COMPUTER SYSTEMS LABORATORY

STANFORD ELECTRONICS LABORATORIES
DEPARTMENT OF ELECTRICAL ENGINEERING
STANFORD UNIVERSITY · STANFORD, CA 94305

SU-SEL 79-O 12 SU-326-P.39-32

A THEORY OF INTERPRETIVE ARCHITECTURES: SOME NOTES ON DEL DESIGN AND A FORTRAN CASE STUDY

by
L. W. Hoevel
and
M. J. Flynn

February 1979

TECHNICAL REPORT NO. 171

The work described herein was supported in part by the Army Research Office-Durham under contract no. DAAG29-78-0205, using laboratory facilities developed with the support of the Department of Energy under contract no. EY-76-S-03-0326-PA 39.

SU-SEL 79-012 SU-326-P.39-32

A THEORY OF INTERPRETIVE ARCHITECTURES:

SOME NOTES ON DEL DESIGN AND A FORTRAN CASE STUDY

by

L. W. Hoevel

and

M. J. Flynn

February 1979

TECHNICAL REPORT NO. 171

This technical report is a revised version of TR 130 (March 1977) and includes as an appendix an abstracted version of TN 108 (March 1977).

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 94305

The work described herein was supported in part by the Army Research Office - Durham under contract no. DAAG29-78-0205, using laboratory facilities developed with the support of the Department of Energy under contract no. EY-76-S-03-0326-PA 39.

ABSTRACT

An interpretive architecture is a program representation that peculiarly suits a particular high level language or class of languages. The architecture is a program representation which we call a directly executed language (DEL). In a companion paper we have explored the theory involved in the creation of ideal DEL forms and have analyzed how some traditional instruction sets compare to this measure.

This paper is an attempt to develop a reasonably comprehensive theory of DEL synthesis. By assuming a flexible interpretation oriented host machine, synthesis involves three particular areas : (1) sequencing; both between image machine instructions and within the host interpreter, (2) action rules including both format for transformation and operation invoked, and finally, (3) the name space which includes both name structure and name environment.

A complete implementation of a simple version of FORTRAN is described in the appendix of the paper. This DEL for FORTRAN called DELtran comes close to achieving the ideal program measures.

KEY WORDS

Computer Architecture

Directly Executed Languages

Emulation

Formats

Interpretation

INTRODUCTION

In our companion paper [5] we examined the role of high level languages in defining computer architectures. Traditional machine architectures and program representations that are written for these architectures make reference to objects and operations which are presumed to be present in the host machine, i .e. the physical system that actually interprets the instructions of the image machine or architecture. An alternate model is possible where the architecture is specifically designed to be an ideal representation for a particular high level language. These program representations which we call Directly Executed Languages or DELs use operations which are in direct correspondence with operations in the higher level language. Also object identifiers are in correspondence with names used in the high level language program. Objects are coded using concise coding techniques to log, of the number of objects in a particular environment. Environment is a notion chosen to match high level language semantics. Thus, the model (Figure 1) perceives a program representation which is quite similar to the source. Only a simple one or two pass compilation process is required to create this representation and representation is decompilable, i .e. the source can be recreated given the DEL form. In this model the burden is placed on the interpreter since now execution and interpretation will occur at higher levels than hitherto. The actual interpretation process is no more cumbersome than the interpretation of a traditional host oriented architecture but one gains significant advantages with the DEL approach simply because one

has and uses more information concerning the source than is possible with the universal architecture.

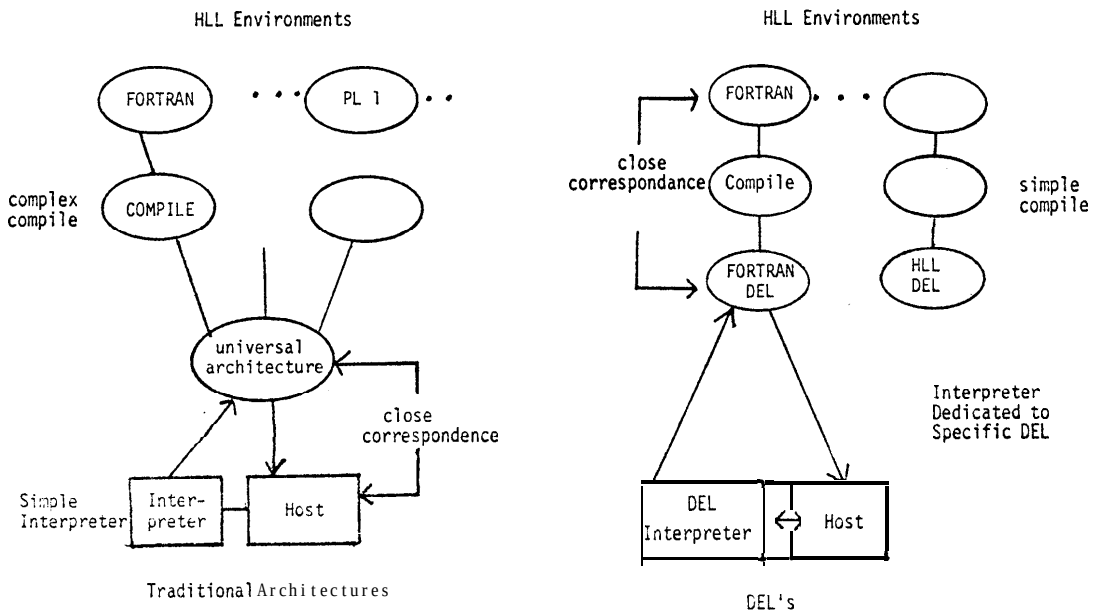


Figure 1: Directly Executed Languages

In order to develop a theory for efficient DEL synthesis, after an initial discussion of host premises, we consider in some detail problems of sequencing and operating on data and name space structure. The problem of DEL synthesis as in any design is a matter of finding a compromise between conflicting demands: in this case the efficient representation of the program and the efficient interpretation of that representation by a host system. In the DEL design we assume that we have a good deal more information about the language environment available than in a traditional machine architecture. Just as the language plays a crucial role so too does the host. However, for our

discussion we deemphasize the role of the host.

A synthesis theory is rather incomplete without a case study. In the appendix we discuss our DEL for a simple FORTRAN language. We describe this DEL and the number of considerations in its creation in sufficient detail for the reader to understand, at least in this particular case, how our theory was applied to practice.

The Host

One may view the host in either one of two ways. (1) it is an ideal mapping of a DEL representation into hardware state transitions, i.e. it is a host dedicated to a particular DEL and its corresponding higher level language. (2) It is a special purpose machine designed to have high performance for general interpretation. While it is true that the former approach would be the most efficient, i.e. present the most expeditious execution of programs, it would refer only to a particular high level language. Further, since the basic objective of this paper is to synthesize DEL representations we have an expository difficulty in assuming that both the host and the image machines are free variables. Thus, we will assume for the remainder of our discussion that we have selected approach 2: we have a high speed interpretively oriented processor as the host system.

It is important to realize how such a host differs from ordinary conventional machines. The differences are subtle yet very important. We assume that the host has:

(1) The register stack with conventional ALU operations based on them. Conventional ALU operations based on the registers form the

basic time quanta or microinstruction which defines, for our purposes, the host state transitions.

(2) The interpreter consists of programs written in this "microinstruction" or state transition form.

(3) The interpreter is contained in a special interpretive storage. Clearly, the interpretive storage is a very high speed storage, termed microstorage, since it must control the state transitions. Because of its high speed it is assumed to have a fixed size of modest proportion, perhaps 16 to 32 thousand bytes.

(4) This microstorage is also presumed to have read-write capabilities, i.e. it is a primary executable storage resource in the system. It may, and frequently does, interact with the registers not only to provide instructions for the interpreter but also for data storage. It holds all of the parameters for the interpreter and intermediate data required by the interpreter.

(5) Beyond the microstorage there is the "main storage". This is the image store and active pieces of this program may be brought into the microstorage together with active data sets. The main storage then is the container for the image program representation and its associated data.

(6) The host instruction set is designed to accomodate rapid interpretation. Thus, such operations as mask, shift, rotate, etc. are performed at high speed.

DEL SYNTHESIS

This section addresses the problem of designing high performance DEL's. We focus on three particular areas:

Sequencing, which has two aspects --

a. Sequencing between actions (program control) .
b. Sequencing within an action (context).

Action Rules, which also have two aspects --

a. The format or transformation used by the rule.
b. The operation invoked.

Name Space, which addresses two issues --

a. Name structure -- the syntax and semantics of identifiers.
b. Name environment -- referencing of variables and operators.

Terms and Assumptions

In order to synthesize simple "quasi-ideal" DELs, let us make some obvious assignments and assumptions.

* The DEL program representation lies in the main storage of the host machine

* The interpreter for the DEL lies in microstorage. The interpreter includes the actual interpretive subroutines as well as certain parameters associated with interpretation.

* Only a small number of registers exist in the host machine that can be used to contain local and environmental information associated with the interpretation of the current DEL instruction. Further, it is assumed that communications between interpretive strorage and this register set can be overlapped (Figure 2(a)).

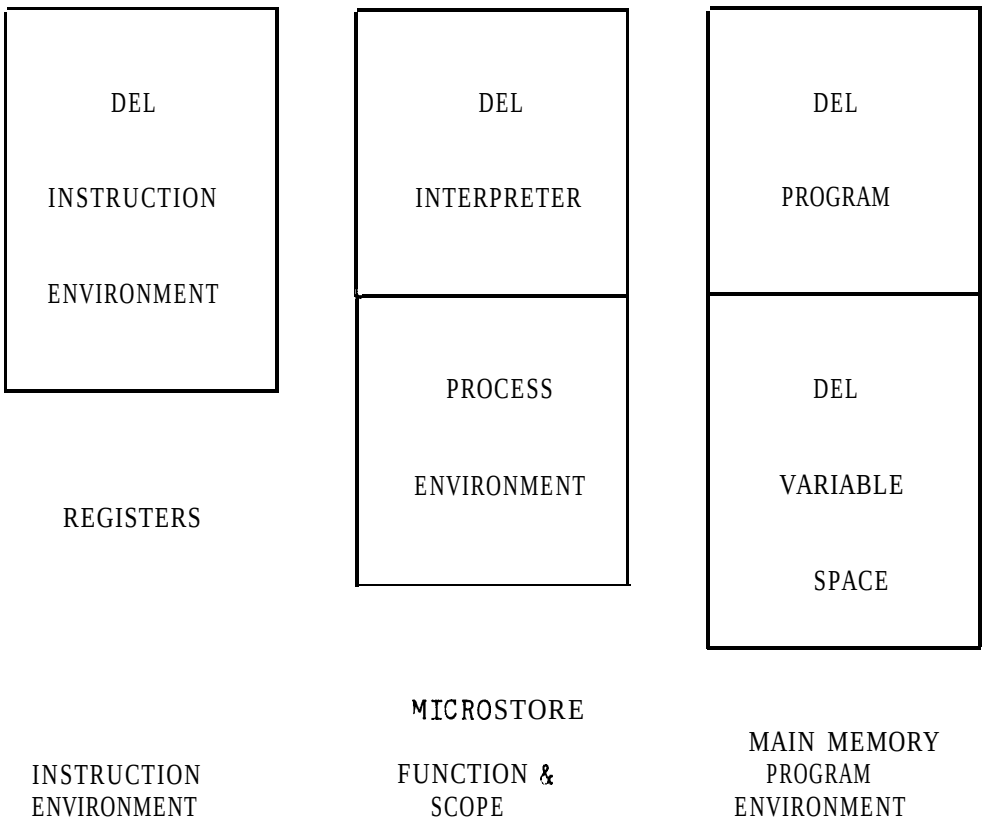


Figure 2(a): DEL/Host Storage Assignment

An instruction is a binary string partitioned into identifiers under action of the interpretive program. An identifier is an element of the vector bit string specifying one of the following:

i. format and (implicitly) the number of operands

ii. the operands

iii. operations to be performed (of at most binary order) on the identified operands

iv. sequencing information, if required.

