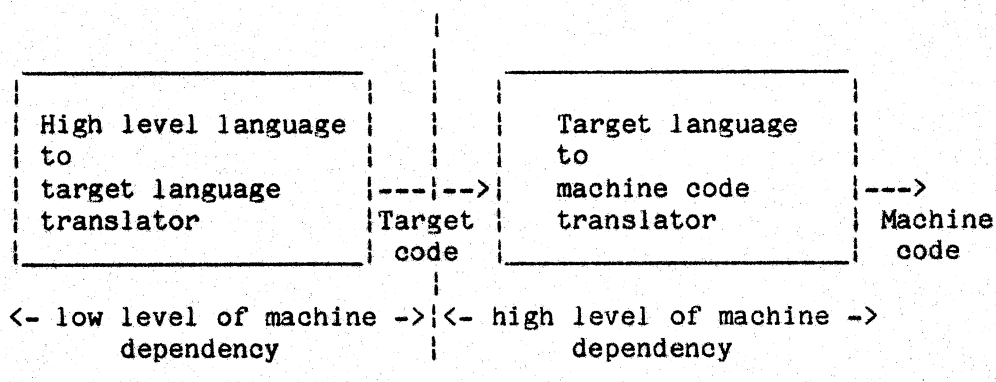
JOHN TAYLOR

CONTENTS.

1.1 THE COMPILER TARGET LANGUAGE

The tasks performed by a compiler in translating a high level language program to some object form may be divided into two parts. Some tasks, such as lexical and syntax processing, are largely independent of the machine on which the compiler is running; while other tasks, such as code generation, are very much more dependent on the type of machine that the program is being compiled for. The concept of a compiler target language is to present an abstract machine model to which the compiler targets its code, thereby increasing the proportion of the compiler which is almost machine independent. Thus a compiler may be considered in the two parts shown below.



The compilers which run under the MUSS system are organised in this way and the target language is the Manchester University Target Language (MUTL).

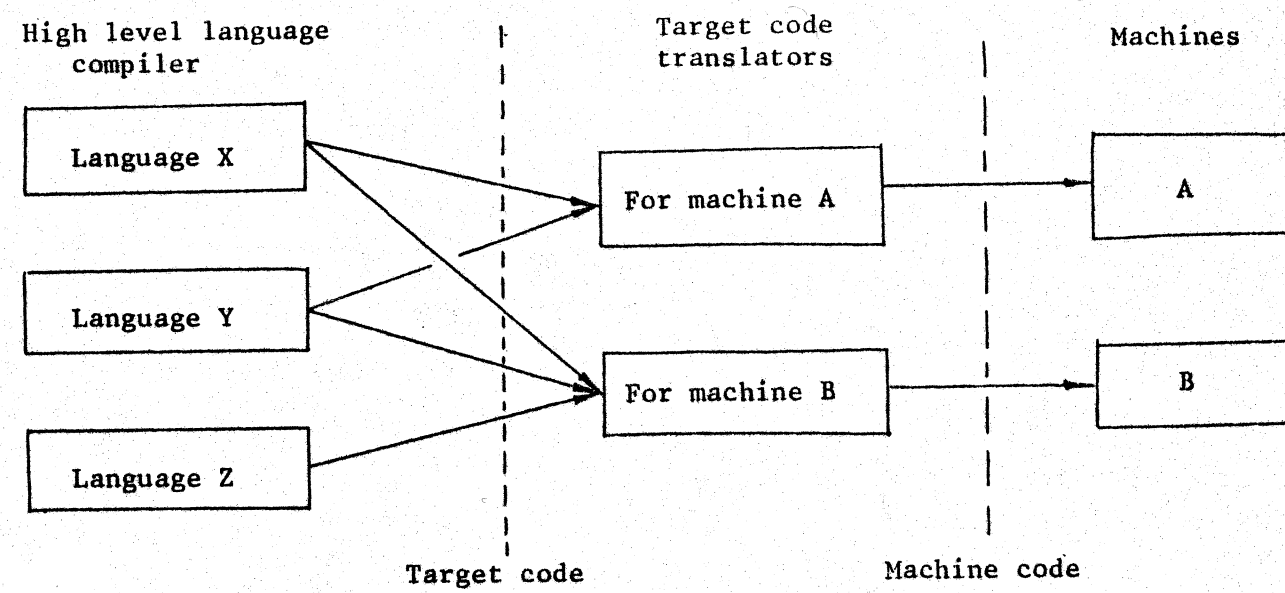
The justification for this separation of a compiler is mainly threefold.

1. The abstract machine model can be the target machine for several high level language compilers, thus all code generation tasks of these compilers is centralised, and they are thereby simplified.

2. A single compiler can produce code for several different machine types, by having several abstract machine code translators available to the compiler.

3. On transferring the compilers to another type of computer, only the abstract machine code translator need be rewritten for the new environment.

The diagram below illustrates points 1 and 2.



Although the main function of a target language is code generation, it may to advantage perform other subsidiary functions. For example, the task of compilation mode control and the provision of high level language run time diagnostics may be functions of the target language.

Experience of the design of several abstract machine models for multi-language multi-machine usage has led to the following conclusions.

1) The design of the target model is strongly influenced by the efficiency of the code required. If the best object code efficiency possible for a multiple register machine is to be achieved the compiler must be aware of the number and type of registers available on a particular machine in order to make optimum use of registers. For single accumulator (2900, MU5) and stack based machines, abstract models based on a single accumulator or stack architecture are equally effective, and such models are capable of reasonably good object code efficiency for multiple register machines.

2) Optimisations such as removing multiple evaluations of sub-expressions and moving invariant expressions outside loops should be in the machine independent part of the compiler and not in the target language translator. As a consequence of this the unit of code generation of the target language can be in terms of single instructions in the abstract model.

3) Compiler target languages which are easily portable normally achieve this by restricting their data types and data structures, consequently these languages often have considerable inefficiency in data storage mapping and inefficient data access code. In order to obtain efficient data access code on different machines,

and allow heterogeneous data structures, the target language must be sympathetic to the type and structure of data.

4) The MUTL abstract model caters primarily for Fortran, Pascal, Algol and Cobol as well as the system implementation language MUSL. An actual implementation of the model may omit some of the capabilities of the model when they are not required for the set of high language compilers required at a particular installation. For instance if only Fortran and Algol are required, the target language need not have decimal arithmetic. The target model should also be adaptable in an extensible way. As it may not cater for all the code production requirements of all high level languages.

5) The communication between the compiler and the target language should be essentially one way. Restricting the target language in this way means that the high level language to target code translation and target code to machine code translation may, where necessary, be completely separated. There are several advantages of this approach:

   a) an encoded form of the target language (MUBL) can be utilised in self bootstrapping the high level language compilers,

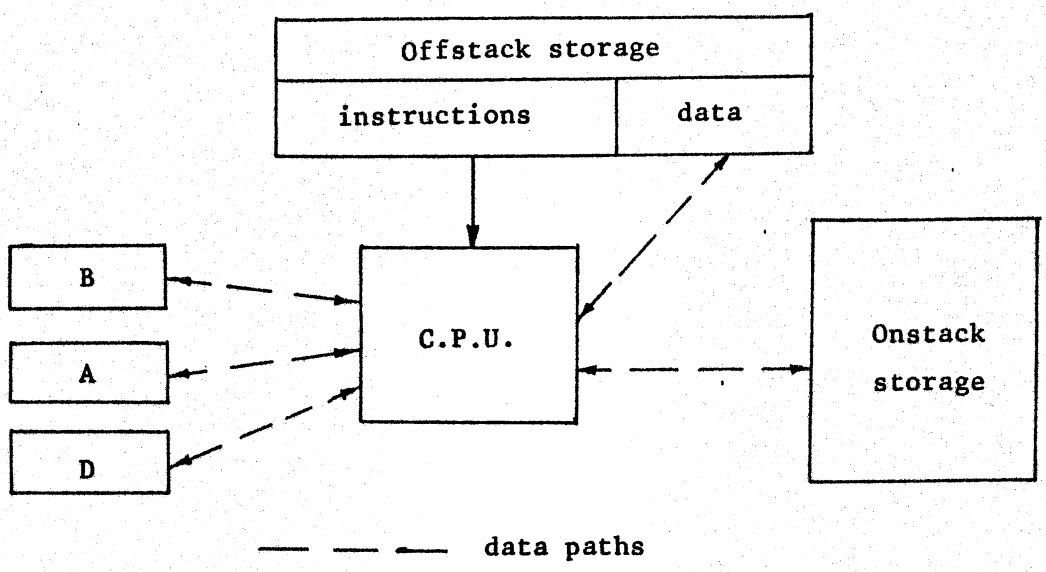
   b) in a computer network, compilation for several different machine types may be done on a central machine, and if the output of the compiler is in the encoded target language code, it can be assembled into machine code on the appropriate execute job machine in the network, and

   c) for small machines, which have a small address space it provides a natural division of the compiler into two passes.

In the abstract model data is allocated in terms of scalar or vector quantities, and from these basic data components more complex data structures are built. The model provides addressing functions to access such data structures, but the target language provides the declarative functions for the allocation of data in the abstract model. As the target language deals with data at this high level, data accessing on the actual machine can be handled efficiently.

1.2 THE MUTL MODEL

Described here is the base model of MUTL, which is implicit in all actual MUTL implementations. The diagram below illustrates a simplified view of the model, it basically consists of a set of registers, a set of stores and a CPU.



1.2.1 Registers

As mentioned briefly in the introduction, optimum register usage on a multiple register machine is best achieved if the compiler can adapt to the number of different registers available. In the MUTL model described here there is just one A, B and D register. However, the single register model could be considered as a special case of a more general model having multiple B, A and D registers (which is the eventual goal).

The A is the computational register and it operates in one of several modes:

integer
unsigned integer
floating point
fixed point decimal
pointer
typeless

The B register is used primarily for the subscripting of data structures. It only operates in integer mode.

The D register is an address register used for explicit referencing into data structures and for the manipulation of data items containing address type quantities.

1.2.2 Arithmetic precision.

To allow MUTL to be implemented on micro, mini and mainframe computers the base model has a wide range of arithmetic lengths for the A register. These lengths are:

1, 8 and 16 bits (with integer mode only)
32, 64 and 128 bits (all modes of A).

An actual implementation of the MUTL model will implement only the arithmetic modes and arithmetic precision lengths as dictated by the machine and the high level languages to be supported. B register arithmetic is of integer mode of sufficient length for data subscription purpose.

1.2.3 Instruction and Data Stores.

The model incorporates off-stack storage and on-stack storage. Instructions are fetched in sequence from the off-stack store and executed in the CPU, until a control transfer order is executed at which point instruction fetches continue from some other 'labelled' place in the store.

Data is contained in both stores. The off-stack store generally contains data that does not belong to any particular procedure e.g. Fortran Common, and permanent data of a procedure e.g. OWN in Algol, SAVE in Fortran. The on-stack store holds the data of all the currently active procedures. Within the stack frame of a procedure, its data is split into three groups:

a) parameters of the procedure
b) variables (scalars, vectors and aggregates) local to the procedure
c) partial results stored in a LIFO basis.

This allows freedom for an actual implementation to use multiple stacks.

1.2.4 Operand types and lengths.

Data in store may be considered as a sequence of simple typed operands. Each of these contains either an arithmetic or an addressing quantity.

An arithmetic operand is either:

a) 1 bit long
b) or between 1 and 16 bytes long

c) a string of up to 32 bits.

At operand declaration time type is specified. Arithmetic operands may be of real, signed or unsigned integer and decimal type. A 1 bit operand is always of unsigned integer type, and real types are restricted to operands of lengths 2, 4, 8 and 16 bytes.

The length of an addressing operand depends on the item it addresses and whether it is a scalar or vector. The following entities may be addressed: data items, labels and procedures. However, the information content required for such entities differs between high level languages, consequently six types of addressing operand are distinguished.

a) pointer to a data item
b) bounded pointer to a data item
c) pointer to a label
d) pointer plus environment information of a label
e) pointer to a procedure
f) pointer plus environment information of a procedure.

1.2.5 The CPU.

The instruction set is mainly one address and most instructions consist of a function part and an operand part. It is fully described in Section 2.8. Implementors of MUTL code generates should beware that whilst many MUTL instructions translate into one machine instruction, and some may translate into several machine instructions, others such as those concerned with operand selection and arithmetic mode selection are not expected to generate any code.

CHAPTER 2 THE MUTL SPECIFICATION

2.1 INTRODUCTION

In addition to providing the imperative functions of code generation for compilers, MUTL provides declarative functions for data items, procedures, labels etc. Before a specification of the procedural interface is given, it is necessary to introduce briefly; compilation units, storage control, the MUTL concept of types, names, machine code efficiency, diagnostic information and MUTL code output.

