On Instruction Sets and Their Formats

MICHAEL J. FLYNN, JOHN D. JOHNSON, AND SCOTT P. WAKEFIELD, MEMBER, IEEE

Abstract — **Central to instruction set design is the issue of the instruction format. We discuss some common format encoding techniques, and introduce a method of representing high-level language parse trees by means of formats that cover successively larger portions of a tree. We then introduce variations on the method that represent directed acyclic graphs as well as simple parse trees, and that encode constants in a special fashion. For a particular representation, we measure the number of times each format is executed to run a sample program to completion.**

Index Terms — **Computer architecture, execution architecture, instruction format, instruction set, program representation.**

I. Introduction

OVER the years there has been significant attention and interest in the literature in the design of instruction sets. Central to instruction set design is the issue of the instruction *format*. Indeed, the instruction format plays a fundamental role in determining the basic attributes of a resultant instruction set design. Yet, the meaning of *format* as used in instruction set design is unclear. It has been used to distinguish the number of explicit operands that appear in an instruction (4 address, 3 address, 2 address, etc.) as in Bell and Newell [1]. But even within a format classification, as in the case of the 2 address formats of the IBM System 370, size and syllable partitioning is used to distinguish basic instruction formats (e.g., $RX, RR, SS, SI, \cdots$). The purpose of this paper is to explore the role of formats in instruction set design. As the format provides key information to the interpreter-executor of the program, the representation of this information determines a number of important execution parameters. Even the recent controversy concerning minimum instruction sets versus complex instruction sets underscores the role of the format in determining implementation cost and speed.

In this paper we will first look at the meaning and role of the format syllable in an instruction. Then we will consider the universe of information that the instruction format can provide to the processor. Finally, we present some usage data on a sample program which could give guidance in selecting a reasonable format set.

Manuscript received September 26, 1983; revised June 8, 1984. This work was supported in part by the U.S. Army Research Office under Contract DAAG29-82-K-0109, using facilities supported by NASA under Contract NAGW 419.

M. J. Flynn and J. D. Johnson are with the Computer Systems Laboratory, Department of Electrical Engineering, Stanford University, Stanford, CA 94305.

S. Wakefield was with Stanford University, Stanford, CA 94305. He is now with the IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598.

II. The Instruction Set

The instruction is a specification of logical entities consisting of a format, an operation, a number of source operands, a result operand, and a next instruction location. Specification can be done in several ways.

1) By coordinate address;
2) by implication;
3) immediately; or
4) by association (by partial record).

Practically speaking, most object specification is done by either coordinate address or implication. Occasionally, specialized information such as shift amount is specified immediately. The format and operation are specified by codes which provide information to the processor control unit. For many familiar organizations, the opcode is an encoding of both format and operation information to be used during the execution of the instruction. For our purposes the operation simply refers to the semantics of the transformation specified in the instruction. The format then specifies everything else — the combination is referred to as the opcode.

III. The Format

The format provides two types of information: template information and transformation information (Fig. 1). While these two roles can be separated, in most instruction sets the format is the object designated to supply both template and transformational information.

The template identifies the starting point and size of each explicit field (or *syllable*) contained in the instruction (Fig. 2). If the template is not trivial (i.e., a single template for all instructions in the instruction set) it is usually computed from transformational information and a known starting point and size of the elements. For some instruction sets the size of an element varies by its type. Thus, in this case the format must supply information not only about the transformation but also about the type of the object or operand referred to by the instruction.

The transformation specifies the functional order and rule and, thus, is the primary focus of this paper. The functional

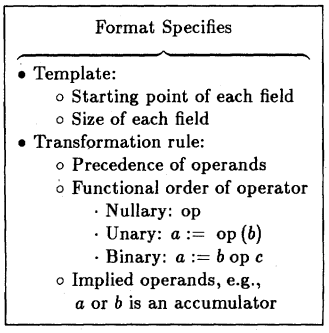

Fig. 1. Information conveyed by the format field.

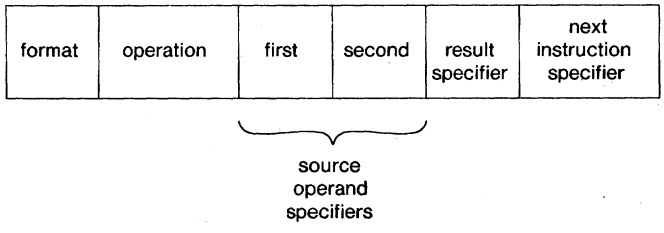

Fig. 2. The instruction vector.

order is simply the number of source operands involved in the computation; i.e., the number of arguments taken by the operation. Regardless of the functional order of the operation, it is assumed that the operation produces at most one result (it may produce none). The functional order of typical computer operations can be classified as follows.

1) Nullary—The operation to follow has no arguments, e.g., HALT or START.

2) Unary—The operation has exactly one argument, e.g., $a := \text{op}(b)$ where op is negation.

3) Binary—The most common operations have binary functional order, e.g., $a := b$ op c where op may be add, subtract, multiply, or divide.

4) Higher functional order—Higher functional order is possible, as in the case of vector operations. However, the instruction usually treats the vectors as single entities, reducing the semantic functional order of the operation to a binary or unary case.

The transformation must also specify operational precedence where appropriate. Operations with binary functional order may have a notion of precedence of application of the operation to the source operands. If the operation is noncommutative, such as subtract or divide, the left argument must be distinguished from the right argument by the format.

The transformation rule is probably that function of the format that is most familiar to machine designers. Traditionally, designers have categorized architectures by differences in the transformation rule: the number of addresses that are explicitly used in the instruction versus those that are implied (Fig. 3).

The number of explicit addresses is of primary interest in operations of binary functional order. There are, of course, no format differences for operations of nullary functional order.

Architecture type	Fields in the instruction	Transformation rule	Next instruction
Three plus one address	op, a, b, c, d	$a := b$ op c	d
Three address	op, a, b, c	$a := b$ op c	$*+1$
Two plus one address	op, a, b, d	$a := a$ op b	d
Two address	op, a, b	$a := a$ op b	$*+1$
One plus one address	op, a, d	$acc := a$ op acc	d
One address	op, a	$acc := a$ op acc	$*+1$
Stack	op	$t := t$ op $(t-1)$	$*+1$

Note that $*$ indicates the present instruction address.

Fig. 3. Categorizing architectures by transformation rule.

IV. Format Coding

There are several approaches to format representation. The simplest approach, implied in the preceding discussion, is a separate identifiable format field which contains the required information. This rarely appears *per se* in familiar architectures. The format is either trivial or coded in some fashion together with the opcode. Consider four possible approaches for format coding.

1) The trivial format—In this case no format field is required since only one template and only one transformation are used throughout the entire vocabulary of the image machine. The responsibility of the format field to distinguish functional order must now rest in the opcode itself since even in the simplest machines nullary, unary, and binary operations must be present.

Note also that the branch operation is a different form of transformation than other operations. Thus, while it appears as a unary (or perhaps binary) operation, it is more accurately regarded as a new format requiring a transformation in the next instruction sequence.

2) Derivation of the format from the opcode—Since the opcode and the format are so intimately related, the image machine designers frequently will regard all aspects of format properties as belonging to the opcode itself. Thus, for example, the interpretation of a floating point operation will by its nature imply the use of a special long template. A branch instruction may similarly require a special template and certainly a special transformation.

3) Encoding of the format and the opcode together—A very common approach to a format specification is to combine this information with the opcode into a single formatted opcode. Thus, when this opcode is decoded, two separate pieces of information are derived: the format with its specifications, and the operation to be invoked.

4) A two-dimensional opcode—A separation of the opcode into two segments, format and operation segments, presents the interpreter with the effect of a separate format field independent from the operation field.

V. A Simple High-Level Language

To best illustrate various kinds of formats, we use a simple high-level language that includes unary and binary order functionality. Consider a language which consists of only assignment statements with the following production rules, expressed in extended Backus–Naur form (EBNF):

assignment-statement = variable-name ":=" expression.

expression = variable-name
| unary-operator "("expression")"
| "("expression binary-operator expression")".

