HIGHER LEVEL SYSTEM

(IBM CONFIDENTIAL)

Final Report
The Machine Organization Concept Study Group
John C. McPherson, Chairman

IBM Corporate Headquarters
Armonk, New York
May 18, 1970

IBM CONFIDENTIAL

HIGHER LEVEL SYSTEM

Final Report

The Machine Organization Concept Study Group
John C. McPherson, Chairman

IBM Corporate Headquarters
Armonk, New York
May 18, 1970

(1) Is vN invoked to indicate something special about machines executing instruction? All do, whatever its level.

(2) The Scots say: not proved

(3) All machines executes statements interpretively

(4) Have always disagreed 100% with this concept.

(5) why?

SUMMARY

The Machine Organization Concepts Study Group recommends that an effort be made to explore the possibilities of a Higher Level System (HLS) of the character described in this report.

(1) We have come close to the limit of the von Neumann instruction-based machine as the vehicle to meet the expanding needs of electronic data processing; the more direct, powerful HLS approach may be needed to sustain IBM growth in the years ahead and extend the use of computers to more people.

HLS raises the man-machine interface from the present instruction level to the (2) level of procedural languages, at which people communicate more naturally and more effectively.

The system uses statements whose operators handle variables identified by name rather than address. Variables are not confined to single values; instead entire arrays and structures can be processed as units of information as well.

(3) HLS executes the statements interpretively, through the help of descriptors which dynamically define the data object - its type, size, precision and current location.

LSI and its memory-like character are exploited in the systematic use of associative techniques for both statement scanning and orderly control of machine functions.

The system design gives the user full access to storage. The convenience of a name-oriented single level store is maintained for a multi-level hierarchy.

(4) System control will be achieved more efficiently by a command language which is simply a facet of the procedural machine language of the system. Use of HLS will thus extend to the full range of problems and to background as well as foreground control.

The user and IBM should both gain substantially from the easier coding and debugging of concise programs. We expect that the cost of programming, and size of complex programs will sharply reduce, as both program quality and programmer productivity are enhanced. This should have particular relevance to IBM in the Systems Control Program and Program Product areas. Interactive computing will feature better turnaround and higher efficiency.

(5) The Field Engineering cost for maintaining the system will be markedly reduced. There naturally will be fewer chances for errors. Automatic meaning-dependent checks allow errors to be detected and localized sooner; simple errors will tend not to propagate. In the face of errors the system can redeploy its resources. Diagnostic programming will be simpler. The self-monitoring features of HLS apply equally to machine failures and programming errors, and further permit selective protection of user information.

With HLS, programmers will be able to exploit important techniques heretofore handicapped by inefficiency, and to explore new techniques. The self-defining information enhances meaningful communication in a complex environment, and HLS further will be able to reconcile minor inhomogeneities in the program, thus taking a step towards the transferability and combinability of programming material.

The new system will form a sound basis for growth, and should facilitate the implementation of data-base systems and large shared systems, both of which have the potential to expand the future pattern of computer usage.

The following statement was unanimously endorsed by the Study Group on February 25, 1970:

> "The Machine Organization Concepts Study Group has studied the question of feasibility and advisability of a higher level system and concludes that such a change of direction is both feasible and necessary and very advantageous to the Company's expansion, both to new fields of application and to larger numbers of users. It offers a way for consolidating the advances in the knowledge in use of machines in the past 25 years and forms a firm base for future development and will use to advantage new technologies."

HIGHER LEVEL SYSTEM (HLS)

Final Report of the Machine Organization Concepts Study Group

CONTENTS

1. Introduction: The Machine Organization Concept Study

The objectives of the study group were to examine the feasibility and potential benefit in developing a computer with a higher level instruction set. It had been suggested that the high cost of programming might be reduced by exploiting the steady improvement in the performance/cost ratio of electronic technology.

From our discussions, additional objectives emerged. Central questions were, whether we should rethink the basic concepts for a stored program computer, and how to introduce features which would enhance both communications between man and machine, and reusability of programs.

A review of the various recent and current higher level machine efforts showed that it might be possible to consolidate 25 years of computer experience into a new architecture basis, through a re-examination of the size of the unit of machine procedure. The instruction, which specifies one operation at a time, appeared to be too small a unit; increasing numbers of people are writing concise procedural language programs using statements containing a number of operators, each of which may operate on an array of numbers.

Several members of the study group have direct experience in the study of procedural-language machines. Further, we heard from all the groups that we could identify who had ideas in this area. In particular we were interested in five such current efforts: the interpretive machine approach in Large Advanced Systems, Poughkeepsie; the SOMAC proposal from RD, San Jose; the Intermediate Systems FS architecture effort in Endicott; Hursley's programming experiment of an incremental compiler for a PL/I subset; and the APL/25 machine at Palo Alto Scientific Center, DPD.

All these efforts were found to have many common features. During our discussions a conviction developed that there exist intriguing possibilities for a higher level machine complementing the S/360 - NS series, implemented from a fresh viewpoint without too great a departure from present computer organizations.

In order to arrive at an answer to the basic questions of feasibility and potential benefit we outlined and discussed such a machine as a basis for evaluation, and described it in this report.

2. Current Instructions versus Future Needs

The first systematic development of computer instructions was published by Burks, Goldstine and von Neumann in 1946, 24 years ago.

Since then hardware technology has seen three revolutions (transistor, SLT hybrid, and integrated circuits), and we are experiencing a fourth (large-scale integration). Machine organization also has shown great diversity since the von Neumann days. But it is a tribute to his genius that the basic instruction orientation has continued to this day, despite important additions (indexing, floating-point arithmetic, interruption features) and numerous changes of details. The S/360 design, for example, features many different machine organizations sharing the same instruction set.

The original computing environment a quarter century ago can be contrasted with the present:

	Then	Now
a)	Memory was expensive and small; circuitry was expensive, slow and space consuming.	Inexpensive and large; inexpensive, superfast and small.
b)	Activities outside the CPU were infrequent.	System activity is such that CPU is only 30% utilized.
c)	Computing was mainly numerical.	Numerical work is small part of computing; even "numerical" problems have large data-processing load.
d)	Machine time was the most precious item.	Cost of human effort most important.
e)	Users were willing and able to conform to machine rigidity.	Few users are willing, able to perform this contortion.
f)	Programming was mainly done in absolute binary (octal, hex).	Use of procedure languages is widespread.
g)	Machine handles one user at a time.	Many users' jobs handled concurrently.

Stripped to the barest essentials, an instruction in a current machine is a quantum of procedure specifying an operation code and one or more addresses. Each address locates a single operand, which is a string of bits of fixed length with no a priori meaning. The operation code alone provides the detailed action on the operands.

Computing with instructions is characterized by unlimited freedom to use operands independent of context; indeed context has no meaning if an operand is not currently being referred to by an instruction. Thus it is possible to perform arithmetic on an operand, then use it immediately as a branch target.

Such context freedom had been necessary when the user had to make do with a small memory and very limited hardware; it was literally the only flexibility available to him. Instruction modification, for instance, once was the only good technique to specify a loop.

In time, however, machines became powerful enough that the handling of things out of a prescribed context is no longer needed, and is being consciously avoided as a major potential source of programming error.

The processing of featureless bits, while practicable for small, static programs of short duration of usefulness, ill-suits the dynamic communication needs in future computing. Procedural languages, such as FORTRAN, COBOL, PL/I and APL, are all based on associating meaning to information. Complex programs today commonly attach descriptors to information, as an interpretive software technique. With expected availability of powerful, low-cost electronic logic, this technique can be hardened into an architecture discipline in which descriptors are usually provided, queried and updated dynamically, and are omitted or ignored only when the meaning is understood by the machine.

Computation proceeds in many stages, program execution being just one of these. The most critical stages in the future are probably those connected with the human user, including the origination, editing, debugging and reuse of programs, also the documentation, filing, indexing and inquiry on information.

The spectacular growth of programming systems software (language processors and system control programs), serving as a cushion between man and machine, is a major development since von Neumann. They are predicated on the assessment of values in the entire computing process, such that local efficiencies may be traded for overall system economy involving hardware, software and the user. Despite new added instruction features, the required services demanded of programming systems have caused them to grow into entities of great size and high complexity.

As computers assume an ever increasing portion of the complex data-handling needs of the human society, the value trade-offs will need to be reassessed. Machine usefulness will tend to be judged more and more on (a) the enhancement of human convenience, and (b) built-in orderliness for reducing complexity. It is submitted that current instruction sets may not be conducive to future overall system economy. A critique is found in Appendix A.

Essentially, current instructions, aside from having a "context freedom", appear to be too small a procedural unit for future needs. They mask the true cause-and-effect relationships, demand what has now become unrealistic details (addresses, register assignments). They entail the linear address restrictions, and require excessive user attention on memory, I/O requirements. An array is not treated as a unit of data, though human and hardware would both benefit by so doing.

Programmers now seldom use instructions. They employ mostly procedure languages and expect the machine to be a procedure interpreter or a virtual machine. The programmer is frustrated whenever the system fails to behave as a proper emulator. Poor emulation is practically axiomatic with instructions, as context is deformed and information lost in the mapping into machine language. As a result, compiling cannot easily achieve source language debugging, even with a sizable processing overhead (500 executions to provide one compiled instruction). Interpreters try to maintain the original context but are slow. The expected future trend towards interpretation means that machines should try to handle procedure code more directly, and should embody generalized interpretive mechanisms.