2.1.1 Compilation Units (Modules)

The basic unit of compilation is a module which normally contains one or more procedures, and a program or library may involve several such modules. A module contains the specifications for the entities which are considered local to the module. Some of these may be indicated as 'exports' in which case they are available for import into other modules. In order that modules may be compiled independently they must contain specifications of their imports, which at compile time are taken on trust. However, these import specifications are passed to MUTL, so that at load time it can check their consistency with the corresponding export specifications. The entities referenced in import or export specifications are called interface entities. Only areas of store, data items, data types, literals, procedures and subroutines, and labels may be interface entities. In fact areas of store are slightly different from other interface entities in that they do not belong to any particular module, but are shared between all modules using them.

It is the symbolic name of the interface entity, specified in characters, that enables inter-module references to be resolved. Therefore, each interface entity of one kind must be uniquely identifiable by its symbolic name.

2.1.2 Storage control

As discussed earlier storage is split into off-stack and on-stack storage. On activation of a dynamic procedure, a new stack frame is created to contain its parameters and local variables. Further space on the stack is obtained as required at run time by the TL.MAKE function. The off-stack store is considered by MUTL as a number of areas into which code is planted and data is statically allocated. There are 255 such areas numbered from 1 upwards, from which current

code and data areas are selected as appropriate. The amount of store allocated in an area while selected as a current area is known as a partition. The selection of an area as current defines the start of a partition, and the selection of another area terminates the partition. An area thus consists of one or more partitions.

The procedure TL.S.DECL is used to statically allocate off-stack and on-stack data. A portion of an area (off-stack or on-stack) may be statically allocated as a SPACE so that at run time variables may be dynamically allocated within it, this is achieved by the TL.MAKE function.

2.1.3 MUTL concept of types

In MUTL there are a set of predefined data types known as the basic types. Additional types known as aggregate types may be specified. These consist of a concatenation of several basic types or previously defined aggregate types. A variable of aggregate type may, when required, be considered as a single entity. In general aggregate types may be considered as a hierachic structure of types. For variables of aggregate types, individual fields are identified by their relative position, counting from zero (at each level in the hierachy). Within an aggregate type, fields may have several alternative type definitions, thus allowing variables to contain variant fields.

Basic types are classified into arithmetic and pointer types. An arithmetic basic type specifies size and mode information (e.g. 16 bit integer) for the data item. Pointer types specify the pointer mode, and also the type of the data item at which it points. The pointer modes distinguish between pointers to data, labels and procedures. There is also a typeless data pointer, but whenever a pointer of this kind is loaded into the D register type information must first be given by calling TL.D.TYPE.

An extension of the typing mechanism allows for bit string operands. These permit a number of consecutive bits in store to be processed as an arithmetic item of type integer or logical. A bit string type is specified like a vector of bits but with additional information stating whether the bit string is logical or integer.

2.1.4 Allocation of names

MUTL names are used to identify and reference many different kinds of entities, for example data variables, literals, procedures, type specifications, etc.. In order to maintain the one way communication of MUTL, for reasons specified earlier, MUTL and the compilers use the following simple algorithm for allocating names. Names exist as numbers in the range 2 to 4031. Names are allocated consecutively from

2 as entities are declared. At the end of procedures and blocks all names allocated within the procedure or block become undefined, and they are re-assigned consecutively as new declarations occur.

Subject to the non-local accessing restrictions of variables and labels of procedures (discussed later in section 2.6) all names currently defined can be referenced regardless of the level of procedure nesting at which they are defined.

2.1.5 Machine code efficiency.

The register use of the abstract model is deterministic, so that the registers of the model may be mapped efficiently onto actual machine hardware. This is accomplished with information about (MUTL) register usage supplied by the compiler.

2.1.6 Program error detection and diagnostics.

The checking for possible program errors is normally done by a mixture of hardware and software. MUTL normally provides some of the software checking, and control of this checking by compilers may be required. The MUTL procedure TL.CHECK provides this control. [Some changes to MUTL concerned with the control of checking are anticipated in late 1981].

The run time diagnostic procedures in MUTL provide primitives to support machine independent diagnostic software for such tasks as producing compile and data maps, debugging (batch and interactive), and profiling. Therefore source language reference information such as data item symbolic names, procedure symbolic names and source line numbers are passed into MUTL during compilation, this and the relevant declarative information is retained at the end of compilation if run time diagnostic support is required.

2.1.7 MUTL Code Output.

The forms of code output produced by MUTL depends largely on the requirements of a particular installation. The parts of MUSS that have dependencies on the MUTL output have been isolated so that an implementor of MUTL can select the most appropriate output forms for his installation. They are as follows.

```
RUN             : This procedure runs a compiled program.
LIBRARY         : This procedure makes available to the job
                  the procedures of a library.
DELETE.LIBRARY  : This releases a library.
```

FINDN	: This searches the directories of the current libraries
FINDP	: This returns a specification of a parameter of the result of a previously found library procedure.
INIT.DIR	: This procedure is used to initialise a directory for a library.
ADD.PROC	: This procedure is called for each procedure which is to be added to a library.
CLOSE.DIR	: When a library directory is complete this procedure is called.
MERGE.DIR	: This provides a facility to merge libraries compiled seperately.

When compiling a library, a directory is automatically generated for the library procedural interface. Apart from the directory structure, programs and libraries are similar. The code at the outer textual level of a compiled library (i.e. not embedded in a procedure) is executed, as initialisation code for the library, whenever it is loaded. The library interface at present is purely procedural. Procedures nominated as library procedures should appear in the module interface.

MUTL views a program or library as a collection of MUTL segments, these segments are numbered 0, 1, 2 etc. The model of a program is of MUTL segment 0 containing code while other segments normally contain data, but may contain code. Entry to a program is always via MUTL segment 0, therefore any entry and exit sequence code concerned with running a program is added to segment 0 by MUTL. The model of a library is of MUTL segment zero containing a library entry table, followed by code for the library procedures. There is a directory structure created for the library, but its placement and organisation is dependent on the library organisation for a particular machine. Entry to the library initialisation code is always via MUTL segment 0, therefore, any entry and exit sequence code necessary is added to segment 0 by MUTL.

Many situations arise where it is necessary for the user to exercise control over the mapping of the module areas into actual storage. Examples of this need are in overlaying for machines with limited address space, and the creation of operating systems where strict control of the placement of areas is required. In some high level languages such control is a language feature, while in others, where it is necessary, control is provided either by compiler directives or by job commands. The procedures in paragraph 2.3 below enable the compiler to control the placement of compilation output.

2.1.8 Target Language Procedure Specifications.

The procedures of the target language are grouped on the following functional basis.

a) 2.2 Type definitions.

2.2.1 TL.TYPE([SYMB.NAME],NATURE)
2.2.2 TL.TYPE.COMP(TYPE,DIMENSION,[SYMB.NAME])
2.2.3 TL.END.TYPE(STATUS)

b) 2.3 Area selection and equivalencing.

2.3.1 TL.SEG(SEG.NUMBER,SIZE,RUN.TIME.ADDR,
COMP.TIME.ADDR,KIND)
2.3.2 TL.LOAD(SEG.NUMBER,AREA.NUMBER)
2.3.3 TL.CODE.AREA(AREA.NUMBER)
2.3.4 TL.DATA.AREA(AREA.NUMBER)
2.3.5 TL.EQUIV.POS(POSITION)
2.3.6 TL.INT.AREA(AREA.NUMBER,[SYMB.NAME])

c) 2.4 Data declarations.

2.4.1 TL.SPACE(SIZE)
2.4.2 TL.S.DECL([SYMB.NAME],TYPE,DIMENSION)
2.4.3 TL.V.DECL([SYMB.NAME],POSN,PRE.PROC,
POST.PROC,TYPE,DIMENSION)
2.4.4 TL.MAKE(SPACE,TYPE,DIMENSION)
2.4.5 TL.SELECT.VAR()
2.4.6 TL.SELECT.FIELD(BASE,ALTERNATIVE,FIELD)
2.4.7 TL.SET.TYPE(MUTL.NAME,TYPE)
2.4.8 TL.ASS(MUTL.NAME.OR.TYPE,AREA.NUMBER)
2.4.9 TL.ASS.VALUE(NAME,REPEAT.COUNT)
2.4.10 TL.ASS.END()
2.4.11 TL.ASS.ADV(NO)

d) 2.5 Literal declaration and data initialisation.

2.5.1 TL.C.LIT.16(BASIC.TYPE,VALUE.16)
2.5.2 TL.C.LIT.32(BASIC.TYPE,VALUE.32)
2.5.3 TL.C.LIT.64(BASIC.TYPE,VALUE.64)
2.5.4 TL.C.LIT.128(BASIC.TYPE,VALUE.128)
2.5.5 TL.C.LIT.S(BASIC.TYPE,[VALUE])
2.5.6 TL.C.NULL(TYPE)
2.5.7 TL.LIT([SYMB.NAME],KIND)

e) 2.6 Program structure declaration.

2.6.1 TL.PROC.SPEC([SYMB.NAME],NATURE)
2.6.2 TL.PROC.PARAM(TYPE,DIMENSION)
2.6.3 TL.PROC.RESULT(TYPE)
2.6.4 TL.PROC(PROC.MUTL.NAME)
2.6.5 TL.PARAM.NAME(MUTL.NAME,[SYMB.NAME])
2.6.6 TL.PROC.KIND(KIND)
2.6.7 TL.END.PROC()
2.6.8 TL.ENTRY(MUTL.NAME)
2.6.9 TL.ENTRY.PARAM(MUTL.NAME)
2.6.10 TL.ENTRY.RESULT(MUTL.NAME)
2.6.11 TL.BLOCK()
2.6.12 TL.END.BLOCK()

f) 2.7 Label declaration.

2.7.1 TL.LABEL.SPEC([SYMB.NAME],LABEL.USE)
2.7.2 TL.LABEL(MUTL.NAME)

g) 2.8 Code planting.

2.8.1 TL.PL(FN,OP)
2.8.2 TL.D.TYPE(TYPE,DIMENSION)
2.8.3 TL.CHECK(STATUS)
2.8.4 TL.INSERT(BINARY)
2.8.5 TL.CYCLE(LIMIT)
2.8.6 TL.REPEAT()
2.8.7 TL.CV.CYCLE(CV,INIT,MODE)
2.8.8 TL.CV.LIMIT(LIMIT,TEST)
2.8.9 TL.REG(REG.USE)

h) 2.9 MUTL initialisation.

2.9.1 TL(MODE,FILENAME,DIRECTORY.SIZE)
2.9.2 TL.END()
2.9.3 TL.MODULE()
2.9.4 TL.END.MODULE()
2.9.5 TL.MODE(MODE,INFO)

i) 2.10 Diagnostics

2.10.1 TL.LINE(LINE.NO)CODE.ADDR
2.10.2 TL.BOUNDS(MUTL.NAME,[BOUNDS])
2.10.3 TL.PRINT(MODE)

j) 2.11 Run time diagnostic primitives

Initialisation

2.11.1 TL.INIT()

Procedure selection

2.11.2 TL.NEXT.PROC(STATIC)TL.ACTIVATION.NAME
2.11.3 TL.YIELD.PROC()TL.PROC.NAME
2.11.4 TL.YIELD.LINE()PAGE.LINE
2.11.5 TL.SET.PROC(TL.ACTIVATION.NAME,TL.PROC.NAME)

Variable identification

2.11.6 TL.NEXT.VAR(TL.VAR.NAME)TL.VAR.NAME
2.11.7 TL.FIND.VAR([SYMB.NAME])TL.VAR.NAME

Variable and component selection

2.11.8 TL.SEL.VAR(TL.VAR.NAME)TL.TYPE.NAME
2.11.9 TL.SEL.FLD(N)TL.TYPE.NAME
2.11.10 TL.SEL.EL(N)TL.TYPE.NAME
2.11.11 TL.SEL.ALT(N)TL.TYPE.NAME
2.11.12 TL.UP()
2.11.13 TL.YIELD.DIMENSION()DIMENSION

2.11.14 TL.YIELD.VAR.REF()VAR.REF
2.11.15 TL.YIELD.VEC.REF()VEC.REF
2.11.16 TL.SEL.DYN.VAR(VAR.REF,TL.TYPE.NAME)
2.11.17 TL.SEL.DYN.VEC(VEC.REF,TL.TYPE.NAME)

Printing

2.11.18 TL.PRINT.NAME(TL.NAME)PRINT.WIDTH
2.11.19 TL.PRINT.VALUE()PRINT.WIDTH
2.11.20 TL.YIELD.NAME(TL.NAME)[SYMB.NAME]

Dynamic debugging

2.11.21 TL.YIELD.VALUE(VALUE.RECORD)
2.11.22 TL.SET.VALUE(VALUE.RECORD)

Breakpoint handling

2.11.23 TL.INSERT.BREAKPOINT(LINE.NO)
2.11.24 TL.REMOVE.BREAKPOINT(LINE.NO)
2.11.25 TL.GO(LINE.NO)

Tracing

2.11.26 TL.TRACE.ON(MODE)
2.11.27 TL.TRACE.OFF(MODE)

2.2 TYPE DEFINITIONS

The procedures in this section enable aggregate types to be defined and associated with a MUTL name. In other contexts where the type of an entity has to be given, a MUTL TYPE SPECIFIER is used. This is an encoded integer whose least significant two bits indicate whether the specifier relates to

0 an entity of the given type
1 a pointer to an entity of the given type
3 a bounded pointer to a vector of entities of given type.