A format is a rule defining:

i. the instruction partition (i.e. number and meaning of identifiers).

ii. the order of the operation (i.e., whether the operation is in nullary, unary or binary).

iii. precedence among operands (i.e., binding of operand identifiers to functional operands).

In this paper, it is assumed that DEL instructions are use ordered -- i.e., that the internal sequence of identifiers within an instruction is the same as the sequence in which these identifiers will be required during interpretation. The 370 architecture is not use ordered, since the format/operation code appears before operand identifier information. This forces the interpreter to "save" the operation code during computation of effective addresses -- wasting, at least temporarily, a scarce host register.

The size of an identifier is the width of the field it occupies within an instruction. It is determined by the number of elements required in a locality; the structure of a typical DEL instruction is illustrated in Figure 2(b) .

Sequencing Rule

Usually, a program consists of a sequence of action rules. The sequencing rule provides the ordering relation among the action rules -- i.e., it defines the sequence of the action. While it is possible to conceive of DEL's with unordered action rules (no sequence rule), this form is of little value in representing familiar high level language programs.

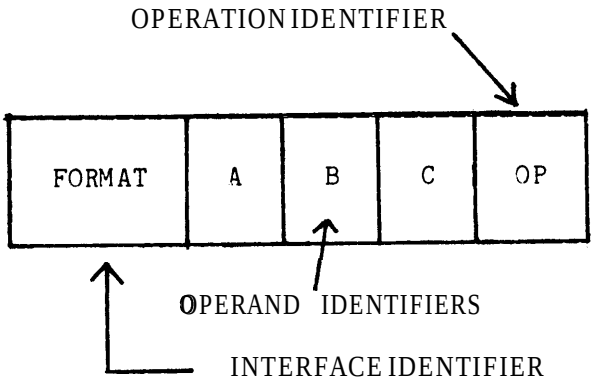


Figure 2(b) : Layout of a Typical DEL Instruction

Sequencing Between Actions (Image Machine Sequencing)

In practice only a few sequencing rules have been used with any degree of success :

Linear: individual instructions are stored in a one dimensional array within the main store. Execution order is the same as the array ordering unless modified by a branch instruction.

Binary Tree: instructions are mapped into the nodes of a tree structure maintained in main store. Leaf nodes normally correspond to data references; ancestor nodes to semantic functions. A standard traversal algorithm defines the default order of execution, which can be modified by visiting a branch node.

Linked List: instructions are stored at the links in a chain structure maintained in main store. The default execution order is again specified by a traversal algorithm, and can be modified by the semantics associated with the most recently visited link.

These three forms are abstracted from well known programming structures. Most traditional machine language DELs are based on a linear form. Tree form are widely used as intermediate data structures by compilers. Linke3 lists are the fundamental program and data structures for LISP and PPL (McCarthy [15], and Standish [18]). Tree

and list data structures are widely used in the algorithms employed in artificial intelligence and information retrieval applications. Figure 3 illustrates program representations in the linear, tree, and list forms.

The particular DEL organization used in these examples is arbitrary, for purposes of illustration only, and is not necessarily optimal. Similarly, neither the operators nor data structures are completely specified; they should be assumed to have the same general interpretation for all three DEL forms. These fragments are constructed so that the order of execution will be identical (i.e., the sequence of functional operations and storage accesses will be the same).

Linear Forms

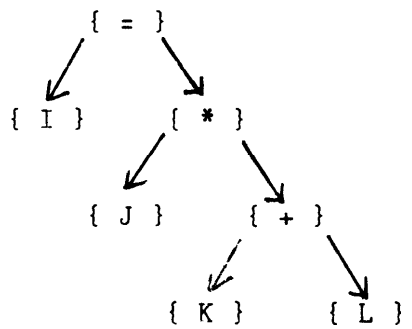
The sequencing rule for a DEL governs the way in which control is passed from one instruction to another. If a linear form is used, for example, the normal sequence of execution is implied by the placement of DEL instructions within the main store. A program counter is usually maintained within the interpreter, as part of the DEL program status vector, which points to the word containing the next DEL instruction to be executed. When the contents of the current instruction word are interpreted, the word pointed to by the program counter is fetched, the counter incremented appropriately, and execution continues. Interpreting a branch instruction causes the DEL program counter to be loaded with a new address that points to the next instruction to be executed. The set of branching instructions in a

Figure 3: Three Representations of " I = J * (K + L) ; "

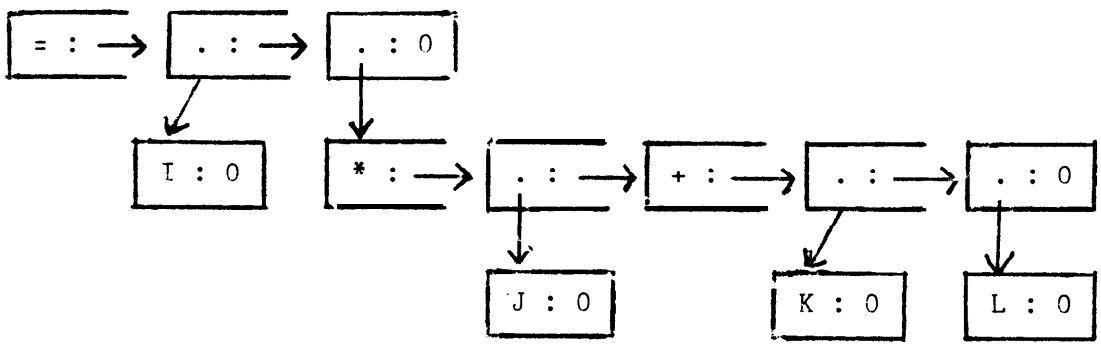
(a) -- Linear

```
push @I
push J
push K
push L
+ (add)
* (multiply)
= (assign)
```

(b) -- Tree



(c) -- List



DEL is not confined to the simple GOTO, but may also include more complex program control operators such as CALL, RETURN, DO, and IF-THEN-ELSE.

The natural ordering of addressable storage cells can be used to induce a default order of interpretation, thus, eliminating the need for explicit sequencing of pointers in linear segments of DEL code. As individual instructions are more highly compressed, fewer main store accesses are required to maintain a given DEL instruction stream.

Tree Forms

Tree structures are used by many compilers as an intermediate form from which the final, executable code is generated. Intuitively, ancestor nodes refer to operators (non-terminals in the source language syntax), while leaf nodes refer to variables (syntactic terminals). The operation code associated with a node is combined with two or tree pointers to form a unit of fixed, uniform size. These units constitute the physical realization of a tree structure within the main store of the host machine. The units for a binary tree DEL need contain only two pointers in a minimal realization: (1) the address of the unit for the left descendent of a node; and (2) the address of the unit for its right descendent.

The left and right descendents of an ancestor node which is associated with a binary operator correspond to its left and right operands, respectively. Usually, the operators in a DEL are binary if a tree structure form is selected -- unary operators are treated as degenerate binary operators, with null right descendent pointers. Some auxiliary pointers (usually to the ancestor of a node) may be included

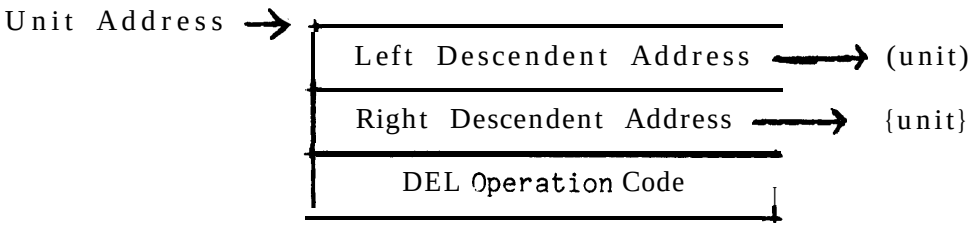


Figure 4: Typical Binary Tree Unit

to facilitate tree traversal, however.

List Forms

The simplest examples of linked lists look much like unary or binary trees; in fact, most of the above tree related comments are equally applicable to linked list DELs. However, the links within a list (its nodes) may be their own ancestors -- i.e., cycles are allowed. Again, instructions are associated with the links in a list representation. They contain a pointer to a successor link, and either an atomic value or a pointer to a value link. A unique pointer, NIL ("0" in Figure 3(c)), is used as the successor pointer in such terminal links.

Because of their generality, linked lists are not easily address encoded. While the relative spatial cost of link pointers depends on the average size of a DEL instruction; a linked list DEL almost always requires more space than an equivalent linear form DEL, barring extensive factoring of common sublists. However, the marginal cost of incorporating additional address references is low for a linked list DEL representation, and hence it is comparatively easy to implement

complex operators that do not easily fit in the binary operator order.

In most cases, the pointers required by tree and list structures makes them less desirable than the linear array as a potential DEL form: both because of the space these pointers occupy, and because of the extra main store access needed to determine the location of successor instructions. It is usually far faster to increment a DEL program counter (normally maintained in a host register) than to fetch an address from main store. Unless the flexibility of tree and list forms can be exploited in an innovative manner, the spatial and temporal overhead associated with this single negative aspect may be of overriding importance in selecting the form for a DEL.

SEQUENCING WITHIN AN ACTION (HOST MACHINE SEQUENCING)

Defining a sequence rule within an action is primarily a problem of exploiting execution context during an action rule interpretation. Context information may be used to significantly improve action rule representation at the expense of some additional complexity in the interpretation process. We consider five distinct, types of context.

No Dependencies

The simplest program representations involve no dependencies, and an example of such DELs is "threaded code" -- in which each field occupies a full word of storage, and is itself a direct pointer to either a cell in the DEL data store (operand references) or to a semantic routine in micro store (operator references). This straightforward encoding may, in fact, be optimal if the host has little or no field extraction capability, since each syllable starts on a word

boundary and need not be processed before use during interpretation.

Threaded code programs are similar to highly subroutinized host programs in which there is one subroutine for each semantic routine within the threaded code interpreter.

The time needed to fetch a threaded code instruction, in main memory accesses, is k+l ; where k is the average number of operands per instruction. If we let b denote the number of bits per word of storage, then the space required to represent a threaded code instruction is b * (k+1).

Memory Dependencies

Given a word oriented host, we view instructions as fixed length "records" containing a fixed number of subfields at known boundaries. In this case, use ordering is of minimal importance, since the syllable positions are always known. Selecting an optional instruction layout is basically an alignment problem; instructions should be stored on bit addresses that minimize the number of main store accesses required to extract critical fields. This problem is examined from the perspective of the computer architect in Flynn and Henderson [6].

This analysis can be applied directly to the DEL synthesis problem, although there are fewer free variables in this case since the host machine is fixed. The relevant result is an analytic expression for the average number of accesses required to retrieve a group of F characters with character address I into a record of length L.