3. Architecture Highlights of HLS

The basic theme in HLS (the Higher Level System) is information with personality. The aim is to:

Bring the system to users' level,
Enhance system performance,
Exploit technology advantages,
Establish man/program/machine system communicability,
Form a new system basis for the 1970s and beyond.

Highlights of HLS are as follows:

(HLS)	(Contrasted with current systems)
1. HLS operates on statements;	not instructions.
2. HLS references procedures, data by name;	not physical location (address).
3. Descriptors of attributes are held with information;	not featureless bits with no prescribed meaning.
4. Attributes are examined dynamically at execution time;	no proliferation of op codes, conversion routines, intractable errors.
5. Descriptors may be modified by operators;	no code modification or extra data processing.
6. Operators are valid on arrays and structures;	no element-by-element looping.
7. Automatic storage hierarchy management;	no static assignment, preplanned overlays, explicit hardware address.
8. Direct support for operation in PL/I, COBOL, APL, FORTRAN;	no hex dumps, delphic error messages.
9. Executes efficiently a new interpretive language embodying best semantic features of the four above;	no proliferation of language interpreters.
10. Supports hierarchical control structure and asynchronous processes;	no unbounded freedom.
11. Performs correct highspeed decimal arithmetic;	no conversion errors, uncertainties.

3.1 Statement Orientation

The unit of machine procedure is a multi-operator statement, e.g.,

A = B + C * D

The statement is a unit of work specified by the user. The exact length is not a primary concern, though processing efficiency should be geared to commonly used lengths, with a few operators.

Because of self-described data (Section 3.2), the detailed specification of the action will be associated with the operands themselves; the arithmetic operators will be fewer in number, and more general in scope (array operation is automatic, and "mixed-modes" can be accommodated). Consequently the statement will be concise and comprehensible to the reader.

The statement delineates the cause-and-effect of the execution sequence clearly. The use of temporary registers for intermediate results is at the choice of the system, requiring no effort by the user (who does not want to do it at all) or the compiler (for which optimum register assignment is a major unsolved problem). The system can dynamically optimize the assignment strategy much better (witness the common data bus on the M91/195).

The omission of intermediate register assignments has other important consequences. During the generation of (B+C*D) the operands will not be destroyed; nor will A be if the temporary result (say T) is made distinct from A. The machine error created by arithmetic will be fully recoverable. Only the quantity T would be in error, and it can be voided; the whole computation can be retried.

Usually, the computation needs to be validated only once per statement, and interruptions will tend to occur at statement boundaries, allowing meaningful memory dumps and comprehensible error messages.

The contrast with current processing is striking. The end of every instruction currently is a potential interruption point, requiring machine tests consuming time and/or circuitry. When the interruption does occur, the machine has no concept of the user's intentions. The contents of all registers (indeed all storage of the user) have to be treated as valid, causing elaborate saves and reloads. The interruption performed out of user's context often creates error messages unreadable by the user.

Another important aspect of statement orientation is protection against unauthorized branch action. In standard machines every address is a potential branch target. A wrongly executed branch can be disastrous.

In HLS only the beginning of a statement is a suitable branch target, reducing the branch error probability by an order of magnitude. Attempts to branch into mid-statement or data will lead to unexecutable situations, and will trigger an error signal.

3.2 Self-describing Information

Information is referred to by names, or machine-assigned identifiers. Their use in lieu of addresses bypasses the "linear address" problem. Each name refers to a descriptor, the latter summarizes the properties of the information and either contains, or points to, the information itself.

Self-description applies to programs, procedures, subprograms, hardware features, and branch targets.

The size of descriptors varies, with a large upper limit. The encoding allows the most common types of information to have short descriptors. There is an "escape hatch" encoding of a short descriptor, which may point to an extension of itself. The design is expected roughly to correspond to Huffman encoding:

> best service for most frequently encountered requirements,
> lower quality service for rare occurrences,
> almost open-ended spectrum of service.

Computation details are based on the descriptors of the operands. Typically a data descriptor contains specifications of:

> type (e.g., floating hex)
> structure (e.g., 3 x 15 matrix)
> constraints (e.g., read only), etc.

In a dyadic operation the descriptions of the two operands are examined for compatibility before the operands are processed. The result is given a descriptor appropriate for the computation. Security enforcement, error checking, even the monitoring of usage frequency, can be done during attribute examination.

With data descriptors many different formats can coexist and be reconciled dynamically.

Nonnumbers can be accommodated using descriptors, without expending storage otherwise. Important ones are, "null" and "undefined".

The descriptors can be manipulated by qualified users (Section 3.3). And, although HLS normally handles information based on descriptors, operations outside the standard context could be an important supervisor function, and should be permitted with restrictions.

Information transmittal includes the movement of descriptors, and is a meaning-preserving operation. The implications on asynchronous processing are profound, and the attachment of meaning to data may be the only rational basis for a data-base system.

The use of names and descriptors instead of direct addresses may seem to require an added level of indirection. This level is not needed if the information is short enough to fit into the descriptor. Also, this level need not be visited a second time in a "lookaside" machine if the information left over from the first visit remains valid. For array processing the one indirection cost enables the processing of many elements.

3.3 Descriptor Manipulation

In HLS the manipulation of descriptors can take the following forms:

a) During computation, the descriptor of the result operand may change automatically to reflect the new structure requirement (see Fig. 3.3-1).

b) Users may inquire about part of the descriptor contents (e.g., what are the dimensions of the matrix DOG?).

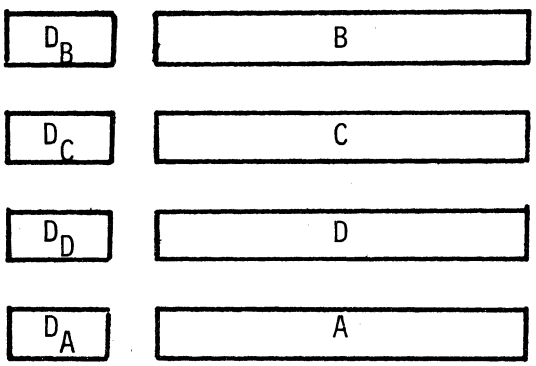
c) Privileged users (e.g., the Supervisor) can read, alter, create, and destroy descriptors.

d) A descriptor may point to another descriptor. Sometimes this is the "escape hatch" mechanism to obtain arbitrarily extensible descriptor sets; at other times it may be employed to achieve indirection, when the descriptor acts like a pointer in list processing. Also several descriptors may point to the same object, in principle, to achieve synonymy and cross-indexing.

In structure processing and in data-base systems we tend to deal with descriptors for a long while before accessing the data object. Large systems in the past have often created internal descriptors to facilitate processing when operands are inaccessible; this is systematized in P. Abrams' "APL Machine", which can postpone execution through descriptor manipulation, often reducing large computations to trivia. It is therefore expected that HLS will stimulate studies on the representations of objects, not just the objects themselves.es.

A Possible Execution of the HLS Statement

A←B+C*D



Prior to execution D_A,A may or may not exist.

Schematic HLS Action:

Step 1. Check D_B, D_C, D_D for data consistency;
- if inconsistent but comformable, go to fix up measure (return to Step 2).
- if irreconcilable, go to error interruption.

Step 2. If consistent, B+C*D is performed, yielding T (say).

Step 3. Check to see if there had been a computing error;
- if intermittent error, re-try from Step 2.
- if unfixable error, go to error interruption.

Step 4. If no error, create D_A, invalidate previous D_A,A; associate T to D_A, T thus becomes A.

(Note: Steps can be made concurrent, and can be omitted based on prior history)

Fig. 3.3-1. HLS Descriptor Orientation

3.4 Processing of Arrays and Structured Data

Like the user, HLS views an aggregate (i.e., a data collection) as a unit of information, and automatically processes it according to their descriptors. The standard array element may be a bit, a character, or a number. Non-standard elements could be other arrays, pointers, "null", or "undefined".

Users should not have to specify detailed space requirement or dimensional information. The operation will be interpretive, based on descriptor contents. All else should be automatic, including the adjustment of the result descriptors, and creating space to house the result array.

The success of APL attests to the power and convenience of descriptor-driven array processing.

A minimum array capability may consist of (a) extending the use of standard arithmetic operators (defined for scalars) to variable size vectors and arrays, (b) concatenation, selective deletion and expansion of vectors, (c) conversion between vectors and arrays, (d) generation of named arrays, (e) extracting subsets of an array, and (f) obtaining properties of an array.

A structure (as in PL/I) is an array of possibly dissimilar elements, each of which could also be a structure. A payroll file is a structure (character strings mixed with numeric data). The full "explosion view" description of, say, an automobile, down to the nuts and bolts would correspond to a large complex structure, with well-defined substructures (parts).

The structure and substructures are processed through their descriptors. Typical structure operations involve the generation of structures, attachment/detachment of substructures, and the matching of forms and/or values.

The dynamic reshuffling of structure forms, important in engineering interactive graphic processing, is not present in PL/I, yet is easy to achieve via the HLS interpretive action. The act of replacing an engine part, for example, would correspond to the detachment and re-grafting of substructures. The structural details are usually unaffected; the requirement is actually one of descriptor manipulation.