The other part of TYPE SPECIFIER either defines a basic type or the MUTL name of a user defined type. Basic types have values less than 64 (shifted left two places to accomodate the two bits defined above). Thus in order that they may be distinguished user types are actually represented by their MUTL name + 64 (again shifted left two places). Thus in a TYPE.SPECIFIER for basic types of sizes greater than 1 bit

Bits 2 - 5 specify the size of the type as the number of bytes less one (e.g, 1 means a 2 byte type)

Bits 6 - 7 specify the type mode.
0 - real
1 - signed integer
2 - unsigned integer

3 - decimal

As operands of real type are restricted to 2, 4, 8 and 16 bytes, the above leaves 48 spare encodings. Some of these are utilised as follows.

%20 - denotes a 1 bit scalar

%01 - Typeless data pointer.
%03 - Typeless unbounded data pointer.

%24 - Pointer to a procedure.
%28 - Pointer to a procedure and its environment.
%2C - Pointer to a label.
%30 - Pointer to a label and its environment.

Throughout MUTL a typed entity is either a scalar, a vector of a particular type or a bit string. In the relevant MUTL procedures this type information is supplied by two parameters TYPE and DIMENSION. For scalars and vectors TYPE contains a TYPE SPECIFIER, and DIMENSION is zero for a scalar and it specifies the number of elements for a vector. For a bit string DIMENSION specifies the number of elements in the string, and TYPE has the value %34 for a logical bit string or %38 for an integer bit string. Note that for vectors and pointers to bit strings it is necessary first to define an aggregate type for the bit string.

2.2.1 TL.TYPE([SYMB.NAME],NATURE)

This starts the specification of an aggregate type and allocates a MUTL name for the type, unless one had previously been allocated (see 2.2.2) for it. Type specifications themselves may not be nested, but an aggregate type may include previously defined types and pointers to types which are not yet fully specified. The constituent types of the aggregate are added by calling TL.TYPE.COMP repeatedly. During the specification of a type only calls to the MUTL procedures TL.TYPE.COMP and TL.END.TYPE are permitted.

The second parameter specifies whether the type is internal to the module, an export from the module, or an assumed type from another module. A type may be imported without its type detail being re-specified in this module in which case there are no further calls to TL.TYPE.COMP or TL.END.TYPE; this situation arises when the language compiler need not be aware of the type details in compiling a module.

Note that a type specification containing no fields is permitted.

The position of a component, counting from zero, within the type

definition is used in accessing it.

Parameters:-

[SYMB.NAME]	Bounded pointer to a byte vector containing the symbolic name in characters of the type. When the type is an interface entity the symbolic name identifies the type, it is also retained in the symbol table for run time diagnostic purposes.
NATURE	Bit 14 = 1 type exported from module. Bit 15 = 1 type of an assumed entity to be imported. Bit 13=1 means that a MUTL name already exists for the type, but a type specification has not yet been given to MUTL, in this case bits 0-11 contain the MUTL name for the type. Bit 12 = 1 type details not specified in this module, but are those of a type import.

Example:

To create a type T consisting of a vector of two integers (4 bytes each) TVI, and a vector of three reals (8 bytes each) TVR.

```
TL.TYPE(["T"],0)
TL.TYPE.COMP(%4C,2,["TVI"])
TL.TYPE.COMP(%1C,3,["TVR"])
TL.END.TYPE(0)
```

A type S consisting of an integer (4 bytes) SI, and a ten element vector SVT where each element is of type T can then be created by:

```
TL.TYPE(["S"],0)
TL.TYPE.COMP(%4C,0,["SI"])
TL.TYPE.COMP(MUTL NAME of T*4+256,10,["SVT"])
TL.END.TYPE(0)
```

The ten element vector is referenced as the second component of S (i.e. SEL FLD 1).

2.2.2 TL.TYPE.COMP(TYPE,DIMENSION,[SYMB.NAME])

Adds the next component to the current aggregate type specification.

The first two parameters specify the type of the component.

To permit 2 different types to contain pointers to one another, MUTL allows fields to point to types with an incomplete specification.

Parameters:-

TYPE - Bits 0-13 give a TYPE SPECIFIER as in 2.2
If bit 14 is 1 this means that the field contains a pointer to a type not yet declared to MUTL and in this case MUTL allocates a MUTL name for this type; note that the only valid settings of bit 0-13 are 1 and 3.

DIMENSION - 0 scalar
>0 vector or bit string, parameter specifies number of elements or bit string length.
<-1 vector, the parameter is the negated MUTL name of a literal whose value gives the vectors dimension or bit string length.

[SYMB.NAME] - Bounded pointer to a byte vector containing the symbolic name in characters of the component, it is retained in the symbol table for run time diagnostic purposes.

2.2.3 TL.END.TYPE(STATUS)

Terminates the definition of the aggregate type currently being defined. If, however, an alternative type is required for the field, setting STATUS equal to one allows an alternative definition to be given immediately.

Parameters:-

STATUS 1 - another alternative type specifications follows
0 - no further alternatives

2.3 STORAGE MAPPING

As previously explained in 2.1 MUTL plants code and literals within the current code area and allocates statically mapped storage within the current data area. All areas are placed in MUTL segments. The procedure TL.SEG introduces a MUTL segment into the compilation, while the procedure TL.LOAD specifies into which MUTL segment a particular area is placed. Thus a modules areas may be mapped to several MUTL

segments and a MUTL segment may contain areas from several modules. The characteristics of a MUTL segment are that it occupies, when in store at run time, a contiguous address space and all parts of it have the same level of protection (e.g. same level of protection and overlay type) and similar items throughout the space are accessed in an identical manner. Note that if there is more than one area mapped to a MITL segment the partitions of an area may not be allocated contiguously within the MUTL segment, and generally the currently selected data and code areas should not be mapped to the same segment as interleaving may occur of code and data. Normally each data declaration can be considered as unrelated from other data declarations, but for off-stack data involved in a Fortran type EQUIVALENCE it implies a specific ordering of the data. Therefore the TL.EQUIV.POS is provided to permit the declaration point of a variable to be specified explicitly within an off-stack data area.

2.3.1 TL.SEG(SEG.NUMBER,SIZE,RUN.TIME.ADDR,COMP.TIME.ADDR,KIND)

Introduces a segment into the compilation. These are known as MUTL segments and have numbers in the range 0 to 31. On introducing MUTL segment 0 initialisation code concerned with compiling a program or library is planted automatically by MUTL.

Parameters:-

SEG.NUMBER	Number of MUTL segment.
SIZE	Size of MUTL segment in bytes. A size of zero means segment size not known.
RUN.TIME.ADDR	of MUTL segment specified in bytes. A value of -1 means that the segment's address will be allocated automatically.
COMP.TIME.ADDR	of MUTL segment specified in bytes. A non negative value specifies the compile time address for the MUTL segment. A value of -1 means the compile time address will be automatically allocated; a value of -2 means that the MUTL segment need not be allocated at compile time as no code production or static data initialisation occur for this MUTL segment; and a value of -3 is similar to -2 except that on loading a program or library for execution the MUTL segment is not introduced automatically by the loading mechanism into the execution environment.
KIND	This parameter specifies the properties of the MUTL segment. Bit 1=1 means that the MUTL segment at run time requires execute access. Bit 2=1 as Bit 1 but for read access. Bit 3=1 as Bit 1 but for write access.

2.3.2 TL.LOAD(SEG.NUMBER,AREA.NUMBER).

This procedure specifies into which segment the partitions of an area will be mapped. The mapping of an area must be given before the area is used. If a partition of an area is nominated as an interface entity, then the mapping may be overridden by a previously specified mapping in another module.

When modules are bound together then once an area segment mapping is established it applies to all modules following until a new mapping is given.

Parameters:-

SEG.NO	Segment number.
AREA.NO	Area number of current module, if between modules then of next module compiled.

2.3.3 TL.CODE.AREA(AREA.NUMBER)

Nominates the current code area and sets the current position to the next available location within it.

Parameters:-

AREA.NUMBER

2.3.4 TL.DATA.AREA(AREA.NUMBER)

This procedure nominates the current data area and sets the current position within the area. Statically allocated variables are mapped to the current position in the currently selected data area.

Parameters:-

AREA.NUMBER - 0 stack frame of the current procedure
- Area number

2.3.5 TL.EQUIV.POS(POSITION)

This procedure resets the current position of the current area to the position specified. The data area to which it applies must be off-stack.

Parameter:-

POSITION: Position from start of area specified in bytes.

2.3.6 TL.INT.AREA(AREA.NUMBER,[SYMB.NAME])

This procedure nominates the area as an interface entity. It is called prior to the area being selected as the current data area.

Parameters:-

AREA.NUMBER Area number

[SYMB.NAME] Bounded pointer to a byte vector containing the symbolic name in characters of the area.

2.4 STATIC DATA STORAGE DECLARATION

Variables declared at the outer level of a module are available to all procedures within the module. For variables declared within a procedure, the kind of procedure determines whether its variables may be accessed non-locally.

2.4.1 TL.SPACE(SIZE)

Space is statically allocated in the current data area, which could be either on-stack or off-stack, and an integer variable is declared to keep track of the run time usage of the allocated space. A MUTL name is allocated for the SPACE and this name can be used in imperative functions (e.g. TL.PL) to control usage of the space. Structures are created dynamically in the space by the TL.MAKE procedure.

Parameters:-

SIZE When positive it specifies the size in bytes of the SPACE
When negative it is the negated MUTL name of a literal whose value gives the size of the SPACE in bytes.

2.4.2 TL.S.DECL([SYMB.NAME],TYPE,DIMENSION)

This procedure declares a variable which may be a scalar or a vector of basic or aggregate type. A MUTL name is allocated for the variable. For variables of a known dimension then, unless the declaration is for an assumed interface entity of another module, storage is allocated for it within the currently selected data area. For variables of an unknown dimension then it is the set of values that statically initialise the variable that determines its size, and in this case storage is allocated at value initialisation time.

Parameters:-

[SYMB.NAME] - Bounded pointer to a byte vector containing the variables symbolic name in characters. It is used to identify the interface entity and for run time diagnostic identification.

TYPE - Bits 0 - 13 is a TYPE SPECIFIER, see paragraph 2.2 for its encoding.

Bit 14 = 1 Data item to be exported as an interface entity
Bit 15 = 1 Declaration of an assumed interface entity imported from another module

DIMENSION

0 scalar
> 0 vector dimension or bit string length defined explicitly
-1 Dimension unknown. Value initialisation determines actual dimension.
< -1 vector, parameter is the negated MUTL name of a literal whose value gives the vectors dimension or bit string length.

2.4.3 TL.V.DECL([SYMB.NAME],POSN,PRE.PROC,POST.PROC,TYPE,DIMENSION).

This procedure enables MUSL V-store variables to be defined and MUTL names allocated to them. After declaration they may then be treated by MUSL as ordinary variables in imperative statements. A V-store variable is a special control/status register of the MUSS ideal machine (refn. MUSS VOL.1) that can in general be read or written. If the real machine has an actual control/status that is exactly equivalent, the V-store variable may be mapped directly to the actual register. If the correspondence is not exact then the V-store is mapped onto an ordinary store line. In this case it may be necessary to update the store line before a read access and to propagate the implied actions after a write access. Therefore, associated with each V-store of the ideal machine there is, a store line address, and optionally a procedure (PRE.PROC) to be called before a read access, and optionally a procedure (POST.PROC) to be called after a write access.

The read and write subroutines associated with V-store variables which are vectors generally need to know the index number of the element to be acted upon. Therefore, on entry to such procedures MUTL makes the index number available. Within the subroutine the index number is obtained by specifying an operand of %1002, known as V.SUB, in the operand description of TL.PL.

In the implementation of index number passing, whenever the actual machine architecture permits it, the index should be passed in the register that corresponds to the B register in the MUTL model, and an implementor of MUTL may assume that any B register use within a PRE.PROC or POST.PROC destroys the passed index value.

Like other data variables V-store variables may be interface entities.

Parameters:-

[SYMB.NAME] - Bounded pointer to a byte vector containing the variables symbolic name in characters. It is used solely to identify the interface entity.

POSN — Actual store line associated with V-store variable. This may either be a previously declared variable or the address of a store line. Thus the parameter may either be the MUTL name of a variable or a literal, or a zero which means the current literal gives the store line address.

PRE.PROC — MUTL name of the subroutine to be called before a read access. A value of zero means no subroutine call on a read access.

POST.PROC — MUTL name of the subroutine to be called after a write

access. A value of zero means no subroutine call on a write access.

TYPE — Bits 0-13 is a TYPE SPECIFIER, see paragraph 2.2 for details.
Bit 14=1 V-store variable to be exported as an interface entity.
Bit 15=1 V-store variable to be imported as an interface entity.

DIMENSION = 0 — V-store is a scalar variable.
> 0 — V-store is a vector variable, the parameter specifies the dimension.
-1 — V-store is a vector variable but its dimension is unknown, this is only permitted on imports.
< -1 — V-store is a vector variable, the parameter is the negated MUTL name of a literal whose value is the vector's dimension.