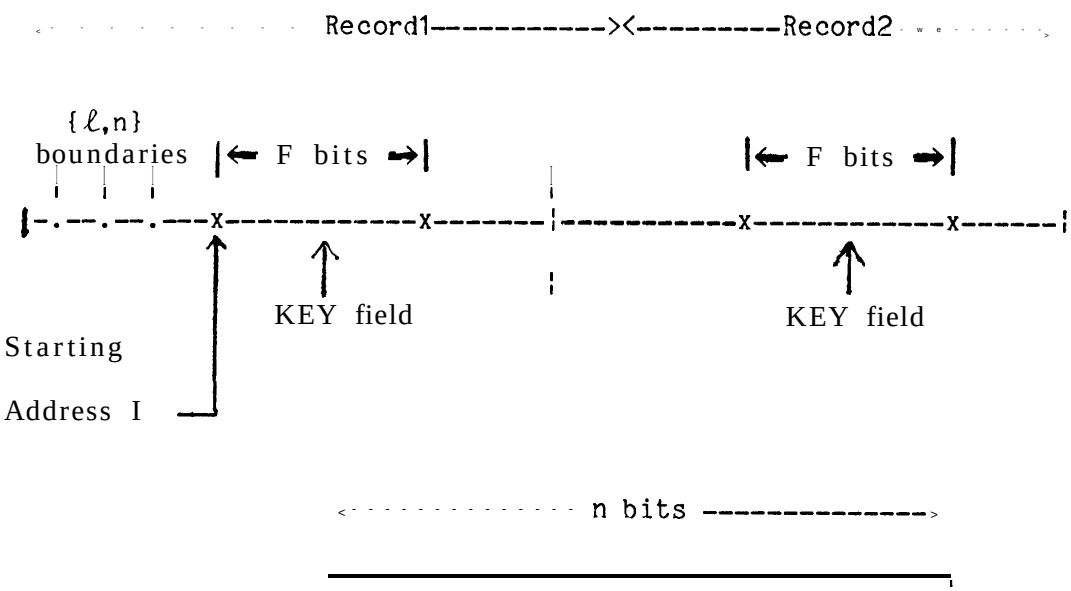


Figure 5: Accessing KEY Fields in DEL Instructions

The group of F characters can be thought of either as an entire DEL instruction -- in which case the notion of a record also corresponds to an instruction -- or as a critical syllable (e.g., the KEY code) within an instruction. In the latter case, the instruction is itself the L character record. If each main store access retrieves n characters of data, the number of accesses needed to fetch the critical portion of an instruction is

$$\text{Accesses} = \left\lceil \frac{F}{n} \right\rceil + \frac{\left\lceil \frac{i+f}{\{\ell,n\}} \right\rceil - 1}{n/\{\ell,n\}}$$

where: f = F Mod n (least positive residue; i.e., x Mod x = x), ℓ = L Mod n (least positive residue); i = I Mod $\{\ell,n\}$ (least

residue, including 0), and $\{\ell,n\}$ = greatest common divisor of ℓ and n.

Although formidable in appearance, this equation is not difficult to interpret. Clearly, the number of accesses required to fetch a DEL instruction of length F from a unit of length L will be either $\lceil F/n \rceil$ or $\lceil F/n \rceil + 1$, depending on the number of word boundaries crossed. This is determined by the starting address of the instruction. The second term is an analytical representation of the average effect of this placement, assuming that fields occupy integral multiples of the basic storage quantum (e.g., eight bit bytes for a 360/370 environment). While this is a reasonable assumption for a machine designer, character size is often a free variable to the DEL designer (Hoevel and Wallach [10]).

If the host is strongly biased toward a particular character size, then it is probably best to use this as the basic storage quantum for DEL encodings. If the host is unbiased, however, the size of a character should be selected to minimize F/n. The Flynn-Henderson equation shows that it is best to start instructions on character addresses that are integer multiples of $\{\ell,n\}$. In this case, the time needed to fetch a typical DEL instruction, in main storage accesses, is:

$$\text{Access Time} = \lceil F/n \rceil + \frac{f - i\,b\,\}}{n}$$

while the space needed to represent it is:

$$\text{Program Size} = \ell * b/n = w * (k+1) \text{ bits}$$

As above, k is the number of syllables that must be fetched and decoded to execute the entire instruction, and b is the number of bits per word; w is the average number of bits per syllable.

In most cases F is less than n, and so the average fetch time is minimal when F is minimized -- i.e., when pointers and/or instructions occupy as few characters as possible. Decoding algorithms for this type of DEL are usually straight forward. Since instructions are word aligned, the exact bit offset of each subfield is known, and decoding is at worst a simple combination of mask and shift operations.

In some cases, special features of the host can be exploited -- such as the transform board capability of the CDC 5600 series, which allows the contents of a micro register to be "exploded" (i.e., distributed accross several other micro registers in a single micro instruction). This board must be physically rewired for each such explosion desired, however, and cannot be changed dynamically during an emulation.

Inter Instruction Dependencies

Both the sequence in which instructions are encountered and their placement can affect their interpretation for certain DELs. The primary reason for selecting a form with inter-instruction dependencies is to minimize the size of a typical DEL program and, thus, indirectly reduce the average fetch overhead.

To exploit the similarity between integer addressable stores and linear program structure, a design permitting multiple DEL instructions to be placed in a single word of storage must be devised. Minimizing the size of individual DEL instructions is quite important here, although if an execution time advantage is to be realized the encoding must be simple to recognize and decode.

Usually, the DEL program state vector is augmented so that the interpreter can remember unused, but previously fetched portions of the DEL instruction stream. Specifically, a residual control cell called the current instruction word (IW) is needed. This word contains those bits in the DEL instruction stream that were brought into host storage registers during the last instruction stream access to main store, but which have not been decoded.

This type of dependency is most effective for hosts with wide storage resources and a large ratio between main and micro store bandwidths. For an average of m instructions packed into a single word, the time needed to fetch a given instruction stream may be reduced by a factor of m compared to a fully independent technique if the image machine sequence is linear.

Interpreters for instruction stream dependent DELs must maintain at least two elements of residual control: a DEL program counter (PC); and current instruction word (IW). If full prefetch is implemented, and additional residual control cell is needed -- a successor instruction word (SW) . The interpreter attempts to maintain the next word of instruction stream bits in SW (i.e., keep SW equal to the con-

tents of the successor to the word last loaded into the IW). When all of the bits in the IW have been decoded, its contents are replaced by the contents of SW, the PC is updated, and most of the time needed to transfer instruction words from main store into the internal resources of the host to be overlapped, but this implies that the PC, IW, and SW must be maintained in the fastest storage resource (i.e., host registers). Use ordering of syllables is important in a strongly context dependent DEL, since such a large fraction of the micro level storage resources must be dedicated to maintaining the DEL instruction stream.

For example, decoding an operator specification prior to the specifications of its operands (as in the natural sequence of interpretation for the 360/370 architecture) forces the interpreter to store the operation code across the operand fetch portion of the interpretation cycle, thus increasing execution time. Also, instructions need not be word aligned. This means that it may be more difficult to decode the syllables, since it can no longer be assumed that they are aligned on specific address boundaries.

Memory Mapping and Word Boundary Dependencies

For the moment, assume that a DEL instruction consists of a sequence of as yet undifferentiated syllables. These syllables may be of a single, uniform width (often the case for polish DELs), any of a fixed number of different widths, or even of dynamically varying widths. How does the interpreter fetch the next instruction? Consider three strategies:

i. Dynamically concatenate successive words in the DEL program store, in effect creating a "bit stream" memory.

ii. Code the fact that the next n syllables lie within the current instruction word as part of the semantic interpretation of the first (or last) syllable in the instruction.

iii. Reserve one syllable code (usually all zeroes) to signify "end of instruction word" -- i.e., that the current instruction word is exhausted (i.e., has been interpreted), and a new instruction word fetch is required.

The first technique is used in the Burroughs S-language implementation for the B1700, a defined field host capable of accessing arbitrary sized fields at bit addresses. By packing DEL instructions at the bit level means that "every bit is fully utilized", and "appears to account for half of all the program compaction which has been realized on the B1700" (Wilner [22]).

There can be a high interpretation time penalty associated with frequency encodings, however, since several sequential levels of decoding may be required to correlate a syllable code with the proper semantics. Wilner outlines an "SDL" encoding that is claimed to obtain most of the compaction resulting from Huffman's code [11], while still permitting reasonable decode times. The resulting polish form instructions are about thirteen bits in length (averaged over both operator and data instructions), and require a maximum of three stages of decode. Wilner estimates that a pure Huffman code would be fourteen per cent slower to decode, but would only reduce the size of a typical surrogate by one per cent.

These tine estimates may be unique to the B1700 and the specific interpretation algorithm used to process the S-languages. Although

Wilner claims only a 2.6 per cent slow down from a straight n-way binary code to a 4-6-10 staged encoding, the manner in which this is computed is not clear. It may be that little or no retention is used by S-language interpreters, or that instruction fetch time is included in the computation of decode time -- which would certainly tend to equalize differences between various techniques. Decoding SDL codes on an EMMY [173 based system (available at our Stanford Emulation Laboratory) would require more than double the time needed by a simple n-way binary code. This is equivalent to more than 40 per cent of a typical instruction execution; if a pure Huffman code were used, this factor could double again. At least some direct hardware assistance appears to be necessary for this technique to achieve high performance.

The second strategy is nothing more than the familiar fixed field organization used by most second and third generation "machine languages". Once the first few bits of such a DEL instruction have been decoded, the exact length and placement of all the subfields within that instruction can be determined. In this case, the Flynn-Henderson equation can be used to adjust the overall length of the various instruction types so as to minimize the time needed to fetch a given instruction stream -- i.e., minimize the time needed to access the critical fields defining the tranformations to be performed.

The last technique was developed independently during the synthesis of DELtran (Hoevel [9], described later in this paper). It approximates the bit stream packing capability of the B1700, but

requires only two registers, the instruction index IX and instruction word IW, and is easily implemented on hosts with flexible memory arrangements. Each DEL instruction is treated as a string of syllables that is fetched and decoded as follows:

1. A syllable is extracted from the IW using either of the two methods described above.
2. If the IW is now zero, transfer of the next word in the instruction stream into the IW is initiated.
3. The appropriate routine is invoked, depending on the contents of the IX, and execution continues with step one.

Using this technique, the all zeros code must be reserved to indicate that the current instruction word has been exhausted.

The generation algorithm for this is to simply place successive syllable codes into a word until the next code does not fit within that word. The current word is then filled with zeros, and the process is repeated for the next word in the DEL program store.

The following is a simple technique, hinging on the definition of "fit", that can save some execution phase time and space (Fig. 6). Suppose that there are M bits in the next syllable code to be packed into a word that has only N bits remaining, where M is greater than N. The first N bits of this syllable can be packed into the current word if its M-N trailing bits are zero -- they will be supplied automatically by the algorithm outlined above. This results in individual syllables being logically, if not physically, contained within individual program store words, but permits entire instructions to cross word boundaries.