Operators in this language are restricted to arithmetic operators. This results in simple semantics for the appearance of a variable name, namely, that its appearance on the right-hand side of a statement means that a processor should fetch its value, whereas if it occurs on the left-hand side, the processor will store a value into it. Any statement in this language can be represented as a parse tree whose general form is shown in Fig. 4. The root of the parse tree is the assignment symbol ":=" which has two descendants. The left-hand side is always a variable name (shown as "var" in the figure), and the right-hand side may be a variable name, in which case there are no further descendants, or it may be an operator (shown as "op"). If it is an operator, then the operator has one argument, which again may be either a variable name or another operator, and it may have a second argument if the operator is a binary operator. If the operator is a unary operator, there is no second argument and this possibility is indicated by the word "nothing" in Fig. 4.

Strictly speaking, the tree in Fig. 4 is actually a directed acyclic graph (DAG) since each node may compute a subexpression which is common to multiple statements. We will exclude such optimization models for the moment, and in a later section we will extend the discussion to include DAG nodes.

Such a tree can be arbitrarily large; this is represented in the figure by ellipsis marks ("· · ·"). However, any real hardware to execute actions called for in the tree must be of finite size. Rather than building hardware that can accommodate trees up to a certain size and no larger, a better strategy would be to break up the tree into some number of smaller parts.

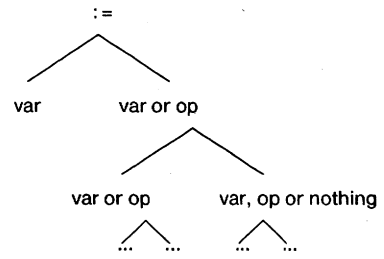

Fig. 4. General form for trees representing simple statements.

VI. A Format Notation

The selection of instruction formats involves first constructing a "format universe" that covers the combinatorial bindings found in traditional zero, one, two, and three address architectures. We distinguish only between two general classes of operand references: *explicit references,* which are defined by distinct (operand) syllables appearing within an instruction; and *implicit references,* which are defined by the format code of the instruction.

A. Architecture

Partition: In a strict sense, the only architectural partition requirement is that the leading format specify sufficient fields to begin instruction execution. The values of the syllables defined by the leading format may then be used to interpret subsequent fields. However, the normal case is that the format syllable defines *all* of the remaining syllables in an instruction.

Association: A three-letter mnemonic code describes all possible associations for operators with at most two arguments and one result. By convention, the first letter will identify the operand to be bound to the result (if any); the second letter will identify the operand to be bound to the left-hand argument of the operator (if any); while the third letter will identify the operand to be bound to the right-hand argument (if any).

Seven one-letter designations are sufficient to describe all relevant possibilities.

S — Denotes implicit specification of the cell just above the current top of the evaluation stack as a result. This cell is presumed to contain garbage at the beginning of any given instruction interpretation, and hence cannot be used as an operand.

T — Denotes an implicit specification of the cell currently on top of the evaluation stack as an operand or result. This cell contains the last intermediate value assigned to an implicit result prior to execution of the current instruction.

U — Denotes an implicit specification of the cell just below the current top of the evaluation stack. This cell contains the next-to-last intermediate value assigned to an implicit result prior to execution of the current instruction.

A — Denotes the variable associated with the first operand identifier in the current instruction.

B — Denotes the variable associated with the second operand identifier in the current instruction.

C — Denotes the variable associated with the third operand identifier in the current instruction.

φ — Denotes that *no* variable is associated with this argument or result. This symbol means "not applicable."

Using this notation, a use-ordered analogue to the typical 360/370 instruction

$$AR\ R1\ R2$$

(meaning "add register *R*1 to register *R*2, and store the result back in *R*1") would be written

$$ABA\ R1\ R2\ +$$

A zero address expansion for the same computation might appear as

$$SA\varphi\ R1\ =$$

$$SA\varphi\ R2\ =$$

$$UTU\ +$$

$$AT\varphi\ R1\ =$$

This notation covers various hybrid formats specifying both implicit and explicit references in a single instruction. For

example, *ABT xy* − means "subtract the value of *X* from the value currently on top of the 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 architectures using this notation. Zero-address architectures use the *UTU* format, one-address architectures use the *TTA* format, two-address architectures use the *AAB* format, and three-address architectures use the *ABC* format. The restrictive nature of monoformat execution architectures is clear when compared to all of the designations suggested by the above mnemonic system. There are $7 \times 7 \times 7 = 343$ potential format names in this notation.

VII. Levels of Formats

Breaking up the tree necessitates the transmission of data values between the parts. One way to do this is to assign names to the parts and to the data which are transmitted, as is done in dataflow architectures. This contrasts with directly executed languages [2], [3], which use a Łukasiewicz ("Polish notation") push-down stack for the temporary storage of the results of partial evaluations of expressions. The use of a stack eliminates some explicit source and destination name fields that would otherwise be needed, resulting in a smaller program representation. Given that the tree is to be broken up into instructions, it remains to be determined how large a part of the tree should be represented by an instruction.

VIII. Level-One Assignment Formats

One way to subdivide the statement tree into instructions is to map each leaf and node of the tree into an instruction. Because the part of the tree corresponding to an instruction covers only one horizontal level of the entire tree, this encoding is called a *level-one format*. In the simple language that has been considered thus far, the only kind of statement is assignment, and assignment statements are the kind most frequently used in real languages as well [4], [5]. Because of the importance of the action of assignment [6], it is represented implicitly in certain kinds of instructions as opposed to, say, being encoded as an operator. Thus, using level-one format instructions to represent a given tree, there is one instruction for each leaf or node in the tree with the exception, for the above reasons, of the node labeled with the assignment symbol itself.

There are four level-one formats. One format encodes both the variable name on the left-hand side and the node labeled with the assignment symbol. This format is $AT\varphi$ where t means the top of the stack before the instruction is executed. A second format $SA\varphi$ pushes the value of a variable name onto the stack and is equivalent to "*s* := *var*" where *s* represents a new stack entry above *t*. The remaining two formats are $TT\varphi$ for unary operators and *UTU* for binary operators

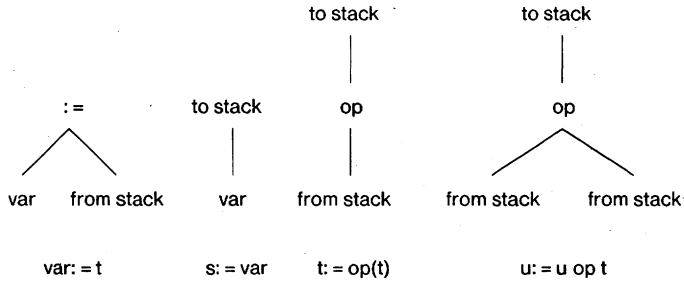

Fig. 5. Tree fragments corresponding to level-one formats.

where *u* means the stack entry just below *t* before the instruction is executed. (Recall that unary operators are operators that take a single argument, as shown in the EBNF for the example language, and binary operators are those that take two.) The parts of the parse tree corresponding to these formats are depicted in Fig. 5.

As an example of this notation, the statement "*factor* := *c*2 × *c*0," which is henceforth called Example 1, is represented using level-one formats as the following four instructions:

$$SA\varphi,\ 'a' = c2;$$
$$SA\varphi,\ 'a' = c0;$$
$$UTU,\ \text{'op'} = \text{multiply};$$
$$AT\varphi,\ 'a' = factor$$

In this example the first field of each instruction is its format, fields are separated by commas, and instructions are separated by semicolons.

IX. Level-Two Assignment Formats

In the case of *level-two formats*, an instruction corresponds to up to two levels of the parse tree. To represent a given tree, there is one instruction for each operator in the statement. There are 21 level-two formats, plus two special cases discussed later in this section, needed to represent assignment statements without redundant use of identifiers for the example high-level language. Actually, an additional 11 formats are required to span nonassignment formats — operations which use or produce side effect information [7] (e.g., HALT, TIME, PRINT, etc.). We shall present these generalized formats in a later section. Of the 21 formats required for assignment statement representation, 14 include the assignment action and the parts of the parse tree corresponding to these formats are shown in Fig. 6.

While a level-one format refers to only one variable name and so can simply refer to it as "*var*", level-two formats are more complex. They may refer to up to three different variable names; these are labeled "*a*," "*b*," and "*c*" in Fig. 6. Thus, the figure shows that if the right-hand side of the statement is a variable name, the statement has the form "*a* := *b*" where "*b*" is a variable name different from "*a*." Similarly, if the right-hand side is an operator, its left-hand argument is one of three possibilities. If it is a variable name

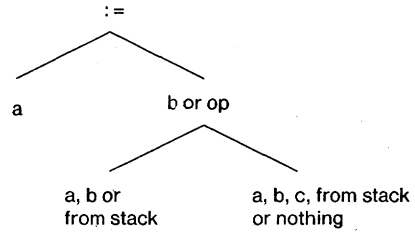

Fig. 6. Tree fragments corresponding to level-two formats that include the assignment action.