3.5 Automatic Storage Hierarchy

In the past few years it has been recognized that a clear conceptual distinction exists between storage and ultimate (source-sink) I/O. The latter deals with the world outside the system, and is subject to the users' explicit choices of format and the information-carrying medium. Storage, on the other hand, is like CPU memory, perhaps in a less accessible form. The need is to make large amounts of storage as accessible as CPU memory.

The user prefers to access all backing stores in the same way as accessing CPU memory, without expending additional specifications which are not only a chore, but a major barrier to program portability. The need for uniformly accessible storage hierarchy is well-known, and partial success has already been achieved. Especially noteworthy is the "cache" buffering technique, exemplified by the M85 and M195.

The storage hierarchy requirements in the HLS system are distinct from the others due to the name/descriptor approach. Access in HLS is through the names, and the descriptors give meaning to the information; this feature extends to the entire storage hierarchy.

Within the HLS system, format, location and detailed access techniques may be changed from time to time. These are not the users' main concern, but are automatic system functions. The user treats the entire storage as one homogeneous CPU memory, accessed uniformly via names.

This way meaning is preserved as information is displaced from one echelon to another. Bit configurations, on the other hand, need not stay fixed; they can be altered reversibly to reflect the detailed trade-off between information density and remapping cost. Automatic data compression for archives is definitely permitted, accompanied by an appropriate descriptor. As archival material moves toward the CPU, reversible decompressions can occur somewhere along the path without necessarily any performance penalty.

Examples of data compression are: (a) recoding of decimal digits from 8-bit characters to 4-bit characters, or to 10-bits for three characters; and (b) full textual data compression using Huffman encoding.

Members of the storage hierarchy should have logical autonomy, which for larger systems may become true physical local autonomy; each echelon being served by a powerful, yet inexpensive, controller device. It now becomes possible to have a form of "vertical" division of labor: simple processing can be done by the local controller far from the CPU, avoiding staging penalties. An example might be the simple updating of a payroll file.

Still another technique is to update the descriptor, but postpone detailed action, thus avoiding both staging and untimely processing.

3.6 A Programmable Machine Language

The HLS machine language should allow the full control of pertinent machine facilities and be easily decoded and executed by the intended machine. It further must be a programming language in its own right, dealing with names, descriptors and multi-operator statements.

Moreover, we expect the HLS language to contain key features of FORTRAN, COBOL, PL/I and APL. These four languages are extremely important in the computing community, and exhibit diversity as well as commonality. A "semantic union" may be large, unmanageable and possibly not self-consistent. The HLS language is required to be a concise language into which the above four can be mapped with little information loss, and (with the help of tables) with full recoverability (see Fig. 3.6-1).

Other primary production programming languages, notably RPG, LISP and ALGOL, can be handled easily, using either interpreters or compilers written in the HLS language. The latter thus behaves like a language funnel or a common language interface to the machine, capable of dealing with current languages as well as new languages yet undevised. Further, current machine languages (such as S/360) can be emulated.

It would not be possible to define such a language but for the descriptors, which can tolerate format differences, and can form the basis for generalized interpretation.

The common major languages do not satisfy the requirements of this kernel language; the descriptor concept tends to be underexploited, and system control functions are minimally provided. APL does have descriptors, is concise and decodable, and handles arrays admirably. However, system control is still largely unspecified, and certain well-regarded features of PL/I, COBOL and FORTRAN are absent.

We hold the opinion that the correct HLS language can be formulated, and this is a subject of the highest urgency.

HLS as a machine system is to be a faithful procedure interpreter of the four languages. The user who submits programs in those languages can expect debug messages, dump and error fixups in source language terms. Source language data formats will be honored; this is possible because of data descriptors.

A simple consequence of the multilingual nature of HLS is that programs in different languages can be linked and executed together despite format and syntax inhomogeneities.

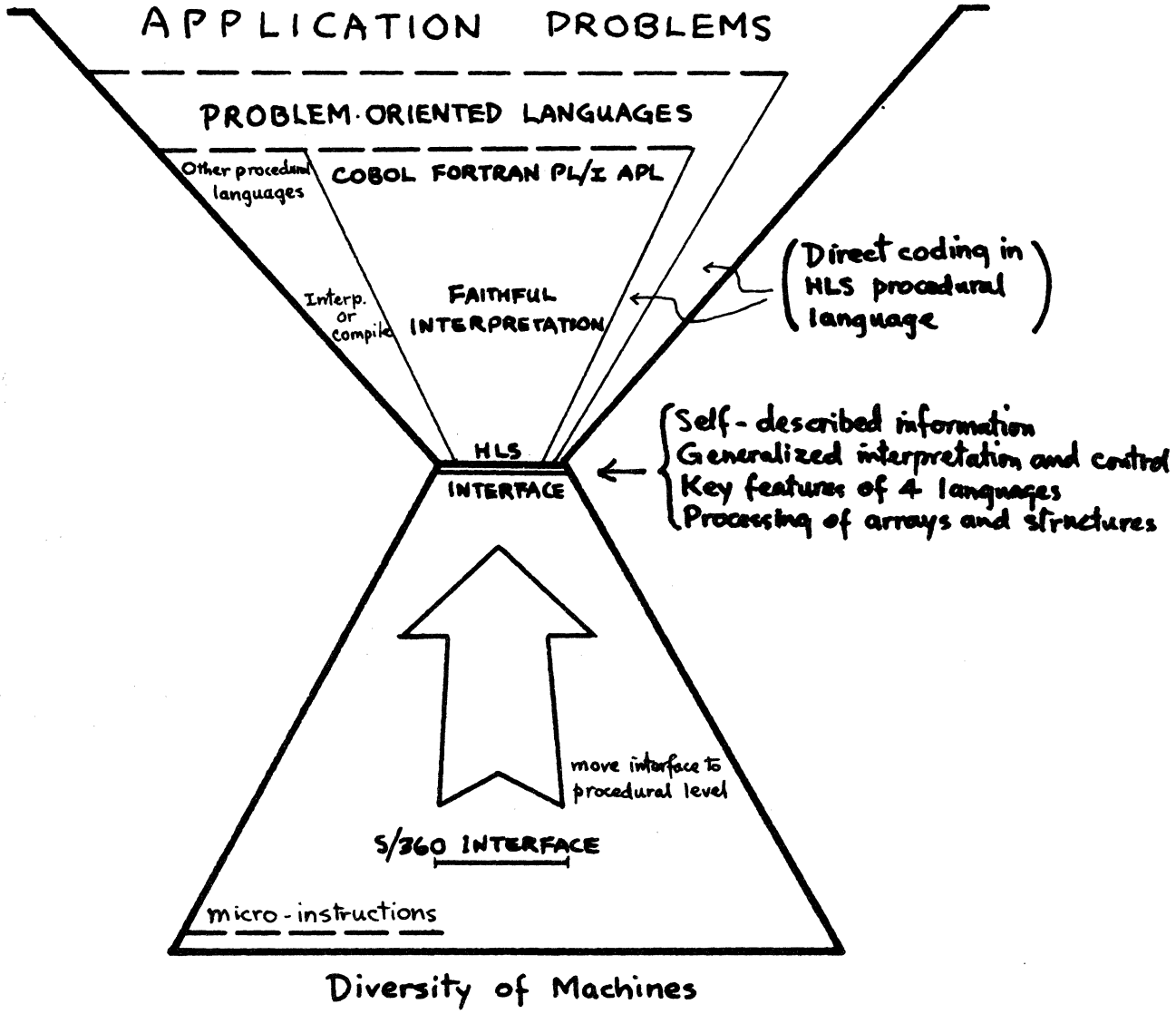


FIG. 3.6-1. A Programmable Machine Language

3.7 Structured Control

HLS uses the same language for both procedure and control. This has been successful in simple language systems, and should extend to this general context. Every (or nearly every) statement should be usable as part of a procedure specification or as a command.

The language should include systematic handling of cumulating condition indicators, editing statements to control source text, command features for interactive computing, and system control features such as suspending or restarting program execution, breakpoint control, storage hierarchy management, exception handling, supervisor functions, diagnostics, and communication aids.

The language should specify clean interfaces for the initiation, monitoring, and termination of asynchronous functions.

Hardware units behave like asynchronous functions, and should be handled on the same basis and be nameable. Their descriptors can furnish information about the hardware function, and this way we can achieve self-declaration for processing units, storage extension units (disks, tapes, etc.), I/O units (printer, etc.), and communication equipment.

User-defined functions should have the same syntax and execution environments as built-in arithmetic operators. Special functions may demand a special environment, and language features should be found to permit this, leaving little trace of the host environment and yet retaining the capability to monitor the process. The system is expected to be a generalized interpreter apparatus providing orderly transition of asynchronous decentralized control.

It is expected that the HLS features cumulatively form a basis for a new pervasive system architecture, unifying the handling of hardware, software and user requirements. This is seen in the preservation of meaning (name orientation, self-defined information), choice of units (statements, arrays and structures), monitoring of paths (descriptor handling, interpretive action), deployment of resources (device independence, asynchronous control) and the scope of the HLS language.

The total operational degrees of freedom within this architecture will be significantly fewer than in the von Neumann machine. Yet each allowed channel of action tends to be more meaningful, more orderly, less error-prone and will be backed by the total system resources.

3.8 Decimal Arithmetic

In the past quarter century we have tried to get people to use numbers to the base 2^n, n = 1 for binary and 4 for hexadecimal.