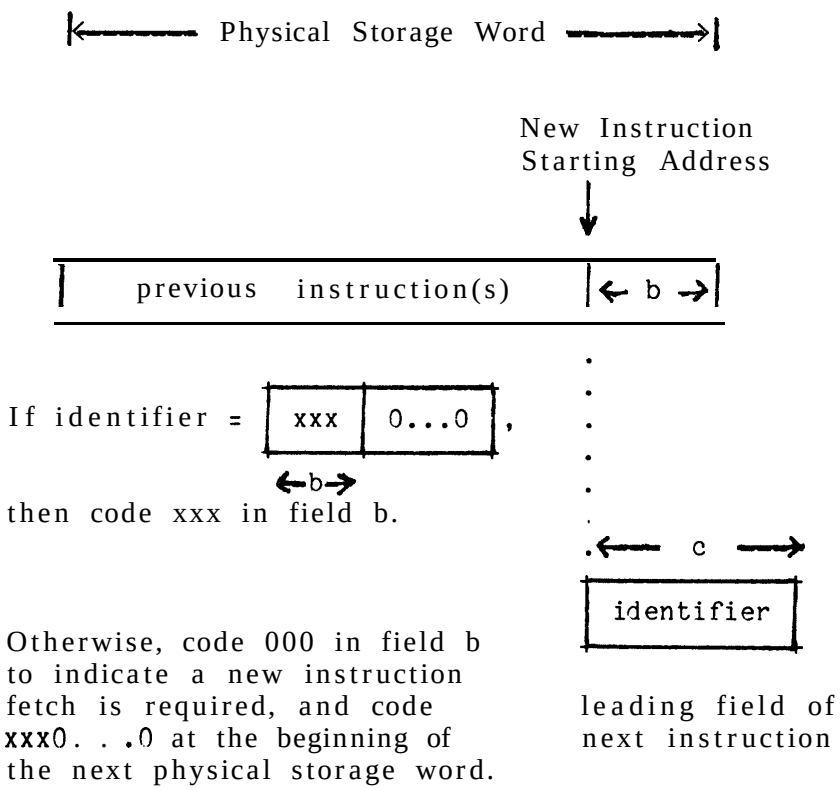


Figure 6: "Fitting" Syllables at the End of a Storage Word

By assigning these codes such that frequently occuring codes have a greater number of trailing zeros, the beneficial effects of this technique should be significantly improved.

Intuitively, this gains some of the spatial advantage of Huffman like codes (at word boundaries) for the simple straight binary code, yet permits rapid decode. In theory, it could also be used in conjunction with more highly encoded forms (either SDL or pure Huffman): the relative time gain would be smaller since decode overhead would dominate the instruction fetch, however; and the space gain would be reduced due to the reservation of the all zeros code. Time and space

estimates for this form are:

Access Time = (k+1) * (R*w/b + shift(w) + test)

Program Space = w * (k+1)

R, k, b, and w are again the same as before; "shift(x)" is the number of host instructions required to extract an x bit field; and "test" is the number of host instructions needed to check for the all zero code (which should be 0 or 1 for a well designed DEL host) .

Field Dependencies

So far, we have discussed only static dependencies. It is also possible to take advantage of locality by dynamically changing the interpretation of specific codes. That is, the semantics associated with special DEL operators may be used to change the tables used by the decode routine within the interpreter. While this generally requires rather sophistocated compilation techniques (see Foster and Gonter [7], and Sweet [19]]), it may be possible to avoid exhorbitant overhead by applying this stratagem only when DEL control passes from one module to another. This is because of the one-to-one correspondence between DEL modules and the lexical "scopes" in the source programs from which they were derived. Fix ing the size of an operand reference upon entry to a DEL module can result in dramatic compression of program size, and should be considered when synthesizing a DEL for any block structured source language.

THE ACTION RULE

The action rule consists of a function applied over a domain of arguments that produces one result. There are two considerations in

synthesizing an action rule: format and operation.

The synthesis objectives for both considerations should be clear from the discussion of canonic interpretive form in our companion paper [5].

* Enough formats should be available to provide transformational completeness: only unique HLL variable names are allowed in the DEL instruction and no "memory overhead" instructions (load, store, move, push, pop, etc) need be introduced.

* Each HLL operation should have a corresponding interpretation within the limits of interpreter size.

The above requirements imply a 1: 1 correspondence between an HLL operation and a DEL instruction. Also, additional variable names (TEMP's) are not introduced. In fact, A + B → A, would have a DEL representation of [F,+, A, B], (F a format indicator) while A * B + C → C would be represented as [F,*,A,B]; [F,+,C] – the single "C" is a result of the uniqueness requirement.

FORMATS

In order to recognize and interpret DEL instructions, the interpreter must be able to determine the size and meaning of at least the next syllable to be fetched and decoded. The leading syllable in an instruction usually specifies its layout and interpretation ; i .e., defines the format of the instruction.

In order to select a format set in an orderly manner, it is necessary to first construct a universe of formats that at least covers the combinatorial bindings found in traditional zero, one, two, and three address architectures. For the moment, we need only distinguish between two general classes of operand references: explicit reference,

which appear as distinct syllables within an instruction; and implicit references, which are defined by the instruction's format code.

We use a three letter mnemonic code to describe associations of implicit and explicit operands with at most two arguments and one result (binary order) . The first letter identifies the operand to be bound to the left argument of the operator (if any); the second letter identifies the operand to be bound to the right argument (if any); while the third letter identifies the operand to be bound to the result (if any). Seven letter designations are sufficient to describe all relevant possibilities:

1. "S", an implicit specification of the cell just above the top of the evaluation stack (value denoted by s).
2. "T", an implicit specification of the cell that was the top of the evaluation stack (value denoted by t).
3. "U", an implicit specification of the cell just below the top of the evaluation stack (value denoted by u).
4. "A", the first explicit operand specification appearing in an instruction (value denoted by a).
5. "B", the second explicit operand specification appearing in an instruction (value denoted by b) .
6. "C", the third explicit operand specification appearing in an instruction (value denoted by c).
7. " ", for null, meaning "not applicable" -- probably due to low functional order.

A use ordered analogue to the typical 360/370 instruction 'AR R1 R2' (meaning "add registers R1 and R2, and store the result in R1) would be written "ABA R1 R2 +" in this notation. A zero address DEL expansion for the same operation might appear as: "AS R1 :=; AS R2 :=;

UTU +; TA RI :=".

This notation also covers various hybrid formats that use both implicit and explicit references in a single instruction; for example, the use ordered hybrid instruction 'TAB X Y -' means "subtract the value of X from the value currently on top of the dynamic evaluation stack, store the result in Y, and decrement the stack pointer" (top of stack is always defined with reference to its state before interpreting the format in question).

It is easy to identify the characteristic formats for traditional zero (UTU), one (TAT), two (ABA), and three (ABC) address architectures using this system. The restrictive nature of these mono format DELs is clear in comparison to the 343 potential formats designations suggested by our three letter mnemonic.

The obvious implementation for all of the formats suggested by this identification scheme, however, would require 7*7*7 distinct interface routines and 9 bits per instruction (assuming a straight forward, n-way binary encoding). Even if the spatial cost were acceptable in the DEL program space the associated interface routines would occupy too great a fraction of micro store for most host machines. Consider the following rules for eliminating formats that are redundant with respect to our notion of transformational completeness.

1. Formats violating standard LIFO stack accessing conventions are not required (this would eliminate such formats as UAB, STU, ABU, etc.).

2. Only one ordering of T and U in the first two (argument) positions is needed--we use the UT ordering, which is consistent with a left to right, depth first post order taversal of the macro-tree representation of a program.

3. Formats that differ only by a permutation of explicit references are equivalent (e.g., ABC, ACB, BCA, BAC, CBA, and CAB are all equivalent; we choose the alphabetized element, ABC in this case).

4. Formats differing only by a permutation of the null designator, "_", in the first two (argument) positions are equivalent--we use formats with a leading null.

All of the above elimination rules can be applied without adversely affecting either the compilation or execution phase. Using these rules, the 343 element format universe suggested by our combinatoric identification rule can be reduced to 30 elements. The table below lists all distinct combinations remaining after these rules have been applied, grouped in order of increasing functional order.

The branches in a macro definition tree [3] may be thought of either as explicit references (if connected to a leaf node), or as implicit references (if connected to an ancestor node). This establishes a connection between format structure and the context of operator nodes in a macro definition tree. By inspection, at least one of the above formats is directly associated with each possible configuration of an ancestor node.

While we have reduced the spatial requirements of multi-format DEL structures to a practical order of magnitude, implementing all 30 formats listed in the table may still be prohibitive for some hosts. The following theorems identify some interesting subsets of this format universe.

Table of Potential Formats

MNEMONIC	TEMPLATE	SEMANTICS	STACK
___	<OP>	call op	
__S	<OP>	s := op	+1
__A	<x> <OP>	x := op	
_T_	<OP>	call op(t)	-1
_A_	<x> <OP>	call op(x)	
TT	<OP>	t := op(t)	
--AS	<x> <OP>	s := op(x)	+1
-TA	<x> <OP>	x := op(t)	-1
-AA	<x> <OP>	x := op(x)	
-AB	<x> <Y> <OP>	y := op(x)	
UT_	<OP>	call op(u,t)	-2
TT_	<OP>	call op(t,t)	-1
AT_	<x> <OP>	call op(x,t)	-1
TA_	<X> <OP>	call op(t,x)	-1
AA_	<x> <OP>	call op(x,x)	
AB_	<x> <Y> <OP>	call op(x,y)	
UTU	<OP>	u := op(u,t)	-1
TTT	<OP>	t := op(t,t)	
UTA	<x> <OP>	x := op(u,t)	-2
TTA	<x> <OP>	x := op(t,t)	-1
TAA	<x> <OP>	x := op(t,x)	-1
ATA	<X> <OP>	x := op(x,t)	-1
TAT	<X> <OP>	t := op(t,x)	
AAS	<X> <OP>	s := op(x,x)	+1
TAB	<x> <Y> <OP>	y := op(t,x)	-1
ATB	<x> <Y> <OP>	y := op(x,t)	
AAB	<x> <Y> <OP>	y := op(x,x)	
ABB	<x> <Y> <OP>	y := op(x,y)	
ABS	<x> <Y> <OP>	s := op(x,y)	+1
ABC	<x> <Y> <Z> <OP>	z := op(x,y)	

Theorem 1: The canonic interpretive form requirements can be satisfied using only eleven formats, up to the level of diadic operators, if "reverse" forms for all non-commutative operators are included in the set of action functions.

Proof: Consider the following DEL restrictions and interpreter coding conventions.

1. Semantic routines for monadic operators must increment the pointer to the top of the DEL evaluation stack before performing their normal processing.

c. "Reverse" forms for all non-commutative (diadic) operators must be included in the repertoire of DEL action functions.

Given these restrictions, we may eliminate all format codes whose mnemonic contains the "_" by using the binary format containing a "S", "T", or "U" in the same position, but which is otherwise identical (interpreter convention). Formats differing only by a reversal of the left and right argument binding (e.g., ABA and ABB) are redundant under the DEL restriction; only one element of each such pair is needed. Finally, no format whose code begins with "TT" can be generated by a naive compiler, since this would require recognition of the use of an intermediate value as a repeated argument.

The set {UTU, UTA, TAT, TAA, TAB, AAS, ABS, AAA, AAB, ABA, ABC) satisfies the theorem by inspection.