2.4.4 TL.MAKE(SPACE,TYPE,DIMENSION)

This procedure plants code to create at run time a mapping for a variable either at the top of the current stack frame or in a previously declared space. For the latter the integer control variable of the SPACE is updated to the next available storage location. Yielded in the D register is a pointer to the variable.

Parameters:-

SPACE — The MUTL name of a SPACE (i.e. SPACE > 1) or zero means allocate at the top of the stack frame.

TYPE — TYPE SPECIFIER, see paragraph 2.2 for its encoding.

DIMENSION — A value of zero means make a scalar variable and yield in the A register an unbounded pointer to it. A non zero value means make a vector variable and yield in the A register a bounded pointer to it. A positive DIMENSION specifies the number of elements that the vector is to contain, whereas a value of -1 means that the value of the B register on 'making' the vector determines the number of elements in the vector.

2.4.5 TL.SELECT.VAR().

This procedure notionally declares a SELECT variable and allocates a name for it. A SELECT variable is used to save data structure selection information held in the D register for later use. This variable may only be used in D= and D=> instructions. In a selection instruction sequence involving SEL EL, SEL FLD and SEL ALT, a 'D=> SELECT variable' saves the current selection information of D. A 'D= SELECT variable' reloads into the D register the previously saved selection information.

Before a 'D= SELECT variable' instruction is planted in the code a corresponding D=> to the same SELECT variable must have already been planted. During run time of the code on executing a 'D= SELECT variable' instruction, the SELECT variable must have previously been assigned by the immediately preceding D=> to the same SELECT variable in the code; in other words, information in a SELECT variable has a static scope.

2.4.6 TL.SELECT.FIELD(BASE,ALTERNATIVE,FIELD)

This procedure allocates a MUTL name for a field within a data structure pointed to by the BASE parameter.

Parameters:

BASE	MUTL name of a Select variable which specifies the containing data structure of the required field.
ALTERNATIVE	If the containing data structure is of union type, this specifies the alternative, counting from zero, required.
FIELD	Number, counting from zero, of field required.

2.4.7 TL.SET.TYPE(MUTL.NAME,TYPE)

A variable or parameter may be declared to be a pointer of typeless type. This procedure is used to set the type of the variable or parameter.

Parameters:-

NAME	MUTL name of variable or parameter.	
TYPE	Bits 0, 1	1 Scalar instance of a type 3 Vector instance of a type
	Bits 2-13	encoded as in paragraph 2.2.

2.4.8 TL.ASS(MUTL.NAME.OR.TYPE,AREA.NUMBER).

This procedure is used to nominate a variable or a user defined type for the current user defined type literal to which value assignment procedures refer. During value assignments there is the concept of a current assignment position. Normally value assignments start at the first component of the item, but any point can be selected as the current assignment position by calling TL.ASS.ADV appropriately.

If the variable was declared in TL.S.DECL as an vector of unknown dimension, then storage is allocated for the vector when initialisation commences, and the size of the vector is then determined by the set of values assigned. For such variables MUTL assumes that complete value assignment occurs at the same time.

Parameter:-

MUTL.NAME.OR.TYPE — If bit 14 is zero then bits 0-11 is a MUTL name of an off-stack data variable. If bit 14 is one then bits 0-13 is a TYPE SPECIFIER, see 2.2 for its encoding, of the current user defined type literal.

AREA.NUMBER — This specifies the area number in which to allocate storage for vectors of unknown size. A value of -1 means allocate in current code area, and -2 means allocate in current data area.

2.4.9 TL.ASS.VALUE(NAME,REPEAT.COUNT)

Assigns the values as specified by the NAME parameter to the components starting at the current assignment position within the literal or variable being assigned to, and advances the current assignment position accordingly. The second parameter allows this to be repeated.

Normally the NAME represents a single value, the exception to this is when the components being assigned to have a size of one byte, in which case the NAME can specify the current basic type literal which had previously been defined to represent multiple byte size values (see Section 2.5).

Parameters:-

NAME The type of name permitted is dictated by the type of the current component having its value assigned.
For components of arithmetic basic type, the name is that of a previously defined literal, and a NAME of zero means the current basic type literal. It is the type of the literal that dictates how the component is assigned a value. This means that logical literals are right justified within the component, with zero padding and truncation where necessary; similarly for integer literals, but with sign extension instead of zero padding; and for real literals, generally, these are left justified with zero padding and truncation where necessary.
For components of label type, the name is that of a label, or the current literal with a null label pointer value.
For components of procedure type, the name is that of an actual procedure or the current literal with a null procedure pointer value.
For components of pointer type, the name is that of an statically allocated data item, in which case the assigned value is a pointer to the data item, or the current literal with a null data pointer.

REPEAT.COUNT Number of times the values associated with the first parameter are assigned.

2.4.10 TL.ASS.END().

This procedure is called to indicate the end of assignments.

2.4.11 TL.ASS.ADV(NO).

This procedure advances the current assignment position over the specified number of basic type components of the literal or variable being initialised.

Parameter:-

NO Number of basic type components to be advanced over.

2.5 LITERAL DECLARATION AND DATA INITIALISATION

The procedures in this section provide for the declaration of

literals. First a value must be assigned to the current literal, then a MUTL name is assigned to the current literal. This two stage approach is necessary because the current literal is used in other contexts. Literals of basic type and user defined type are defined in this way. There are two current literals, one of basic type and one of user defined type.

2.5.1 TL.C.LIT.16(BASIC.TYPE,VALUE.16)

2.5.2 TL.C.LIT.32(BASIC.TYPE,VALUE.32)

2.5.3 TL.C.LIT.64(BASIC.TYPE,VALUE.64)

2.5.4 TL.C.LIT.128(BASIC.TYPE,VALUE.128)

These procedures are used to define the value of the current basic type literal. The first parameter specifies the type of the current literal. The size of the basic type value can be smaller than the size of the value parameter, in which case the value is right justified within the parameter.

Parameters:-

BASIC TYPE - Bits 2-7 specify a Basic arithmetic scalar type for the value. These are encoded as bits 2-7 of basic type specifications.
Bit 0 = 1 means value specified as a character string.
Bit 0 = 0 means value in binary.

VALUE.16 - Value of current literal
VALUE.32
VALUE.64
VALUE.128

2.5.5 TL.C.LIT.S(BASIC.TYPE,[VALUE]).

This procedure provides an alternative way of defining the current basic type literal value. To expediate the initialisation of byte vectors, this procedure also permits the 'current literal' to be a sequence of byte sized values.

Parameters:-

BASIC.TYPE See BASIC.TYPE of above procedures TL.C.LIT.16, etc...

[VALUE] A bounded pointer to a byte vector. If the literal type does not have a size of one byte the current literal consists of a single VALUE, the size in bytes of the literal type and VALUE must be the same. Otherwise the length in bytes of the [VALUE] parameter determines the number of byte sized values associated with the current literal.

2.5.6 TL.C.NULL(TYPE)

This procedure assigns a 'null' pointer value to the current basic type literal.

Parameters

TYPE Specification of a pointer type, see paragraph 2.2 for its encoding.

2.5.7 TL.LIT([SYMB.NAME],KIND).

This procedure allocates a MUTL name for a literal which has the value and type of the current literal. For literals of basic type then the above procedures (2.5.1 to 2.5.6) specify the literal value. For literals of user defined type then the TL.ASS procedures (2.4.9 to 2.4.11) specify the literal value.

For imported literals the value is optional. When the value is present it is checked against the exported literal value.

Parameters:-

[SYMB.NAME] Bounded pointer to a byte vector containing the literal symbolic name in characters. This is also used if the literal is an interface entity.

KIND Bit 14=1 Literal to be exported as an interface entity.
Bit 15=1 Literal to be imported as an interface entity.

Bits 0-13 A value of zero means it is a basic type literal and a value of 2 means it is a user defined type literal, in both these cases the literal value is that of the appropriate current literal. For imported literals any other value is interpreted as a TYPE SPECIFIER, see 2.2 for its encoding, of the literal and no value is given.

2.6 PROGRAM STRUCTURE DECLARATIONS

There are basically three types of program structures; namely procedures, subroutines and blocks.

A procedure may nave parameters and a result. Procedures are further classified with regards to the permitted non-local availability of their variables, and also whether a procedure is static or potentially recursive. The local namespace is created dynamically on entry to a potentially recursive procedure. The non-local accessibility of a procedures variables is at one of the following levels.

I) No non-local variable access.

II) Non-Local variable access permitted.

III) Restricted non-local access. Variables of the most recently entered procedure of this kind are accessible non-locally, but the variables of all other currently active procedures of this kind are inaccessible.

If the procedure kind is not specified explicitly, then by default its variables may not be referenced non-locally.

A subroutine has no parameters and no stack frame of its own, but it may have a result and be called recursively. For subroutines that are only invoked from the code of the immediately enclosing procedure the code of the subroutine may access (as local variables) the variables of the enclosing procedure, for all other subroutines such access is prohibited.

The same set of procedures are used to define procedures and subroutines.

A block is a delimited sequence of code within a procedure or a subroutine. The block has no name and is not referenced from control instructions. Any labels or variables declared within the block are only accessible within the block, but the variables of the enclosing procedure are still accessible as locals from within the block.

2.6.1 TL.PROC.SPEC([SYMB.NAME],NATURE)

A specification must be given before a procedure (or subroutine) is either defined or referenced. A procedure (or subroutine) specification consists of one call to TL.PROC.SPEC, followed immediately by the appropriate number of calls to TL.PROC.PARAM to specify its parameters, and terminates with a call to TL.PROC.RESULT which indicates the end of the parameters and also specifies the result type. Of course for a subroutine specification there are no calls to TL.PROC.PARAM.

A procedure (or subroutine) definition starts with a call to TL:PROC. For procedures its kind is specified, unless the default kind of procedure is required, by calling TL.PROC.KIND before the first imperative instruction of the procedure is planted. Finally, TL.PROC.END is called at the end of the procedure (or subroutine).

The symbolic names of parameters, these are used for diagnostic purposes only, are supplied to MUTL by calling TL.PARAM.NAME during the procedure definition, i.e. in between calls of TL.PROC and TL.END.PROC.

This procedure normally allocates a name for the declared procedure (or subroutine).

Parameters:-

[SYMB.NAME] Bounded pointer to a byte vector containing the characters of the procedures (or subroutines) symbolic name. It is used to identify the procedure (or subroutine) if it is an interface entity, for runtime diagnostic identification, and for constructing a directory of a library if this procedure is part of its interface. To permit abbreviated names to be used to identify library procedures, then the name supplied to MUTL contains both lower and upper case characters. The abbreviated name consists of these upper case characters. The only use of this abbreviated name is in constructing the directory of the library, and for interface and diagnostic purposes the name is handled as if it were all upper case characters.

NATURE

Bit 0 A value of zero means the procedure is dynamic i.e. a potentially recursive, and a

one means it is static.
Bit 1 A value of zero means this is a procedure specification, and a one means it is a subroutine specification.
Bit 3 A value of one means this is a library procedure specification.
Bit 4 A value of one means this is a procedure specification associated with a procedure variable.
Bit 13 A value of one means that a MUTL name is required but the parameter and result specification is to be supplied later.
Bit 12 A value of one means that a MUTL name already exists for the procedure, but it does not have a specification, in this case bits 0-11 give the MUTL name, and calls to TL.PROC.PARAM and TL.PROC.RESULT will follow to give its specification.
Bit 14 A value of one means the procedure (or subroutine) is an interface entity that is to be exported from the current module.
Bit 15 A value of one means this is a specification of an assumed interface procedure (or subroutine) of another module.

To reference a procedure from a library then its specification must be given. Bits 3 and 5 of NATURE are set to one to indicate this.

When compiling a library bits 3 and 14 are set to one to indicate that this procedure is in the libraries interface.

2.6.2 TL.PROC.PARAM(TYPE,DIMENSION)

This procedure defines the type of the next parameter in the current procedure declaration. A MUTL name is not allocated for the parameter by this procedure. Names for a procedures parameters are allocated by TL.PROC when it is called to commence the definition of the procedure.

Parameters:-

TYPE		TYPE SPECIFIER of parameter, see paragraph 2.2 for its encoding.
DIMENSION	0	scalar parameter
	>0	parameter is a vector containing the DIMENSION elements
	<-1	parameter is a vector its dimension is the value of literal whose MUTL name is

DIMENSION negated.

2.6.3 TL.PROC.RESULT(TYPE)

This procedure defines the result type for the procedure (or subroutine).

Parameters:-

TYPE — Bit 0-13 contain a TYPE SPECIFIER of the result, see paragraph 2.2 for its encoding, in addition a zero value means the procedure has no result. If bit 14 of TYPE is one then MUTL automatically allocates a result variable at the start of the procedure. Within the procedure the result variable may be accessed as a scalar variable. A zero operand of the RETURN opcode then loads the result variable (if there is one) into the appropriate register before returning. With multiple entry point procedures, a RETURN opcode with a zero operand loads the result variable associated with the entry point used to execute the procedure. Note that on the TL.PROC call of procedures which have a result variale automatically allocated, an additional MUTL name is allocated for the result.