This attempt has not proved successful. Decimal notations, traditional for millenia, continue to predominate. Practically all procedural languages now use decimal input and output. Some languages (like COBOL) demand decimal arithmetic internally; most, however leave the internal radix unspecified, which most implementers have chosen to be a power of two.

The conversion between two radices creates errors (1/5 is exact in decimal, but not in power-of-two radices), which become "apparent bugs" in the users' program. (The tolerance of these apparent bugs, on the other hand, may leave genuine bugs undiscovered.)

The reason for choosing power-of-two radices was once efficiency, now it is mainly compatibility with the immediate past.

Arithmetic units in most machines today are but a small fraction of the total system, and their performance is seldom the bottleneck to computation. The choice of radix is no longer a basic issue for system efficiency or economy.

Indeed the implementation of decimal arithmetic on binary circuitry can be more efficient than 2^n-radix and more LSI adapted, by going to a redundant number representation, exploiting the extra code points in each (4-bit encoded) digit. This is the subject of a working paper (Section 9.1). The best system now seems to be one with 12 states (0 to 11) per digit.

The new representation allows computation on an independent digit-by-digit basis, without carry propagation chains or long carry-lookahead wiring. Highspeed arithmetic units can be highly modular, and add time will be independent of word length. Additions can proceed from left to right, obtaining early overflow indication, avoiding back-and-forth sweeping.

The capacity loss in the decimal notation can be redressed by mapping reversibly into a base-1000 system for archival storage, coming to within 2.5% of binary efficiency. The format change can be indicated in the descriptors.

In HLS decimal arithmetic should be emphasized. Power-of-two radix arithmetic should also be available for compatibility. It must be stated that bit pattern manipulation is not a problem for arithmetic, but a logic function for arrays, and will become easier and more powerful in HLS.

4. Qualitative Advantages of HLS

It has been emphasized that HLS seeks to retain the personality that information normally possesses when it is outside the machine, and avoids flattening it into featureless bits, as has generally been the case since von Neumann.

Important advantages should result; these cannot be properly quantified today, but would appear to lie in the following main areas:

a) machine efficiency
b) programmability
c) protection of system and user
d) system advantages

It is essential that the realizability of these potential values be tied down as quickly as possible.

4.1 Machine Efficiency

HLS requires special emphasis on associative hardware techniques for table management, variable field-length data handling and simultaneous testing of sets of data. The systematic LSI arrays have memory-like characteristics, and will be well-exploited here.

Some HLS features benefit machines of all sizes; these include the elimination of array processing decode overhead, reduction of interruption tests, better error control, freedom from linear address constraints, shorter codes, and sharp lowering of compiling cost and software overhead.

In addition, HLS should offer particular efficiency advantages to "large" and "small" machines. A typical small machine today has a narrow data path (8 or 16 information bits), microcoded writable control store, small memory capacity and limited I/O. An example is the S/360 M25.

A typical "large" machine is exemplified by the M195. It has CPU concurrency, branch/storage anticipation, word-length insensitive processing power, large memory and extensive I/O.

As computer designs continue to evolve, the distinction between the two classes may disappear. An HLS feasibility study may even accelerate this process.

4.1.1 Large Machines

On large machines the direct use of HLS, or the interpreting of procedural languages on HLS, should be highly efficient. There is reason to believe that to these machines, instructions are an unnecessary understructure; their removal would eliminate bottlenecks.

A very large machine often devotes part of itself to manage the resources at hand, to achieve self-optimization. This is difficult for the instruction-oriented machines, but is easier for HLS.

There will be no intermediate result register fixations in the procedure, and the entire storage hierarchy, including registers, can be brought under system control. Full pipelining becomes a more common occurrence. Array processing allows the system to reserve equipment in advance to exploit repetitions.

For large arrays, memory requirements are not based on access, but bandwidth. It is reasonable to put most of the arrays on a slow but wide memory, which can deliver a "line" of many consecutive words at the same time, with excellent bandwidth when all or most of the words are needed.

Descriptor handling can take place concurrently with arithmetic, without slowing down the latter. Lookahead/lookaside mechanisms permit bypassing the pointer mechanisms for often-used information or often-invoked procedures.

The system can adhere to the human-oriented causality chain contained in statements, and thus remove bottlenecks arising from the previous need to examine every instruction for conditional branch or interruption. Interruptions will be fully recoverable if the storing of the results is preceded by a test.

There are new possibilities for HLS efficiency. It is possible to "crack tokens" in a procedural code at the rate of one token per CPU cycle or better, using memory chips and associative techniques. Also there is a new way to do multiple adds at 4 words/CPU cycle or better, using array logic. Another item is "associative search for match". These actions, involving operands of unspecified length or multiple operands, are hard to specify using instructions.

The user of HLS need no longer lay claim to large tracts of memory for possible data insertion; he simply gets what he needs. The memory hierarchy is efficiently used, and multiprogramming on a large scale becomes more meaningful.

For large machines self-autonomy of major units is the key to performance. With self-described information, the parcelling of workload to subprocessors becomes more well-defined, more efficient, and less risky.

4.1.2 Small Machines

At the other end of the scale, a smaller machine views the S/360 instructions as an unneeded superstructure. Direct interpretation of procedural language code in microcode "cuts out the middleman", so to speak, and enables a higher degree of efficiency.

Small machines often have narrow data paths (8, 16 bits), which lend themselves naturally to character string processing. This is typical in procedure languages, especially with the assistance of new LSI hardware. "Token cracking" at one character per cycle is an instance.

As an illustration of these points, work just completed in Palo Alto indicates that a S/360 M25, interpreting APL directly in microcode, performs roughly at a par with the much costlier S/360 M50 APL system using S/360 instructions. The M25 is 25 times faster in syntax analysis (100 microseconds per token vs. 2500 in the M50) and its APL performance is limited only by arithmetic speed.

Micro-instructions depend heavily on hardware details, and do not form an adequate basis for architecture. However, the emulation of instructions by microcode requires, in effect, the normalizing to S/360 boundary conditions at the end of every S/360 instruction. The emulation of the HLS procedures would require normalization only at statement boundaries, sharply reducing red-tape action.

In small machines the instruction decoding cost is often high. With array processing, one decode can serve many useful arithmetic operations.

Memory is in critical supply for a small machine. We expect HLS codes in general to occupy less space than S/360 codes. It is pertinent that the users will not be forced to overclaim territory; dynamic data handling supplies the users' needs. Storage hierarchy, at the very least, should allow larger programs to be run, allowing multiprogramming on a modest scale.

The small machine user of the past has had to be resigned to slow processing of one-shot jobs. Even with compile-and-go techniques, such jobs have been compile-bound. With HLS the compile cost will be revised sharply downward, and the quality of the computing service improves because of the interpretive nature of the system and debug facilities.

A possible penalty is the data descriptor manipulation cost for scalar operands. It is hoped that this can be avoided by added hardware count, new hardware, and/or better microcode. In any case, this overhead should generally be lower than the S/360 emulation overhead for the small machine.

4.2 Programming

The need for programmers is steadily increasing, while the average prior computer training per programmer declines. There is a critical need to relieve the programming problem, by making the machine more human-oriented and hence more useful. The problem has two facets: (a) the problem solver and (b) the systems programmer.

4.2.1 HLS and the User

HLS is believed to be well-attuned to human needs, and is expected to greatly enhance programmer productivity and the quality of programming. This is not only because HLS handles procedure languages (many software systems also do this), but because it is expected to handle them faithfully and efficiently. This is possible only with hardware support. As a result standards programs should run faster, with fewer bugs and better debugging aids. Interactive computing turnaround will greatly improve.

In addition, the HLS machine language is in itself a comprehensible procedure language, capable of generalized interpretation, control of machine features, also flexible array and structure processing; these should have a major favorable influence on the future scope and effectiveness of programming.

The storage hierarchy allows name-oriented computation to proceed with little regard for program size. Computing techniques which have not been popular due to the space and time costs, can now be reconsidered. In particular, dynamic structure manipulation and descriptor processing, in conjunction with storage hierarchy, will lend impetus to the handling of representations of objects. Examples are list processing, graphic processing and information retrieval. A closely related field is simulation, which is further assisted by the generalized interpretation capabilities.

The dynamic management of resources using descriptors allows the systematic debugging of programs, changing array sizes and format with each rerun, to selectively test different aspects of the program.

Potentially of the highest importance for programming, but not as yet well understood, is the effect that HLS could have on program reusability.

From the earliest days programmers have attempted to bring about a situation in which the borrowing of program material, rather than the originating of it, is paramount. These ideas have found expression in the building up of libraries of program material, and in the idea of macro-processors. The limited success which these ideas have found (for most program material is still originated, not borrowed) is at least partly due to the ultrasensitivity of present-day systems to inhomogeneities which might exist in program-program, program-data, or program-machine interfaces. Almost any such abnormality is sufficient to prevent or invalidate execution, and in the borrowing mode such abnormalities are of course expected.

One route out of the difficulty lies in the development of tolerant systems which can bridge differences in machine configurations, data types, formats and programming languages, and compensate for minor errors. Through its data-descriptors and interpretive mode of execution, HLS takes a forward step towards the combinability of programs.

4.2.2 Systems Programming and Complex Programs

The HLS hardware machine already is largely a high level system. The requirements of system software are expected to be much easier to satisfy than in other machines. Further, the powerful interpretive processing, formalized asynchronous control, better protection, all contribute to simplify the software programming using the HLS language directly.