Theorem 1 demonstrates that the individual advantages of both stack and register oriented architectures can be merged at a gross cost of only four bits per instruction, which compares favorably with typical polish DELs (in which each instruction contains two form bits to distinguish between "push", "pop", "operate", and "literal"). For example, a single TAB format is equivalent to the polish sequence "push A, operate, pop B"; the first requires one instruction and four format bits, the second requires three instructions and six format bits.

Theorem 2: Only four formats are required if the DEL evaluation stack is eliminated.

Proof: The set {AAA, AAB, ABA, ABC} is sufficient, by inspection.

Compilation is somewhat more difficult in this case, however, since "dummy" variables must be synthesized in order to evaluate compound expressions. Although fewer bits would be needed to indicate the format code, it is likely that the space and time required during

execution would increase because of these extra explicit operand syllables.

Theorem 3: Only six formats are needed to satisfy all but the "unique variable" requirement of the canonic interpretive form.

Proof: The set {UTU, UTA, TAT, TAB, ABS, ABC) is sufficient, again by inspection.

It is difficult to determine whether or not execution phase time and space would increase or decrease if this reduced format set is used, however, since the question is sensitive to user behavior. The smaller format sets are interesting because of their coding compatibility with hosts strongly biased toward 8 bit storage quanta. If only two or three bits are needed to define the format of an instruction, then it is possible to combine both the format and operator code in a single byte.

PROCESS NAME SPACE -- GENERAL ISSUES

A name used by a process is a surrogate for a value. The set of all names that can be accessed by a process is the name space for that process. Source level names are usually just alphanumeric strings imbedded within a program text; DEL level names are operand identifiers appearing within executable instructions and host level names are simply addresses of accessable elements of the host storage hierarchy. Values are associated with names via a "contents map"--at any point during a computation, the contents of a name is its correct value. In this discussion, we are concerned only with the properties of names themselves, not with the form of identifiers for these names or the

problem of interpreting identifiers within an executable instruction; the contents mapping is assumed to be established externally--e.g., by a loader.

Name Space Synthesis

Providing a flexible and effective name space structure helps minimize the space and time requirements of a DEL. Good designs are characterized by both a simple correspondence between the source name space and the DEL name space (to simplify compilation and preserve transparency), and a simple correspondence between the DEL name space and the host name space (to maintain efficiency during execution).

High level language name spaces generally involve effectively unbounded ranges, one dimensional reference structures (viewing subscripted arrays and other qualified references as "expressions" rather than primitive symbols), and discrete granularity (i.e., reference structure does not induce a fixed relation between referands in the memory space). The identifiers used as references at this level are syntatically homogeneous, but semantically inhomogeneous--i.e., interpretation of the contents map for a referand depends on the context in which its reference appears. In particular, the referand associated with a particular source name may be different for different occurrences of that name. This is because the name space of most source programs is partitioned into distinct scopes of definition (or "scope" for short; intuitively, a scope is simply a natural grouping of references within which the association between references and referands is fixed, unless altered explicitly by dynamic allocation or

redefinition statements).

On the other hand, most host level name spaces are structurally inhomogeneous, being partitioned into register sets, storage modules, etc. References to elements in these partitions are rarely interchangeable within a host instruction. The association between references and referands is usually fixed at this level, however, even though it may be parameterized in terms of the current contents map (e.g., as in indexed or indirect referencing). Such discrepancies between the source and host name spaces account for much of the difficulty in synthesizing an effective DEL name space.

DEL organizations may be classified according to the placement of different portions of the information needed to bind reference to a referand (Chevance [2]). Data is characterized by three distinct pieces of information: type, locator, and value. The type of a referand defines the range of values it may assume; its locator defines the address to be used when accessing its contents; and its value is the bit pattern assigned by the current contents map, which must be interpreted according to its data type.

Incorporating locator information in the reference itself also leads to complications in handling changes in scope (e.g., storage management, passing parameters, and accessing externally defined referands). Perhaps the best known model for describing the effects of scope is the Contour Model developed in Johnson [12]. This model is rich enough to describe the address map transformations required by the allocation, release, and retention rules of most source languages,

and captures all practical methods of binding actual arguments to formal parameters as well and suggests itself as a good design base for DEL name spaces.

Environment and Contours

The notion of environment is fundamental not only to DELs but also to traditional machine languages as evidenced by widespread adoption of cache and virtual memory concepts. What is proposed for DELs is to recognize locality as an important property of a program name space and handle it explicitly under interpreter control. Thus, locality is transparent to the DEL name space but recognized and managed by the interpreter. Thus:

1. The DEL name space is homogeneous and uniform with an a priori unbounded range and variable resolution.

2. Operations, involving for example the composition of addresses which use registers, should not be present in the DEL code but should be part of the interpreter code only. Thus, the register name space and the interpreter name space are largely not part of the DEL name space and it is the function of the interpreter to optimize register allocation.

3. The environment locality will be defined by the higher level language for which this representation is created. In FORTRAN, for example, it would correspond to function or subroutine scope.

4. Unique to every environment is a scope which includes:

    i. a label contour,
    ii. an operand contour,
    iii. an operation table.

Following the Johnston model, we define a contour to be a vector (or table) of object descriptors. When an environment is invoked, a contour of label and variable addresses must be established (if not already present) in the interpretive storage. For a simple static

language like FORTRAN this creation can be done at load time. For languages that allow recursion , etc., the creation of the contour would be done before entering a new environment. An entry in the contour consists of the (main memory) address of the variable to be used; this is the full and complete DEL name space address. Type information and other descriptive details may also be included as part of the entry.

The environment must provide a pointer into the current contour, and must define the width of identifiers for labels and variables. Typically, the contour pointer and identifier width would be maintained in the register of the host machine. We denote identifier width by W and the pointer to the base of the current contour by EP; Figure 7 illustrates the process of referencing a DEL entity using this terminology. Both labels and variables may be indexed off the same environmental pointer. Subfields within DEL instructions, then, are actually containers for immediate values that define indices in the current contour; contour entries at the indexed location define the mapped address of the desired variable or label in the host name space. In other words, the operand identifiers within DEL instructions are simply contour indices that select a particular description for the image of a given source level object in the host name space.

The Contour Model differs from other high level architectures in that the function of references is separated from that of descriptors. References are one dimensional indices into a current declaration array, which we call the current contour. The current contour is

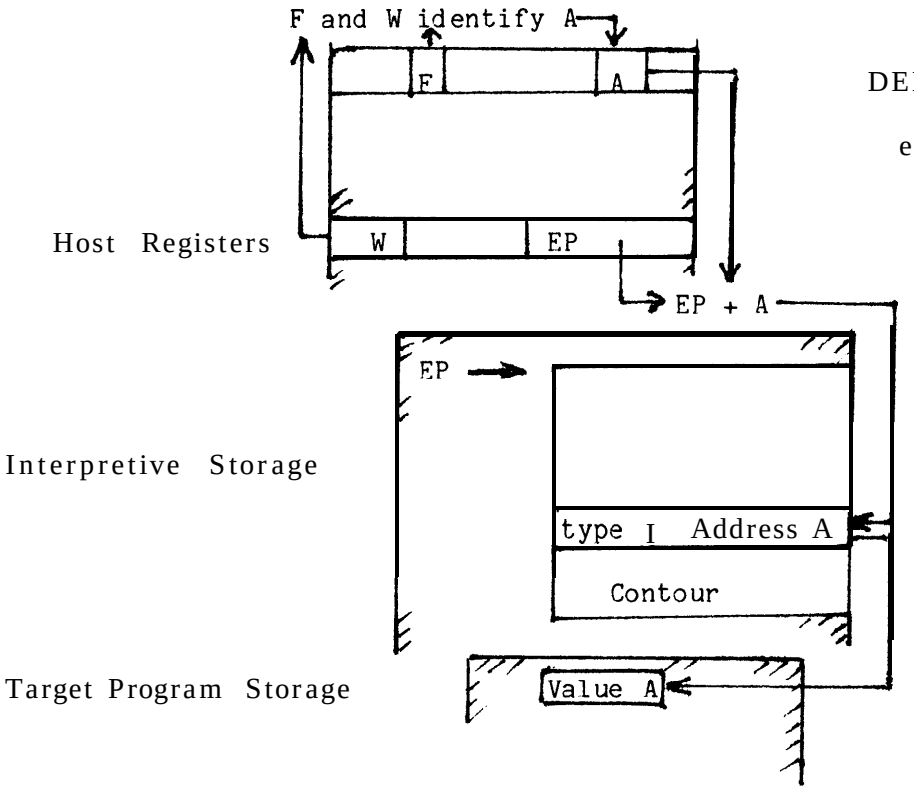


Figure 7: Referencing a DEL Variable

always maintained within the host micro store, and a new contour is created for each distinct incarnation of a source scope. Only W bits are used to represent a reference--where W is the smallest integer such that there are less than 2^W distinct referands in the current access environment .

Each contour is uniquely identified by an environment pointer that, at least logically, denotes its zeroth element. The environment pointer for the current contour is part of the DEL program state vector, and must be saved/restored when entering/leaving a scope of definition. The address map is computed by adding the reference code to the current environment pointer, and then accessing the appropriate

referand descriptor (Figure 8) :

```
descriptor ( reference N ) = micro store ( ep + N )
value ( reference N ) = main store ( descriptor N )
```

Figure 8: Normal DEL Addressing Structure

This analysis can be extended by noting that the logical type of a referand (integer, floating point, logical, or character) can be separated from its physical type (single, double or varying percision). We refer to the physical type as "shape". Elements of contours are descriptors, each of which is itself a vector that defines the shape, type, and locator of a particular DEL entity--or, more precisely, the algorithm used to access that entity. Distinguishing shape within the descriptor allows us to use semantic routines designed for the general case, rather than having one per type:shape combination.

Given a fully static source language (like BASIC or FORTRAN) a unique contour for each distinct scope of definition may be preallocated during compilation. In this case, only the descriptors for formal parameters need be modified during execution. For most source languages, however, a new contour will have to be created each time a new scope is entered; particularly if the source language supports recursive procedure invocation.

Since the header entries need be evaluated only once per contour creation, if procedure entry is infrequent in an HLL it can be relatively complex and difficult to evaluate. However, this factors out the common calculations needed to compute effective addresses; there will be a substantial time savings whenever variables are accessed repeatedly within a contour, and the possibility of a time loss when variables are not accessed at all. The penalty can be avoided by marking descriptors in the current contour as "unbound" until they are actually referenced. Each time a DEL reference is processed, its descriptor must be checked for validity; this usually means that some form of hardware support is required for this stratagem to work efficiently.

The contour technique is easily adapted to most existing parameter passing conventions. Parameters may be passed "by reference" simply by copying the appropriate descriptors from the caller's contour into the callee's contour. Parameters are passed "by value" by initializing a variable created either in the caller's environment (call by copy value), or in the callee's environment (call by value copy), with the value of the argument referand in the caller's contour. "By name" parameter passing involves moving an IP:EP pair into the appropriate descriptor in the callee contour; the IP:EP, where IP is an instruction pointer into the time invariant algorithm, and EP is an environment pointer identifying a particular access environment. No transformation identified by the IP can depend upon or alter the contents of a memory cell unless that cell is in the address mapping

image of the current access environment.