Left-hand argument			Right-hand argument
a	b	From stack	
$a := \text{op}(a)$	$a := \text{op}(b)$	$a := \text{op}(t)$	Nothing
$a := a$ op a	$a := b$ op a	$a := t$ op a	a
$a := a$ op b	$a := b$ op b	$a := t$ op b	b
	$a := b$ op c		c
$a := a$ op t	$a := b$ op t	$a := u$ op t	From stack

Fig. 7. Level-two formats that involve the assignment action and an operator.

and if it is the same as the one on the left-hand side of the statement, it is denoted by "a." The case of a variable name different from the left-hand side is shown as "b," and the last possibility is a result from another operator, "from stack." Also, as in Fig. 4, if the upper operator is a unary operator there is only one argument and this is denoted by the word "nothing" among the possibilities for that operator's right-hand argument. Notice that this means that the format can give information about repeated variable names, reducing the number of fields required in the rest of the instruction.

One of the formats that includes the assignment action is simply "$a := b$," mentioned above. The other 13 are tabulated in Fig. 7. The table has a space for every combination of the left-hand argument with the right. Two of them are blank: "$a := a$ op c" is the same as "$a := a$ op b" with "b" renamed to "c," and similarly "$a := t$ op c" is not needed given the existence of "$a := t$ op b." As before, s stands for a new element on the stack above the element that was the top of the stack before the instruction was executed, t is the top before execution, and u is the element just below the top before execution.

The remaining seven formats transmit their results onto the stack. The parts of the parse tree corresponding to these formats are shown in Fig. 8, and the formats are tabulated in Fig. 9. The blank space in Fig. 9 would contain "$t := t$ op b," but that format is no different from "$t := t$ op a." Consider Example 2, the statement "*discriminant* $:= sqr(c1) - 4.0 \times c2 \times c0$." This is represented using level-two formats as the following four instructions:

$SA\varphi\ s := \text{op}(a), \text{'}a\text{'} = c1, \text{'op'} = sqr;$

$SAB\ s := a$ op $b, \text{'}a\text{'} = 4.0, \text{'}b\text{'} = c2, \text{'op'} = \text{multiply};$

$TTA\ t := t$ op $a, \text{'}a\text{'} = c0, \text{'op'} = \text{multiply};$

$AUT\ a := u$ op $t, \text{'}a\text{'} = discriminant, \text{'op'} = \text{subtract}$

In this example, "$sqr(c1)$" means the square of $c1$. Once again, fields are separated by commas and instructions by semicolons.

The 21 level-two formats described in this section are exactly the formats used to represent Fortran assignment statements in DELtran [2]. To these we may add "$a := t$ op t" and "$t := t$ op t" as further useful formats, although they do not follow the basic discipline of a Łukasiewicz stack and require an optimizing compiler in order to be generated. The above are summarized in Fig. 7.

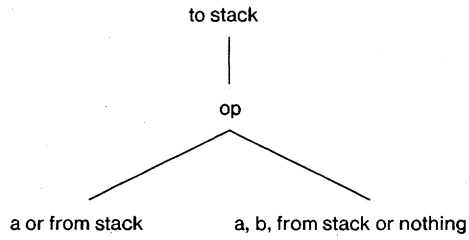

Fig. 8. Tree fragments corresponding to level-two formats that transmit their results to the evaluation stack.

Left-hand argument		Right-hand argument
a	From stack	
$s := \text{op}(a)$	$t := \text{op}(t)$	Nothing
$s := a$ op a	$s := t$ op a	a
$s := a$ op b		b
$t := a$ op t	$u := u$ op t	From stack

Fig. 9. Level-two formats that transmit their results to the evaluation stack.

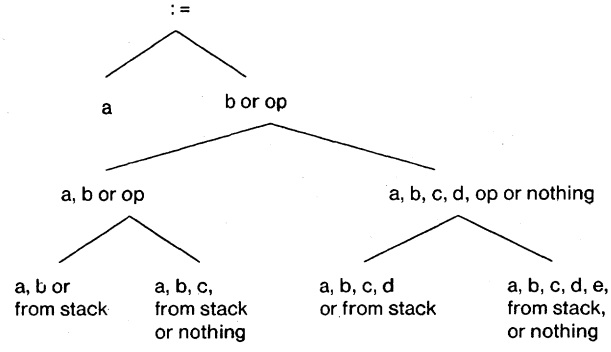

Fig. 10. Tree fragments corresponding to level-three formats that include the assignment action.

X. Level-Three Assignment Formats

Each *level-three format* instruction can represent a part of the parse tree that is up to three levels deep. One operator can be represented at the top level of its corresponding tree fragment, it can take results from up to two operators at a second level, and those operators in turn can have up to four descendants which can be leaves or results transmitted from other instructions.

Fig. 10 shows the parts of the parse tree corresponding to formats that include the assignment action. As before, the formats give information about repeated variable names. However, if the same operator occurs more than once in an instruction, then that operator must be specified more than once in the remainder of the instruction. Even so, there are 324 formats that include the assignment action. If information about multiple occurrences of operators were included in the formats there would be even more of them.

Fig. 11 shows those parts of the parse tree corresponding to the remaining formats, which transmit their results to the

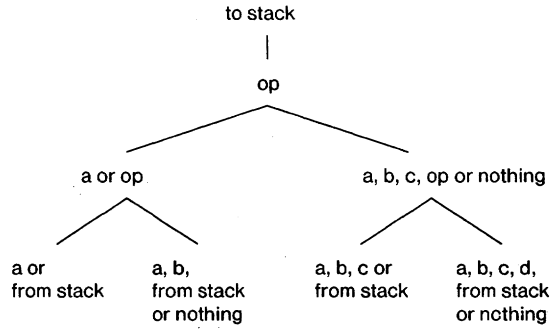

Fig. 11. Tree fragments corresponding to level-three formats that transmit their results to the evaluation stack.

stack. They number 123, so there are 447 level-three formats for assignment statements. Consider the following statement, Example 3:

"$x := (-c1 + sqrt\{sqr(c1) - 4.0 \times c2 \times c0\})/(2.0 \times c2)$."

This is represented using level-three formats as the following five instructions:

$s := \text{op}(a), a = c1, \text{'op'} = \text{negate};$

$s := \text{op}(a), a = c1, \text{'op'} = sqr;$

$s := (a \text{ op1 } b) \text{ op2 } c, \text{'}a\text{'} = 4.0, \text{'}b\text{'} = c2, \text{'}c\text{'} = c0,$
$\text{'op1'} = \text{multiply}, \text{'op2'} = \text{multiply};$

$u := \text{op1}(u \text{ op2 } t), \text{'op1'} = \text{subtract}, \text{'op2'} = sqrt;$

$a := (u \text{ op1 } t) \text{ op2}(b \text{ op3 } c), \text{'}b\text{'} = 2.0, \text{'}c\text{'} = c2,$
$\text{'}a\text{'} = x, \text{'op1'} = \text{add}, \text{'op2'} = \text{divide}, \text{'op3'} = \text{multiply}.$

XI. A Summary of Assignment Format Types and Characteristics

Fig. 12 summarizes some characteristics of the first four basic assignment format levels. The first four characteristics of the first three format levels have already been discussed. A *level-four format* instruction can represent one operator at the top level of its corresponding tree fragment, up to two operators at its second level, and four at its third, for a total of seven operators in one instruction. The operators at the third level can each have up to two descendants, giving a maximum of eight leaves (excluding the left-hand side) that can be represented in an instruction. The entry in the table for the number of level-four formats is a minimum for that value. It is obtained by simplifying the general form of the tree fragments corresponding to level-four formats. The possibilities at each node are simplified by considering a variable appearing at the node as a single possibility, without regard for whether or not the variable has already appeared in the instruction format. The simplified trees are shown in Figs. 13 and 14.