The language processor creation will concentrate on the detailed support of the four selected languages (FORTRAN, PL/I, COBOL, APL), and facilities to map other languages into HLS.

The system control program (SCP) will provide human interfaces, interactive command language handling, resource allocation and accounting, input/output, teleprocessing, also other facilities for multiprogramming and multiprocessing. Under the new structural discipline, the size of SCP for HLS is expected to be a fraction of OS/360 size, requiring far less programming effort.

It is believed that not only the HLS systems software but large complex programs in general will benefit by HLS features. Examples of complex programs are:

simulation programs
transaction programs (e.g., SABRE)
design automation programs
artificial intelligence programs
information retrieval - data base programs.

Some of the HLS advantages have already been given (Section 4.2.1).

All large complex programs can gain from the unrestricted symbolic addressability afforded by the storage hierarchy. No overt filing/swapping effort is needed.

Systematic control of asynchronous processes is explicitly required in many of the complex programs. In HLS the ability to treat autonomous units as a subroutine will be a powerful technique for SCP, transaction programs and simulation.

Interpretive processing using descriptor-like objects is a natural technique to delineate orders, preserve tractability and facilitate interchange. For language processors the generalized descriptors are symbol table entries or dictionary entries, containing the names and attributes of language variables/structures. For system control programs the control blocks are the descriptors, describing the states and interrelations of system variables (logical and physical resources, units of control, etc.).

HLS hardens these software schemes into a systematic architectural discipline, and should result in better economies, protection and efficiency. Problems with much higher complexity can now be handled with greatly reduced risk.

Current complex systems, especially programming systems, tend to have a high density of conditional branch occurrences. The HLS-supported procedural code will permit recasting entire networks of these conditional requirements into memory-like actions, in effect permitting modelling in terms of finite-state machines. An opportunity thus presents itself to bring formal computer science techniques to bear, imposing a systematic discipline from the outset, with prescribed global behavior. This can prevent "I didn't think of that combination" bugs. Finite state machines as formal models may also be a basis for rigorously proving required properties of system components.

4.3 Protection of System and User

The critical availability of Field Engineering resources, and the increasing complexity of computing tasks, demand high reliability, availability and serviceability in future systems. The latter should be less prone to failures, should be resilient in the face of errors, and further should make errors more localized, less propagable, more tractable and easier to correct.

HLS goes a long way along these lines, by enforcing an integrated system architecture discipline, instead of relying on user self-restraint, software techniques and add-on hardware features.

The descriptor-driven processing leads naturally to a hardware-implemented self-monitoring system. Misdirected occurrences, usually uninterpretable, automatically triggers the supervisor control. This error monitoring requires no overt testing action by the decoding control, and has no extra timing overhead.

For instance, if A is a standard matrix of 3 rows and 4 columns, then A(1), A(5,2), A(1,2,3) elements do not exist. A(3,4) is a valid element, but A(4,3) is not. The operation (A+B) has no meaning unless B and A obey rigorous compatibility rules. Branching into A, like branching into mid-statement, makes no sense. The importance of these meaning-dependent checks are seen by their current availability as software features, despite the consequent sacrifice of both program conciseness and execution efficiency. HLS would just treat these cases as exceptions to normal processing, and alert the supervisor. The self-monitoring applies equally to hardware malfunction, software failure and programming bugs.

Catastrophic propagation of mishappenings tends not to occur at all in a meaning-controlled environment. The supervisor (with software help) can usually identify the error cause, and notify the user in meaningful terms, and take appropriate countermeasures.

The automatic storage management and systematic asynchronous control both lend resiliency to the HLS system. Faulty devices or memory areas of known error occurrences can be bypassed, and the entire collection of hardware boxes can be dynamically reconfigured. The workload will be equitably shared by components of the new system. These features enhance asynchronous multiprocessing, which in itself is a major means to achieve RAS.

The system, together with the control capabilities of the procedural machine language, can influence diagnostic programming towards self-diagnosis, self-correction and interactive maintenance.

Through the descriptors, each piece of named information can be independently checked for accuracy and protected for security. Therefore selective emphasis can be applied on important programming material. A new opportunity arises to replicate important items and file away copies for safekeeping; the original descriptor can record the existence of duplicates.

For the purposes of security, especially in a multiprogramming environment, descriptors can be selectively locked to bar access (read, write) from users without proper passwords. This protection scheme can also be dynamically altered.

4.4 System Advantages

The advantages of HLS as a system are seen in the following aspects:

Improved performance per user dollar,
Influence on IBM costs,
Communicability,
Future processing.

4.4.1 Improved Performance per User Dollar

The total dollar outlay to sustain a typical current installation is divided into three roughly equal components: system rental, application programming cost, and installation operation cost. The CPU rental for a typical M65 installation is about 20% of the systems rental, or less than 7% of the total cost.

Therefore, if the projected LSI is used in the obvious way to reduce only CPU costs, the benefit to the user will be minor, and the value to IBM is uncertain. The more rational way to maintain IBM growth and to leapfrog the competition, is to adapt the new flexibility of hardware to enhance the overall economy of the user.

By increasing programmer productivity, reducing operation overhead and exploiting non-CPU equipment more efficiently, the user's total outlay will decrease and a larger fraction of the user's dollar will be channeled into IBM revenue.

For example, the price of the CPU is equal to 1/5 that of programming. It would be worth doubling the CPU price if the user's programming cost is reduced by 25%.

The programming improvement through HLS is expected to be very significant (Section 4.2.1-2). The programmer can concentrate on problem solving, unencumbered by the previous lack of rapport between language and machine. Even for languages outside the five official ones, it will be far easier to construct efficient interpreters. The programmer shortage will be alleviated as programming output is enhanced in quality and quantity.

The HLS software overhead will be very low; excellent turnaround and better throughput are expected. Currently the CPU is only busy 30% of the time; the rest of the time is spent in "I/O" and "wait" states. With hardware-assisted system control (Section 3.7) one expects much higher overlap and fewer artificial interlocks. Non-CPU equipment, currently costing 3.5 times the CPU, can expect to be much better utilized.

The system is better self-checked and is easier to maintain, tending to create fewer machine-room disasters. Software maintenance and updating will be simpler and non-obtrusive. The activity known as SYSGEN, which establishes the correlation between the machine configuration and the software, formerly takes hours to complete; with HLS self-declaration of asynchronous units this should be materially simplified.

There will be transitional problems for the users of standard machines to change over to HLS. The path is made smoother in three ways. First, HLS is procedural language compatible, especially concerning FORTRAN, COBOL, PL/I and APL. Secondly, S/360 data formats will be honored by HLS. Furthermore, the machine will be able to emulate S/360 or NS in detail.

4.4.2 IBM Costs

HLS probably represents a sizable increase of circuit count. This should not mean a proportionate increase of CPU cost, as other factors (packaging, power supply, software) need to be considered.

With LSI the circuit count may not even be a measure of circuit cost. High density regular arrays with low interchip connections, typified by memory chips and associative logic, tend to cost less and be more efficiently packaged than "random logic", with a much higher interconnection/circuit ratio. The flexibility in HLS requires "memory-like" circuitry, and LSI advantages will be well-exploited.

Software development and maintenance costs are an important part of the prorated system cost to IBM. With the HLS machine already at the procedural language level, the software costs will decrease (Section 4.2.2). The delivery time lag between hardware and software, heretofore a factor difficult to control, will be noticeably narrowed.

The HLS system being inherently reliable (Section 4.3), Field Engineering maintenance cost will be lowered, reducing further IBM's cost for providing the system to users.

4.4.3 Communicability

Computer systems are handling the needs of an increasingly large segment of the human society. The latter, at the same time, is becoming increasingly complex due to a large measure to the ever-increasing need to communicate, and to interact across boundaries.

In HLS information and its descriptor form a self-defined entity, and information transmittal is a meaning-preserving process. In this way communication is put on a formal, new basis.

For man-to-man and man-to-program communication, HLS offers comprehensible code, self-documented data, and selective security enforced through descriptors. For program-to-program interaction, HLS offers the prospect of combinability, through an ability to reconcile minor differences without enforcing a unique language convention. For program-to-machine communication, HLS offers faithful interpretive execution, easy reruns with altered formats and memory requirements, and program transferability despite configuration differences. HLS offers a generally higher quality man-machine interface, featuring the procedural machine language (Section 4.2). For the machine-machine interface, HLS aims to enforce a system discipline across all hardware/software linkages (Section 3.7).

4.4.4 New Techniques and New Applications

HLS will give new impetus to interacting computing, list processing, language processing, graphic display handling, and is a good vehicle for practicing multiprogramming and multiprocessing.

IBM has identified the class of data-base applications as a promising class for growth; HLS offers direct hardware support for several important requirements, and should prove to be a natural vehicle.

From the start, HLS aims at a simple, device-independent method of accessing information throughout the storage system, via names and descriptive material rather than rigid addresses. The use of descriptors promotes automatic data compaction for low-usage archives, and leads naturally to the handling of representations of information. These features, plus the ability to handle structural or textual material interactively, are key to the data-base problems.

Another possible new application area is the large shared system, which practices multiprogramming on a vast scale, overcomes geographic limitation, and averages out peak usages of a multitude of users. The basic requirements are communication in a complex processing environment, and asynchronous multiprocessing.