2.6.4 TL.PROC(MUTL.NAME)

Defines that a procedure (or subroutine) starts at the current position in the code segment. For procedures a name for each of its parameters as specified in the associated procedure specification is automatically allocated at this point.

Parameter:-

MUTL.NAME — MUTL name of procedure.

2.6.5 TL.PARAM.NAME(MUTL.NAME,[SYMB.NAME])

This procedure is called during the definition of a procedure to supply symbolic names to its parameters.

Parameters:-

MUTL.NAME MUTL name of parameter.

[SYMB.NAME] Bounded pointer to a byte vector containing the symbolic name of the parameter. It is used only for diagnostic purposes.

2.6.6 TL.PROC.KIND(KIND)

This procedure is called to specify the non-local accessibility of the current procedures labels and variables.

Parameters:-

KIND

bits 0 - 1 Specify non-local accessibility of the procedures variables.
0 - Prohibited.
1 - Allowed.
2 - Restricted.

2.6.7 TL.END.PROC()

Defines the end of the code for the procedure (or subroutine) most recently started.

2.6.8 TL.ENTRY(MUTL.NAME).

On calling this procedure the current position in the code is a multiple entry point for the enclosing procedure, the parameter supplies a procedure specification for the multiple entry point.

Parameters:-

NAME MUTL name of a specification for a static procedure.

2.6.9 TL.ENTRY.PARAM(MUTL.NAME)

This procedure is called for each parameter associated with the entry

point.

Parameter:-

MUTL.NAME A non-zero value means that the parameter appears in the parameter list of a previous entry to the enclosing procedure. It gives the MUTL.NAME of the associated parameter and in this case another name is not allocated. A value of zero means that this is a 'new' parameter and a MUTL name is allocated for it.

2.6.10 TL.ENTRY.RESULT(MUTL.NAME)

The multiple entry points to a procedure may yield results of different type. To cater for this TL.PROC.RESULT has been extended as follows:

If bit 14 of TL.PROC.RESULT parameter TYPE is zero then TL.PROC.RESULT and the RETURN function act as specified in Chapters 2 and 3.

If the parameter of TL.ENTRY.RESULT is zero then, providing the entry has a result, a MUTL name is allocated for a result variable for this entry point. If the parameter is non-zero this means that a previous entry point to this procedure has a result of the same type as this entry point, therefore the result variable allocated for a previous entry point is also the result variable of this entry point. The parameter gives the MUTL name of the previously allocated result variable, and in this case no MUTL name is allocated.

Parameter:-

MUTL.NAME Zero or MUTL name of a result variable.

2.6.11 TL.BLOCK()

This procedure is called to indicate the start of a block. No name is allocated for the block.

2.6.12 TL.END.BLOCK()

This procedure is called to indicate the end of the block most recently started.

2.7 LABEL DECLARATION

Labels in code may be classified by their use and their origin, i.e. whether the label is source language or compiler generated. As most compiler generated labels are referenced in a very restricted manner, some MUTL implementations may be able to preserve actual machine registers across such labels. Labels are classified as follows.

i) Source language labels. Non-local (i.e. from textually embedded procedures) to these labels is not permitted. For local jumps to labels of this kind use the -> orders.

ii) Source language restart labels. These labels are always accessible non-locally from textually embedded procedure.

iii) Compiler generated labels.

As some labels need to be forward referenced, a specification of a label is always given before the declaration of the value of the label. The specification and declaration of the label must appear in the same procedure.

2.7.1 TL.LABEL.SPEC([SYMB.NAME],LABEL.USE)

This procedure gives a specification for a label and allocates a MUTL name for the label.

Parameters:-

[SYMB.NAME] Bounded pointer to a byte vector containing the characters of the labels symbolic name. It is used to identify the interface entity and for run time diagnostic purposes.

LABEL.USE

Bits 0 to 13. Values interpreted as

0 - Source language label.
1 - Source language Restart label.
>1 - Compiler generated labels.

Bit 14 = 1 Label to be exported as an interface entity.
Bit 15 = 1 Specification of an assumed interface entity which is to be imported from another module.

2.7.2 TL.LABEL(MUTL.NAME)

This procedure declares a label value by assigning the current code address value to the label.

Parameters:-

MUTL.NAME MUTL name of a label.

2.8 CODE PLANTING

Most instructions consist of a function and an operand as specified in the TL.PL procedure. Optimisation of loop control, when possible, is desirable for reasonable object code efficiency, therefore, MUTL also provides special code planting procedures for the most frequently used types of loops. MUTL distinguishes two special kind of loop.

1) Loops that are executed a number of times, where this number can be determined at the start of the loop, and no explicit control variable is required.

2) Loops that require a control variable, but the increment for the control variable is either +1 or -1.

For (1) the procedures TL.CYCLE and TL.REPEAT are called, for (2) the procedures TL.CV.CYCLE, TL.CV.LIMIT and TL.REPEAT are called. For other loops the appropriate sequence of TL.PL, TL.LABELS, etc. calls must be used.

2.8.1 TL.PL(FN,OP)

This procedure specifies one MUTL instruction. It may translate into several actual machine instructions or it may sometimes be optimised out. The latter is very often the case with the D instructions concerned with stepping through data structures. The general form of MUTL instructions is one-address, and they are in the main orthogonal in the sense that any function can be used with any operand. The exceptions to this rule are given in 2.8.1.3. A function (FN) specifies an operation and an operation type (or MUTL register) as shown in 2.8.1.1.

The operand part a MUTL instruction (OP) is in principle a MUTL

name, but there are special encodings of this name to provide for literals, registers, stacked quantities etc, as described in 2.8.1.2.

2.8.1.1 Operation Codes

The FN parameter is encoded thus

bits 0 - 4 specify the operation
bits 5 - 6 specified the operation type
0 indicates B operation
1 indicates A operation
2 indicates Organisational operation
3 indicates D operation

There is in effect a greater range of operation types than is apparent in the above since the A-register may be defined to be in any of the following 19 basic modes

REAL(SIZE 32 or 64 or 128 BITS)
INTEGER(SIZE 8 or 16 or 32 or 64 or 128)
LOGICAL(SIZE 1 or 16 or 32 or 64 or 128)
DECIMAL(SIZE 32 or 64 or 128)
PTR(SIZE UNBOUNDED or BOUNDED)
TYPELESS(ANY SIZE OR TYPE)

There are other modes for the A-register which cater for specific high level languages. These are known as extended A modes and at present include the following:

COMPLEX - FORTRAN
CHARACTER STRING OPERATIONS - FORTRAN

Thus in the operation table given below, the A operations are given for each basic mode which is selectable.

		A FUNCTIONS						
D	B	REAL	DEC	INT	LOG	TYPELESS	PTR	ORG.FN
0	=>	=>	=>		=>	=>	=>	=>	STK L
1	=REF	=-	=-		=-			=REF	STK PAR
2	=	=	=		=	=	=	=	ENTER
3	SEL FLD	-=			-=	-=			RETURN
4	SEL EL	&			&	&			
5	SEL ALT	V			V	V			ACONV
6	BASE	<<				<<		BASE	AMODE=
7	LIMIT	>>				>>		LIMIT	STACK
8		+	+		+				STK LB
9		-	-		-				-> IF=
10		-:	-:		-:				-> IF/=
11		*	*		*				-> IF>=
12		/	/		/				-> IF<
13		/:	/:		/:				-> IF=<
14		≡			≡	≡			-> IF>
15		COMP	COMP		COMP	COMP	COMP	COMP	SEG ->
16									ISEG ->
17									AECONV
18			MFN		MFN				AEMODE=
19		-=>			-=>	-=>			
20		&>			&>	&>			
21		V>			V>	V>			
22		=TYPE			=TYPE				
23		=ALIGN			=ALIGN				
24		+>	+>		+>				
25		->	->		->				
26		-:>	-:>		-:>				
27		*>	*>		*>				
28		/>	/>		/>				
29		/:>	/:>		/:>				
30		≡>			≡>	≡>			
31		**	**		**				

The operators used in the above table have the following meanings.

=>	Store
=-	Load negative.
=	Load.
-=	Logical 'non-equivalence'.
&	Logical 'and'.
V	Logical 'or'.
≡	Logical 'equivalence'.
<<	Logical left shift. The operand must be a positive integer.
>>	Logical right shift. The operand must be a positive integer.
+	Add.
-	Minus.
-:	Reverse minus.
*	Multiply.
**	Exponentiate.
/	Divide. For integer division this yields the nearest integer

in the range zero to the mathematical result.

/: Reverse divide.

COMP The operand is compared with the register concerned and the T register set accordingly. The T register indicates one or more of the following six states namely, =, /=, >=, >, =< and <.

=TYPE The operand is a literal, bits 0-13 is TYPE SPECIFIER and a bit 14 is the V field. The size of the type in bytes is loaded into the register; a size of zero means 1 bit. A value of zero in V means yield the size of a scalar of this type, while a value of one means yields the size of a vector element of this type.

=ALIGN The operand is a literal, bits 0-13 is a TYPE SPECIFIER and a bit 14 is the V field. The byte allignment in bytes for allocation of variables of this type is loaded into the register. A value of zero in V means yield the allignment for a scalar of this type, while value of one means yield the allignment of a vector whose elements are of this type.

=> Logical 'non-equivalence' into store.

&> Logical 'and' into store.

V> Logical 'or' into store.

=> Logical 'equivalence' into store.

+> Add into store.

-> Subtract into store.

-:> Reverse subtract into store.

*> Multiply into store.

/> Divide into store.

/:> Reverse divide into store.

STK L Stack link. The operand is the name of a procedure specification. This function is required at the start of a procedure call sequence.

STK LB Similar to STK L except the operand is the FIND.N value of a library procedure contained in the current literal.

STK PAR Stacks a parameter in a procedure call sequence. The operand must be a register whose mode is compatible with the parameter specification.

ENTER Instruction used at the end of the procedure call sequence to enter the procedure as specified by the associated STK L. A zero operand enters the procedure associated with the specification given by STK L or STK LB, or it specifies the name of a procedure variable.

RETURN Returns control from a called procedure to the instruction immediately following the ENTER used to call the procedure. The operand specifies the result for functional procedures. The result is phased back via the A register. An operand of %3000 means the A register contains the result and a zero operand either means there is no result or MUTL loads result automatically (see 2.6.3).

ACONV Sets a new basic mode and precision for the A register, and the contents of A are converted into the new mode as described in 2.8.1.4. The operand specifies the mode.

AMODE= Sets the basic mode and precision of A at the start of an evaluation in A. The operand is an encoded mode as described in 2.8.1.4.

AECONV Operates as ACONV but selects an extended mode for the A register.

AEMODE= Operates as AMODE= but selects an extended mode for the A

register.

STACK — Stacks a register as specified by the operand. A stacked operand is removed by specifying an UNSTACK operand in the instruction.

->IF= ->IF< ->IF> ->IF/= ->IF=< ->IF>=
Transfers control to a label in the same MUTL segment if the status of T implies the condition of the instruction. Operand is a name of label type.

SEG -> — Transfers control to a label in the same MUTL segment. Operand is of label type.

I.SEG -> — Transfers control to a label in another MUTL segment. Operand is of label type.

=REF — Load a reference to the entity specified by the operand. The operand must be the MUTL name of a data item or D[].

SEL FLD — Selects a sub-field of the current field. The operand is a non-negative literal integer.

SEL EL — Selects the B'th element of the current sub-field. No operand is required.

SEL ALT — If a field has multiple type definitions this selects the required type definition. A SEL ALT is not required if the first of the multiple type definitions is required. The operand is always a non-negative literal integer.

BASE — The B register specifies an element of a vector addressed by a pointer in the A or D register. This pointer is modified by this instruction so that the base of the vector now starts at the element specified. The operand specifies whether to maintain an unbounded or a bounded pointer (i.e. = 0 or 1).

LIMIT — The B register specifies an element of a vector addressed by a pointer in the D or A register. The pointer is modified so that the last element of the vector becomes the element specified. No operand is required.

MFN — Mathmatical and built-in functions that operate on A. The operand is taken to be a function number and the available functions are given in 2.8.1.5.

The operation codes for the extended modes are given in the table below

	COMPLEX	STRING
	COMP	CHAR
0	=>	L.STR
1	=-	R.STR
2	=	MOV
3	R=	E.MOV
4	I=	COMP
5		E.COMP
6		SRCH
7		E.SRCH
8	+	LEN
9	-	E.LEN
10	-:	ASC.COMP
11	*	E.ASC.COMP
12	/	
13	/:	
14		
15	COMP	
16		
17		
18	MFN	
19		
20		
21		
22		
23		
24	+>	
25	->	
26	-:>	
27	*>	
28	/>	
29	/:>	
30		
31	**	

Where the operation codes mean

R= Load real part only of complex accumulator.
I= Load imaginary part only of complex accumulator.

The other operators are as for the basic modes except they operate on the complex accumulator.

Five character string functions are provided:

MOVE This involves concatenation of one or more strings (called right strings) to form a result string and storing this in a destination string

(called a left string). If the result string is longer than the left string, then the string starting at the lefthand end of the result string of the same length as the left string is stored. If the result string is shorter, then the result string is transferred to the lefthand end of the left string and blanks inserted in the remaining part of the left string.