Variables are shown as "*v*," and operators and results from the stack as "op." Cases of a node not being present, shown as "nothing" in previous figures, are shown as "*n*" in these figures. The number of possibilities at each leaf is shown next to the edge connecting it to its parent. The number of possibilities at an inner node that arise when that node is an opera-

Characteristic	Level one formats	Level two formats	Level three formats	Level four formats
Levels of the statement tree represented in one instruction, maximum	1	2	3	4
Operators represented in one instruction, maximum	1	1	3	7
Leaves represented (excluding lhs) in one instruction, maximum	1	2	4	8
Number of assignment formats	4	21	447	> 6,613
Instructions to represent a statement of n operators and m leaves on rhs	$n+m+1$	n	$\lceil n/3 \rceil$, at best	$\lceil n/7 \rceil$, at best
Instructions to represent Example 1 ($n=1, m=4$)	4	1	1	1
Instructions to represent Example 2 ($n=4, m=4$)	9	4	2	1
Instructions to represent Example 3 ($n=9, m=7$)	17	9	5	1

Fig. 12. A summary of the characteristics of format levels.

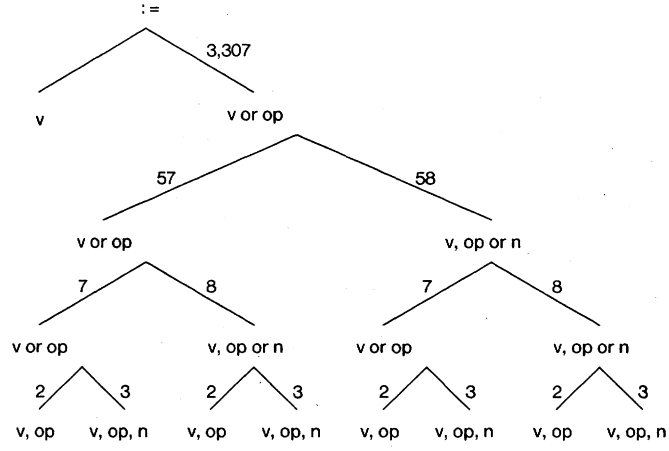

Fig. 13. Simplified tree fragments corresponding to level-four formats that include the assignment action.

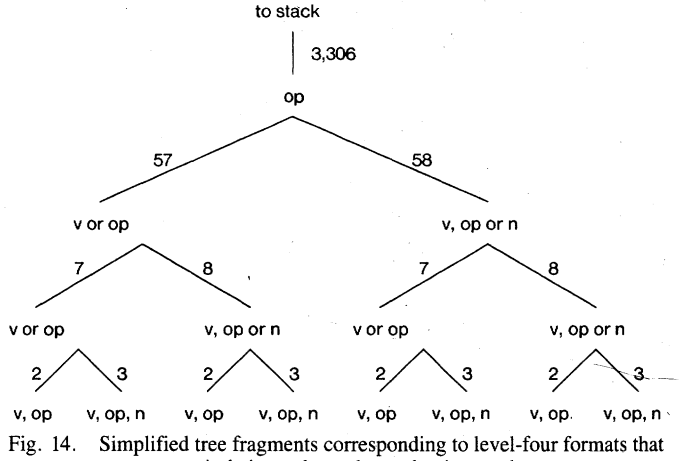

Fig. 14. Simplified tree fragments corresponding to level-four formats that transmit their results to the evaluation stack.

tor is equal to the product of the number of possibilities at the node's descendants. The number shown next to the top edge of each tree is then the total number of formats needed to represent all possible forms of the tree, and their sum appears in Fig. 12. This simplified technique applied to level-three formats indicates a minimum of 113 formats, which is 25 percent of the actual number. If that ratio holds for level four formats, then there are over 25 000 such formats.

If a parse tree has n operators and m leaves on its right-hand side, then level-one formats represent it as m instructions that push values onto the stack, plus n instructions that operate on the stack, and one instruction to pop the final result off of the stack—a total of $m + n + 1$ instructions. By design, level-two formats represent such a statement in n instructions unless $n = 0$, in which case one instruction is required. Since a level-three format instruction can represent up to three operators, some statements can be represented in $\lceil n/3 \rceil$ instructions. However, Example 3 in the previous section had nine operators, yet it required five instructions. This is because the shape of the parse tree did not allow every instruction to represent three operators. The same situation holds for level-four formats. The last three rows of Fig. 12 refer to the examples of the previous sections and give some specific values for n and m. Examples 2 and 3 should not be considered to be typical, though, as statements as complicated as those are infrequent [4], [5].

XII. Address Computation

The discussion to this point has been restricted to the representation of a simple high-level language whose only operators are arithmetic. This gives simple semantics associated with the appearance of a variable name, signifying the fetching of its value should it appear on the right-hand side of an assignment statement, while a variable name on the left-hand side would mean storing a value into the named variable. If now address computations are included in the language, the occurrence of a variable name has more complex semantics which must be reflected in the design of formats for representing a high-level program.

Consider a second simple language extended to include address computation as defined in the following production rules:

```
assignment-statement = variable ":=" expression.
expression = variable
           | unary-operator "("expression")"
           | "("expression binary-operator expression")".
variable = variable-name
         | variable "["expression"]".
```

In this syntax, the square brackets ("[" and "]") enclose an index into an array. While this example language does not contain such address calculations as those needed for pointer variables and record variables, the approach considered here can be applied in a similar way with fewer formats than those needed for array indexing. Any statement in this language can be represented as a parse tree whose general form is shown in Fig. 15. As in Fig. 4, "*var*" in the figure stands for a variable name, "op" is an operator, and the word "nothing"

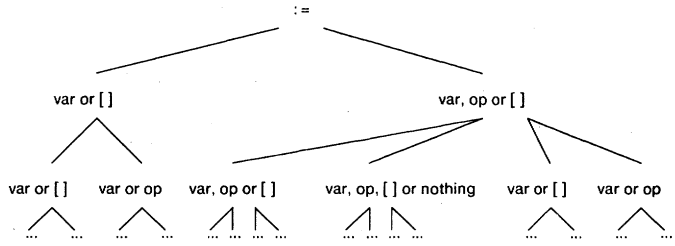

Fig. 15. General form for trees representing statements in the extended language.

means that the operator is a unary operator and so has no second argument. Also, the ellipsis marks ("· · ·") again indicate that the tree can be arbitrarily large. This figure differs from Fig. 4 in that array indexing, shown as square brackets ("[ ]"), is added to the possibilities at certain nodes.

By the same arguments that were applied to the problem of representing a tree involving only arithmetic operators, this tree is better represented for purposes of execution as some number of smaller parts. One way to subdivide the tree is to have an instruction represent up to two levels of the tree. If the left-hand side is simply a variable name and the top node on the right-hand side is either a variable name or an arithmetic operator, then the top part of the tree is of the form shown in Fig. 6 in the discussion of level-two formats. If the top node on either the left-hand side or the right-hand side indicates array indexing, then Fig. 16 or 17, respectively, shows the form of the top part of the tree.

Fig. 18 gives the form of parts of the tree involving array indexing that transmit an address of an array element as their result to a parent node, and Fig. 19 shows those that give a parent an array element's value. An array element is here taken to mean a part of the array of one less dimension than the part before the operation. Thus, an element of a two-dimensional array is a row of that array, and an element of a one-dimensional array is a simple variable.

Notice that by means of these formats, the representation of a statement involving arithmetic and array indexing will consist of as many instructions as the sum of the number of arithmetic operators and the number of dimensions of array indexing. Thus, this approach to address computation is analogous to the level-two format approach to arithmetic operators. For example, the statement "$x[i,j] := 0$" involves two levels of array indexing and so requires two instructions, as follows:

$$SA[B]\ s \text{ equ } a[b], \text{`}a\text{'} = x, \text{`}b\text{'} = i;$$

$$T[A]B\ t[a] := b, \text{`}a\text{'} = j, \text{`}b\text{'} = 0$$

As before, s on the left-hand side of an instruction format means that the result is to be pushed onto the stack, and t in the format refers to the current top of the stack. The first instruction uses one of the formats from Fig. 18 in which an address is placed on the stack, so the top of the stack is made equivalent to the ith element of x. Note that this instruction includes a multiplication of i by the size of the elements of x before it is added to the base address of x. That multiplier could be included explicitly in the instruction, or it could be incorporated into the mechanism for identifying the object x, as is assumed in the above instruction.