Operation Contours

Each verb or operation in the higher level language identifies a corresponding interpretive operator in the DEL program representation (exclude for the moment control actions which will be discussed shortly). The routines for interpreting all familiar operations are expected to lie in interpretive storage. Certain unusual operations, such as transcendental functions, may not always be contained in the interpretive storage. A pointer to an operator translation table must be part of the environment; the actual operations used are indicated by a small index container off this pointer (Figure 9). The table is also present in the interpretive storage. For simple languages, this latter step is probably unnecessary since the total number of operations may be easily contained in, for example, a six bit field and the saving in DEL program representation may not justify the added interpretive step.

DELTRAN

Deltran is an intermediate language, described in the appendix, tailored to a FORTRAN source language EMMY host machine, and typical community of scientific programmers. Its design is intended to minimize execution time and space, subject to the limitations imposed by a one pass compilation that performs only single statement optimization. Our primary objective in synthesizing this language is to demonstrate the practicality of the DEL design principles, rather than to advance the state of the art in FORTRAN execution.

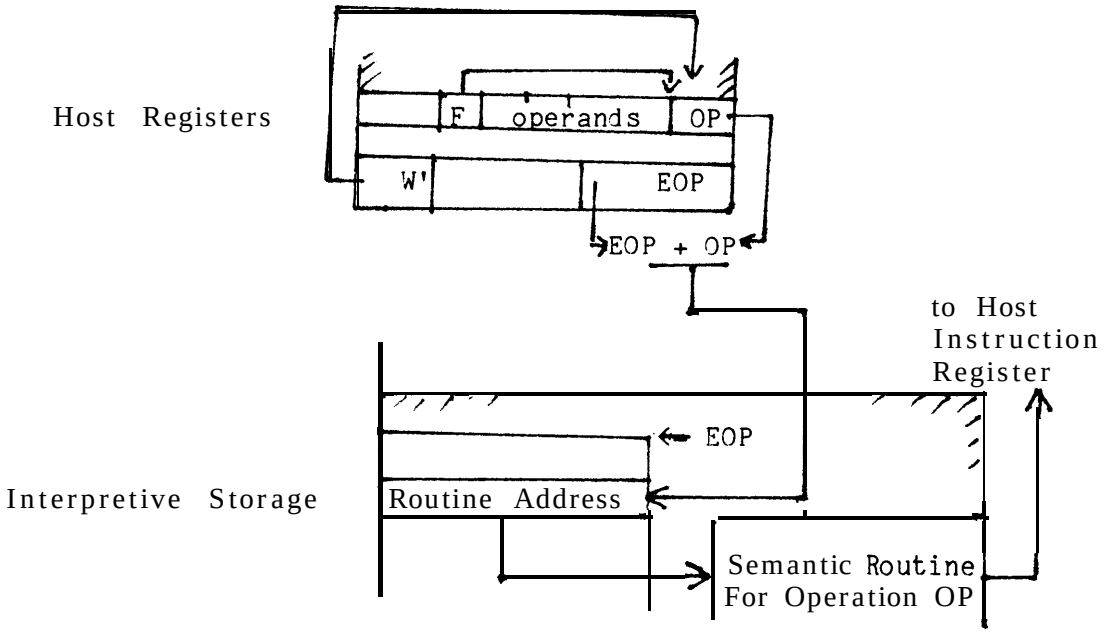


Figure 9: Referencing a DEL Operator

The DELtran as described in the appendix was implemented at the Stanford Emulation Laboratory in early 1977. The compiler as developed later is actually a one plus one pass process. One pass in required for the translation while another is required for completion of the symbol table. The compiler is non optimizing in the sense that it does not attempt to rearrange the source code (i.e. factor statements out of DO loops, etc.). Of course, optimization in the sense of register assignment for allocation is meaningless in the DEL construct.

The DELtran interpreter consists of 800 32 bit words excluding the trignometric functions and I/O. The trig functions are brought in separately as required from main memory. This resulting size compares favorably to implementations of PDP-11 and System 360 emulators. The PDP-11 emulator consists of 1200 words while the System 360 consists

of 2100 words of microstorage (all excluding I/O). The 800 words of interpreter leaves over 300 words available to handle scope entries as defined in the preceding section. In view of the limited vocabulary in FORTRAN II, operation tables were not employed hence only labels and operand entries are required in scope tables. FORTRAN is a static language (no recursion possible) ; scopes, then can be created at load time up to 3000 variable names. Beyond 3000 variables the compiler would either have to be modified or the program partitioned.

For a variety of different sample program material we achieve a static code as reduction of between 4 to 1 and 10 to 1. The better code compaction coming in problems that have a large number of array operations. This code compaction excludes both prologue and eiplogue information in traditional machine code as well as scope information in DELtran. The ratio of header information to scope information seems quite variable and while the ratio is in DELtran's favor, it is not as formidable as the static code size itself.

In dynamic code DELtran interprets between 1/3 and 1/5th the number of instructions as would be required by a conventional machine organization. These numbers also exclude a scope creation time much of which could be overlapped with a suitable host in DELtran. For a relatively unbiased host system such as MMY, the time to interpret an instruction unit is approximately twice as fast with DELtran when compared to either a System 360 or PDP-11 emulation on EMMY. Of course, EMMY would be considerably slower in interpreting image machines whose data paths exceed 32 bits and hence would not represent a fair compar-

ison. The reason for DELtran's improvement over traditional machines is rather simple. Traditional image machines represent the frozen environment. In DELtran one can make compromises in creation of the intermediate form and in recognition of the features of the host. The net effect of minimizing the number of instructions to be interpreted and in improving the interpretation time is a program execution improvement factor of between 6 and 10 to 1 when compared to the emulation of a traditional program representation of the same program on our host system. Of course, hosts dedicated to a particular image machine will naturally run faster than EMMY emulating that system. Thus, the actual time improvement ratio is somewhat less clear. Just as it is possible to build a host dedicated to a traditional image machine it would also be possible to build a host dedicated to DELtran. We would expect such a host to interpret DELtran instructions at least at the rate that traditional machine instructions could be processed. Such DEL tailored hosts are subjects of continuing research and evaluation at Stanford. A number of other open questions remain. A more comprehensive evaluation of the header vs scope creation philosophy is underway. In addition, dynamic languages are being studied such as ALGOL and PASCAL. Actually the run time differences between languages are not nearly so great as at precompiled time. However, dynamic languages, i .e. languages which support recursion, cannot use, of course, a scope creation at load time and must in turn create scope on entry.

CONCLUSIONS

At least for most familiar high level languages our synthesis theory seems to be adequate and comprehensive enough to allow the careful DEL implementor to achieve something close to the "canonic form" measures described in our companion paper. Indeed DELtran can, with proper host support, achieve almost all measures missing only on program size due to the required 5 bit format syllable included in each instruction.

FORTRAN, being an especially simple and static language, does not really tax the synthesis theory. DELs for dynamic languages (which support recursion) are being implemented at the Stanford Emulation Laboratory. Preliminary indications are that, when compared to conventional architectures, the improvement factors in more complex language may be expanded over those achieved in FORTRAN. This is due to the relatively complex code required on most traditional machines, whereas the DEL program representation is largely little changed from that which was described herein.

Architectural synthesis, like any design process, is a series of tradeoffs and compromises between user behavior and host characteristics. Thus, of necessity, the strategum, algorithms and design example included here may assume different relative weightings in other environments.

APPENDIX

DELTRAN: A CASE STUDY IN A FORTRAN DEL

As noted previously, the primary purpose of developing a FORTRAN DEL is described in the application of DEL design principles. We limited the magnitude of our task by addressing only a subset of the full FORTRAN language (Basic FORTRAN), and ignoring a number of questions relating to a production environment such as higher level data, task, and job management. The resulting design does not preclude extension to features like named COMMON, additional data types and structures, or random access external files, multiple named COMMON blocks, complex variables, logical variables, relational operators. The instruction unit structure and operand referencing mechanism described below should be compatible with the modifications needed to capture the full FORTRAN language.

Source Influence

The FORTRAN subset of interest here is usually referred to as Basic FORTRAN (Heising [8]). The adjective "basic" is not applied lightly; it is indeed a rudimentary programming language. This turns to our advantage, however, by holding the size of the design problem within reason. Some assumed source language features and restrictions affecting the design of DELtran are:

(1) Its name space is entirely static, except for the binding of actual arguments to formal parameters.

(2) The natural range of scope of definition is a procedure specification (i.e., SUBROUTINE or FUNCTION block).

(3) Few primitive data types are needed (e.g., only single and double precision forms of fixed and floating point numbers).

(4) Unstructured program control is permitted (i.e., DO loops need not be one-in one-out control structures).

(5) Parameters are uniformly passed "by reference", although this is equiv al ent to "by copy va1ue" when expressions are used as actual arguments (this is not required by the standard, but follows the long established IBM tradition).

These observations are extracted from the preliminary ANS specifications for FORTRAN vs Basic FORTRAN [1]. Immediate implications are: recursive procedure invocation need not be supported; both global and local storage can be statically allocated during compilation; all type checking can be performed during compilation (ignoring parameters to procedures, as is conventional); and program flow analysis can involve arbitrarily complex constructs.

Host Influence

The basic architecture of the EMMY host and its surrounding laboratory environment are described in Neuhauser [16] and [17]. In general EMMY is a microprogrammable "universal host" with a 200 ns. micro store and an 800 ns. main store (50 and 400 ns. access times respectively). Both stores are 32 bits wide; 4K words of read/write micro store and 16K words of main store were available during the development of DELtran.

User Influence

The intended user community is assumed to be composed of general purpose scientific programmers. User characteristics most relevant to the design of DELtran are:

(1) About half the statements in a typical source program deal with program control, and about half are assignment statements (Wichman [21] and Lunde [14]).

(2) The single, most frequent type of statement is "A = B", followed at some distance by "A = A + B" (Knuth [13]).

(3) DO statements almost always use an implicit increment (stepping value of one (Knuth [13]).

(4) Three distinct branches are usually specified for the arithmetic if statement (implied by the distribution of branch statements noted in Flynn [4]).

While these assumptions appear applicable to a variety of user communities and source languages, specific programs could deviate from the implied statistical distribution of operators, names, etc. A more detailed behavioral model could, of course, be extracted from installation-specific trace-tape data.

General Description

Due to the sequential nature of FORTRAN, both at the source and machine code level, a linear sequencing rule is used. The natural scope of definition for source level identifiers is the program or subprogram -- i .e., MAIN, SUBROUTINE, or FUNCTION blocks. Indeed, the lack of any other structured control units leaves little choice in this matter, especially in light of our intent to minimize compilation

complexity.

Individual DELtran instruction units are broken down into independently encoded syllables. Three classes of syllables were required: operand syllables, which denote DELtran variables (or labels); operator syllables, which denote transformation rules to be applied to the DELtran data store; and format syllables, which denote initializations to be performed in preparation for a deferred operator syllable.

Word boundaries may be crossed immediately before or immediately after either operator or format syllables: i.e., sequences of operand syllables must lie within a single word. (operand lists for n-ary immediate operators such as CALL, READ, and WRITE excepted). These syllables may be combined in three general syntatic sequences to form DELtran instruction units:

Leading Operator: <OP> [<A> [<B> C.. .]]]

Leading Format: <F> [<A> [...] 1 <OP>

Compound: <F> [<A> [...1] <OP> [<D> [...] 1

Leading operator forms generally deal with program control, involving functions that do not fit well within the familiar molds of binary or unary operators (the leading MOVE operator is an exception; it is coded in this form because of its high frequency of occurrance). The leading format construction factors out the operand decode and fetch computations required by common operator functionalities: diadic (two arguments, one result); monadic (one argument, one result), and onadic (no argument, no result). The compound form is used only with a few high order operators, or with array access primitives that require

information about explicit operand references not provided by the standard format interface. The normal sequence of interpretation is for leading format constructions:

Decode leading syllable -- extract 5 bit leading syllable from the current instruction word (IW); and transfer control to the appropriate interface routine.