COMPARE — This compares two strings and sets the T status bits accordingly. The strings involved are a left and a right string, and each string consists of a concatenation of one or more strings. The left string is compared with the right string. If one of the strings is shorter than the other then it is extended to the right with blanks so that both strings are of the same length. Comparison proceeds by comparing the strings from left to right using the collating sequence of the character set.

SEARCH — This searches one string for the leftmost occurence of another string. The left string is searched for the leftmost occurence of the right string. If found its position, counting from one is put in the B register, otherwise B is set to zero.

LENGTH — This gives the length of a string in B. The left string is used to hold the string.

ASCII COMPARE — Similar to COMPARE but the comparison uses collating sequence described in ANSI, X3.4-1968 (ASCII).

Each one of these operations is specified as a sequence of simpler character string functions.

Example

E = F//G//H

MUTL instructions

```
MOV       :: Start of a MOVE operation
L.STR E   :: Left string
R.STR F   :: Result string of right strings
             F, G and H
R.STR G   :: Concatenated
R.STR H   :: Together
E.MOV     :: End of MOVE operation.
```

The character string functions provided are described below. An operand is always a vector of byte sized elements.

L.STR — This function is used to specify the left string. If the character operation in which this is used permits concatenation of strings, then a sequence of L.STR functions specify that the left strings will be

concatenated together and then act as a single string in the operation.

R.STR — As L.STR except it applies to the right string.

MOV — Start of a MOVE operation, this is followed by functions to specify the left string and then the right string. The left string consists of a single operand, whereas the right string may consist of several concatenated operands.

E.MOV — This function terminates the specification of the right string and MOVE operarion.

COMP — Start of a COMPARE operation, this is followed by functions to specify the left string and then the right string. Both the left and the right strings may consist of several concatenated operands.

E.COMP — This function terminates the specification of the right string and the COMPARE operation.

SRCH — Start of a SEARCH operation, this is followed by functions to specify the left and then the right string. Both of these strings may consist of several concatenated operands.

E.SRCH — This function terminates the specification of the right string and the SEARCH operation.

LEN — Start of a LENGTH operation, this is followed by functions to specify the right string, which may consist of concatenated operands.

E.LEN — This function terminates the specification of the right string and the LENGTH operation.

The MOV, E.MOV, COMP, E.COMP, SRCH, E.SRCH, LEN and E.LEN functions do not have an operand. The stack may be used during an operation for holding temporary data.

ASC.COMP — As COMP but for ASCII collating sequence.

E.ASC.COMP — As COMP but for ASCII collating sequence.

2.8.1.2 Operands

For most functions the operand is one of the following:

1. A zero which means the operand is a basic type literal having a value and type of the current basic type literal.

2. A name of a previously declared item (1<OP<%1000)

3. A special operand (OP>%FFF).

%1002 V.SUB see TL.V.DECL 2.4.3.

%1003 The operand is to be popped from the top of the current stack frame. MUTL requires that the use of the stack for temporary variables is determinable at translation time. This means that the corresponding STACK function must statically precede a %1003 operand, and that associated pairs of STACK functions and %1003 operands must be statically nested.

%1004 D[ ] the item addressed by the D register.

%1005 specifies an operand that is normally used with the STACK and the D= instructions. This operand allows the stack to be used for temporary select variables (see paragraph 2.4.4). This operand with a STACK instruction pushes the current selection information of the D register onto the stack, and on the corresponding paired D= instruction with this operand, the stacked selection information is popped into the D register. Obviously several items of this type may be on the stack simultaneously, but as with select variables there are static scope implications. The STACK and D= with this operand may be statically nested, and the D= is paired with the immediately preceding unpaired STACK in the compiled code.

For the next six special operands the value of the operand for the instruction depends on the state of the T register.

%1008 a 1 if T implies =, otherwise 0
%1009 a 1 if T implies /=, otherwise 0
%100A a 1 if T implies >=, otherwise 0
%100B a 1 if T implies >, otherwise 0
%100C a 1 if T implies =<, otherwise 0
%100D a 1 if T implies <, otherwise 0

%2000 the B register.
%3000 the A register.

2.8.1.3 Restrictions on the Use of Operands

There are a number of exceptions to the general rule that any function can be combined with any operand.

1) Some functions have no operand as stated in the function description
2) Some functions have specially encoded operands as stated in the function description

3) Operands of B orders must be of type integer or logical

2.8.1.4 Accumulator Mode Conversions

The operand of AMODE= and ACONV operations is taken as an encoded integer where the bits 0-13 give a TYPE SPECIFIER as in 2.2. In the case of ACONV bit 14 are also relevant and it specifies a KIND of conversion. Note that basic arithmetic modes have a value less 256 and have bits 0, 1 zero, while PTR modes have a value of 1 or 3 in bits 0, 1.

For the extended modes the operand of AEMODE= and AECONV operations is encoded as for AMODE= and ACONV, but the TYPE SPECIFIERs are as follows:

Complex - TYPE SPECIFIER of an aggregate type containing 2 reals. Fortran always uses a TYPE SPECIFIER of %108.

Character string - TYPE SPECIFIER of %83.

When the type given by the ACONV operand or the existing mode of the A register is unsigned integer then the precision of the A register is first changed to that of the new mode, and then the conversion is applied. This conversion changes the A register type and leaves in the register a binary equivalent value of its value in the old mode. Similarly when converting from an existing mode of unsigned integer, the precision of the A register is first changed to that of the new mode, and a type conversion, which leaves in the A register a binary equivalent value, is then applied.

The conversions available are given below

INTEGER TO REAL

INTEGER TO DECIMAL

REAL TO INTEGER

KIND = 0	Truncated conversion yielding the nearest integer value selected from the range zero to the floating point value.
KIND = 1	Rounded conversion yielding the nearest integer value to the floating point value.

REAL TO DECIMAL

KIND = 0	Truncated conversion.
KIND = 1	Rounded conversion.

DECIMAL TO INTEGER

DECIMAL TO REAL	
INTEGER TO INTEGER	Change in precision
REAL TO REAL	Change in precision
DECIMAL TO DECIMAL	Change in precision
INTEGER TO UNSIGNED INTEGER	
REAL TO UNSIGNED INTEGER	
DECIMAL TO UNSIGNED INTEGER	
UNSIGNED INTEGER TO UNSIGNED INTEGER	Change in precision
UNSIGNED INTEGER TO INTEGER	
UNSIGNED INTEGER TO REAL	
UNSIGNED INTEGER TO DECIMAL	
INTEGER TO PTR	The integer value becomes the address origin of a pointer.
PTR TO INTEGER	
KIND = 0	The address origin of the pointer becomes an integer value.
KIND = 1	The bound of the pointer becomes an integer value.
INTEGER/REAL TO COMPLEX	
KIND = 0	The integer/real accumulator value is converted and becomes the real part, and the imaginary part is set to zero.
KIND = 1	The integer/real value is converted and becomes the real part, and the imaginary part is undefined.
COMPLEX TO INTEGER	A truncated conversion is applied to the real part of the complex value to yield an integer value.
COMPLEX TO REAL	
KIND = 0	The conversion yields the real part of the complex value.
KIND = 1	The conversion yields the imaginary part of the complex value.

2.8.1.5 Mathematical Functions

MFN NO	REAL	INTEGER	DECIMAL	COMPLEX
0	ABS	ABS		ABS
1	MOD.1	MOD.1		CONJG
2	MOD.2	MOD.2		
3	SIGN.1	SIGN.1		
4	SIGN.2	SIGN.2		
5	DIM.1	DIM.1		
6	DIM.2	DIM.2		
7	MAX.S	MAX.S		
8	MAX.M	MAX.M		
9	MAX.F	MAX.F		
10	MIN.S	MIN.S		
11	MIN.M	MIN.M		
12	MIN.F	MIN.F		
13	SQRT	ODD		SQRT
14	SIN			SIN
15	COS			COS
16	LOG			LOG
17	EXP			EXP
18	LOG.10			
19	TAN			
20	ASIN			
21	ACOS			
22	ATAN			
23	ATAN.1			
24	ATAN.2			
25	SINH			
26	COSH			
27	TANH			
28	X SIGN			

Some of the functions (e.g. SIN,ABS) operate solely on the current contents of the accumulator. While others are used in a sequence (e.g. MOD.1,MOD.2) to provide a 'function' that operates on several values.

Example

For the Fortran statements

```
REAL X,Y,W,Z,V
Z = MAX (X,Y,Z,V)
```

The equivalent MUTL instructions are:

```
A.MODE = real
A = X
MAX.S
A.MODE = real
A = Y
MAX.M
```

```
A.MODE = real
A = Z
MAX.M
A.MODE = real
A = V
MAX.F
A => Z
```

In such a sequence the mode of the accumulator must be the same for each of the functions in the sequence.

The functions provided are as follows:

ABS — Yields absolute value. For complex mode the result is yielded in the A register whose mode and precision is that of the real part of the complex mode.

MOD.1, MOD.2 — Yields a1 - int(a1/a2)*a2, where a1 is the value of the A register for MOD.1, and a2 is the value of A for MOD.2.

SIGN.1, SIGN.2 — Yields |a1| if a2 >= 0, and -|a1| if a2 < 0.

DIM.1, DIM.2 — Yields a1-a2 if a1 > a2, and 0 if a1 =< a2.

MAX.S, MAX.M, MAX.F — Yields the maximum of (as,am ..... am,af), where as is the value of the A register for MAX.S, am is the value for MAX.M, and af for MAX.F.

MIN.S, MIN.M, MIN.F — Yields the minimum of (as,am ..... am,af)

SQRT — Yields square root

SIN — Yields sine

COS — Yields cosine

LOG — Yields natural logarithm

EXP — Yields exponential

LOG10 — Yields common logarithm

TAN — Yields tangent

ASIN — Yields arcsine

ACOS — Yields arccosine

ATAN — Yields arctangent

ATAN.1 Yields arctangent of a1/a2
ATAN.2

SINH Yields hyperbolic sine

COSH Yields hyperbolic cosine

TANH Yields hyperbolic tangent.

X SIGN Extract sign. Yields -1 in A register if A value negative, otherwise yields 1 in A register.

ODD Converts the A register into 1 bit mode and loads it with a value of one if a contains an odd integer, otherwise loads a zero.

CONJG Yields the conjugate of the complex value.

The following groups of functions operate on a pair of values, each function is used once and in order.

MOD.1	MOD.2
SIGN.1	SIGN.2
DIM.1	DIM.2
ATAN.1	ATAN.2

The following groups of functions operate on two or more values. The function suffixed .S is used for the first value, the one suffixed .F for the last value, and the .M functions for all values inbetween.

MAX.S	MAX.M	MAX.F
MIN.S	MIN.M	MIN.F

Note that intermediate values in the above group functions are held temporarily on the stack, and that nested use of functions is permitted.

2.8.2 TL.D.TYPE(TYPE,DIMENSION)

The type of an operand referenced by the D register must be set by the compiler whenever a typeless data pointer is loaded into the D register. This procedure is called immediately before the call to TL.PL which loads a typeless data pointer into the D register.

Parameters:-

TYPE TYPE SPECIFIER of operand addressed by D register, see paragraph 2.2 for its encoding.

DIMENSION 0 - Scalar addressed by D register.
>0 - No. of elements in vector addressed by D

register or length of bit string.
<0 - D addresses a vector but number of elements unknown.

2.8.3 TL.CHECK(STATUS)

This procedure changes the level of program error checking. Note, for a change in hardware error detection that it effects all subsequent instructions executed, until there is another change at run time. Whereas for a change in software error detection, the subsequent instructions that are planted are only effected.

At the start of compilation all hardware checking is permitted, and all software checking is inhibited.

Parameters:-

STATUS

Bit 7 = 0(1) means change level of software (hardware) checking
Bit 8 - 14 specify which of bits 0-6 are acted upon
If bit (8+i) = 1 then act on bit i
Bit 0 = 0(1) inhibit (permit) array bounds checking
Bit 1 = 0(1) inhibit (permit) overflow checking on B
Bit 2 = 0(1) inhibit (permit) overflow checking on A in integer mode
Bit 3 = 0(1) inhibit (permit) overflow checking on A in floating point mode
Bit 4 = 0(1) inhibit (permit) overflow checking on A in decimal mode.

2.8.4 TL.INSERT(BINARY)

The binary specified by the parameter is planted in the next available location in the current code area. Thus special operating system functions of an order code, such as dump and reload register, can be used from the systems implementation language MUSL.

Parameter:-

BINARY Value of binary to be planted. The unit of binary planted is machine dependent.

2.8.5 TL.CYCLE(LIMIT)

This indicates the start of a loop which will be executed LIMIT times.

Parameter:

LIMIT The MUTL name of an integer variable or an integer literal. A value of zero means use the current literal. In addition the names %3000 and %1004 (see TL.PL) may be used.

2.8.6 TL.REPEAT()

This defines the end of a loop.

