THE PERFORMANCE ADVANTAGES OF A HIGH LEVEL LANGUAGE MACHINE

by

JAMES WALTER RYMARCZYK

Submitted in Partial Fulfillment

of the Requirements for the

Degree of Bachelor of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June, 1972

Signature of Author . . . . . .
Department of Electrical Engineering; May 12, 1972

Certified by . . . . . .
Thesis Supervisor

Accepted by . . . . . .
Chairman, Departmental Committee on Theses

ABSTRACT

This paper identifies and discusses a number of mechanisms by which a machine with a suitable high level interface language might achieve a level of performance which exceeds that of a machine with a conventional von Neumann architecture. A high level language machine is characterized which is somewhat less flexible than a conventional design, but which significantly out-performs the conventional machine when used in a way that exploits the high level language.

Keywords and Phrases: Computer Architecture,
Machine Organization,
High Level Language Machine,
Language Oriented Computer Design,
Computer Command Structures,
Hardware Implementation
of Programming Languages,
High Performance Computer Design.

ACKNOWLEDGEMENT

I am grateful to Stu Madnick, my thesis supervisor, and to Dave Kelleher and Steve Zilles of IBM for their generous efforts in reviewing and commenting upon this paper.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

Peter Landin is reported to have once said that most papers in Computer Science describe how their author learned what someone else already knew [MoseJ70]. This paper is no exception to that rule. It combines a collection of notions that have been extracted from the literature with a number of my own ideas that, although independently derived, have surely been noted before. While credit for many of the concepts presented belongs to others, I accept full responsibility for any misrepresentations or ambiguities which may exist in this presentation.

This paper is intended to serve primarily as an aid to others in the field of computer design by collecting, organizing and commenting upon these ideas. The scope of this paper does not permit much in the way of demonstrable results. It is hoped that the absence of physical realizations for the mechanisms that are discussed will be compensated for by a somewhat broader perspective than is customary in the literature.

The reader is assumed to be familiar with the principal semantic features of the APL, PL/I and LISP programming languages [M9; M10; M13]. In addition, he should have some knowledge of the goals of contemporary programming systems, the problems that are encountered and the hardware/software engineering techniques used in seeking to attain these goals.

I.A. Historical Perspective

There have been many diverse efforts to design a computer that directly interprets a high level language [AbraP70; BashT67; BashT68; BerkK69; ChesG71; DennJ71; McFaC70; MeggJ64; MullA63; RiceR71; RossC64; ShawJ58; ThurK70; WebeH67; ZaksR71; M3]. The motivation for doing so has generally been based upon either of two assumptions.

First, it has been postulated that a machine with a high level interface language can provide a significant increase in useful programming function without a prohibitive increase in cost. One would like to create a well disciplined programming environment in which run-time programming errors are detected [IlifJ68, p. 14; BerkK69, p. 60], in which such errors need not culminate in machine dumps [BashT67; McFaC70], in which programming generality is guaranteed or at least likely [DennJ69; DennJ71], etc. However, it is widely recognized that, while desirable features such as these can be programmed upon contemporary low level machines (indeed, upon Turing machines!), the performance cost of implementing such features with interpretive software is unacceptable [IlifJ68, pp. 4,13; BerkK69].

Second, it has often been assumed that an overall system performance improvement can be attained by implementing a high level language more directly. Various designs have sought to avoid the costly overhead that is normally incurred by software implemented compilers and interpreters [BashT67; BashT68; BerkK69; RossC64; ThurK70; WebeH67]. At least one recent effort has included several basic operating system functions such as main storage management and process dispatching [RiceR71].

But whether the goal was enhanced functional capabilities or the accelerated performance of conventional functions, most designs of high level language machines to date have been based upon existing language-independent machine organizations. Recently, an alternative and more promising strategy for achieving these goals has been to employ a machine organization which is specifically designed for the efficient semantic interpretation of a particular high level language [AbraP70; ThurK70; ZaksR71].

I.B. Objectives

The primary objective of this thesis is to identify the performance advantages that may be attained, in principle, by a processor whose machine language is high in level. Most previous work in this area has sought to achieve higher computational rates by reducing the number of interfaces, or

layers of interpretation, between the high level language and the machine circuitry. Some efforts have gone further by tailoring the machine organization to the primary semantic characteristics of an existing programming language (e.g., to the array features of APL). This paper will concern itself with the performance improvements that may be attained over and above those that accompany a brute-force reduction in the number of interfaces. It will not restrict its treatment to the features of any particular existing language. Instead, a set of loosely compatible language features will be proposed with computational performance in mind. Then, by the consideration of relevant computational mechanisms, an attempt will be made to identify the general ways in which a machine with a suitable high level interface language can achieve a level of performance which exceeds that of a machine with a conventional von Neumann architecture.

A high level language machine will be characterized which is somewhat less flexible than a conventional design, but which significantly out-performs the conventional machine when used in a way that exploits the high level language.

It should be noted that no attempt will be made to address the difficult problems of translating from other languages into the high level machine language. In most practical systems, the machine language would have to serve as an effective target

language for such translations. Here, the sole concern is with the high speed interpretation of a single, although exceptionally powerful, language.

THIS PAGE INTENTIONALLY LEFT BLANK

II. FAVORABLE LANGUAGE CHARACTERISTICS

This section characterizes a machine language which possesses many features that are desirable in a programming system, that are impractical to implement upon contemporary machine architectures, but that favorably affect the performance capabilities of the high level language machine. The language characteristics are discussed in terms of their relation to the structure of programs and the nature of primitive objects.

This section does not comprise the definition of a new language. Such a formidable task would require a lengthy dissertation in itself. What is sought is a language characterization that is sufficient to support the following sections of this paper.

For precision, and to avoid the thorny (and here extraneous) issues of syntax, the program examples that follow in this document will use a simple LISP-like notation:

(operator operand1 operand2 ... operandN)

Notation variables and constants will be indicated by lower-case and upper-case symbols, respectively.

II.A. Program Structure

In terms of its program structure, the envisioned high level machine language most closely resembles the language LISP. Each of its programs is a structured expression consisting of an operator and an optional list of operands; and each operand is, in turn, a structured expression.

As in LISP, the operators and operands within the text of a program are merely symbols whose meaning depends upon the environment (or context) in which the program is invoked. Of course, various static and dynamic symbol resolution mechanisms are possible; but what is important is that, in general, the semantics of a program cannot be determined prior to symbol resolution time (which would typically be as late as program activation time).

The motivation for these features is twofold. First, it is desired that there be an equivalency between values and expressions. It should be possible to replace any value with an expression that evaluates to (or "returns") that value, and vice-versa. In other words, all language constructs should be closed under composition. Second, there is to be no sacred distinction between builtin operators and user-defined programs. It should be possible for a user to redefine any "system" operator, within his own local environment, by providing a

program with the appropriate name. Moreover, the redefinition of an operator should not in itself require changes to those programs that use the operator.

The language also possesses a large and powerful set of builtin operators (a superset of the operators of APL). Of singular importance are the execution-sequencing operators which perform functions analogous to those of the PL/I DO and IF statements, the COBOL PERFORM statement, etc. For example, there would be some sort of SEQUENCE operator which evaluates its operands in strict sequence, a PARALLEL operator which evaluates its operands without regard to order (perhaps employing concurrent hardware processing), a REPEAT operator which would repetitively evaluate one of its operands either some number of times, or until some condition is met, and so forth. (see Figure 1 on page 10)

It is further stipulated that the programs for the high level language machine be "pure". That is, an executing program may in no way modify itself. This constraint promotes the generation of shareable software, outlaws many "tricky" programming practices, and eliminates certain common types of execution-time programming errors. It also has implications regarding the organization of the language interpreter.

```
(SEQUENCE expression1 expression2 ... expressionN)
(PARALLEL expression1 expression2 ... expressionN)
(REPEAT expression1 expression2 TIMES)
(REPEAT expression1 WHILE expression2)
(IF expression1 THEN expression2 ELSE expression3)
```

Figure 1 : Typical execution-sequencing operators

Another important feature of the proposed high level language is its facility for processing exceptional conditions. In this regard, it is unlike either APL or LISP, but similar to PL/I [M10, pp. 104-106] with its conditions, ON-units, and SIGNAL statement. A single exception handling mechanism is provided which handles both builtin and user-generated exceptions in a uniform way. Upon the occurrence of an exception, the current activation chain is sequentially searched (from the most recent activation to the system "root" activation) for an exception-action-specification (which consists primarily of a program to be executed). If an exception-action-specification which corresponds to the exception is found, then it is executed as if it were invoked in the context in which the exception occurred. It may choose to inspect or modify any variable in that context (subject to the authorization mechanism), to signal another exception, to return to the point of interruption, to execute a return (with value)

from the interrupted program, to suspend the process (which results in a SUSPENSION exception in the process which owns this process), and so forth. If no corresponding exception-action-specification is found, then an EXCEPTIONERROR exception is signalled. The system root activation always provides an exception-action-specification for this exception.