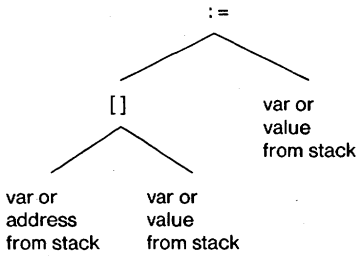

Fig. 16. Tree fragments corresponding to level-two formats that include the assignment action and have array indexing on the left-hand side.

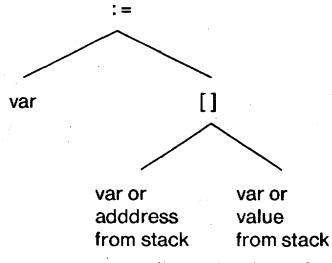

Fig. 17. Tree fragments corresponding to level-two formats that include the assignment action and have array indexing on the right-hand side.

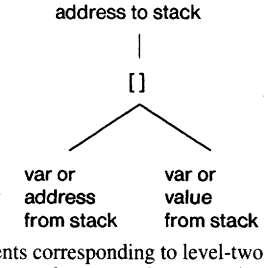

Fig. 18. Tree fragments corresponding to level-two formats that transmit the address of an array element to the stack.

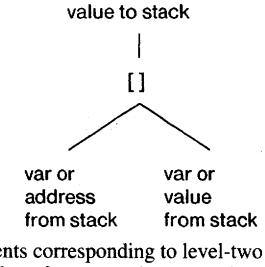

Fig. 19. Tree fragments corresponding to level-two formats that transmit the value of an array element to the stack.

An alternative approach is to break down array indexing into smaller parts and to encode those. In this approach, the statement "$x[i,j] := 0$" can be stated as "$M[adr(x[i,j])] := 0$" where M means main memory and is considered to be an array and the notation $adr(v)$ means the address of v. Breaking up this statement into smaller parts results in the following equivalent statement:

$$M[x.base + i \times elementsize + j] := 0$$

Encoding these smaller parts requires the following instructions:

$SAB\ s := a$ op b, 'a' $= i$, 'b' $= elementsize$,
'op' = multiply;

$TAT\ t := a$ op t, 'a' $= x.base$, 'op' = add;

$T[A]B\ t[a] := b$, 'a' $= j$, 'b' $= 0$

This sequence is one instruction longer than the method previously presented.

XIII. Extending Level-Two Formats

A. Representing DAG Nodes

To handle common subexpression optimization we need to expand the formats to include the possibility of sending the result of an operation to at least two destinations. The second destination may be the same as one of the source operands or different from all other operands or the evaluation stack. The stack as a destination should appear at most once per format because cases which require a result to be used immediately as both operands of another operator are covered by the "$a := t$ op t" and "$t := t$ op t" formats. Note that general DAG's may require more than two destinations, but that these cases can always be handled with compiler-created temporaries. Only the operations which produce a result need to be expanded. The expanded formats are shown in Figs. 20 and 21, except for the simple assignment format "$a, b := c$."

B. Generalized Operations

So far we have restricted our attention to arithmetic assignment operations. These require at least one source operation and produce a single result in the data space. A number of operations cannot be included since the transformation each represents either uses or produces only side effect information. There are 11 such possibilities listed below [7]:

1	op	$\varphi\varphi\varphi$
2	$s :=$ op	$s\varphi\varphi$
3	$a :=$ op	$a\varphi\varphi$
4	op(t)	$\varphi T\varphi$
5	op(a)	$\varphi A\varphi$
6	u op t	φUT
7	t op t	φTT
8	a op t	φAT
9	t op a	φTA
10	a op a	φAA
11	a op b	φAB

The first case includes operations such as HALT; the second and third cases produce results based on side effect (status-type) information, e.g., TIME. The remaining cases include both unary and binary operations that do not assign a result to the data space (e.g., READ, PRINT).

XIV. Subsets of Level-Two Formats

It is not always possible, or even advisable, to implement all formats (Fig. 7 and Fig. 9) due to encoding problems or host limitations. However, by starting with this format universe, we can judiciously eliminate those formats not needed

Left-hand argument			Right-hand argument
a	c	From stack	
$a,b := \text{op}(a)$	$a,b := \text{op}(c)$	$a,b := \text{op}(t)$	Nothing
$a,b := a \text{ op } a$	$a,b := c \text{ op } a$	$a,b := t \text{ op } a$	a
$a,b := a \text{ op } b$			b
$a,b := a \text{ op } c$	$a,b := c \text{ op } c$	$a,b := t \text{ op } c$	c
	$a,b := c \text{ op } d$		d
$a,b := a \text{ op } t$	$a,b := c \text{ op } t$	$a,b := u \text{ op } t$	From stack
		$a,b := t \text{ op } t$	

Fig. 20. Level-two formats that assign to two variables.

Left-hand argument			Right-hand argument
a	b	From stack	
$a,s := \text{op}(a)$	$a,s := \text{op}(b)$	$a,t := \text{op}(t)$	Nothing
$a,s := a \text{ op } a$	$a,s := b \text{ op } a$	$a,t := t \text{ op } a$	a
$a,s := a \text{ op } b$	$a,s := b \text{ op } b$	$a,t := t \text{ op } b$	b
	$a,s := b \text{ op } c$		c
$a,t := a \text{ op } t$	$a,t := b \text{ op } t$	$a,u := u \text{ op } t$	From stack
		$a,t := t \text{ op } t$	

Fig. 21. Level-two formats that both involve the assignment action and transmit their results to the evaluation stack.

for a specific system. The following observations, taken from Flynn and Hoevel [7], identify some interesting subsets.

Observation 1: We begin by excluding for the moment array computation formats since $a[b] := c$ or $A[B]C$ can be represented by the $a := b$ op c or ABC format. However, since array ops may be frequent in certain computations, we will reconsider this issue later. We also exclude formats for common subexpression elimination as being infrequently used. This leaves the level two operations of both assignment and generalized type.

Observation 2: All formats containing φ can be replaced by otherwise similar formats containing a stack designator in the same position. Formats such as $AT\varphi$ would be replaced by ATT; the semantics of the operator would discriminate between the two.

Example 1: To repeat the earlier example of a solution for the discriminant, we would have:

$$\text{discriminant} = \sqrt{c_1^2 - 4c_2c_0}$$

$SA\varphi$	$a = c_1$		op = square
SAB	$a = 4.0$	$b = c_2$	op = multiply
TAT	$a = c_0$		op = multiply
AUT	$a = disc.$		op = subtract

Observation 3: Operators can be used to decide precedence for noncommutative operations, eliminating TAT, AAB, ATB, and ATA.

Example 2: Continuing the example of the discriminant, we have

SAA	$a = c_1$		op = square
SAB	$a = 4.0$	$b = c_2$	op = multiply
TAT	$a = c_0$		op = multiply
AUT	$a = disc.$		op = subtract

Notice that now the interpreter must distinguish $SA\varphi$ from SAA by the op semantics.

Observation 4: The φTT, TTT, and ATT formats, all of which specify the *same intermediate source value* of an operator, may be eliminated since the use of equivalent expressions as the two operands of a single binary operator is very infrequent.

If all of the above observations are employed, only eleven formats remain:

$UTU, AUT, TAT, ATA, ATB, SAA,$

$SAB, AAA, AAB, ABA,$ and ABC.

This subset is sufficient to leave our discriminant example unaffected.

Observation 5: A subset of eight formats results if identical explicit source operands are treated as distinct, eliminating the SAA, AAA, and ABB formats and leaving only UTU, AUT, TAT, ATA, ATB, SAB, ABA, and ABC.

Observation 6: If no redundant identifiers are allowed, only six formats are needed: UTU, AUT, TAT, ABT, SAB, and ABC. The loss of formats such as AAB requires redundant identifiers to be used in the ABC format.

Example 3: With the loss of the SAA format, we replace

SAA	$a = c_1$		op = square

with

SAB	$a = c_1$	$b = c_1$	op = square

increasing the program identifiers by one.

Observation 7: Finally, we can create a stackless format set of three elements: ABA, AAB, ABC. Further elimination of overlapped identifiers produces the classic three-address instruction architecture format, ABC.

Example 4: Now our discriminant program is significantly altered:

ABA	$a = \text{temp}_1$	$b = c_1$		op = square
ABC	$a = \text{temp}_2$	$b = 4.0$	$C = c_2$	op = multiply
AAB	$a = \text{temp}_2$	$b = c_0$		op = multiply
AAB	$a = \text{temp}_1$	$b = \text{temp}_2$		op = subtract

(temp_1 is also the final value of discriminant).