2.8.7 TL.CV.CYCLE(CV,INIT,MODE)

This indicates the start of a loop which has an explicit control variable and a unit increment.

Parameters:

CV The MUTL name of the integer control variable.

INIT The initial value to be assigned to the control variable at the start of the loop. It can be the MUTL name of a variable or a literal, or zero meaning use the current literal, or %3000 or %1004 (see TL.PL).

MODE
- Bit 0=0 Increment is +1.
- Bit 0=1 Increment is -1.
- Bit 1=0 Control variable may be left undefined at the end of the loop.
- Bit 1=1 Control variable is defined at the end of the loop.

2.8.8 TL.CV.LIMIT(LIMIT,TEST)

This is called after TL.CV.CYCLE, but before the start of the code for the loop body. This procedure specifies the condition which terminates loop execution, and it is evaluated by comparing the control variable with LIMIT and exiting the loop if TEST is true. If this condition is initially true then the loop body is not executed.

Parameters:-

LIMIT This can be the MUTL name of an integer variable or an integer literal, or zero meaning use the current literal.

TEST
- 2 Greater than ie control variable greater than LIMIT.
- 3 Less than.
- 4 Less than or equal.
- 5 Greater than.

2.8.9 TL.REG(REG.USE)

Once a register is loaded it is defined to be in current use. A register is no longer in current use after one of the following events:

EVENT	REGISTER
a) => and into store orders	B, A or D
b) COMP order	B, A or D
c) ST PAR with register operand	B,A or D
d) STACK with register operand	B,A or D
e) B or A used as a register operand when function is not =>	B or A
f) access of operand via D[]	D
g) SEL.EL	B
h) after a TL.MAKE function call	B
h) SEL.EL	B
i) D = A	A
j) A = D	D
k) If orders	T
l) BASE and LIMIT orders	B

or events a) to g) inclusive, the A and B registers may be retained in current use after the event by calling TL.REG immediately before the TL.PL call which causes such an event. The D register is only maintainable in current use after event a) when the operand is not a select variable and after f).

In other situations the compiler may need to inform the target code translator that a register is no longer in use.

Parameters:-

REG USE
bit - 0 = 1 B }
1 = 1 A } retain register in current use

```
      2 = 1 D    }
      3 = 1 T    }

bit - 4 = 1 B    }
      5 = 1 A    }  register no longer in current use
```

2.9 MUTL INITIALISATION

2.9.1 TL(MODE,FILE.NAME,DIRECTORY.SIZE)

Initialises MUTL for a compilation. This initialisation includes creating MUTL segment 0, this may be overridden where necessary by a call to TL.SEG to replace this default creation of MUTL segment 0. At the start of each module, area 1 of the module is automatically mapped to this code segment, and area 1 selected as the current code segment while area 0 (i.e., the stack) is selected as the current data area. The action of MUTL in this mode can be described more precisely in terms of MUTL procedure calls.

1) At TL time, MUTL takes the following action:-

   a) TL.SEG (0, Default Segment Size, -1, -1, 6).

2) At TL.MODULE time, MUTL takes the following action:-

   a) Map area 1 (code area) of the module to segment 0: TL.LOAD(0,1).
   b) Select area 1 as the current code area: TL.CODE.AREA(1).
   c) Select area 0 as the current data area: TL.DATA.AREA(0).

3) At TL.END time, MUTL plants in code segment 0 any code required for exiting from a program or library.

There are some restrictions when compiling in a one pass mode. In order to map data declarations as they occur then generally interface literals and anonymous types must be exported in the compilation run before they are imported into any other modules, Furthermore on some machines there may be some restrictions on the use of interface data variables.

When compiling a library the DIRECTORY.SIZE parameter specifies the maximum number of procedures expected in the library interface.

Parameter:-

```
MODE Bit 0 = 0  produce necessary data structures to support
                run time diagnostics
     Bit 0 = 1  do not produce data structures
     Bit 1 = 0  normal compilation run
     Bit 1 = 1  MUSS type compilation, not that this means
```

that all on-stack declarations are within procedures.

Bit 2 — Only used for a normal compilation run. A value of zero means compilation of all or part of a program, and a value of one means compiling all or part of a library.

Bits 4-7 — The use of this is machine dependent. For example when compiling MUSS control of the stack position may be required. See the MUTL implementation manual of the appropriate machine for details.

FILENAME — This gives the name of the file into which the output (i.e. code and data areas etc.) of a successful translation will be placed at the end of a translation (i.e. at TL.END).

DIRECTORY SIZE

Bits 0-11 — Specify the number of procedures in library interface.

Bit 15 — A value of one means that MUTL will not create a library entry table.

Bits 12-14 — Specify library organisation dependent information. See appropriate Library Interface implementation manual for details.

2.9.2 TL.END()

Terminates a compilation.

2.9.3 TL.MODULE()

Initialises the compilation of a module.

2.9.4 TL.END.MODULE()

Terminates the compilation of a module.

2.9.5 TL.MODE(MODE,INFO)

This procedure allows the translation mode to be altered. The mode is set first by the MODE parameter of TL.MODE.

MODE Bits 8-15 specify which bits of the translation mode to alter. If bit 8 is set then change bit 0, if bit 9 is set then change bit 1, etc.
Bits 0- 7 specify the altered bit values of the translation mode. E.g. a mode of #8480 would change bit 3 to a zero and bit 7 to a one.

INFO This parameter permits further machine dependent control over the translation. See the implementation manual of the appropriate machine for details.

2.10 DIAGNOSTICS

Run time diagnostics refer to items in terms of the source language program. Most of the necessary information to accomplish this is in the existing declarative procedures within MUTL, e.g. TL.S.DECL, TL.TYPE etc. Alternatively or in addition to high level language diagnostics a compile and data map of the compilation may be produced from the diagnostic information known at compile time. The procedures in this section exist solely to supply information to support diagnostics.

2.10.1 TL.LINE(LINE.NO)CODE.ADDRESS

MUTL notes the specified source language line number and this should be called for each line of source program that generates code.

TL.LINE yields the address of the next location in the code area to enable compilers to incorporate this information in a compile map.

Parameters:-

LINE.NO 0 means that the line number is obtained by the basic system IN.LINE procedure.
/=0 means that the parameter itself specifies the line number.

2.10.2 TL.BOUNDS(MUTL.NAME,[BOUNDS])

A vector variable in MUTL consists of a single dimension with a lower bound of zero. Languages that support multi-dimensional arrays, non-zero bounds or adjustable dimensions may elect (for reasons of efficiency) to map these arrays into a single dimension with a zero

Bit 6, 7 reserved for use by implementor of MUTL
for debug monitoring control.

2.11 RUN TIME DIAGNOSTIC PRIMITIVES

2.11.1 TL.INIT()

This procedure initialises the MURD diagnostic system. All currency indicators are reset to undefined. This is called from the trap procedure (TL.TRAP), not by the user.

PROCEDURE SELECTION

2.11.2 TL.NEXT.PROC(STATIC)TL.ACTIVATION.NAME

This selects the activation of a procedure as specified by STATIC and sets the currency indicators CURRENT.PROC and CURRENT.ACTIVATION accordingly. The other currency indicators are reset to undefined.

Parameters:-

STATIC	A value of -1 means select the procedure that was currently active when the fault or breakpoint occured. A value of 0 means the next procedure in the dynamic chain associated with the current procedure activation is selected. A value of 1 means the next procedure in the static chain associated with the current procedure activation is selected.
TL.ACTIVATION.NAME	An RD name containing dynamic information of the newly selected current procedure activation.

A result of -1 means the end of the static or dynamic procedure chain has been reached.

The trap procedure makes a call to this procedure with a value of -1 for STATIC, initialising the MURD system to the procedure/block in which the program failed.

2.11.3 TL.YIELD.PROC()TL.PROC.NAME

This procedure supplies the RD name containing the static information of the current procedure activation.

A result of -1 indicates the diagnostic information of the procedure is not available.

2.11.4 TL.YIELD.LINE()PAGE.LINE

This procedure yields the page and line of the point within the currently selected procedure activation that control was last at.

2.11.5 TL.SET.PROC(TL.ACTIVATION.NAME,TL.PROC.NAME)

This procedure selects the activation of a procedure as specified by the parameters, and sets the currency indicators CURRENT.PROC and CURRENT.ACTIVATION accordingly. The other currency indicators are reset to undefined.

Parameters:-

TL.ACTIVATION.NAME — as yielded by previous call to TL.NEXT.PROC.

TL.PROC.NAME — as yielded by previous call to TL.YIELD.PROC.

VARIABLE IDENTIFICATION

2.11.6 TL.NEXT.VAR(TL.VAR.NAME)TL.VAR.NAME

This supplies the RD name of a variable.

Parameters:-

TL.VAR.NAME — A value of -1 means supply the name of the first variable of the currently selected procedure, otherwise supply the name of the variable following the

one specified by the parameter.

TL.VAR.NAME A result of -1 means that there are no more variables.

2.11.7 TL.FIND.VAR([SYMB.NAME])TL.VAR.NAME

This finds the requested variable and supplies its RD name. Only variables within the CURRENT.PROC are searched.

Parameters:-

[SYMB.NAME] Bounded pointer to character vector containing symbolic name in characters.

TL.VAR.NAME A result of -1 means variable not found.

VARIABLE AND COMPONENT SELECTION

2.11.8 TL.SEL.VAR(TL.VAR.NAME)TL.TYPE.NAME

This selects the variable as specified by TL.VAR.NAME as the CURRENT.COMPONENT. The result indicates the type of the variable. Only variables within the CURRENT.PROC may be selected.

Once a variable is selected the procedures TL.SEL.FLD, TL.SEL.EL and TL.SEL.ALT are called to manipulate the position of the CURRENT.COMPONENT, within the variables data structure and these procedures yield type and positional information about the CURRENT.COMPONENT. The TL.SEL procedures may be used to step through the basic components of a variables data structure, or alternatively select only components of interest.

Parameters:-

TL.VAR.NAME Name of a variable.

TL.TYPE.NAME Type information is encoded as follows.

Bits 0,1
0 - Scalar
1 - Pointer to a scalar instance of a type

3 - Bounded pointer to a vector instance of a type

For basic types TL.TYPE.NAME has a value less than 256, the encoding is described in 3.2 of the MUTL manual.

2.11.9 TL.SEL.FLD(NO) TL.TYPE.NAME

When the parameter is non-negative the Nth field, counting from zero, within the current component is selected as the CURRENT.COMPONENT. The newly selected component is thus one level lower than the previous selected component in the variables data structure. For a negative value of N, the [0-N]th field following the CURRENT.COMPONENT becomes the new CURRENT.COMPONENT. In this case the new and previous CURRENT.COMPONENT are at the same level within the variables data structure.

A result of -1 indicates that the component requested does not exist, but when further alternatives follow at the same level as the component search a result of -2 is yielded. For unsuccessful selection the CURRENT.COMPONENT remains unchanged. A positive result indicates successful component selection and supplies type information.

Parameters:-

N
TL.TYPE.NAME Type information as in 2.11.8.

2.11.10 TL.SEL.EL(N) TL.TYPE.NAME

When the CURRENT.COMPONENT is a vector a non-negative value of N selects the Nth element, counting from zero, of the vector as the new CURRENT.COMPONENT. When the CURRENT.COMPONENT is an element of a vector a negative value of N selects the [0-N]th following element as the CURRENT.COMPONENT. Thus in the first case the new CURRENT.COMPONENT is one level deeper in variables data structure than the previous CURRENT.COMPONENT, where as in the latter both are at the same level.

A result of -1 means the element does not exist, and the CURRENT.COMPONENT remains unaltered. For successful element selection the result yields type information.

Parameters:-

N

TL.TYPE.NAME Type information as in 2.11.8

2.11.11 TL.SEL.ALT(N)TL.TYPE.NAME

When N is non-negative the first field of the Nth alternative, counting from zero, within the CURRENT.COMPONENT is selected as the new CURRENT.COMPONENT. For a negative value of N, the first field of the [0-N] alternative following the CURRENT.COMPONENT becomes the new CURRENT.COMPONENT. Note once again that for a non-negative N the new CURRENT.COMPONENT is one level deeper, and for a negative N the level within the variables data structure is the same.

A result of -1 means the alternative does not exist in which case the CURRENT.COMPONENT is unaltered. For successful selection the result yields the type information of the CURRENT.COMPONENT.

Parameters:-

N

TL.TYPE.NAME Type information as in 2.11.8

2.11.12 TL.UP()

This procedure reselects the component at one level higher in the variables data structure that contains the CURRENT.COMPONENT as the new CURRENT.COMPONENT.

2.11.13 TL.YIELD.DIMENSION()DIMENSION

When the CURRENT.COMPONENT is a vector or element, this procedure yields the number of elements in the vector.

2.11.14 TL.YIELD.VAR.REF()VAR.REF

When the CURRENT COMPONENT contains a pointer to a scalar instance of a data structure, this procedure yields the value of the pointer as a result. Thus dynamic selection of data structures is possible.

2.11.15 TL.YIELD.VEC.REF( )VEC.REF

Similar to TL.YIELD.VAR.REF except that CURRENT.COMPONENT contains a bounded pointer to a vector. The result is a bounded pointer.