II.B. Primitive Data Types

The set of primitive objects that are recognized by the machine includes such aggregate objects as vectors, arrays and tuples. This implies that each object in the system has an associated descriptor which contains information regarding its type (e.g., program, character, integer, real or complex) and shape (e.g., scalar, vector, tuple), as well as an indication of its ownership, persistence and access-authorization.

Consequently, the machine is able to examine the attributes of the objects that it manipulates, and thereby perform such functions as operator distribution, domain-rule enforcement and data protection.

The internal encodings of the object descriptors and values are inaccessible to the user. Appropriate builtin operators are provided for the purpose of converting an object from one type to another (e.g., from REAL to COMPLEX). Predicate operators,

such as ISINTEGER or ISPROGRAM, are also provided for the purpose of accessing the information in the object descriptors. Information is written into the object descriptors only by means of the BUILDOBJECT operator, the sole means for constructing objects (the conversion operators employ BUILDOBJECT).

As a result of having self-describing data which is manipulated by an attribute examining machine, it is possible to have objects whose type and/or value is undefined. Thus, there exists an object of undefined type and undefined value, an object whose type is constrained to be INTEGER but whose value is undefined, etc. Furthermore, it is reasonable to permit such undefined objects to be components of an aggregate object without requiring that the entire aggregate object be undefined. By definition, an attempt to use an undefined part of an object results in the signalling of an appropriate exception, such as UNDEFINEDVALUE. However, certain operators, such as the builtin operator which copies objects from one place to another, and programmed operators which quote their operands (by using the builtin QUOTE operator), can be applied to undefined objects or objects that contain undefined objects and will not signal an exception because they do not actually use the undefined information.

III. MECHANISMS FOR ACHIEVING IMPROVED PERFORMANCE

This section discusses the performance improvement mechanisms that become available to a machine because it has a suitable high level interface language. For the purpose of this exposition, these mechanisms are grouped into three classes: those that apply to what has traditionally been called the execution-unit (E-unit or ALU), those that apply to the instruction-unit (I-unit or CU), and a class of execution-monitoring mechanisms that do not relate to any part of a conventional machine. This classification is somewhat arbitrary; it will sometimes be the case that a particular mechanism could be viewed as belonging to more than one of these classes.

III.A. Optimization of Expression Evaluation

Part A of this section deals with the techniques that a high level language machine may use to increase the rate at which it evaluates expressions. The contextual structure of programs, the presence of primitive aggregate objects and the implicit management of temporaries are three language features which are viewed as contributing to a higher expression evaluation rate.

III.A.1. Context-Sensitive Optimizations

Because its programs are structured expressions, a high level language machine may employ a top-down method of program execution in which each encountered operator is executed in a well defined context. Such a machine possesses a wealth of knowledge concerning its computation that is not determinable prior to execution time. As a result, it is able to optimize its performance dynamically in several ways that are not possible on conventional context-free computing machines.

III.A.1.a. Avoiding Unnecessary Operations

First, the machine may use the available contextual information to avoid performing unnecessary operations. For example, in order to minimize the cost of their operation, certain builtin operators may behave differently depending upon the context in which they are executed. The results that they ultimately produce must be the same, of course, but context-sensitive "short cuts" may be used internally to improve performance. Thus, in the evaluation of the expression

(LENGTH (CONCATENATE STRING1 STRING2))

there is no need to actually concatenate the two strings [ElsoM70, p. 167]. All that is required is the length of the result of concatenating the strings. On a high level language machine, the CONCATENATE operator could recognize that it was

invoked as a direct argument to the LENGTH operator and that it need not concatenate the strings. Instead, it could return as its value a string descriptor that contained the correct result length but whose value component was undefined. This could be accomplished by accessing the string descriptors alone, with no need to even fetch the (possibly lengthy) strings.

Many other optimizations of this sort are possible. Note that this optimization could not be performed prior to execution time, as by a compiler, since the resolution of the symbols LENGTH and CONCATENATE is not then known.

However, it is not possible for context-dependent operators such as these to avoid all unnessary operations. There is a large class of more global work reduction transformations that may only be performed by the instruction stream interpreter. Abrams [AbraP70, pp. 66-68] has identified a number of these which he separates into the two processes of drag-along and beating.

Drag-along is the process whereby the machine defers the evaluation of each operator and operand for as long as possible. By deferring the evaluation of an expression it becomes possible to simplify the expression in ways which are impossible when only small parts of the expression are available. Beating consists of manipulating the deferred expressions, and

particularly the the object descriptors, in order to reduce the amount of work that needs to be done. For example, the expression

(TAKE 3 (TIMES (NEGATIVE v) vector1))

might be reduced to

(TIMES (NEGATIVE v) (TAKE 3 vector1))

by the deferral of the non-select type operator TIMES. This particular transformation avoids (MINUS (SHAPE vector1) 3) unnecessary multiplication operations.

III.A.1.b. Reordering Operations

Ramamoorthy [RamaC71] has noted that expression execution time can be minimized only if consideration is given to the ordering of subexpressions. In particular, he has shown that subexpressions should be evaluated in the order of their decreasing memory and processor time requirements. But if subexpressions are to be reordered to minimize execution time, the reordering process must be performed after symbol resolution. The overhead involved in such a dynamic process is unacceptable when the high level language is implemented upon a conventional machine, but the process may well be viable upon a high level language machine. Thus, when faced with the execution of several unordered expressions, and when unable to execute all of the expressions simultaneously, the machine could rationally choose to tackle the most resource-demanding

expressions first.

III.A.1.c. Exploiting Special Cases

The technological development of writeable control stores suggests that the microprogram for a particular machine might be many times larger than the capacity of the control store. If a facility could be provided for paging microcode between some backing store and the control store, it would be useful in implementing a high level language machine. Essentially, it would permit the machine to employ a library of highly specialized micro-procedures. Depending upon the particular operation to be performed and the context, the machine could invoke a micro-procedure that is specifically designed to handle that situation. In effect, the machine would be capable of extensive "special casing" [similar to the OMD mechanism described in ElsoM69] without requiring a larger than normal control store.

III.A.2. Parallel Processing of Aggregates

Since aggregate objects are primitive within its machine language, a high level language machine may employ specialized hardware techniques to efficiently deal with them. Homogeneous aggregates are particularly amenable to high-speed streaming through a pipeline, or direct parallel processing by cellular

logic arrays.

Indications are that, with the advent of LSI technology, logic-in-memory components will be more economical than conventional "random" logic [for justifications see HenlR69]. It will be possible to incorporate logical functions directly in the memory because the size (and complexity) of the circuit that may be placed on a chip is becoming large relative to the constraint on the number of chip-to-chip interconnections. Thus, the most cost-effective computer organizations will employ regular arrays of memory with builtin logic capabilities.