Both the data-base system and the large shared system call for meaningful security of RAS, and a major common concern is the construction of adequate software to handle the complex environments. HLS has provided for these requirements.

5. Feasibility and Practicability

We have thus far given only strong indications of the architecture advantages of HLS. The full definition of HLS has not been made; and there is as yet no proof of feasibility nor detailed evaluation of practicability. All these are beyond the scope of the present short-range study, and depend on follow-up efforts in the future.

HLS has far-reaching implications on virtually all aspects of computing, including language specification, CPU organization, storage hierarchy and human interface. As a completely new coherent system, the combined merits of the HLS characteristics are believed to greatly exceed the sum of the individual points. By the same token, a local failure in handling one of the aspects may severely compromise the worth of the remainder.

The study of HLS feasibility and practicability is handicapped by both the sheer magnitude of the problem and the paucity, to date, of detailed specifications. While we view the architecture to be feasible and advantageous, one must settle the questions by actually trying to define the HLS language, construct prototype hardware/software and test under realistic environments. This calls for an exploratory program lasting several years, engaging quality personnel.

This exploratory program very probably will succeed; but it might also fail. It is not free of risks.

There have been numerous studies on higher level language machines, both within and outside IBM (Bibliography, Section 9). Many HLS features already exist in current software. Actual microcode emulation of higher languages has been achieved, notably in Endicott ("Euler" on M30), Boeblingen (RPG on M20 submodel 5) and Palo Alto (APL on M25). All these efforts give encouraging indications of feasibility and potential worth, and at the same time point to the non-trivial nature of the HLS study.

We recommend that the HLS study be launched with adequate resources, as soon as possible. Either it will demonstrate the basic soundness of concept, quantify the application range and produce prototype machines; or conversely, (and in our view improbably) it may give an early proof of infeasibility.

We have studied alternative schemes and partial measures, but take the position that HLS (if feasible) is superior to any of these, and NS can provide the needed time buffer for the HLS study.

The program can cover aspects of both large and small machines. By a small machine is meant one with 8 or 16-bit data width, writable control store and "microcode". By a large machine we mean one with CPU concurrency, branch/storage anticipation, and word-length insensitive processing. The M25 is thus a small machine, and the M195 a large machine. It is entirely possible that "large" and "small" properties may reside in the same future machine.

Reasons for considering these two classes of machines initially:

a) There are a priori advantages to implement HLS for these machine classes (Sections 4.1.1-2).

b) The "study cycle" on small machines is short enough that major redefinitions can be contained with low risk and minimum time loss. Indeed there can be two kinds of redefinitions: remicrocoding, and rebuilding the entire hardware.

c) Small machines can probably find a ready, profitable market segment implementing a subset of HLS architecture.

d) The importance, magnitude, and complexities of data-base system problems call for large HLS system study. The same is true for large shared systems and computer utility.

e) LSI is expected to be particularly advantageous for "large" and "small" systems.

In the exploratory program the most crucial aspects are:

a) the consistent definition of the HLS language, including both data handling and control; and,

b) implementation feasibility as a hardware/software system.

Both require special emphasis in order to obtain early answers of feasibility.

Next come the questions of a lower urgency, but which are needed to delineate the role of HLS in the market:

c) range of applicability and projected economy (which markets? which type machines?);

d) usefulness in new markets (e.g., data-base systems); competitive edge in the mid-1970's.

We feel that the system will be feasible and will make economical sense for a large segment of the future market; we actually expect HLS to prove competitive even in general purpose computation of the future, not just limited to interactive-type markets.

It is expected that concurrent to the HLS study, related study projects in data-base systems, large shared systems and storage hierarchy will take place regardless of HLS interests, furnishing valuable data to the HLS study.

The study, if feasible, naturally may evolve, as a byproduct, entire low-cost machine systems or at least new microcodes for known systems. "Large" systems can benefit from the "small" system construction effort. In both cases the study cannot be considered complete until prototypes are available for measurements. However, questions of feasibility will probably be answered long before this time. Measurements and projection for future worth must be based on entire systems, including system efficiency, programmer productivity, user installation management, and RAS.

6. Conclusions

The study has led to the belief that it is indeed feasible to move the machine/user interface from the original machine instruction level up to the level of general purpose programming languages.

Our views have been strongly influenced by reports from, and visits to, two pilot study projects. The Palo Alto Scientific Center demonstrated APL/25, an APL machine microprogrammed on the M25. The Advanced Programming Study Group at Hursley studied the construction of a PL/I incremental compiler using functional memory.

During initial measurements for array processing, APL/25 ran at half the speed of the corresponding FORTRAN-compiled code, not counting the very severe compiling overhead in the latter. The microprogram is being improved, and should lead to an overall speed one-third that of assembly code, and 40 to 50% of compile code for programs conveniently expressible in FORTRAN. In the "unimproved" state APL/25 is already comparable to the Yorktown APL/360 interpretive system, coded in S/360 assembly language, using the much faster M50 machine; the syntax analysis part turned out to be fully twenty-five times faster.

Though not to be equated to the HLS language, APL embodies many basic HLS requirements (names, descriptors, dynamic arrays, interpretive handling). The Palo Alto study indicates that even with a machine not designed for the task, the direct emulation of HLS can be very rewarding. The various reports on the programming efficiency using APL (completion time reduced by as much as 90%, comparing with assembly coding and often even with FORTRAN programming) also indirectly comments on the programmability of the more general HLS language.

The Hursley study showed that judicious use of functional memory can have a ten-fold advantage over a conventional machine in syntax scanning and associative logic, reinforcing our belief that the HLS interpretive handling will exploit the memory-like character of inexpensive LSI.

In any case, the HLS cost should be offset by the greater efficiency in treating arrays and structures, greater programmer productivity, and better protection of system and users even in complex environments.

We believe the proposal systematizes the hardware/software of the computer system to a greater degree than has been achieved. The overall consistency will make the system much more tractable and will greatly reduce IBM system programming. The new interface can further allow the convenient handling of current and future programming languages.

The ultimate result should be a computing system working directly with its users in a common language that is simpler, less costly, more efficient, more reliable and better adapted for continued growth.

7. Appendix A: A Critique of Current Instructions

The following is a critique of current instructions in the light of current and projected future needs.

a) Information without a priori meaning. The concept of a "floating point number", for instance, does not exist except as the operand of a floating-point arithmetical operation. The same quantity, if referred to as the target of a branch instruction, behaves like an instruction. The total freedom to treat the same quantity as entirely different things in different occasions was once a necessity when machines were otherwise inflexible and memory was small. The very same freedom has now been identified as a major source of programming bugs, and is conscientiously avoided by programmers.

A characteristic of most procedural languages if the attachment of meaning to data, yet machines still persist in attaching meaning to actions. So to speak the machine favors the use of adverbs (modifiers of verbs) yet the procedure languages (and human users) favor the use of adjectives (characterizers of data). This dichotomy runs deep, and is the main reason for the current difficulty in communication between man and machine.

The adjectival technique is more proper because information has meaning even when unused.

Data loses meaning whenever separated from the instructions using it. This certainly happens when data is transmitted; but worse, it even occurs from one instruction to the next. As it stands, therefore, data cannot be shared and be understood without special artifice, and analysis of machine and/or program errors is extremely difficult.

b) The machine maintains the fiction of an address despite repeated mappings. The compiler maps the user's symbolic names into addresses, then the relocation loader, the dynamic storage allocation mechanism, and (for large machines) the high speed memory buffer each map from one address to the other, treating the prior address as a name. The last mapping, by the memory buffer, is not even unique, yet is the most useful. IBM has unique know-how to exploit buffering techniques down to small machines, and thus true physical address assignments should best be left to real-time hardware.

c) The finite, linearly addressed memory. The total CPU memory is limited, and the addresses form a vector of sequential integers. Thus a unit of data is hemmed in by the left and right neighbors, and insertion (say to produce a longer vector) is virtually impossible. In practice users tend to claim "enough territory" to minimize insertions, at the cost of poor memory usage. The inflated claims, in turn, make multiprogramming unrewarding. Users also perform memory overlays, specifying several different uses of the same area. These require conscious programming effort and lead to debugging complications. "Paging" and "segmentation" are attempts to create a large "virtual memory" mappable onto the CPU memory.

d) Unnatural register assignments. The compiler management of registers apparently is not a perfected art, much less a science. In S/360, the distinction among XR/BR/GPR is not clean, and the separation of GPR from FLR, on hindsight, is too drastic. Because of operand lengths, several XRs often are needed for the same notion (say 7: 7 bytes, 7 half-words, 7 words, 7 double words are four distinct things, each calling for an index register or equivalent action).

For GPR and FLR, Model 91 and 195 experience shows that pipelined machines would prefer not having to specify intermediate registers (which adds more bottlenecks to the processing and increases interruption restore burden), but rather to execute the procedural language statement directly. In smaller machines "registers" are just standard memory, addressed differently; the specification of registers does not imply extra efficiency. Actually, registers are just one more form of storage whose management is most meaningful when done by hardware during computing real-time. There is no need to limit the number of registers; seldom-used ones can be placed in slower memory.

e) Loss of causality information. Traditional instructions form a distinct unit of machine processing. In compiling, the causal chain explicit in procedure language statements is broken up, with each piece capable of becoming a branch target. The follower of machine code may, with difficulty and some luck, divine the execution sequence starting from a certain instruction (say A); but he cannot decide, short of reading the entire program from top to bottom, the predecessor of A, even in the absence of machine error.