Form interface -- extract all W bit operand reference syllables; fetch values of arguments into registers; compute address of result, if any.

Decode operator -- extract operator code, and transfer to appropriate semantic routine.

Execute -- compute designated transformation ; store result, if any; and begin another cycle of interpretation.

The mechanism for communicating information between interface and semantic routines consists of three host registers: P, Q, and R. For binary formats, P will contain the value of the left argument, Q the value of the right argument, and R the address of the result. Lower functionality requirements are derived from this standard interface by deleting specifications. In the unary case, for example, Q contains the value of the only (and hence still right most) argument, and R the result address.

This "PQ" interface has meaning only within the interpretation of a leading format of an instruction unit. Some residual control infro-mat ion, called the DEL program state vector, must be maintained across instruction interpretations, however. The internal DELtran program

state is defined by:

(1) Instruction Word (IW) : a buffer for the DELtran instruction stream.

(2) Instruction Pointer (IP): a pointer to the next word of instruction units in the DELtran program store.

(3) Control Pointer (CP): a pointer to a linear definition table for all accessable labels, variables, constants, etc.

(4) Stack Pointer (SP): a pointer to the top of a dynamic evaluation stack.

(5) Syllable Width (W): A specification for the number of bits in an operand reference syllable.

(6) Evaluation Stack (ES): A LIFO queue containing the results of intermediate computations.

Five of these six entities are encoded in three micro registers; the current instruction word is kept in micro register I, and the current instruction pointer is kept in micro register IP. The control pointer CP, the current stack pointer SP, and the current syllable width W are all encoded in a single micro register S:

This assignment leaves four micro registers available for general use. Three of these (P, Q, and R) are temporarily dedicated to the "PQR" interface when interpreting leading format or compound instruction forms; but may be reassigned when the standard interface is not required. The remaining micro register, X, is used for general purpose indexing and scratch storage.

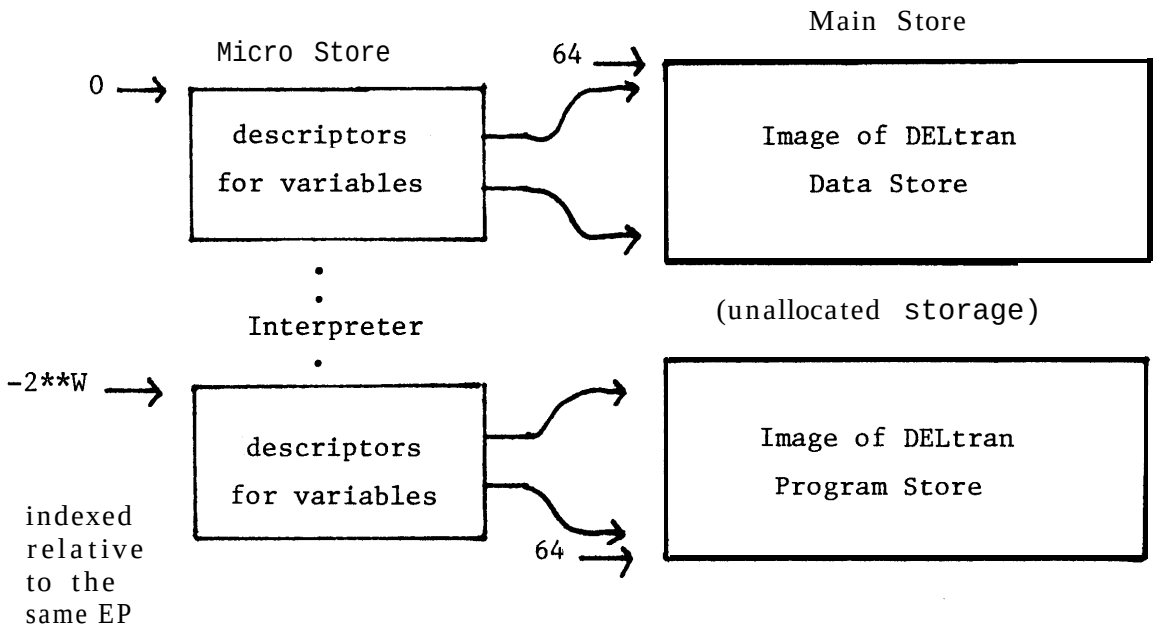
The association between DELtran operand references and referands in the data or program stores is defined by a single linear table called the current contour. Each element of this table, called a descriptor, contains two pieces of information -- a shape and a locator. Shape specifiers (high 8 bits) define an entity's size, justification, and the granularity of type in the classical sense. Locators (low 24 bits) are directly the address of a referand in EMMY's main store.

The current contour is physically divided into two parts: a data table located at the bottom of micro store; and a label table located at the top of micro store. Since the current contour is always located in a fixed position, a dynamic environment pointer (i.e., the ep in Johnston's Contour Model [12]), is not required -- the control pointer serves as an environment pointer for CALL and RETURN, but is not normally used to interpret DELtran reference codes.

Because it is possible to distinguish between references to variables and references to labels syntactically (for the given FORTRAN source language), judicious placement of descriptors can reduce the number of bits required in operand syllables. An operand reference code N denotes the descriptor at location N if it refers to a variable, and the descriptor at location -2**W+N if it refers to a label -- where W is the number of bits in an operand reference, and micro store is treated as a circularly addressable memory. This means that W may in fact be the least integer such that there are less than 2**W distinct labels and less than 2**W distinct variables, rather than the

least interger such that there are less than 2**W distinct entities (both labels and variables) in a given scope of definition. This addressing scheme is illustrated below.

DELtran Reference Structure



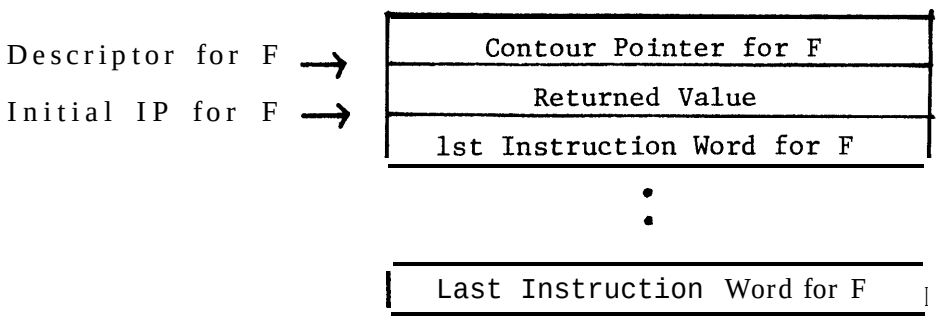
This figure also illustrate3 the general layout of DELtran pro-gram3 in EMMY's main store; with COMMON and LOCAL storage allocated

just above the 64 word evaluation stack, and program modules allocated at the upper end of main store. If more than one procedure is included in a module, COMMON is extended toward the higher addresses and LOCAL for the n+lth procedure is allocated just above that for the n-th procedure (MAIN is the 1st procedure). The actual bodies and skeletal contours for procedures are allocated beginning at the high end of main store and moving toward the lower addresses.

The current contour is initialized by the CALL and RETURN operators from skeletal contours pre-allocated during compilation. There is one skeletal contour for each separate scope of definition ; i.e., for each SUBROUTINE or FUNCTION (including MAIN). Each skeletal contour consists of a label definition table, linkage area, and a data definition table:

The table-of-contents word defines the number of formal parameters, dynamic (overlay) variables, static variables, and label descriptors for the associated block. The **Caller's ..." words in the linkage contain the DELtran program state vector elements that must be restored upon encountering a RETURN instruction. Skeletal contours are themselves identified by the "-1 th" word of a DELtran module; the "0th" word contains the returned value, if it is a FUNCTION; while the "1st" word is the actual beginning of the executable code for the module :

Letting the descriptor for a FUNCTION module identify the referand of its returned value, as well as its entry point, helps to minimize the number of distinct entities in a given scope of definition.



Layout of a DELtran Module (F)

Syllable Descriptions

All DELtran instruction units begin with a 5 bit leading syllable. The 32 distinct codes for this key syllable specify either an immediate operation or a format that describes the preliminary processing

required to establish the standard interface for a deferred operator.

Five Bit Lead Syllable Encoding

Code	Immediate Syntax	Immediate Semantics
00000	FETCH	fetch next instruction
10000	MOVE <A><B>	b := a
01000	_TT <OP>	t := OP(t)
11000	_AB <A> <B> <OP>	b := OP(a)
00100	_TA <A> <OP>	a := OP(t)
01100	-AS <A> <OP>	s := OP(a)
10100	_AA <A> <OP>	a := OP(a)
11100	___ <OP>	execute OP
00010	UTU <OP>	u := OP(u,t)
00110	UTA <OP>	a := OP(u,t)
01010	ATT <A> <OP>	t := OP(a,t)
01110	TAT <A> <OP>	t := OP(t,a)
10010	ABS <A> <B> <OP>	s := OP(a,b)
10110	ABC <A> <B> <C> <OP>	c := OP(a,b)
11010	TAB <A> <B> <OP>	b := OP(t,a)
11110	ATB <A> <B> <OP>	b := OP(a,t)
00001	ABA <A> <B> <OP>	a := OP(a,b)
00011	ABB <A> <B> <OP>	b := OP(a,b)
00101	ATA <A> <OP>	a := OP(a,t)
00111	TAA <A> <OP>	a := OP(t,a)
01001	AAS <A> <OP>	s := OP(a,a)
01011	AAB <A> <B> <OP>	b := OP(a,a)
01101	AAA <A> <OP>	a := OP(a,a)
01111	CALL n <F> <A1> . . . <An>	invoke F(A1, ..., An)
10001	RETURN	return from invocation
10011	GO <L>	goto l
10101	CGO <I> <L>	goto (l+i-1)
10111	IFE <E> <L>	goto (l+(e=0)+2*(e>0))
11001	IFT <L>	goto (l+(t=0)+2*(t>0))
11011	END0 <N> <I> <M> <L>	goto l if n=n+i < m
11101	END1 <N> <M> <L>	goto l if n=n+1 < m
11111	BREAK	trap to monitor function

In practice, lead syllable codes are extracted from the residual instruction word register (I) using the double shift technique, and then added to the microprogram counter ($) to effect an indexed branch.

Instruction units in the table following the extraction may perform useful computations as well as transfer microprogram control to the remaining body of the appropriate routine (due to the semihorizontal nature of EMMY's native language).

The DELtran format set permits full exploitation of repeated operands (either as arguments alone, or in combination with the result specification), and is tranformationally complete in the sense that any binding of explicit operands (i.e., primitive variables) and implicit operands (i.e., stack elements) can be generated by local combinatorial analysis of the FORTRAN source code. Note also that deferred operators are partitioned into disjoint classes according to their functionality by the inclusion of distinct, binary, unary, and nullary formats; and that reverse forms of deferred operators are not needed, since all required argument permutations are contained in the format set.