III.A.3. Parallel Processing of Special Cases

There are many operations, such as the operation of inverting a matrix, for which there exist several algorithms which exhibit various degrees of speed and applicability. It is commonly the case that there is a particular algorithm which will "work" whenever it is applied to an argument for the which the operation is defined, although it executes rather slowly. And there are several other algorithms which execute significantly faster, although they only work for special cases from the domain of arguments. Thus, in the case of matrix inversion, any nonsingular n-by-n matrix may be inverted by triangular decomposition, requiring slightly more than n^3 scalar multiplications and divisions. However, if the matrix is

symmetric, then its inverse may be obtained in only $n^3/2$ scalar multiplications and divisions [RalsA65, p. 446, p. 462].

A high level language machine whose performance is of paramount importance could exploit this situation as follows: To invert a matrix, it could execute two or more matrix inversion algorithms in parallel with a domain-test algorithm which would select the result from the fastest algorithm that properly applies.

Some other operations that have special case algorithms of this sort are the calculation of the determinant of a matrix, the eigenvalues and eigenvectors of a matrix and the zeroes of a polynomial.

III.A.4. Dynamic Management of Temporaries

In the evaluation of expressions by a high level language machine, perhaps the greatest potential performance advantage results from the ability to dynamically manage temporaries. It is customary on conventional machines for each procedure to somewhat statically possess its own set of reserved temporary cells. These cells are usually allocated at compile-time, load-time or activate-time, due to the computational expense of software implemented dynamic storage allocation. Consequently, the number of storage cells that are dedicated to use as

temporaries throughout the system far exceeds the minimum number that are actually needed.

On a high level language machine, it is possible to allocate temporaries on demand and release them immediately after their use. Since a temporary is only used within the immediate context of an enclosing subexpression, a relatively small amount of storage may be used to efficiently satisfy the temporary storage requirements of a large system.

Because the instantaneous storage requirement for temporaries is generally small, a very high speed local store that is integrated with the processor could be used to contain the temporaries. This implies that references to temporaries need not contribute to the processor-to-storage data transfer bottleneck.

III.A.4.a. Increasing the Use of Temporaries

Since references to temporaries may be far more efficient than references to operands in main storage, it may be worthwhile to attempt to increase the ratio of temporary-references to storage-references. One method for doing so is to employ a machine language that is expression oriented and has an abundance of builtin monadic operators.

The APL language possesses these characteristics; it contains a large number of monadic operators which are merely dyadic operators that assume a default value for one of their operands (e.g., the reciprocal and exponential operators). The expression oriented nature of APL is demonstrated by the large percentage of nontrivial programs that consist of a single expression. In contrast, low level machine languages are ill suited to exploit the efficiencies of temporaries, particularly when the values involved are nonscalar.

III.B. Instruction Stream Efficiencies

Part B of this section discusses four ways in which a machine with a high level interface language can benefit from having a high level instruction stream. The presence of operators for explicit execution-sequencing control, the absence of detailed and unnecessary tactical specifications, the higher density of program encoding and the freedom from much needless interlocking are presented as factors contributing to higher instruction-issuing rates.

III.B.1. Explicit Procedural Control

On high performance machines such as the Control Data 7600 and the IBM System/360 Model 195, pipelining and parallelism are

used within the E-unit to achieve a major improvement in the instruction execution rate. However, much of this increased power is wasted because the I-unit is unable to decode instructions and issue them to the E-unit at a commensurate rate [M1, pp. 10-13, 31-34; ThorJ71, pp. 124-125]. Attempts are made to decode several separate instructions simultaneously, but the nominally sequential nature of the instructions being decoded severely limits the effectiveness of this process [BuchW62; ThorJ71; M1; M4; M8]. The I-unit is continually "surprised" by conditional branches and other discontinuities which require a reloading of instruction buffers and cause a disruption in the E-unit pipeline streams. As Flynn [FlynM72, p. 21] has observed:

> ... Thus the IBM System/360 Model 91 had execution resources in excess of 70 MIPS (million instructions per sec) while this was immediately restricted at a maximum instruction decode rate of 16 MIPS; further with an average incidence of branch and data dependencies this was reduced to 6 MIPS. Thus the discrepancy between available resources of 70 MIPS and average request rate of 6 MIPS.

If a machine has a high level interface language with a program structure as described in section II.A., then most of these difficulties with the instruction stream can be surmounted. By requiring that all procedures be pure, the I-unit can be relieved of the responsibility of supporting write-operations into prefetched instructions.

Of greater importance is the reduction in the number of branch instructions that need be encountered. While the structured programming aspects of the language can be justified in user-oriented terms alone [DijkE68; DijkE70; MillH70], they have promising machine performance implications. Language constructs such as those described in Figure 1 on page 10 can convey valuable information to the processor regarding the content of an iteration, the number of times an iteration will be performed, the extent of the true and false clauses on a conditional, etc. The machine can, in principle, use this information to organize the use of its resources and thereby optimize its own performance.

III.B.2. Deferral of Tactical Decisions

A problem that plagues compiler based systems is that of allocating unique machine resources at a level of detail that requires overspecification. The statements from a high level language must be mapped into a sequence of low level instructions which reference specific machine registers and storage locations. This is a complicated task to perform, and generally requires an optimizing compiler to perform it well. Even then, what is "good code" for a System 360 Model 50 may be inefficient on a Model 85, and vice-versa. In fact, on high performance machines such as the 360/195 these a priori tactical decisions are a severe handicap. As Chen [ChenT71a, p. 74] has

note1:

> ... a piece of procedural language code retains a wealth of job independence information. A FORTRAN statement essentially describes a string of causally connected events; but adjacent statements are often locally independent of each other, and can be executed concurrently. Yet the conventional compiling process obscures causality. The resultant machine instructions are tactical prescriptions, imposing unrealistic causality demands (one instruction at a time) and arbitrary facility assignments ("register 2", "address 32768"); they becloud human understanding and impede the debugging process, and are such potential sources of computer inefficiency that machines are known to reconstruct the original statements internally for better traffic flow.

A machine with a high level interface language may avoid this problem entirely. The programs that it interprets can be free from purely machine-oriented constraints.

III.B.3. Algorithmic Encoding Density

On a conventional machine, the high level language operations that manipulate non-scalar objects generally must be implemented by means of the repetition (either iterative or recursive) of some sequence of low level instructions. Consider the addition of two vectors with the vector sum replacing one of the argument vectors. To be specific, consider a PL/I statement of the form A=A+B where A and B are vectors of length n. As Figure 2 (page 25) indicates, there are four logical parts in the program loop structure which underlies such an operation. The first of these consists of several setup instructions which are executed only once at the beginning of the vector operation.