Format completeness, producing a minimum number of identifiers, can be traded for the size of each format syllable.

XV. Level-Two Formats with Separate Constants

Johnston [8] defines a contour to be a vector describing the named objects in a local environment. When an environment is invoked, a contour for the named data objects in this new environment must be established before interpretation of the program may proceed. For simple languages, such as Fortran, all objects are statically allocated and the contour for every environment can be determined at load time. This allows the interpretive mechanism to establish the dynamic contour by simply setting a base register that selects the appropriate contour for the desired environment. Objects within a contour may be given an initial value without additional overhead to the interpreter. Initial values can be set at load time since their position within the contour is known. Objects which have a constant value or a defined initial

value, such as variables that have been set by a data statement in Fortran, may be treated in the same manner as other objects when this scheme is used.

Languages which allow recursion must have their contour established during the dynamic execution of the program. When a recursive call is made, the semantics of the high-level language require that a new contour be created for the local variables of the new procedure invocation. Because the flow of control may be data dependent, it is impossible to determine at load time the number or position of the environmental contours which will be required at run time. Therefore, it is not possible to initialize the values of objects within contours of recursive languages at load time. Most recursive languages acknowledge this difficulty by requiring a variable to be set by an assignment statement before it is read. These languages usually lack an equivalent to the Fortran data statement.

Support of recursive languages requires that either a skeletal contour be copied to the dynamic contour every time the environment changes or that a separate contour exists for objects which have constant values. When the copying of the skeletal contour scheme is used, only a partial copying is required if the objects are arranged so that the initialized objects are all in one group. If all objects must be initialized, as when a tagged data scheme is used, then the complete skeletal contour must be copied or set to a known value. This copying may be done on a demand basis if an "uninitialized tag" can be established for all objects in the contour when it is created.

Separate contours for constant and variable objects can be implemented in various ways. There can be two local addressing modes, one which accesses the contour containing the variables and one which accesses the contour containing the constants. When a recursive call is made a new variable contour must be established but the constant contour is reused. The constant contour position can be determined and initialized at load time. It is also possible to include the constants in the instruction stream as immediate data, or a mixture of constant contour and immediate data methods can be used.

When the architecture distinguishes between objects having constant value and objects which vary during program interpretation then there are three general classes of operand references: constant, explicit, and implicit. Constant references refer to a constant contour which is separate from the variable contour and they appear as distinct fields within an instruction. They always appear in one of the source operand positions, never in the destination operand position. Because constants never change in value, only one contour is needed per procedure. It does not need to be copied upon recursive calls. Explicit references refer to the variables within a procedure and a new contour is allocated for them on every recursive call. As before, they appear as distinct fields within an instruction. Implicit references are defined by the format and usually refer to the objects on an evaluation stack.

Additional formats are required when they specify which contour is to be used by each operand. The following notation is used to indicate that the corresponding operand field is an offset from the start of the constant contour.

1) "k" refers to the first explicit constant operand specification appearing in an instruction;

Left-hand argument				Right-hand argument
a	b	k	From stack	
$a :=$ op (a)	$a :=$ op (b)	$a :=$ op (k)	$a :=$ op (t)	Nothing
$a := a$ op a	$a := b$ op a	$a := k$ op a	$a := t$ op a	a
$a := a$ op b	$a := b$ op b	$a := k$ op b	$a := t$ op b	b
	$a := b$ op c			c
$a := a$ op k	$a := b$ op k	$a := k$ op k	$a := t$ op k	k
		$a := k$ op l		l
$a := a$ op t	$a := b$ op t	$a := k$ op t	$a := u$ op t	From stack

Fig. 22. Level-two formats with separate constants that involve the assignment action.

2) "l" refers to the second explicit constant operand specification appearing in an instruction.

Figs. 22 and 23 list the 36 formats required to provide transformational completeness when constants are different from variables. Two additional formats are needed to accommodate simple assignment formats of the form $a := b$ and $a := k$.

Note that the formats which specify constants for all the operands of an operation are included even though these operations could be reduced at compile time. This enables the compiler to avoid dealing with this complexity.

Fig. 24 shows an implementation of separate contours for constants and variables. Object selection is performed by extracting an operand field and adding it to the two registers which establish the constant and variable contours. The format determines which one of the selected objects is forwarded to the execution unit.

Operand field width now becomes a more complex issue. If code size is the only consideration and there are n variables and m constants, the operand field width should be $\lceil \log_2 (n) \rceil$ bits wide for variables and $\lceil \log_2 (m) \rceil$ bits wide for constants. However, in this encoding scheme the field width is only known after the format is decoded, thus making it impossible to extract the operand field in parallel with format decoding. When speed of interpretation is important, then it may be wise to set the operand field width to the maximum of the number of bits required for variables and constants, thereby allowing parallel operand extraction.

Another consideration is whether to use one constant contour for the entire program or to use a constant contour per procedure. If only one contour is available, then the complete program must be compiled before its size is known. The size of this contour will be smaller than the sum of the sizes of all the individual procedure contours if multiple appearances of constants are eliminated, but may require a wider constant operand field in each instruction. Thus, only one constant contour may increase the overall size of the code if there are many different constants. When there is a constant contour per procedure, the requirement of compiling the entire program before any code is generated is removed. A constant contour per procedure is probably a better method since only a small number of constants are likely to appear in multiple contours.

The main advantage of separating constants from variables is that it reduces the overhead in establishing a new environment to loading two registers. One register selects the constant contour while the other establishes the base of a new variable contour on the run time stack. It is not necessary to initialize anything within the variable's contour. If variables are not initialized then using a tag to indicate indirection is

Left-hand argument			Right-hand argument
a	k	From stack	
$s := \text{op}(a)$	$s := \text{op}(k)$	$t := \text{op}(t)$	Nothing
$s := a \text{ op } a$	$s := k \text{ op } a$	$t := t \text{ op } a$	a
$s := a \text{ op } b$			b
$s := a \text{ op } k$	$s := k \text{ op } k$	$t := t \text{ op } k$	k
	$s := k \text{ op } l$		l
$t := a \text{ op } t$	$t := k \text{ op } t$	$u := u \text{ op } t$	From stack

Fig. 23. Level-two formats with separate constants that transmit their results to the evaluation stack.

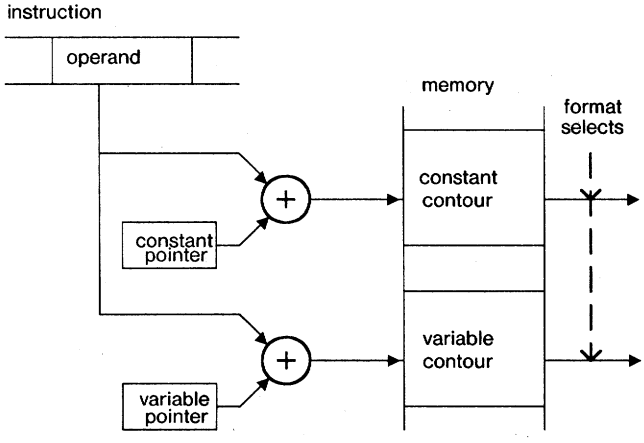

Fig. 24. Accessing objects within constant and variable contours.

inappropriate. Instructions must indicate if an object is to be referenced indirectly through the variable or constant contour. Global objects with known load-time addresses can be accessed by referencing them indirectly through the constant contour. Array elements, pointers, and other references which have addresses computed at run time can be accessed indirectly through the variable contour.

XVI. Encoding Constants in the Instructions

When the architecture distinguishes between objects having constant value and objects whose value varies during program interpretation, it is also possible to place constants in the instruction stream. Tanenbaum [5] reports that 81 percent of integer constants are in the range 0–3 and argues that a reduction in code size is achieved if these constants are created by small instructions. Wilcox [9] reports that 56 percent of the integer constants are in the range of 0–3 and proposes a method of encoding constants based on a bounded-n Huffman encoding [10].

If all constants are in the instruction stream, then the number of required formats for transformational completeness is 38, the same as when a separate constant contour is used. The only difference here is that a k or l in the format implies that the corresponding operand field contains the (encoded) constant itself. To reduce code size, these constant operand fields would use a variable length bounded-n Huffman encoding. Very few different constants are present so very tight encoding occurs. However, this method has the drawback that using variable length fields results in an inability to perform parallel extraction of operands.