The MOVE operator simply transfers the value of the referand identifed by operand reference <A> to the referand identified by operand reference <B>. Simple program control operators such as GO, CGO, IFT, and IFE cause the current instruction word register (I) to be reloaded from the bit address in DELtran's program store identified by the appropriate label descriptor. The single label reference appearing in

the CGO, IFT and IFE constructs is actually the first entry in a subtable of the current contour; the data dependent index into this table is determined by the semantic routine for each of theses operators.

END0 and END1 operators cause the value identified by <N> to be incremented and then execute a GO <L> if the result is less than or equal to <M>. The increment value is assumed to be one for the END1 operator, but is explicitly denoted by <I> for the more general END0 operator. Breaking out the special case of END1 is indicated not only by the default specification rule for FORTRAN looping constructs, but also by empirical user statistics (Knuth [9]).

CALL and RETURN operators are somewhat more complicated, since they involve modification of the internal state of the DELtran executor. CALL causes the volatile portion of the current contour to be paged out to its static image, which is identified by the control pointer CP. The instruction pointer, instruction word, and status registers (IP, I, and S) are also saved in a linkage area within this skeletal contour, thus saving the DELtran program status vector and, hence, all information needed to resume the caller's process. The skeletal contour for the callee is then moved into the current contour, and descriptors for formal parameters copied into the appropriate locations. The TP is set to point to the first word of the callee's program body, the first instruction word is fetched, and the state register S is loaded with the callee's syllable width and control pointer.

RETURN simply undoes a CALL. Only those descriptor elements in the original caller's skeleton contour which were overlayed during the CALL operation need be restored, however, these are easy to determine by comparing the upper and lower reference indi.ce bounds for both programs, which are stored in a linkage area in their skeleton contours . We save and restore the contents of the caller's old instruction word register to avoid wasting static space in the DELtran program store ; the time required to perform this linkage is greater than that which would be required simply to fetch a new instruction word from the program store.

Operand Syllable

As noted above, the width of operand syllables varies from one scope of definition to another. The current number of bits in an operand syllable, W, is maintained in the low order six bits of the DELtran secondary state register, S, which is automatically saved and restored by the execution semantics for CALL and RETURN. For short subroutines or functions, only three or four bits are needed to identify a unique variable; in larger modules, however, six to eight bits may be needed.

The map from reference codes to descriptors for DELtran variables is simple and direct: the descriptor for variable with references code N is located at address N in micro store. It is possible to extract an operand reference and look up the corresponding descriptor in a single EMMY instruction.

Deferred Operator Syllables

Deferred operators are categorized as diadic (two arguments, one result), modadic (one argument, one result), or onadic (no argument, no results). Data types are not checked dynamically because FORTRAN is such a strongly typed language in its own right, and hence, distinct operator codes are used to denote integer and floating functions. Some collapsing of the DEL operator set was possible where only the sign of an operand or equivalence to zero need be checked, as with the IF statement, since these representations are the same for both fixed and floating point (internal value representation consistent with the 370 architecture has been used for pragmatic reasons; see Wallach [20]).

The -A2- operators are perhaps not self defining; in general, the two argument values in the P and Q interface registers to be treated as the first and second subscripts for the array whose descriptor will be in the result register, R. A2E causes the effective address of the indicated array element to be computed, creates a descriptor to this element by combining the shape field from the array descriptor with this address, and stores the result in the contour slot for the deferred reference code D. MA2 and A2M operators work in a similar fashion, but actually cause a state transformation in the DELtran data space -- they are similar to the MOVE operator. TA2 and A2S are "push" and "pop" operators that transfer values between the evaluation stack and array elements.

Bounds checking is not performed, following with the tradition established by IBM. It would be easy to incorporate by modifying the

Four Bit Encoding of Diadic Operators

Code	Deferred Syntax	Deferred Semantics
0000	FETCH	fetch the next instruction word
1000	A2E <D>	associate D with (p,q)-th element of r
0100	F+	r := p+q (floating add)
1100	I+	r := p+q (integer add)
0010	F-	r := p-q (floating subtract)
0110	I-	r := p-q (integer subtract)
1010	F*	r := p*q (floating multiply)
1110	I*	r := p*q (integer multiply)
0001	-A2-	prefix for array accessing operators
" 00	MA2 <D>	r(p,q) := d
" 01	A2M <D>	d := r(p,q)
" 10	TA2	r(p,q) := t
" 11	A2S	s := r(p,q)
0011	F/	r := p/q (floating divide)
0101	I/	r := p/q (integer divide)
0111	F^F	r := p**q (floating to floating power)
1001	I^I	r := p**q (integer to integer power)
1011	FST	r := sgn(p)*q (floating sign transfer)
1101	IST	r := sgn(p)*q (integer sign transfer)
1111	BREAK	trap to monitor

appropriate array accessing routines, and would not involve a high space or time penalty for the MMY host. The multiplier needed to compute the effective address of an indexed array element is stored at the "base" of the array (i.e., is its zero-th element; this works for FORTRAN since array subscripts must begin with one).

Four Bit Encoding of Modadic Operators

Code	Deferred Syntax	Deferred Semantics
0000	FETCH	fetch new instruction word
1000	AlE <D>	associate reference code D with r(p)
0100	FLOAT	r := float(p)
1100	FIX	r := fix(p)
0010	F~	r := -p (floating negate)
0110	I~	r := -p (integer negate)
1010	LOG	r := log(p) (logarithm)
1110	SIN	r := sin(p) (sine)
0001	-Al-	prefix for array accessing operators
" 00	MAl <D>	r(p) := d
" 01	AlM <D>	d := r(p)
" 10	TAl	r(p) := t
" 11	AlS	s := r(p)
0011	COS	r := cos(p) (cosine)
0101	TANH	r := tanh(p) (hyperbolic tangent)
0111	PAUSE	pause with code p
1001	STOP	stop with code p
1011	TIME	r := (current time)-p
1101	not used	
1111	BREAK	trap to nonitor

Three Bit Encoding of Onadic Operators

Code	Deferred Syntax	Deferred Semantics
000	FETCH	fetch next instruction word
100	SET <U> <F>	set Unit = U and Format = F
010	READ n <D1>...<Dn>	input to D1...Dn as per Unit/Format
110	WRITE n <D1>...<Dn>	output from D1...Dn as per Unit/Format
001	REWIND	rewind Unit
011	BACKSPACE	backspace Unit
101	ENDFILE	write end-of-file mark on Unit
111	BREAK	trap to monitor

Compound instruction units of the form " <onadic OP> ..." are really nothing more than a partial frequency encoding of infrequent and/or difficult to handle functions, the bulk of which deal with input out put. Two residual control cells, Unit and Format, are used to maintain the status of I/O operations. Unit corresponds to a logical designation of a specific file/device/channel combination, and would in practice be bound by a surrounding operating system as specified by some external job control language. The Format cell is merely a byte pointer into a string of field specifications produced during

compilation from the appropriate FORTRAN format statement.

Although the full I/O structure indicated above are not implemented the intent is that it should proceed as a subinterpretation, either the EMMY performing conversions under control of the current Format, or with the control device for Unit performing these conversions asynchronously. The Unit and Format residual control cells are, respectively, the environment pointer and instruction pointer for this subinterpretation. An entire byte is used to encode formatted field specifications simply to keep this process as simple as possible; the spatial penalty is low since I/O statements are statically insignificant.

References

[1] American Standards Association Sectional Committee X3, Computers and Information (R. V. Smith, ed.), "FORTRAN vs. Basic FORTRAN -- A Programming Language for Information Processing on Automatic Data Processing Systems," Communications of the ACM, Vol. 7, No. 10, October 1964, pp. 591-625.

[2] Chevance, R. J., "Design of High Level Language Oriented Processors," SIGPLAN Notices, Vol. 12, No. 1, January 1977, pp. 40-51.

[3] Elson, M., and Rake, S. T., "Code-Generation Techniques for Large- Language Compilers," IRM Systems Journal, Vol. 9, No. 3, 1970, pp. 166-M.

[4] Flynn, Michael J., "Trends and Problems in Computer Organizations," IFIPS Congress, Stockholm, Sweden, August 1974, North Holland Publishing Company, 195 pp. 2-10.

[5] Flynn, Michael J., and Hoevel Lee W., "Interpretive Architectures: A Theory of Ideal Lanaguage Machines," Computer Systems Laboratory Technical Report 170, February 1979, Stanford University, Stanford, California.

[6] Flynn, Michael J., and Henderson, D. S.., "Variable Field-Length Data Manipulation in a Fixed Word-Length Memory," IEEE Transactions on Electronic Computers, Vol. EC-12, No. 5, October 1963, pp. 512-17.

[7] Foster, C. C. and Gonter, R. H., "Conditional Interpretation of Operation Codes," IEEE Transactions on Computers, Vol. C-20, No. 1, January 1971, pp. 108-11.

[8] Heising, W. P., "History and Summary of FORTRAN Standardization Development for the ASA," Communications of the ACM, Vol. 7, No. 10, October 1964, p. 590.

[9] Hoevel, Lee W., "DELtran Principles of Operation," Digital Systems Laboratory, Technical Note No. 108, Stanford University, Stanford, California, March 1977.

[10] Hoevel, Lee W. and Wallach, Walter A., "A Tale of Three Emulators," Technical Report No. 98, Digital Systems Laboratory, Stanford University, Stanford, California, October 1975.

[11] Huffman, D. A., "A Method for the Construction of Minimum Redundancy Codes," IRE, Vol. 40, No. 9, September 1952, pp. 1098-101.

[12] Johnston, John B., "The Contour Model of Block Structured Processes," Proceedings of the SDSPL (SIGPLAN Notices, Vol. 6), February 1971, pp. 55-82. --

[13] Knuth, D. E., "An Empirical Study of FORTRAN Programs," Software -- Practices and Experience, Vol. 1, 1971, pp. 105-33.

[14] Lunde, A., "Empirical Evaluation of Some Features of Instruction Set Processor Architectures," Communications of the ACM, Vol. 20, No. 3, March 1977, pp. 143-52.

[15] McCarthy, J., et al., LISP 1.5 Programmer's Manual, MIT Press, Cambridge, Massachusetts, 1965.

[16] Neuhauser, Charles J., "System Description of the JHU Emulation Laboratory and Host Machine," Proceedings of the 7th Annual Workshop on Microprogramming (SIGMICRO 7), September 1974, pp. 28-33.

[17] Neuhauser, Charles J., "An Emulation Oriented, Dynamic Microprogrammable Processor (Version 3)," Technical Note No. 65, Digital Systems Laboratory, Stanford University, Stanford, California, October 1975.

[18] Standish, T. A., "A Preliminary Sketch of a Polymorphic Programming Language," Centro de Calculo Electronico, Universidad Nacional Autonoma de Mexico, July 1968.

[19] Sweet, Richard Eric, Empirical Estimates of Program Entropy, Ph.D. Dissertation, Department of Computer Science, Stanford University, Stanford, California, December 1975.

[20] Wallach, Walter A., "EMMY/360 Cross Assembler," Technical Note No. 74, Digital Systems Laboratory, Stanford University, Stanford,California, June 1976.

[21] Weber, Helmut, "A Microprogrammed Implementation of EULER on IBM System/360 Model 30," Communicatins of the ACM, Vol. 10, No. 9, September 1967, pp. 549-58.

[22] Wichman, B. A., "Five Algol Compilers," Computer Journal Vol, 15, No. 1, January 1972, pp. 579-86.