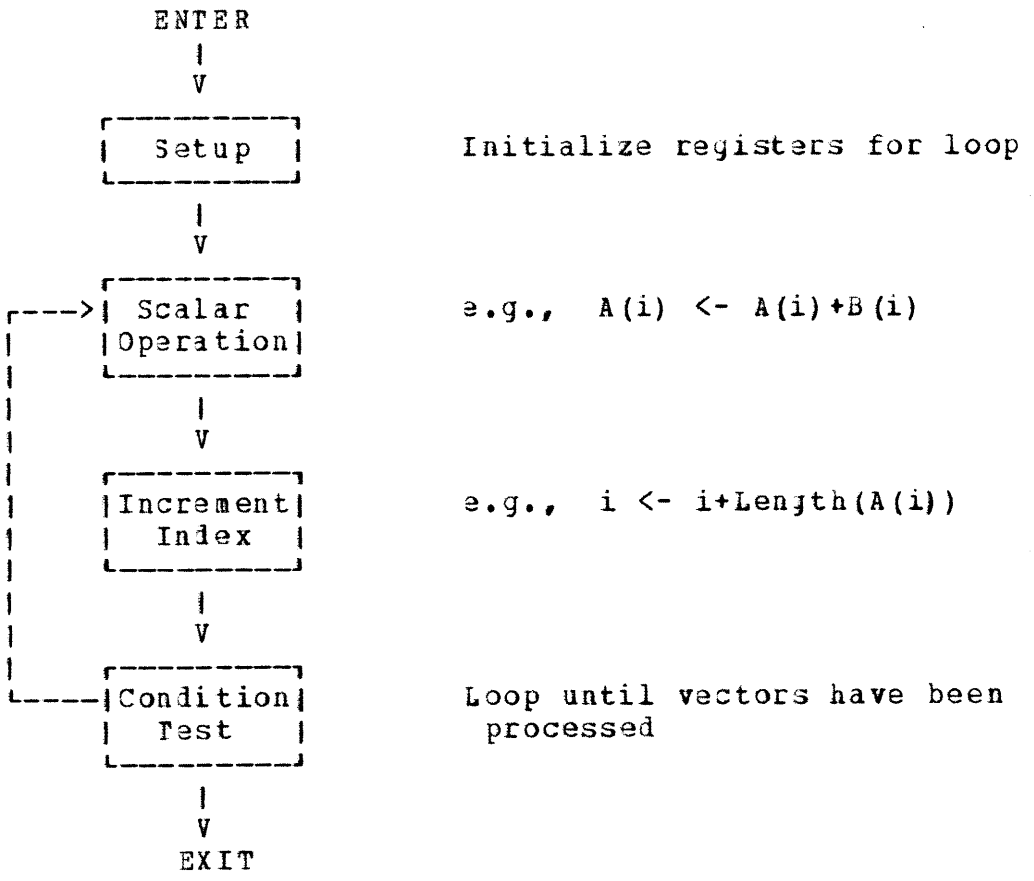


Figure 2: Conventional loop structure for vector operations

The remaining three parts, however, are iterated n times in order to accomplish the operation in an element-by-element fashion. Thus, a certain number of memory references, proportional to n, is required for the purpose of fetching instructions.

Figures 3 and 4 (page 26) contain "optimal" System/360 implementations of the vector addition and inner-product operations. These programs are optimal in the sense that they occupy the fewest bytes possible and have the shortest execution

```
         LM     R3,R5,LOOPCTRL    Setup for iteration
LOOP     L      R2,A(R3)          Fetch A(i)
         A      R2,A(R3)          Add B(i)
         ST     R2,A(R3)          Replace A(i) with sum
         BXLE   R3,R4,LOOP        Loop until A(n) is processed

LOOPCTRL DC     F'0'              Initial value for index R3
         DC     F'm'              Limit value m=n*4-1
         DC     F'4'              Increment
A        DS     nF                Vector of fullword integers
B        DS     nF                Vector of fullword integers
```

Figure 3: System/360 implementation of vector addition

```
         LM     R3,R5,LOOPCTRL    Setup for iteration
         SER    F2,F2             Clear accumulator
LOOP     LE     F4,A(R3)          Fetch A(i)
         ME     F4,B(R3)          Multiply by B(i)
         AER    F2,F4             Add to sum
         BXLE   R3,R4,LOOP        Loop until A(n) is processed
         STE    F2,INNRPROD       Store sum

LOOPCTRL DC     F'0'              Initial value for index R3
         DC     F'm'              Limit value m=n*4-1
         DC     F'4'              Increment
A        DS     nE                Vector of short float
B        DS     nE                Vector of short float
INNRPROD DS     E                 Result of INNERPRODUCT(A,B)
```

FIGURE 4: System/360 implementation of inner-product

time. They are somewhat unrealistically efficient in that they assume convenient addressability to all the required data. Nevertheless, instruction fetching accounts for over 57% of all memory references in the case of the vector addition, and over 63% in the case of the inner-product.

A machine with a high level interface language, as described in Section II, will not be burdened by this overhead of repetitively fetching atomic instructions. Since its operators, such as ADD, are builtin and automatically distribute over vectors, only a single "instruction" need be fetched in order to perform the entire operation.

III.B.4. High Level Interlocking

As noted in Section III.B.1., the presence of data dependencies in the instruction stream results in a major degradation in the instruction execution rate of a conventional high-performance machine [M1, pp. 31-34; FlynM72, P. 21]. Elaborate schemes, such as the Scoreboard on the CDC 6600 [ThorJ71; DennJ70], are required to interlock storage references in order to prevent conflicts (i.e., with respect to a given storage cell, to insure that no operation is interchanged with a write operation).

This problem will continue to exist on a high level language machine but will be of a much smaller magnitude. For one thing, most storage references made by a high level language machine will be generated internally by the machine rather than by the programmer. For example, the programmer's use of a builtin operator applied to vector operands will result in the machine generation of the numerous storage references that are

required to process the elements of the vector. Since these references are generated by a fixed algorithm that can be designed to be conflict free (in the extreme case, a pipeline schemata may be used), the machine may issue these references without the burden of interlocking. Of course, interlocking will still be necessary on a larger scale to prevent conflicts among the high level operators. But the interlocking mechanism will need to be used much less frequently.

III.C. Concurrent Error Monitoring

Hardware reliability has increased manyfold over the past ten years and is expected to continue to improve. This is largely due to developments in component technology and the introduction of sophisticated hardware-error detection and correction schemes.

Unfortunately, software has not experienced a similar improvement in reliability. Moreover, the complex operating systems which have emerged since the days of IBSYS, and which continue to expand in scope, place increasing emphasis upon reliability. It is now commonplace for a simple malfunction in the system software to crash an entire system with its many simultaneous users. Yet, despite the apparent and growing crisis, no widely-used and general-purpose system (e.g., OS/360 or CP-67/CMS) has overcome this problem. It is generally

acknowledged that powerful programming systems, as we know them today, are never completely debugged.

A major reason for the unreliability of software is that many common types of software errors cannot be detected practically on contemporary systems. If all detectable execution-time errors are, in fact, to be detected on a conventional machine with a low level machine language, then some substantial fraction of the machine's instruction execution rate must be expended continually upon error checking [a partial exception is the Burroughs B6700 family of machines which have a low level machine language that does reflect certain software requirements, particularly for block-structured languages, and that is more conducive to software reliability than other contemporary systems, but that provides only a small degree of error checking (e.g., instructions and data are distinguishable); see M5 and OrgaE71]. This cost must be incurred regardless of the machine's internal organization or the technology with which it is implemented. As long as the machine language is low in level, and hence does not convey any high level language semantics to the machine, the machine cannot employ hardware techniques (such as parallelism) to efficiently perform error monitoring. Instead, error monitoring can only be performed by means of the addition of explicit machine instructions which consume some fraction of the machine's computing power. Virtually no general-purpose programming

systems employ extensive run-time error checking because the costs involved are unacceptable.

For example, consider the problem of detecting illegal subscripting operations in a language such as PL/I. There are basically two approaches that are used. First, there is the totally interpretive approach as exemplified by CPS [M12]. In CPS, all PL/I statements are interpreted by software and subscripting errors are therefore easy to detect and handle. But the accompanying performance degradation limits the usefulness of the system to certain program development activities. In particular, it is infeasible to use such a system as the basis for a frequently executed operating system or for computation-intensive application programs.

A second approach, which is compiler oriented, is to perform subscript testing within a particular program only if a program-checkout option was specified at compile time [M11, pp. 172-173]. This scheme is based upon the assumption that one writes a program, fully debugs it using the program checkout facility, and then installs the debugged program with the error tests removed. However, in practice, many non-trivial programming bugs manifest themselves days, or even years, after a program has been in productive operation.

In order to get a rough measure of the overhead involved in performing this type of error checking on a contemporary machine, several "off-the-shelf" PL/I (F) [M11] programs were run both with and without the compiler generated SUBSCRIPTRANGE and STRINGRANGE tests. It was found that this simple type of error checking was accompanied by a 15% to 179% increase in program execution time and a 68% to 97% increase in program size.

Although the compiler generated tests were not as efficient as hand-coded tests, they were reasonably good. Perhaps the overhead could be reduced by at most a factor of two. However, it should also be noted that the test case programs did not make heavy use of subscripting or string manipulations. Programs that make extensive use of these facilities would undoubtably incur a higher penalty.