2.11.16 TL.SEL.DYN.VAR(VAR.REF,TL.TYPE.NAME)

2.11.17 TL.SEL.DYN.VEC(VEC.REF,TL.TYPE.NAME)

These procedures select the typed data item as specified by the parameters as the CURRENT.COMPONENT.

Parameters:-

VAR.REF	As yielded by TL.YIELD.VAR.REF
VEC.REF	As yielded by TL.YIELD.VEC.REF
TL.TYPE.NAME	RD type name of pointer.

PRINTING

2.11.18 TL.PRINT.NAME(TL.NAME)PRINT.WIDTH

Outputs the symbolic name associated with the TL.NAME of a procedure, variable, or the field name of the current component.

Parameters:-

TL.NAME	RD name of a procedure or variable. 0 means print name of CURRENT.COMPONENT.
PRINT.WIDTH	Number of characters output.

2.11.19 TL.PRINT.VALUE( )PRINT.WIDTH

Outputs a value of the CURRENT.COMPONENT in a suitable format as dictated by its type.

Parameters:-

PRINT.WIDTH Number of characters output.

2.11.20 TL.YIELD.NAME(TL.NAME)[SYMB.NAME]

Supplies a bounded pointer to a vector containing the symbolic name of the requested RD name.

Parameters:-

TL.NAME	RD name of a procedure, variable or type. 0 means name of CURRENT.COMPONENT required.
[SYMB.NAME]	Bounded pointer to a byte vector.

DYNAMIC DEBUGGING

These procedures allow the CURRENT.COMPONENT to be yielded or modified. If a complete record is modified a separate call is made to TL.SET.VALUE for each component.

2.11.21 TL.YIELD.VALUE(VALUE.RECORD)

Yields the value in value record.

2.11.22 TL.SET.VALUE(VALUE.RECORD)

Sets the value in value record into the current component.

BREAKPOINT HANDLING

Breakpoints may be inserted dynamically into a program and subsequently removed. When a breakpoint is entered the dump, variable printing and variable modification facilities may then be continued. In some systems the compile mode (breakpoints possibly required) must be set by a compile time call of TL.TL.

2.11.23 TL.INSERT.BREAKPOINT(LINE.NO)

Inserts a breakpoint at the start of a line. When the breakpoint is executed TL.TRAP is entered and a single line printed. The user may then print or alter variables using TL.DEBUG and use TL.GO to resume execution.

2.11.24 TL.REMOVE.BREAKPOINT(LINE.NO)

Removes a breakpoint from the start of the line specified.

2.11.25 TL.GO(LINE.NO)

Restarts the program after a breakpoint. To resume at the next instruction a parameter of zero is sufficient. It may be possible to restart at other points in a program, but in general it is only safe to do so at a line which is labelled in the source program.

TRACING

It is possible to trace the control flow through an executing program at the procedure, flowchart box or line level. It is also possible to trace all changes of designated variables.

Most (but not necessarily all in all implementations tracing requires recompilation of the program with trace information specified to MUTL at compile time. When this is done the production of trace information can be switched on and off dynamically at run time by the following procedures.

2.11.26 TL.TRACE.ON(I)

The parameter P1 controls the type and amount of tracing information produced. This is, in general, a subset of that asked for at compile time. It is bit significant with bits set causing the following action:

```
Bit 0 : print as trace proceeds
Bit 1 : print last 100 trace elements
        after failure
Bit 2 : trace procedure entry
Bit 3 : trace flowchart box entry
Bit 4 : trace every line
```

Bit 5 : trace changes to designated
variables.

2.11.27 TL.TRACE.OFF(I)

The parameter is bit significant in the same way as in TL.TRACE.ON in the relevant bit switches off that tracing.

CHAPTER 3 THE ENCODED TARGET LANGUAGE (MUBL)

The main use of this encoded form of the target language is in bootstrapping MUSS into other machines. One version of the MUSL compiler generates MUBL instead of making direct calls on the MUTL procedures, hence any component of the MUSS system can be delivered in MUBL. Normally MUSS would be established at a remote site by first delivering the MUSL compiler in MUBL and possibly other software tools such as flocoder. A MUBL translator has then to be created at the remote site after which MUSL compilations should be possible. The rest of MUSS is delivered in MUSL form. Since the full set of MUTL procedures have also to be created at the remote site so as to complete the compilers delivered with MUSS, it is perhaps advisable to create these first and use them as the means of processing MUBL. For this reason the MUBL statements are described in terms of the equivalent MUTL procedure calls.

Programs in MUBL exist as files of bytes. These bytes may sometimes be encoded as two hexadecimal characters if binary output is not feasible. Each statement is a sequence of one or more bytes in which the first byte identifies the kind of statement, and hence the MUTL procedures that are to be called, and the remaining bytes supply the parameters. There is a one to one correspondence between MUTL procedures and MUBL statements.

As explained earlier in chapter 2 a MUTL operating in executable binary mode imposes the restriction that the declaration of interface types and data variables must precede any references to them from other modules, so that code can be produced directly from the MUTL procedure calls. The MUBL of the MUSL compiler adheres to this rule. Therefore, a MUBL bootstrap for MUSL can be constructed quite simply by decoding the MUBL statements and calling the appropriate procedures of a MUTL operating in executable binary mode. For any software that breaks the forward referencing restriction, two of the options available are as follows. With a MUTL in loader mode the corresponding MUTL procedures can be called for the MUBL statements. The other option is to produce MUBL output that does not break the restriction. This can normally be achieved by inserting the type declaration for any forward references to types, and sometimes re-arrangement of the module order may be necessary.

The encodings will be given in the MUTL procedure grouping order as specified below.

An encoding is described by adding encoding information enclosed in < > brackets to the MUTL procedure specifications.

For example, <C8 #04>TL.GLOBAL.AREA(<U8>AREA). This means that the MUBL statement for TL.GLOBAL.AREA has two encoding fields. The first is one byte long and contains #04 which identifies the MUTL procedure, and the second is also one byte long and contains an unsigned quantity which is the parameter.

The different types of encoding fields are as follows.

a) <Cn #hex value> Such encoding fields are employed to identify the MUTL procedure. The field width in bits is specified by n, which is followed by the hexadecimal value for the field.

b) <Un> The encoding field is n bits long and contains an unsigned integer.

c) <UV> The width of the encoding field is contained in the field itself. The field contains a width specifier followed by a value specifier, the position of the most significant zero bit in the field indicates the last bit of the width specifier. It is the number of bits in the width specifier that defines the number of bytes in the field. The value specifier part contains an unsigned integer. For example, if the first byte has the value #94, the encoding field is two bytes long and it contains a 14 bit unsigned integer value for the parameter.

d) <IV> Similar to <UV> except the value specifier contains a signed integer.

e) <S> The first byte of the encoding field specifies how many more bytes there are in the rest of the field. This type of encoding field is normally used for parameters which contain a pointer to a byte vector, in which case the length of the vector and the vector contents are put in the encoding field.

Except for some of the code planting procedure, encodings the width of parameter encoding fields is always a byte multiple.

For any MUBL program the representation of any literal values in the program is specified, because on some machines the same literal values are represented differently. MUBL produced on MU5 will have characters specified in ISO, integers in twos complement and reals in

MU5 floating point format.

3.1 TYPE DEFINITIONS

```
<C8 %00>TL.TYPE(<S>[SYMB.NAME],<UV>NATURE)
<C8 %01>TL.TYPE.COMP(<UV>TYPE,<IV>DIMENSION,<S>[SYMB.NAME])
<C8 %02>TL.END.TYPE(<U8>STATUS)
```

3.2 AREA SELECTION AND EQUIVALENCING

```
<C8 %15>TL.SEG(<U8>SEG.NUMBER,<UV>SIZE,<IV>RUN.TIME.ADDR,
                <IV>COMP.TIME.ADDR,<U8>KIND)
<C8 %2C>TL.LOAD(<U8>SEG.NO,<U8>AREA.NO)
<C8 %03>TL.CODE.AREA(<U8>AREA.NUMBER)
<C8 %04>TL.DATA.AREA(<U8>AREA.NUMBER)
<C8 %38>TL.EQUIV.POS(<UV>POSITION)
<C8 %2A>TL.INT.AREA(<U8>AREA.NO,<S>[SYMB.NAME])
```

3.3 DATA DECLARATIONS

```
<C8 %07>TL.SPACE(<UV>SIZE)
<C8 %08>TL.S.DECL(<S>[SYMB.NAME],<IV>TYPE,<IV>DIMENSION)
<C8 %09>TL.V.DECL(<S>[SYMB.NAME],<UV>POSN,<U16>PRE.PROC,
                  <U16>POST.PROC,<UV>TYPE,<IV>DIMENSION)
<C8 %0A>TL.MAKE(<U16>SPACE,<U16>TYPE,<IV>DIMENSION)
<C8 %3B>TL.SELECT.VAR()
<C8 %2E>TL.SELECT.FIELD(<UV>BASE,<UV>ALTERNATIVE,<UV>FIELD)
<C8 %28>TL.SET.TYPE(<UV>NAME,<UV>TYPE)
<C8 %26>TL.ASS(<UV>MUTL.NAME.OR.TYPE,<IV>AREA.NO)
<C8 %0F>TL.ASS.VALUE(<UV>NAME,<UV>REPEAT.CNT)
<C8 %0D>TL.ASS.END()
<C8 %0E>TL.ASS.ADV(<UV>NO)
```

3.4 LITERAL DECLARATION

```
<C8 %10>TL.C.LIT.16(<U8>BASIC.TYPE,<UV>VALUE.16)
<C8 %35>TL.C.LIT.32(<U8>BASIC.TYPE,<UV>VALUE.32)
<C8 %36>TL.C.LIT.64(<U8>BASIC.TYPE,<UV>VALUE.64)
<C8 %37>TL.C.LIT.128(<U8>BASIC.TYPE,<UV>VALUE.128)
<C8 %3C>TL.C.LIT.S(<U8>BASIC.TYPE,<S>[VALUE])
<C8 %39>TL.C.NULL(<U16>TYPE)
<C8 %0B>TL.LIT(<S>[SYMB.NAME],<UV>KIND)
```

3.5 PROGRAM STRUCTURE DECLARATION

```
<C8 %11>TL.PROC.SPEC(<S>[SYMB.NAME],<UV>NATURE)
<C8 %12>TL.PROC.PARAM(<U16>TYPE,<IV>DIMENSION)
```

```
<C8 %27>TL.PROC.RESULT(<UV>RESULT)
<C8 %13>TL.PROC(<UV>MUTL.NAME)
<C8 %0C>TL.PARAM.NAME(<UV>MUTL.NAME,<S>[SYMB.NAME])
<C8 %29>TL.PROC.KIND(<U8>KIND)
<C8 %14>TL.END.PROC()
<C8 %31>TL.ENTRY(<U16>NAME)
<C8 %3A>TL.ENTRY.PARAM(<U16>NAME)
<C8 %3F>TL.ENTRY.RESULT(<U16>NAME)
<C8 %19>TL.BLOCK()
<C8 %1A>TL.END.BLOCK()
```

3.6 LABEL DECLARATION

```
<C8 %1B>TL.LABEL.SPEC(<S>[SYMB.NAME],<U8>LABEL.USE)
<C8 %1C>TL.LABEL(<U16>MUTL.NAME)
```

3.7 CODE PLANTING

```
<C1 %1>TL.PL(<U7>FN,<U0>OP) OR
<C2 %1>TL.PL(<U7>FN,<U15>OP)
<C8 %1D>TL.D.TYPE(<U16>TYPE,<IV>DIM)
<C8 %34>TL.CHECK(<U16>STATUS)
<C8 %2B>TL.INSERT(<UV>BINARY)
<C8 %1F>TL.CYCLE(<U16>LIMIT)
<C8 %20>TL.REPEAT()
<C8 %3D>TL.CV.CYCLE(<U16>CV,<U16>INIT,<U8>MODE)
<C8 %3E>TL.CV.LIMIT(<U16>LIMIT,<U8>TEST)
<C8 %1E>TL.REG(<U8>REG.USE)
```

Note for the TL.PL procedure the first encoding is used whenever the OP parameter has the value zero.

For the functions of TL.PL STLLLB and D=LLB the encoding is

```
<C1 %1>TL.PL(<U7>FN,<S>OP)
```

where <S> specifies the symbolic name of the library procedure.

3.8 MUTL INITIALISATION

```
<C8 %22>TL(<U8>MODE,<S>FILENAME,<UV>DIRECTORY.SIZE)
<C8 %23>TL.END()
<C8 %24>TL.MODULE()
<C8 %25>TL.END.MODULE()
<C8 %30>TL.MODE(<U16>MODE,<UV>INFO)
```

3.9 DIAGNOSTICS

<C8 %21>TL.LINE(<UV>LINE.NO)
<C8 %17>TL.BOUNDS(NO ENCODING YET DEFINED)
<C8 %18>TL.PRINT(<UV>MODE)

The run time diagnostic primitives have no MUBL encodings.

CHAPTER 4. EXAMPLES OF MUBL AND MUTL