Placing the constants in the instruction stream may reduce the number of memory references required to execute an operation. This does not necessarily result in a time benefit.

Left-hand argument					Right-hand argument
a	b	i	k	From stack	
$a := \text{op}(a)$	$a := \text{op}(b)$	$a := \text{op}(i)$	$a := \text{op}(k)$	$a := \text{op}(t)$	Nothing
$a := a \text{ op } a$	$a := b \text{ op } a$	$a := i \text{ op } a$	$a := k \text{ op } a$	$a := t \text{ op } a$	a
$a := a \text{ op } b$	$a := b \text{ op } b$	$a := i \text{ op } b$	$a := k \text{ op } b$	$a := t \text{ op } b$	b
	$a := b \text{ op } c$				c
$a := a \text{ op } i$	$a := b \text{ op } i$	$a := i \text{ op } i$	$a := k \text{ op } i$	$a := t \text{ op } i$	i
		$a := i \text{ op } j$			j
$a := a \text{ op } k$	$a := b \text{ op } k$	$a := i \text{ op } k$	$a := k \text{ op } k$	$a := t \text{ op } k$	k
			$a := k \text{ op } l$		l
$a := a \text{ op } t$	$a := b \text{ op } t$	$a := i \text{ op } t$	$a := k \text{ op } t$	$a := u \text{ op } t$	*FromStack*

Fig. 25. Level-two formats with separate short and long constants that involve the assignment action.

Encoded constants typically will be decoded by indexing into a lookup table which probably has an access time similar to that of a memory cycle. The total data movement for both operations is about the same.

Another possibility is a hybrid scheme where frequently used, small constants are encoded in the instruction stream and infrequent or large constants are placed in a constant contour. This method is similar to a short immediate addressing mode found in many traditional computers. Additional formats are used to specify that the operand field contains a short constant. The two additional letter designators are: 1) i, the first explicit short constant encoded in an instruction, and 2) j, the second explicit short constant encoded in an instruction.

Figs. 25 and 26 list the 56 formats required to provide transformational completeness with short immediate constants and constant contours. Three additional formats are needed to accommodate the simple assignment forms $a := b$, $a := i$, and $a := k$. The short immediate field width could be the same as the variable and constant contour width and could be sign extended to form a full length integer constant.

A high-speed implementation of object accessing with this format set is shown in Fig. 27. A decoding table is used to map a short constant encoded in the operand field into a full-width constant object. In parallel, the operand field is used to index into the long constant and variable contours. The format determines which one of the data items is forwarded to the execution unit.

XVII. Format Frequencies for a Sample Program

In order to gain some insight into how the theory of instruction formats can help in the design of an actual execution architecture, we have implemented compilers and emulators to measure how often formats are actually used. A sample program called Kalman was compiled and executed using these tools. The program, an implementation of a Kalman filter, reads data for the ballistic trajectory of an object to which pseudorandom numbers have been added to simulate radar noise. For each data point, it produces as output an estimate of the object's position and velocity based on the current and all previous data points. Kalman is 1103 lines long and contains 38 procedure blocks.

P-code [11] is used as an example of a level-one format architecture. The compiler accepts Pascal programs and produces a level-one format representation. Each instruction describes at most one data object and all data transformations use the stack. The Stanford Emulation Laboratory [12] was

Left-hand argument				Right-hand argument
a	*i*	*k*	From stack	
$s := \text{op}(a)$	$s := \text{op}(i)$	$s := \text{op}(k)$	$t := \text{op}(t)$	Nothing
$s := a \text{ op } a$	$s := i \text{ op } a$	$s := k \text{ op } a$	$t := t \text{ op } a$	*a*
$s := a \text{ op } b$				*b*
$s := a \text{ op } i$	$s := i \text{ op } i$	$s := k \text{ op } i$	$t := t \text{ op } i$	*i*
	$s := i \text{ op } j$			*j*
$s := a \text{ op } k$	$s := i \text{ op } k$	$s := k \text{ op } k$	$t := t \text{ op } k$	*k*
		$s := k \text{ op } l$		*l*
$t := a \text{ op } t$	$t := i \text{ op } t$	$t := k \text{ op } t$	$u := u \text{ op } t$	From Stack

Fig. 26. Level-two formats with separate short and long constants that transmit their results to the evaluation stack.

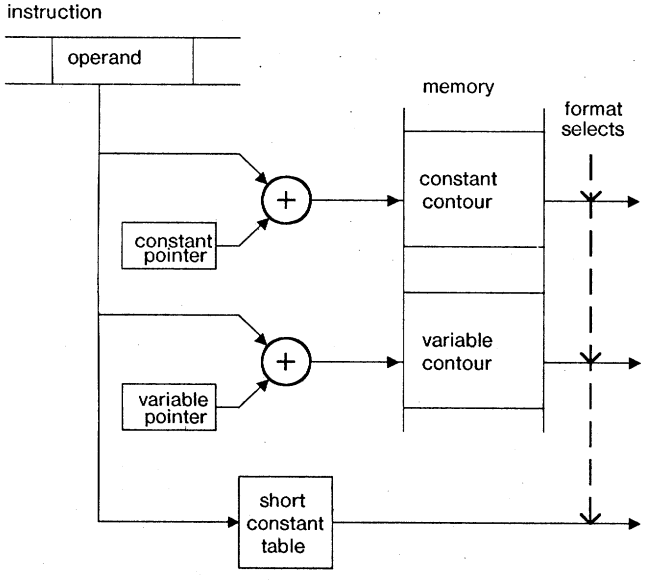

Fig. 27. Accessing objects with encoded constants.

```
There were a total of 292570 counts.

Count Percent   Cum.  Class name    |        10%        20%        30%
117577  40.199  40.188 s := a       |*********************************>
 98844  33.785  73.972 u := u op t  |*********************************
 22552   7.708  81.681 a := t       |********
 21752   7.435  89.116 t := op(t)   |*******
 15276   5.221  94.337 jump false   |*****
  8445   2.886  97.232 jump         |***
  6236   2.131  99.355 callstdproc  |**
   428   0.146  99.501 enter        |
   428   0.146  99.647 return       |
   427   0.146  99.793 call         |
   427   0.146  99.939 mark stack   |
   178   0.061 100.000 case jump    |
```

Fig. 28. Level-one (*P*-code) format frequency for the Kalman program.

```
There were a total of 77035 counts.

Count Percent   Cum.  Class name    |        10%        20%
29044  37.702  37.702 arrayx        |****************************>
11227  14.573  52.276 a := a op b   |**************
 7049   9.150  61.426 u := u op t   |*********
 7021   9.114  70.540 s := a op b   |*********
 5091   6.608  77.149 a := a op t   |******
 3673   4.767  81.917 s := op(a)    |****
 2256   2.928  84.845 a := b        |**
 2100   2.726  87.571 callstdproc   |**
 1762   2.287  89.859 a := b op c   |**
 1720   2.232  92.091 t := t op a   |**
 1414   1.835  93.927 t := op(t)    |*
 1394   1.809  95.737 goto          |*
 1324   1.718  97.455 a := u op t   |*
  427   0.554  98.009 call         |
  384   0.498  98.508 a := t op b   |
  266   0.345  98.853 t := a op t   |
  259   0.336  99.189 a := op(b)    |
  226   0.293  99.483 procreturn    |
  202   0.262  99.745 funcreturn    |
  108   0.140  99.885 a := b op t   |
   39   0.050  99.936 a := op(t)    |
   22   0.028  99.964 arrayt        |
   19   0.024  99.989 a := op(a)    |
    8   0.010 100.000 a := b op a   |
```

Fig. 29. Level-two (Adept) format frequency for the Kalman program.

used to emulate a processor that executes *P*-code programs.

Fig. 28 gives the number of occurrences of each level one format during execution of the Kalman programs. Most frequent formats are listed first. The percentage of the total number of instructions executed and cumulative percentage column are also included. The four most frequent formats in this level-one experiment are the assignment formats. The other formats deal with flow of control and are much lower in frequency.

We have implemented a compiler and emulator for a level-two architecture, called Adept. The compiler accepts Pascal programs and produces a level-two format representation. Descriptors for variables and constants are found in a single contour in the representation used. The Stanford Emulation Laboratory was also used to emulate a processor that executes programs in Adept representation.