On a machine with a high level machine language, this type of error detection could be performed concurrently with the actual computation that is being monitored.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. HYPOTHETICAL HIGH LEVEL LANGUAGE MACHINE

This section describes a machine which is designed for the sole purpose of directly executing a high level language of the type described in Section II. Of necessity, many details are omitted. Some important topics such as object ownership and persistence are not even addressed. The details that are provided are intended to illustrate the nature of the machine; the specific values that are used for design parameters are meant to be reasonable but generally have not been subjected to system-wide tradeoffs.

IV.A. General Machine Organization

The proposed machine is a shared resource multiprocessor with a structure as indicated in Figure 5 (page 34). The functions of each I-unit are to step through a linearized encoding of a program written in a high level machine language, to maintain the current state of execution for that program, and to issue requests for computation to the E-unit and await the results. The E-unit services the computational needs of the I-units. It consists of a collection of specialized functional units (FU's) which are centrally coordinated.

There are a number of reasons for coupling the multiple I-units to a common E-unit. First, because the builtin

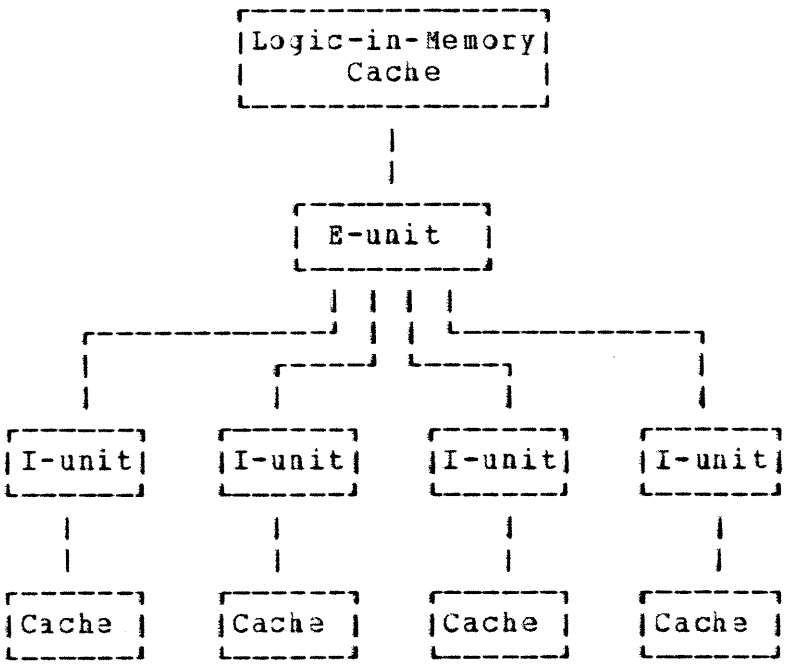


Figure 5: High level language machine structure

operators are numerous and complex, an E-unit is necessarily quite large. Furthermore, most of the FU's (such as the FU's which perform the matrix inversion, square root and index operations) are used irregularly. Thus, it is unreasonable to dedicate a complete E-unit to each instruction stream.

Second, the E-unit operations need to be interlocked in order to prevent conflicts. If multiple E-units were used, they would not really be independent, but would need to be centrally coordinated anyway.

Third, and lastly, the E-unit for a high level language machine can accept requests at a much higher rate than it can

possibly complete them -- this familiar pipelining phenomenon is accentuated by the more substantial operations that are builtin on such a machine.

The interface between the I-units and the E-unit may be either synchronous or asynchronous. Attractive approaches have been investigated by Flynn [FlynM72] and by Plummer [PlumW72]. It appears that the synchrony or asynchrony of the interface protocol is not sensitive to the use of a high level machine language.

Underlying the processor is a one-level storage system which provides an effectively inexhaustable number of uniquely named spaces. Each space consists of an ordered set of fixed length cells (16 bits per cell) which are consecutively addressed. If the space names are 48 bits in length, then up to $2.8 \cdot 10^{14}$ distinct spaces may be addressed without needing to reuse space names. At a space generation rate of one space every five microseconds, the machine could run for about 39 years before running out of unique names. Spaces created for the purpose of holding machine generated temporaries are not implemented in the one-level storage system, and do not contribute to the consumption of space names.

Each I-unit has its own cache store as an interface to the storage hierarchy. Since all programs are read-only, these

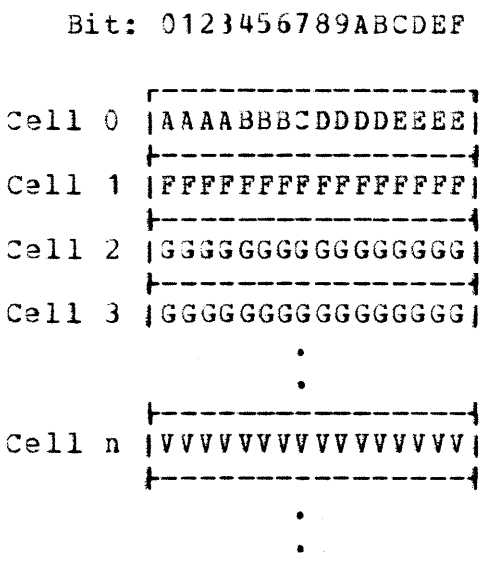
caches are unidirectional and are not interlocked, either with each other or with the activity of the E-unit. The E-unit cache possesses logic-in-memory capabilities and is organized in sectors [as described in StonH70 and ThurK70].

IV.B. Objects

Each object stored within the system consists of a space which contains a descriptor and an associated value. As shown in Figure 5 (page 37), the descriptor specifies the object type, structure and access constraints. For aggregate objects, it also specifies the object rank and dimensions.

IV.C. Aspects of Program Interpretation

There are three temporal phases in the process of interpreting a machine language program on this hypothetical machine: a translation phase in which the character string representation of a program is used to generate a PROGRAM object, an activation phase in which a PROGRAM object and an ENVIRONMENT object are used to generate an ACTIVATION object, and an execution phase in which an ACTIVATION object is executed. This section discusses several key aspects of these different phases of program interpretation.



AAAA - Object Type: Undefined, Logical, Integer, Real, Complex, Character, Label, System or Mixed

BBB - Object Structure: Undefined, Scalar, Vector or Array

C - Value Present Flag

DDDD - Object Access Constraints:
0000 - unconstrained
0001 - value constrained
0010 - type constrained
0100 - rank constrained
1000 - dimension constrained
1100 - structure constrained

EEEE - Object Descriptor Extension

FF...F - Optional rank field (present for arrays)

GG...G - Optional dimension fields (present for vectors and arrays, field repeats for arrays)

VV...V - Value encoded in as many cells as required

Figure 6: Internal representation of an object

IV.C.1. Program Representation

A PROGRAM object, which is an object of type SYSTEM, is created by applying the TRANSLATE operator to an operand which evaluates to a character string object whose value denotes a program. If errors are detected in the source program during translation, then the TRANSLATE operator signals appropriate exceptions. This allows the program that invoked TRANSLATE to decide whether to continue or to abort the operation.

As Figure 7 (page 39) illustrates, a PROGRAM object is comprised of five elements. The first is a copy of the character string object from which the PROGRAM object was derived. The second, a TEXT object, is an object of type SYSTEM which contains an encoded linearization of the program tree. The third, a LINKAGE object, is also a SYSTEM object. It serves as a linkage vector for binding nonlocal symbols. The fourth, a SYMBOL object, is a SYSTEM object which serves as a symbol table, containing such information as the symbolic names for all tokens in the TEXT object. And the fifth component is a boundary address (BDY) which is used to distinguish between local and nonlocal symbol references.

