ITERATE	ITERATE	5,OA,OA,OA,IA
NOT-EQUAL-BRANCH-FALSE	NEBF	6,OA,OA,IA
EQUAL-BRANCH-FALSE	EQBF	7,OA,OA,IA
LESS-THAN-BRANCH-FALSE	LTBF	8,OA,OA,IA
GREATER-THAN-BRANCH-FALSE	GTBF	9,OA,OA,IA
LESS-THAN-EQUAL-BRANCH-FALSE	LEBF	A,OA,OA,IA
GREATER-THAN-EQUAL-BRANCH-FALSE	GEBF	B,OA,OA,IA
ACTIVATE	ACT	C,X,CA1,...CAx
CALL	CALL	D,OA,X,A1,OA1,...Ax,OAx
BRANCH	B	E,IA
COMPLEMENT	COMP	F,OA
ABSOLUTE	ABS	01,OA
DIVIDE	DIVIDE	02,OA,OA
CONCATENATE	CONCAT	03,OA,OA
MOVE-SUBSTRING	MOVESS	04,OA,OA,OA,OA,OA,OA
AND	AND	05,OA,OA
OR	OR	06,OA,OA
INDEX	INDEX	07,OA,OA,OA
LENGTH	LENGTH	08,OA,OA
CONVERT	CONVERT	09,OA,OA
RETURN	RETURN	0A
LOCAL CALL	LCALL	0B,IA,X,A1,OA1,...Ax,OAx
LOCAL ACTIVATE	LACT	0C,X,CA1,...CAx
LOCAL RETURN	LRET	0D
COMPUTE POINTER	CPTR	0E,OA,OA
ALLOCATE	ALLOC	0F,X,OA
UNDEFINE	UNDEF	001,OA
REMAINDER	REMAIN	002,OA,OA
SEARCH	SEARCH	003,OA,OA,OA
DEFINED-BRANCH-FALSE	DEFBF	004,OA,IA
NOT	NOT	005,OA,OA
CHANGE ACCESS	CACC	006,X,OA
DESTROY	DESTROY	007,OA
POWER	POWER	008,OA,OA
LOAD MODULE	LMODULE	009,X,OA,OA
LINK	LINK	00A,OA,OA,OA
SEND	SEND	00B,OA,X,OA1,...OAx
RECEIVE	RECEIVE	00C,OA,X,OA1,...OAx
RAISE FAULT	RFAULT	00D,X
CONTINUE	CONT	00E
COMPUTE ENTRY POINTER	CEP	00F,OA,OA,OA
DISPLAY TAG	DTAG	0001,OA,OA,OA
DISPLAY CONTENTS	DCON	0002,OA,OA,OA
TRACE	TRACE	0003,X,OA

NOTRACE	NOTRACE 0004,X,OA
TRANSFER FAULT	TRFAULT 0005
COMPUTE INDIRECT POINTER	CIPTR 0006,OA,OA
CREATE PORT	CPORT 0007,X,OA
ENABLE	ENABLE 0008,OA
DISABLE	DISABLE 0009,OA
CHANGE LOGICAL ADDRESS	CLA 000A,OA
DESCRIBE	DESC 000B,OA,OA,OA
GUARD	GUARD 000C
UNGUARD	UNGUARD 000D