Every instruction is a potential branch point or interruption point, until proven otherwise. Large machines often have to expend hardware for such eventualities. The M91 (M195 too) takes great pains to reconstruct causality, literally undoing the compiling process and reverting to the procedural code.

f) Instructions do not enhance reliability or security. The adverbial orientation detaches meaning from data, and is inadequate to guard against error or security breach. Errors may compound themselves into unmanageable and intractable situations; a single branch error may initiate a sequence of destructive random walk.

g) Most instruction sets do not have array operations, yet an array is a well-defined unit of information to the user, and capable of being so in a machine. Pipelining in large machines needs array operations, and all machines can take advantage of the reduced decoding overhead.

S/360 has some "vector-like" instructions, such as the VFL class, decimal arithmetic, and load- and store-multiple. In these cases the limited length of the vector has to be explicitly stated (not even indexable), a very confining requirement indeed.

h) Lack of device independence. Input/output instructions employ different addressing techniques for different devices, with little or no interchangeability. Programs written and debugged for a given machine configuration are usually not transferable to another environment, due to small differences in configuration. This situation is somewhat alleviated by software, but I/O rigidity is partially responsible for the complexity of the job control language.

It has been recognized that the classification of access techniques by device should be replaced by a classification by purpose, and ultimate I/O should be separately considered from memory extension. The latter should be addressed like memory, because a major reason for device independence is to allow programs to be insensitive to available CPU memory size.

i) One-at-a-time condition philosophy. In most instruction sets (S/360 included) there is only one representation of condition. Any condition-setting operation (an add in S/360) will replace the previous condition by the new one. Condition handling being a major worry in large machines, it would be extremely desirable to be able to accumulate a sequence of condition occurrences, and then allow the program to be controlled by the total effect of this pattern. (This feature is already present in FORTRAN.)

j) Lack of features. The absence of array instructions has been noted. There are other important operations, useful because of procedural language need. Many are rather easy to achieve by new hardware, but are absent in current sets. Their emulation in terms of current sets tend to be clumsy. These include multiple sum, associative search for match, and even exponential and logarithm.

In sum, an instruction is too small a unit of procedure even for present machines. The complete freedom to perform modifications, while perfectly useful (and even necessary) in bygone years, are inadequate in dealing with the complexities we have today. There is a genuine fear that instruction-based programs beyond a certain size may become undebuggable.

The procedure languages form a more adequate basis for coding complex applications problems. However, execution of procedural programs still require mapping into machine instructions. The mismatch between the adverb-oriented machine world on one hand, and the adjective-oriented human and procedural language world on the other, means inefficiency and misunderstanding will continue to exist, unless the adjectival world becomes the basis of machine architecture. It seems very likely that then, and only then, can we be prepared to tackle the next order of complexity, the large data-base system.

8. Appendix B: The Machine Organization Concepts Study Group

In response to an inquiry from Mr. B. O. Evans, the Machine Organization Concepts Study Group was convened starting November 25, 1969, meeting roughly on a bi-weekly basis. The latest meeting occurred February 25, 1970.

The composition of the group was as follows:

John C. McPherson (CHQ, Armonk), Chairman

Tien Chi Chen (RD, San Jose)
Carl J. Conti (SDD, Poughkeepsie)
Claud M. Davis (SDD, Poughkeepsie)
Albin D. Kolwicz (SDD, Boulder)
John C. Laffan (SDD, Poughkeepsie)
Albert A. Magdall (SDD, Endicott)
Anthony Peacock (SDD, Poughkeepsie)
Anthony Proudman (SDD, Hursley-Poughkeepsie)
Nathaniel Rochester (FSD, Boston)
David Sayre (RD, Yorktown)
Ralph F. Schauer (CD, Poughkeepsie)
William S. Worley, Jr., (SDD, Time-Life)

8.1 What Has Been Accomplished

a) The group has identified the need for HLS as a new architecture basis, and has become convinced of its basic feasibility and potential advantage to IBM.

b) We have agreed that a coherent HLS language and architecture can be developed. We have listed highlights of this language, and supplied much detail. Some important choices are left open, as deeper studies with simulation verification and exploratory development are clearly required.

c) The group recommends exploratory effort towards implementation, knowing that the full exploitation of this type of machine, which is the largest departure thus far from von Neumann principles, will not take place automatically. Research and development on a broad scale will ultimately be needed. Thought should be given as to how other IBM divisions can assist in this process.

d) A presentation to Mr. B. O. Evans has been made on February 26, 1970.

e) With the issuance of the present document, the formal activities of the group are concluded.

8.2 Resolution (Unanimously endorsed February 25, 1970)

"The Machine Organization Concepts Study Group has studied the question of feasibility and advisability of a higher level system and concludes that such a change of direction is both feasible and necessary and very advantageous to the Company's expension, both to new fields of application and to larger numbers of users. It offers a way for consolidating the advances in the knowledge in use of machines in the past 25 years and forms a firm base for future development and will use to advantage new technologies."

9. A Selected Bibliography

The following were selected for their intrinsic pertinence to the problem of procedural-language oriented machines.

The bibliography is divided into three parts.

a) Working papers generated within the group.
b) Documents by IBM authors.
c) External publications.

9.1 Working Papers Generated Within The Group

Some of the working papers generated within the group are cited below.

J. Philip Benkard and Carl J. Conti, "Measurement and Analysis", Memo to Mr. J. C. McPherson, January 15, 1970.

Tien Chi Chen, "On Higher Instruction Sets", Memo to Mr. J. C. McPherson, December 11, 1969.

Tien Chi Chen, "Higher Instruction Set Machines", Flip chart reproduction, January 1970.

Tien Chi Chen, "On Decimal Arithmetic", Memo to Mr. J. C. McPherson, January 22, 1970.

Carl J. Conti, "Large Advanced System", Flip chart reproduction, December 1969.

Carl J. Conti, "Subgroup report and implementation thoughts", Flip chart reproduction, December 12, 1969.

Carl J. Conti, "Qualitative advantages of a machine with a higher level instruction set", Two memos dated January 7 and January 12, 1970.

Albin D. Kolwicz, "System/360 Input/Output Architecture", Memo to T. C. Chen, March 17, 1970.

John C. Laffan, "System Q", Memo to Mr. D. H. Furth and Mr. P. Kolko, January 13, 1970.

John C. Laffan, "System Organization", Memo dated January 23, 1970.

Anthony Peacock, "An Addressing Proposal", Flip chart reproduction, 1970.

Nathaniel Rochester, "Command language for high level language machine", Manuscript dated January 7, 1970.

David Sayre, "Tolerant system", Flip chart reproduction, February 10, 1970.

William S. Worley, Jr., "Preliminary requirements of FORTRAN, COBOL, PL/I in high-level machines", Flip chart reproductions, December 19, 1969.

9.2 Documents by IBM Authors

Most documents in this category are IBM Confidential. The exceptions are mainly those published in the open literature.

The proceedings of two IBM symposia held last year are noteworthy because of their timeliness and technical content.

> The Seventies: A Challenge to Programming (IBM 1969 Programming Symposium Proceedings (2 volumes issued just before the symposium in Boston, October, 1969. (IBM Confidential) (To be abbreviated as TSCP below).
>
> Proceedings of the IBM Symposium on New Directions in Computer Technology (held in Yorktown, April 1969). IBM Poughkeepsie Technical Report TR00.1895, July 1969. Two volumes (IBM Confidential) (To be abbreviated as NDCT below).

John W. Backus (RD, San Jose), "Comments on Programming Development Supporting Strategic Plan", Manuscript, October 18, 1968.

Klaus Berkling (RD, Yorktown), "A Machine Architecture Based on Tree Structures", TSCP, pp. 237-255.

Klaus Berkling, "The Accommodation of APL and ALGOL on the Lambda Calculus Machine", Manuscript 1970.

Willard G. Bouricius (RD, Yorktown) and Keith A. Duke (SDD, Poughkeepsie), "Proposed Functional Descriptions of an Interpretive Data-Driven Array Processor", NDCT, pp. 125-130. (Proposal to implement APL-like language on 64-bit word machine).

Lawrence M. Breed (now non-IBM) and Richard H. Lathwell (DPD, Yorktown), "The Implementation of APL/360", Interactive Systems for Experimental Applied Mathematics, Academic Press (New York 1968), pp. 390-399.

Peter F. Carpenter (IBM Hursley), "Functional Memory Programming Studies - PL/I Incremental Compiler/Interpreter", Memo dated January 16, 1970 (IBM Confidential), addendum dated February 11, 1970. (Hursley's study of PL/I machine).

Tien Chi Chen (RD, San Jose), "A Statement-Oriented Multiprogrammed Automatic Computer (SOMAC)", (Memo to Dr. J. H. Eaton), April 1968, (Proposal for superscale 10-instruction stream machine using APL-like language).

Tien Chi Chen, "Unconventional Superspeed Computer Organizations", NDCT, pp. 149-160. (Reasons for embodiment of statement orientation, array processing and name orientation).

Tien Chi Chen, "Highspeed LSI Arithmetic", Memo to Dr. I. T. Ho, December 10, 1969, (highspeed evaluation of exp(x), log(x)).