An object of singular importance is the TEXT object, which specifies the actual algorithm to be performed in the course of executing a given program. It consists of an ordered set of

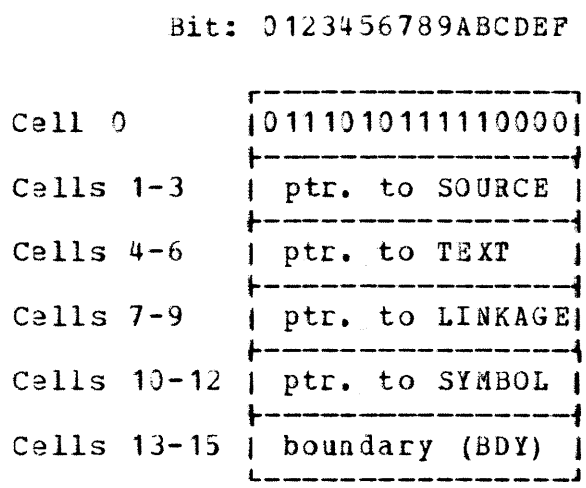
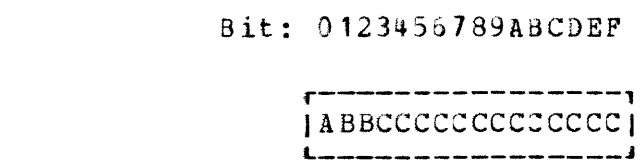


Figure 7: Internal representation of a PROGRAM object

elements that are either operand pointers or TEXT tokens of the form shown in Figure 8 (page 40). Figure 9 (page 41) contains an example of a TEXT object (note that this object has undergone symbol resolution; i.e., it is part of an ACTIVATION object).

IV.C.2. Program Activations

The builtin ACTIVATE operator is used to create an ACTIVATION object given a PROGRAM object and one or more ENVIRONMENT objects. It accomplishes this operation by making a copy of the PROGRAM object and then manipulating the new object in a privileged way. Its functions include allocating an "activation area" of storage, storing the area address into the high order 35 bits of BDY, creating the required instances of local symbols in this "activation area", binding operands to program symbols, and resolving nonlocal symbols by searching the



A - Builtin/Defined Flag

This flag is set to 1 by the symbol resolution mechanism (when creating an ACTIVATION) if this symbol resolves onto a builtin operator. Hence, during execution, builtin operators are immediately self-identifying.

BB - Operand Designator

00 - no operands (symbol is niladic)
01 - one operand (symbol is monadic)
10 - two operands (symbol is dyadic) and first operand has no operands
11 - abitrary number of operands

This field indicates the number of operands that are actually being passed to the symbol. (It does not indicate the number of operands that the symbol will accept; symbols may choose to accept varying numbers of operands) Its purpose is to reduce the number of operand pointers that are required.

CC...C - Token Offset

If this offset is greater than the low order 13 bits of BDY (in the ACTIVATION object), then this offset points to an entry in the nonlocal symbol LINKAGE object; else this offset, when appended to the high order 35 bits of BDY, constitutes the space name of an object that is local to this ACTIVATION. A program may reference up to 8192 distinct objects.

Figure 8: Internal Representation of a TEXT token

Source program:

```
(ASSIGN X (SUM Y (FCN1 (MAX X Y Z) (FCN2 Y))))
```

Corresponding TEXT object:

Bit: 012 3456789ABCDEF	Contains
011 1010111110001	object descriptor
110 'ASSIGN'	builtin dyadic opcode
000 X	niladic reference
110 'SUM'	builtin dyadic opcode
000 Y	niladic reference
011 FCN1	n-adic reference
2	operand pointer
5	operand pointer
111 'MAX'	builtin n-adic opcode
3	operand pointer
3	operand pointer
3	operand pointer
000 X	niladic reference
000 Y	niladic reference
000 Z	niladic reference
001 FCN2	monadic reference
000 Y	niladic reference

Figure 9: Example of a TEXT object

ENVIRONMENT objects in the sequence provided (each ENVIRONMENT object may also specify a successor ENVIRONMENT object).

IV.C.3. Program Execution

An ACTIVATION may be executed by the application of the builtin EXECUTE operator. The execution of a program involves a number of activities in the I-unit and E-unit portions of the machine.

The I-unit is envisioned as consisting of three major parts which operate under central control: a token fetch unit, a linkage fetch unit and an instruction assembler. The token fetch unit has its own cache from which it reads (in a highly sequential manner) the tokens that are contained in the TEXT object component of the program. It is equiped with hardware stacks so that it may conveniently recurse when walking its way through the program in a top down fashion. The linkage fetch unit reads the contents of the LINKAGE object component of the program in order to obtain the space name of an object to which a nonlocal symbol has been bound. The instruction assembler builds logical instructions for the E-unit by collecting an opcode and a list of the space names for its operands. It then issues the logical instructions to the E-unit and awaits the reply, which is either the space name of the resultant object, or an exception.

The E-unit does all the actual fetching of operands and interlocks upon the operand space names. The actual layout of the value component of an aggregate object is determined by the characteristics of the various functional units and the logic-in-memory cache. It is crucial to the performance of such a machine that its objects be internally organized in order to maximize spacial locality since the one-level store will used so extensively.

THIS PAGE INTENTIONALLY LEFT BLANK

V. CONCLUSIONS

This paper has investigated a variety of mechanisms by which a machine that directly interprets a suitable high level language might expect to achieve improved performance. One result of this effort is a catalog of such mechanisms, which may be of some use in the design of high performance computers.

Another result is an increased understanding of the significance (in terms of performance) of adopting a high level machine language. It is now the author's view that the use of a low level machine language, as an intermediate interface between the high level language and the machine, has two inherent effects upon the potential execution rate of the high level language.

First, the low level interface restricts the amount of relevant semantic information which flows from the executing program to the machine. The computer is deprived of most of the intent of the high level operations. While, with yesterday's technology, it was acceptable to decompose a program into a sequence of context-free atomic orders, advances in technology now permit a high performance machine to profitably employ a knowledge of the macroscopic operators and operands.

Second, artificial constraints are imposed upon the the computation because a low level language, by its very nature, imparts detailed tactical prescriptions. These unwanted constraints have long been an obstacle to the design of high performance machines and will become even less acceptable as the functional capabilities of hardware increases.

REFERENCES AND BIBLIOGRAPHY

Abbreviations

Journals:

ACM	Association for Computing Machinery
AFIPS	American Federation of Information Processing Societies
BCS	British Computer Society
EJCC	Eastern Joint Computer Conference
FJCC	Fall Joint Computer Conference
IBM J. of Res. and Dev.	IBM Journal of Research and Development
IBM Sys. J.	IBM Systems Journal
IEEE	Institute of Electrical and Electronics Engineers
IFIP	International Federation for Information Processing
NAECON	National Aerospace Electronics Conference
SIGPLAN	ACM Special Interest Group on Programming Languages
SJCC	Spring Joint Computer Conference
WJCC	Western Joint Computer Conference

General:

Bull.	Bulletin
Comm.	Communications
Conf.	Conference
Cong.	Congress
J.	Journal
Proc.	Proceedings
Pt.	Part
Pub.	Publication
Res. Rept.	Research Report
Supp.	Supplement
Symp.	Symposium
Tech. Rept.	Technical Report
Trans.	Transactions

Books, Periodicals and Reports

AbraP70 Abrams, P. S., An APL Machine, Tech. Rept. No. 114, Stanford Electronics Laboratories, Stanford University, February 1970

AlleF71 Allen, F. E., and John Cocke, A Catalog of Optimizing Transformations, Res. Rept. RC 3548, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, September 1971

AlleM69 Allen, M. W., and T. Pearcy, Developments in Machine Architecture, Proc. of the Fourth Australian Computer Conf., Adelaide, South Australia, pp. 227-230, 1969

Amda364 Amdahl, G. M., et al., The Structure of SYSTEM/360, Part III - Processing Unit Design Considerations, IBM Sys. J., Vol. 3, No. 2, pp. 144-164, 1964

AndeD67 Anderson, D. W., F. J. Sparacio and R. M. Tomasulo, The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling, IBM J. of Res. and Dev., Vol. 11, No. 1, pp. 8-24, January 1967

BartR69 Barton, R. S., Ideas for Computer Systems Organization: A Personal Survey, COINS-69, Third International Symp. on Computer and Information Science -- Software Engineering, December 1969

BashT67 Bashkow, T. R., et al., System Design of a FORTRAN Machine, IEEE Trans. on Electronic Computers, Vol. EC-16, No. 4, pp. 485-499, August 1967