Fig. 29 gives the number of occurrences of each format, with the most frequent formats listed first. For this experiment, formats for address calculation are encoded in two fields, the first of which can take on values listed in the table as "arrayx" and "arrayt." The table also shows formats for Pascal statements that do not appear in the example simple high-level language. These formats are goto, call a procedure or a function, call a standard procedure or function, return from a procedure, and return from a function.

For the Adept architecture, formats that transmit their results to the stack, such as "$u := u$ op t" and "$s := a$ op b" (the third and fourth most frequently occurring formats) are used for some flow-of-control statements as well as for arithmetic evaluation. This dual-purpose nature may account for their high ranking.

A large proportion of the second most frequent format, "$a := a$ op b," may include statements of the form "$a := a + 1$" that can be optimized into instructions of the form "$a := \text{op}(a)$, 'a' $= i$, op $= succ$." The format may also include a large number of statements of the form "$a := a - 1$" for which the compiler can emit similar instructions that use the *pred* operator. A study of the static statistics of 51 Simula programs [13] found that 3.5 percent of all statements were of the form "$a := a + 1$," and a further 0.5 percent were of the form "$a := a - 1$." The sum of the number of returns from procedure calls and from function calls equals the number of call instructions, except that one procedure return is executed to halt the processor at the end of execution.

A level-two format set allows a program to be executed by far fewer instructions than a level-one format set allows. The level-one format set required 3.8 times as many instructions as the level-two format set for the Kalman example. The reason is obvious; level-two format sets encode more of the high-level language statement's code tree in each instruction.

XVIII. Conclusions

Basic to any architectural definition is the instruction format. While format considerations include template (size and field) definitions, it is the specification of a transform regime (operand precedence, etc.) that is most important in deter-

mining architectural effectiveness. The old debate concerning the superiority of one fixed address format over another is evidence of this. This argument misses the point: the format must represent elements of an execution graph (a directed cyclic graph, or DAG). In, for example, the stack fixed format representation, each DAG link and node (two sources, one sink link, and a node) is an instruction. At the other extreme, the three-address format represents the DAG links and node as a single instruction but frequently at the expense of a redundant identifier in the instruction — presenting the dilemma of space versus time.

The basic issue is a notion of transformational completeness: that each DAG node be represented as a single instruction *without* redundancy. This can be accomplished in about 30 formats (depending upon node restrictions) by allowing a robust set of formats, including both three-address and stack types.

While the principle of format completeness can be extended to include multiple nodes (i.e., subgraphs), the combinatorics of completeness of such arrangements are formidable and probably exclude the possibility for an efficient implementation.

As an effect of having a robust format set, one might also use the format to designate the lexical data type of an object reference since: 1) data references must have at least partial sink-source consistency, and 2) certain types (e.g., constants and locals) represent the bulk of references.

Compilers typically generate a DAG as an input to a code generator. Much of the complexity of the subsequent code generation — register optimization, etc. — can be eliminated with a format set in correspondence with the transformational requirements of each DAG node. Thus, the value of a robust format set is clear du ing compilation.

During execution a complete format set will minimize run-time parameters such as dynamic program space (instruction bandwidth) and instruction count, but at the expense of a more complex format decode.

On the other hand, the designer may observe (at least on the basis of our sample programs) that most cases of required DAG transformations can be covered with much fewer format types than the complete set. This allows the possibility of reasonable tradeoffs between hardware cost and run-time and compile-time performance.

REFERENCES

[1] C. G. Bell and A. Newell, *Computer Structures: Readings and Examples*. New York: McGraw-Hill, 1971.

[2] L. W. Hoevel, "Directly executed languages," Ph.D. dissertation, The Johns Hopkins Univ., Baltimore, MD, Apr. 1978.

[3] S. Wakefield, "Studies in execution architectures," Ph.D. dissertation, Stanford Univ., Stanford, CA, Dec. 1982; also available as Comput. Syst. Lab. Tech. Rep. 237, Stanford Univ., Stanford, CA.

[4] S. K. Robinson and I. S. Torsun, "An empirical analysis of FORTRAN programs," *Comput. J.*, vol. 19, no. 1, pp. 56–62, Feb. 1976.

[5] A. S. Tanenbaum, "Implications of structured programming for machine architecture," *Commun. ACM*, vol. 21, pp. 129–134, Mar. 1978.

[6] D. E. Knuth, "An empirical study of FORTRAN programs," *Software—Practice and Experience*, vol. 1, pp. 105–133, 1971.

[7] M. J. Flynn and L. W. Hoevel, "Execution architectures: The DELtran experiment," *IEEE Trans. Comput.*, vol. C-32, pp. 156–175, Feb. 1983.

[8] J. B. Johnston, "The contour model of block structured processes," in *Proc. ACM SDSPL*, Feb. 1971, pp. 55–82.

[9] C. R. Wilcox, "A language-oriented approach to computer architecture," Ph.D. dissertation, Stanford Univ., Stanford, CA, June 1980.

[10] D. A. Huffman, "A method for the construction of minimum redundancy codes," *Proc. IRE*, vol. 40, Sept. 1952.

[11] K. V. Nori *et al.*, "The Pascal ⟨P⟩ compiler: Implementation notes," Instituts fur Informatik, Eidgenossische Technische Hochschule, Zurich, Tech. Rep., July 1976.

[12] C. Neuhauser, "Emmy system processor—Principles of operation," Comput. Syst. Lab., Stanford Univ., Stanford, CA, Tech. Note 114, May 1977.

[13] S. Arnborg, "Simula user program semantics and code generation effectiveness," Nat. Res. Inst. Defense, Stockholm, FOA P Rep. C 8408-M3(E5), June 1974.

Michael J. Flynn was born in New York City, NY, on May 20, 1934. He received the B.S.E.E. degree from Manhattan College, Bronx, NY, in 1955, the M.S. degree from Syracuse University, Syracuse, NY, in 1960, and the Ph.D. degree from Purdue University, West Lafayette, IN, in 1961.

He joined IBM in 1955 and, for ten years, worked in the areas of computer organization and design. He was Design Manager of prototype versions of the IBM 7090 and 7094/11, and later was Design Manager for the System 360 Model 91 Central Processing Unit. He has been a Faculty Member of Northwestern University, Evanston, IL (1966–1970) and The Johns Hopkins University, Baltimore, MD (1970–1974). In 1973–1974 he went on leave from Johns Hopkins to serve as Vice President of Palyn Associates, Inc., a computer design firm in San Jose, CA where he is now a Senior Consultant. In January 1975 he became Professor of Electrical Engineering at Stanford University, Stanford, CA, and was Director of the Computer Systems Laboratory from 1977 to 1983.

Dr. Flynn has served on the IEEE Computer Society's Board of Governors and as Associate Editor of the TRANSACTIONS ON COMPUTERS. He was founding chairman of both the ACM Special Interest Group on Computer Architecture and the IEEE Computer Society's Technical Committee on Computer Architecture.

John D. Johnson was born in Camp Poke, LA, on August 7, 1952. He received the B.S. and M.S. degrees in electrical engineering from the University of Illinois at Urbana in 1975 and 1976, respectively, and is currently a Ph.D. degree student at Stanford University, Stanford, CA, in the Department of Electrical Engineering.

His research area is computer architecture. From 1975 to 1980 he was a Development Engineer at Hewlett-Packard's Data Systems Division, and worked on both hardware and software products. He has continued to work parttime at Hewlett-Packard while pursuing his studies at Stanford University.

Scott P. Wakefield (S'82–M'83) received the B.S.E.E.-C.S. degree from the University of Colorado, Boulder, and the M.S. and Ph.D. degrees from Stanford University, Stanford, CA, in 1975, 1976, and 1983, respectively.

He is with the IBM Thomas J. Watson Research Center, Yorktown Heights, NY, where he is working on direct correspondence architectures. In 1974 he reviewed studies of neural networks from the perspective of a computer designer.

Dr. Wakefield was a Student Member of the Local Arrangements Committee for the Sixth Annual Symposium on Computer Architecture. He shared third prize in the first Mattel Toy Design Contest (1977) and shared second prize in the second one (1978). He received a Fairchild Fellowship for the 1981–1982 academic year and an IBM Postdoctoral Fellowship in 1983.