Edgar F. Codd (RD, San Jose), "Derivability, Redundancy and Consistency of Relations Stored in Large Data Banks", TSCP, pp. 207-218; "A Relational Model of Data for Large Shared Data Banks", manuscript, 1970; "Notes on a Data Sublanguage", Flip chart reproduction, 1970. (A systematic attack on data base problems).

Rex L. Comerford and R. C. Huang (SDD, Endicott), "Status of RP2 Architecture", Architecture File Memo #1, Dept. 663, dated February 3, 1970. (A recent memo on the RP2 attribute-examining machine organization).

Steven W. Dunwell (FED, Kingston), "Computer Systems in 1975: The Shape of the Future", Memo 1969. (Teleprocessing, multiprocessing, microprogramming, memory paging under one primary procedure language).

Robert W. Engles, "A Tutorial on Data-Base Organization", IBM Poughkeepsie Technical Report TR00.2004, March 20, 1970, 108 pages.

Adin D. Falkoff and Kenneth E. Iverson (DPD, Yorktown), "the APL/360 terminal system", Interactive Systems for Experimental Applied Mathematics, Academic Press (New York 1968), pp. 22-37.

M. Flinders (IBM Hursley), "Functional Memory as a System Technology", NDCT, pp. 49-60.

Irving T. Ho (CD, Fishkill) and Tien Chi Chen, "Multiple Addition with Modular Threshold Operator Approach", Array Logic Technical Note #34, January 30, 1970.

Charles L. Gold (SRI, New York), "What to Expect of Future Programming Languages", Memo to Mr. J. C. McPherson, January 16, 1970.

Anthony Hassitt, John W. Lageschulte and Harry F. Smith (DPD, Palo Alto), "A Procedure-Oriented Machine Language, Part I", Report 320-3271, January 1970, 38 pages. (APL microcoded on M25).

Anthony Hassitt and Richard H. Lathwell (DPD, Yorktown), "APL System", file memo March 18, 1970. (APL microcoded on M25 vs. alternatives; implications).

Robert A. Henle (CD, Fishkill) and W. Lee Shevel (CD, Poughkeepsie), Monolithic Semiconductors - Basis for New Direction", NDCT, pp. 1-22.

Robert A. Henle, Irving T. Ho, Gerold A. Maley and Ronald Waxman (CD, Fishkill), "Array Logic", NDCT, pp. 413-430.

Hirondo Kuki (U. of Chicago, IBM consultant), "Advocating Arithmetic Orthodoxy in SOMAC Design", Manuscript, January 27, 1970.

E. E. McDonnell (DPD, Yorktown), "A Formal Description of JCL", TSCP, pp. 219-236. (The job control language codified in APL).

Alan J. Melbourne (WTC Hursley) and John M. Pugmire (IBM France), "A Small Computer for the Direct Processing of FORTRAN Statements", Computer J.8 24-27 (April 1965) (A pioneering proposal).

John E. Meggitt (IBM, now self-employed), "A Character Computer for High Level Language Interpretation", IBM Syst. Journal 3, 68-78 (1964).

A. P. Mullery (RD, Yorktown), R. F. Schauer (CD, Poughkeepsie), and R. Rice (now at Fairchild), "ADAM - A Problem Oriented Symbol Processor", Proceedings SJCC 23 367-380 (1963), (A pioneering effort in both language and hardware).

Alvin P. Mullery (RD, Yorktown), "A Procedure-Oriented Machine Language", IEEE Trans. Elec. Comp. EC-13, 449-455 (August 1964).

John Nichols (RD, Yorktown), "Objectives of CLF", Manuscript 1970. (Central language facility).

Leo O'Leary (SDD, Endicott), "FS System Proposal", Flip chart reproduction, 1969.

H. J. Ollmert (SDD, Boeblingen, Germany), "An RPG Machine", TSCP, pp. 162-178. (Microprogrammed from M20 Submodel 5).

Anthony Peacock (SDD, Poughkeepsie) and Anthony Proudman (SDD, Hursley-Poughkeepsie), "Project Spade Progress Report #1, January 31, 1969. "WP #21: Interpretive Computing Systems and Large Scale Integration. WP #23: The APL Logical Machine. WP #31: The APL Model. (LSI is a good match with interpretive machines; an APL machine can be simulated in APL).

Richard H. Lathwell (DPD, Yorktown), "Relative Frequency of APL/360 Primitive Execution", Memo to Mr. J. C. McPherson dated January 19, 1970. (Usage measured in three consecutive days shows "concatenation" to be the second most used dyadic operator, next to addition).

David Sayre (RD, Yorktown), "On Programming Strategy for the 70's", TSCP, pp. 1-6.

Jean E. Sammet (FSD, Boston), "Programming Languages, History and Fundamentals", Prentice-Hall Inc. (Englewood Cliffs, N. J., 1969), 785 pages. (A major reference volume).

K. W. Van Mechelen (FSD, Gaithersburg), "High Level Language Machine", file memo, October 4, 1968, 38 pages. (Attempt to handle PL/I subset with his own IBM 1620).

Helmut Weber (SDD, Endicott), "A Microprogrammed Implementation of EULER on IBM System/360 Model 30", Comm. ACM 10, 549-558 (1967). Microcoded Model 30 for Algol-like language, generalized, for precedence parsing methods).

William S. Worley, Jr., (SDD, Time-Life), "The Universal Language", Flip chart reproduction 1970. (Language funnelling).

9.3 External Publications

Philip S. Abrams (Stanford U.), "An APL Machine", Ph.D. Thesis, Stanford University 1970, 204 pages. Available as SLAC report No. 114, February 1970, Stanford Accelerator Center, Stanford University, Stanford, Calif.

James P. Anderson (Burroughs; now independent consultant), "A Computer for Direct Execution of Algorithmic Languages", Proc. EJCC 20 184-193 (1961). (A proposal for an ALGOL-60 machine).

Theodore R. Bashkow, Azra Sasson and Arnold Kronfeld (E.E. Dept., Columbia University), "System Design of a FORTRAN Machine", IEEE Trans. Electronic Computers EC-16, 485-499, (August 1967). (A proposal aimed at 1620-level FORTRAN without subroutines).

Edsger W. Dijkstra (Holland), "The structure of the "THE" - Multiprogramming System", Comm. ACM 11, 341-346 (May 1968). (Hierarchical operating system with asynchronous control and proof of well-behavior).

J. A. Gosden (MITRE Corp.), "Software Compatibility: What was Promised, What We have Done, and What We Need", Proceedings FJCC 1968 pp. 81-88. (We need both inter- and intra-software family compatibility).

E. A. Hauck and B. A. Dent (Burroughs), "Burroughs' B6500/B7500 Stack Mechanism", Proceedings SJCC 1968 pp. 245-251. (Extension of B5500).

J. K. Iliffe (ICL, England), "Basic Machine Principles", American Elsevier Publishing Co., (N. Y. 1968), 86 pages. (A descriptor-oriented system outgrown from the Rice University computer design).

J. K. Iliffe (ICL, England), "Elements of BLM", Computer J. 12, 251-258 (September 1969.) (ICL experimental implementation of Iliffe's machine).

Edgar T. Irons (IDA, Princeton), "Experience With an Extensible Language", Comm. ACM 13, 31-40 (January 1970).

William V. Kahan (U.C. Berkeley), "Default Rules for Rounding Fixed Precision Normalized Floating-Point Arithmetic Ignoring Over/Underflow", a one-sheet comment distributed September 25, 1969 at Lake Arrowhead IEEE Workshop.

T. Kilburn, D. Morris, J. S. Rohl and F. H. Sumner, (U. of Manchester, England), "A System Design Proposal", IFIPS 1968 Proceedings D76-D80. (Names rather than addresses).

William M. McKeeman (U.C., Santa Cruz), "Language-directed Computer Design", Proceedings FJCC 1967 pp. 413-417. ("Language that reflects what we want to do and how we do it (for instance, in parallel) and machine structures effective in handling that language".)

Howard L. Morgan (Cornell U.), "Spelling Correction in Systems Programs", Comm. ACM 13, 90-94 (February 1970). (Computer systems efficiently correct spelling errors.)

Saul Rosen, "Hardware Design Reflecting Software Requirements", Proc. FJCC 1968, pp. 1443-1450.

Masakatsu Sugimoto (U. Tokyo, Japan), "PL/I Reducer and Direct Processor", Proc. 24th Nat. Conf. ACM, 1969, pp. 519-538.

Kenneth J. Thurber and John W. Myrna (U. Montana), "System Design of a Cellular APL Computer", IEEE Trans. Computers C-19, 291-303 (April 1970).

S. C. Wang (Hewlett-Packard), "On the Direct Implementation of Algorithmic Scientific Computer Languages", Ph.D. Thesis, U. of Minnesota, June 1968. (COFAC uses SAL, a procedural machine language with features of FORTRAN, ALGOL and PL/I).

Nicklaus Wirth (U. of Zurich, Switzerland), "On Multiprogramming, Machine Coding, and Computer Organization", Comm. ACM 12, 489-498, (September 1969).

E. C. Yowell (NCR), "A Mechanized Approach to Automatic Coding", Automatic Coding, J. Franklin Inst. Monograph No. 3 (April 1957) pp. 103-111. (NCR 304 machine uses autocode, has merge, edit, and summarize instructions).

(End of Report)