BashT68 Bashkow, T. R., et al., Study of a Computer for Direct Processing of List Processing Language, Tech. Rept. No. 103, Columbia University, New York, January 1968

BellC71 Bell, C. G., and A. Newell, Computer Structures: Readings and Examples, McGraw-Hill Book Co., New York, 1971

BerkK69 Berkling, K., A Computing Machine Based on Tree Structures and the Lambda Calculus, Res. Rept. RC 2589, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, August 1969

BjorD70 Bjorner, D., On Higher Level Language Machines, Res. Rept. RJ 792, IBM Research Laboratory, San Jose, California, December 1970

BlocE59 Bloch, E., The Engineering Design of the Stretch Computer, Proc. of the EJCC, pp. 48-59, 1959

BrowJ71 Brown, J. A., A Generalization of APL, Systems and Information Science, Syracuse University, September 1971

BuchW62 Buchholz, W., Planning a Computer System, McGraw-Hill Book Co., New York, 1962

ChamD71 Chamberlin, D. D., The "Single-Assignment" Approach to Parallel Processing, Res. Rept. RC 3308, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, March 1971

ChenT71a Chen, T. C., Parallelism, Pipelining, and Computer Efficiency, Computer Design, Vol. 10, No. 1, pp. 69-74, January 1971

ChenT71b Chen, T. C., Unconventional Superspeed Computer Systems, Proc. of the SJCC, Vol. 38, AFIPS Press, New York, pp. 365-371, 1971

ChesG71 Chesley, G. D., and W. R. Smith, The Hardware-Implemented High-Level Machine Language for SYMBOL, Proc. of the SJCC, Vol. 38, AFIPS Press, New York, pp. 563-573, 1971

ChroG71 Chroust, G., Comparative Study of Implementation of Expressions, Tech. Rept. TR 25.112, IBM Laboratory Vienna, Austria, March 1971

ContC69 Conti, C. J., Concepts for Buffer Storage, Tech. Rept. TR 00.1852, IBM Poughkeepsie Laboratory, February 1969

CurtR71 Curtis, R. L., Management of High Speed Memory in the STAR-100 Computer, Proc. of the IEEE International Computer Society Conf., pp. 131-132, September 1971

DaviR71 Davis, R. L., and S. Zucker, Structure of a Multiprocessor Using Microprogrammable Building Blocks, NAECON Record, pp. 186-200, May 1971

DennJ69 Dennis, J. B., Programming Generality, Parallelism and Computer Architecture, Proc. of the IFIP Cong. 1968, North-Holland, Amsterdam, pp. 484-492, 1969

DennJ70 Dennis, J. B., Modular, Asynchronous Control Structures for a High Performance Processor, Record of the Project MAC Conf. on Concurrent Systems and Parallel Computation, ACM, New York, pp. 55-80, June 1970

DennJ71 Dennis, J. B., On the Design and Specification of a Common Base Language, Symp. on Computers and Automata, Polytechnic Institute of Brooklyn, April 1971

DijkE68 Dijkstra, E. W., Go To Statement Considered Harmful, letter to the Editor, Comm. of the ACM, Vol. 11, No. 3, pp. 147-148, March 1968

DijkE70 Dijkstra, E. W., Structured Programming, Software Engineering Techniques, Scientific Affairs Division, NATO, Brussels 39, Belgium, pp. 84-88, April 1970

ElsoM69 Elson, M., R. A. Larner, et al., A Prototype PL/I Optimizing Compiler, IBM Tech. Rept. TR 44.0071, IBM Systems Development Division, Boulder, Colorado, November 1969

ElsoM70 Elson, M., and S. T. Rake, Code-Generation Technique for Large-Language Compilers, IBM Sys. J., Vol. 9, No. 3, pp. 166-188, 1970

FleiH70 Fleisher, H., A. Weinberger and V. D. Winkler, The Writeable Personalized Chip, Computer Design, Vol. 9, No. 6, pp. 59-66, June 1970

FlinM70 Flinders, M., et al., Functional Memory as a General Purpose Systems Technology, Proc. of the IEEE International Computer Group Conf., pp. 314-324, June 1970

FlynM65 Flynn, M. J., and G. M. Amdahl, Engineering Aspects of Large High Speed Computer Design, Symp. on Microelectronics and Large Systems, Spartan Books, pp. 77-95, 1965

FlynM66 Flynn, M. J., Very High Speed Computing Systems, Proc. of the IEEE, Vol. 54, No. 12, pp. 1901-1909, December 1966

FlynM72 Flynn, M. J., and A. Podvin, Shared Resource Multiprocessing, Computer (A Pub. of the IEEE Computer Society), pp. 20-28, March/April 1972

FostC71 Foster, C. C., Uncoupling Central Processor and Storage Device Speeds, The Computer Journal, A Pub. of the BCS, Vol. 14, No. 1, February 1971

GardP71 Gardner, P. L., Functional Memory and Its Microprogramming Implications, IEEE Trans. on Computers, Vol. C-20, No. 7, pp. 764-775, July 1971

GertJ70 Gertz, J. L., Hierarchical Associative Memories for Parallel Computation, Tech. Rept. MAC TR-69, Project MAC, Massachusetts Institute of Technology, Cambridge, June 1970

HassA71 Hassitt, A., Microprogramming and High Level Languages, Proc. of the IEEE International Computer Society Conf., pp. 91-92, September 1971

HenlR69 Henle, R. A., et al., Structured Logic, Proc. of the FJCC, Vol. 35, AFIPS Press, New York, pp. 61-67, 1969

HobbL70 Hobbs, L. C. (Editor), et al., Parallel Processor Systems, Technologies, and Applications, Spartan Books, New York, 1970

HussS70 Husson, S. S., Microprogramming: Principles and Practices, Prentice-Hall, 1970

IlifJ68 Iliffe, J. K., Basic Machine Principles, American Elsevier Publishing Co.(in Europe: MacDonald, London), New York, 1968

JohnJ71 Johnston, J. B., The Contour Model of Block Structured Processes, SIGPLAN Notices, Vol. 6, No. 2, pp. 55-82, February 1971

JoseE69 Joseph, E. C., Computers: Trends Toward the Future, Proc. of the IFIP Cong. 1968, North-Holland, Amsterdam, pp. 665-677, 1969

LawsH68 Lawson, H. W., Jr., Programming-Language-Oriented Instruction Streams, IEEE Trans. on Computers, Vol. C-17, No. 5, May 1968

MadnS71 Madnick, S. E., An Analysis of the Page Size Anomaly, Project MAC, Massachusetts Institute of Technology, December 1971

McCrD71 McCracken, D., and G. Robertson, C.ai(P.L*) -- An L* Processor for C.ai, Rept. No. CMU-CS-71-106, Dept. of Computer Science, Carnegie-Mellon University, October 1971

McFaC70 McFarland, C., A Language-Oriented Computer Design, Proc. of the FJCC, Vol. 37, AFIPS Press, New York, pp. 629-640, 1970

McKeW67 McKeeman, W. M., Language Directed Computer Design, Proc. of the FJCC, Vol. 31, AFIPS Press, New York, pp. 413-417, 1967

MeggJ64 Meggitt, J. E., A Character Computer for High-Level Language Interpretation, IBM Sys. J., Vol. 3, No. 1, pp. 68-78, 1964

MelP69 Melliar-Smith, P. M., A Design for a Fast Computer for Scientific Calculations, Proc. of the FJCC, Vol. 35, AFIPS Press, New York, pp. 201-208, 1969

MilH70 Mills, H. D., Structured Programming, Unpublished Paper, IBM Corporation, Federal Systems Division, Gaithersburg, Maryland, October 1970

MosJ70 Moses, J., The Function of FUNCTION in LISP or Why the FUNARG Problem Should be Called the Environment Problem, Artificial Intelligence Memo No. 199, Project MAC, Massachusetts Institute of Technology, June 1970

MulA63 Mullery, A. P., R. F. Schauer and R. Rice, ADAM: A Problem Oriented Symbol Processor, Proc. of the SJCC, Vol. 23, AFIPS Press, New York, pp. 367-380, 1963

OrgE71 Organick, E. I., and J. G. Cleary, A Data Structure Model of the B6700 Computer System, SIGPLAN Notices, Vol. 6, No. 2, pp. 83-145, February 1971

PlumW72 Plummer, W. W., Asynchronous Arbiters, IEEE Trans. on Computers, Vol. C-21, No. 1, pp. 37-42, January 1972

RalsA65 Ralston, A., A First Course in Numerical Analysis, McGraw-Hill Book Company, New York, 1965

RamC69 Ramamoorthy, C. V., and M. J. Gonzalez, A Survey of Techniques for Recognizing Parallel Processable Streams in Computer Programs, Proc. of the FJCC, Vol. 35, AFIPS Press, New York, pp. 1-15, 1969

RamC71 Ramamoorthy, C. V., and M. J. Gonzalez, Subexpression Ordering in the Execution of Arithmetic Expressions, Comm. of the ACM, pp. 479-485, July 1971

RiceR71 Rice, R., and W. R. Smith, SYMBOL - A Major Departure from Classic Software Dominated von Neumann Computing Systems, Proc. of the SJCC, Vol. 38, AFIPS Press, New York, pp. 575-587, 1971

RoseS68 Rosen, S., Hardware Design Reflecting Software Requirements, Proc. of the FJCC, Vol. 33, Pt. 2, AFIPS Press, New York, pp. 1443-1449, 1968

RossC64 Ross, C., et al., A New Approach to Computer Command Structures, Tech. Rept. No. RADC-TDR-64-135, Rome Air Development Center, Griffis Air Force Base, May 1964

RuggJ69 Ruggiero, J. F., and D. A. Coryell, An Auxiliary Processing System for Array Calculations, IBM Sys. J., Vol. 8, No. 2, pp. 118-135, 1969

SammJ69 Sammet, J. E., Programming Languages: History and Fundamentals, Section X.6., Prentice-Hall, pp. 717-719, 1969

SchrM72 Schroeder, M. D., and J. H. Saltzer, A Hardware Architecture for Implementing Protection Rings, Comm. of the ACM, Vol. 15, No. 3, pp. 157-170, March 1972

SenzD65 Senzig, D. N., and R. V. Smith, Computer Organization for Array Processing, Proc. of the FJCC, Vol. 27, Pt. 1, AFIPS Press, New York, pp. 117-128, 1965

SenzD67 Senzig, D. N., Observations on High Performance Machines, Proc. of the FJCC, Vol. 31, AFIPS Press, New York, pp. 791-799, 1967

SethR70 Sethi,R., and J. D. Ullman, The Generation of Optimal Code for Arithmetic Expressions, J. of the ACM, Vol. 17, No. 4, pp. 715-728, October 1970

ShawJ58 Shaw, J. C., et al., A Command Structure for Complex Information Processing, Proc. of the WJCC, pp. 119-128, 1958

SingS71 Singh, S., and R. Waxman, Adder for Multiple Operands and its Application for Multiplication, IBM Tech. Rept. TR 22.1356, IBM Components Division, East Fishkill, New York, October 1971

StonH70 Stone, H. S., A Logic-in-Memory Computer, IEEE Trans. on Computers, pp. 73-78, January 1970

SumnF71 Sumner, F. H., Operand Accessing in the MU5 Computer, Proc. of the IEEE International Computer Society Conf., pp. 119-120, September 1971

ThorJ70 Thornton, J. E., Design of a Computer: The Control Data 6600, Scott, Foresman and Company, Glenview, Illinois, 1970

ThurK70 Thurber, K. J., and J. W. Myrna, System Design of a Cellular APL Computer, IEEE Trans. on Computers, Vol. C-19, No. 4, pp. 291-303, April 1970

ThurK71 Thurber, K. J., and R. O. Berg, Applications of Associative Processors, Computer Design, Vol. 10, No. 11, pp. 103-110, November 1971

TjadG70 Tjaden, G. S., and M. J. Flynn, Detection and Parallel Execution of Independent Instructions, IEEE Trans. on Computers, Vol. C-19, No. 10, pp. 889-895, October 1970

TomaR67 Tomasulo, R. M., An Efficient Algorithm for the Automatic Exploitation of Multiple Execution Units, IBM J. of Res. and Dev., Vol. 11, No. 1, pp. 25-33, January 1967

TuckA71 Tucker, A. B., and M. J. Flynn, Dynamic Microprogramming: Processor Organization and Programming, Comm. of the ACM, Vol. 14, No. 4, ACM, New York, pp. 240-250, April 1971

WareW72 Ware, W. H., The Ultimate Computer, IEEE Spectrum, pp. 84-91, March 1972

WebeH67 Weber, H., A Microprogrammed Implementation of EULER on IBM System/360 Model 30, Comm. of the ACM, Vol. 10, No. 9, pp. 549-558, September 1967

ZaksR71 Zaks, R., Microprogrammed APL, Proc. of the IEEE International Computer Society Conf., pp. 193-194, September 1971

Manuals and Miscellaneous

[M1] IBM System/360 Model 195, Functional Characteristics, Form A22-6943-0, International Business Machines Corporation, Poughkeepsie, New York, August 1969

[M2] IBM System/360 Model 195, Theory of Operation: System Introduction and Instruction Processor, Form SY22-6855-0, International Business Machines Corporation, Poughkeepsie, New York, August 1970

[M3] NCR 304 Electronic Data Processing System, Programming Manual, National Cash-Register Company, June 1960 (obtained through the courtesy of Jean Sammet)

[M4] Control Data 7600 Computer System, Preliminary Reference Manual, Pub. No. 60258200, Control Data Corporation, Minneapolis, Minnesota, 1969

[M5] Burroughs B6500/B7500 Information Processing Systems, Characteristics Manual, Burroughs Corporation, Detroit, Michigan, 1967

[M6] IBM System/370 Model 165, Functional Characteristics, Form GA22-6935-0, International Business Machines Corporation, White Plains, New York, June 1970

[M7] A Guide to the IBM System/370 Model 165, Form GC20-1730-0, International Business Machines Corporation, White Plains, New York, June 1970

[M8] IBM System/370 Model 165, Theory of Operation (Volume 2): I-unit, Form SY22-6831-0, International Business Machines Corporation, White Plains, New York, January 1971

[M9] APL/360 User's Manual, Form GH20-0683-1, International Business Machines Corporation, White Plains, New York, March 1970

[M10] PL/I Language Specifications, Form Y33-6003-1, International Business Machines Corporation, White Plains, New York, April 1969

[M11] IBM System/360 Operating System, PL/I (F) Language Reference Manual, Form C28-8201-2, International Business Machines Corporation, White Plains, New York, October 1969

[M12] Conversational Programming System (CPS), Terminal User's Manual, Form GH20-0758-0, International Business Machines Corporation, White Plains, New York, January 1970

[M13] Weissman, C., LISP 1.5 Primer, Dickenson Publishing Co., Belmont, California, 1967

[M14] McCarthy, J., et al., LISP 1.5 Programmer's Manual, MIT Press, Cambridge, Massachusetts, February 1965

[M15] Griswold, R. E., et al., The SNOBOL 4 Programming Language, Prentice-Hall, 1968

[M16] Sussman, G. J., and T. Winograd, Micro-Planner Reference Manual, Artificial Intelligence Memo No. 203, Project MAC, Massachusetts Institute of Technology, July 1970

[M17] Evans, A., Jr., PAL -- Pedagogic Algorithmic Language, Reference Manual and Primer, Dept. of Electrical Engineering, Massachusetts Institute of Technology, October 1970

[M18] Reynolds, J. C., GEDANKEN - A Simple Typeless Language Based on the Principle of Completeness and the Reference Concept, Comm. of the ACM, Vol. 13, No. 5, ACM, New York, pp. 308-319, May 1970

- FINIS -