HLS (HIGHER LEVEL SYSTEM)

INTERIM REPORT

FEB. 26, 1970

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

IBM, CHQ, Armonk, New York

HLS -- HIGHER LEVEL SYSTEM

Interim Report

of the

MACHINE ORGANIZATION CONCEPTS STUDY GROUP

Table of Contents

0. Summary of Report

The Study Group recommends that a Higher Level System of the character described in this report be developed to sustain IBM growth in the years ahead and extend the use of computers to more people. We have reached the limit of the Von Neuman single-operator machine as the form of computer to meet the expanding needs of electronic data processing and must have a more powerful and simpler approach.

The HLS moves the man-machine interface upward from the present machine-oriented instruction language level to the level of today's PL/I and COBOL at which people communicate more easily.

The language uses statements whose operators process variables identified by name rather than address. Variables are not limited to single values. Instead, arrays and structures can be processed as single units.

The HLS handles the execution of program statements interpretively with the help of descriptors stored with the data which dynamically define the data object--its type, size, precision, and its current location.

The HLS is a good design for the wide-scale application of LSI to both storage and logic functions. Functional memory can be of great value in a number of applications: associative tables, complex scanning, and control logic. There are indications of a tenfold increase in performance in these areas from its use.

The system design fits in with the use of a storage hierarchy and gives full advantage of the extra storage space to the user without change of concepts or limitation of generality. A variable at any level is still controlled and located by its descriptor and the convenience of a single level store is maintained for a multi-level hierarchy.

The user and IBM should benefit substantially from the condensed statement of programs and the easier coding and debugging of programs. We expect that the size and cost of programs will be reduced to a small fraction of their present figures. This improvement should be of special significance to IBM in the System Control Program area and in the 'programming product' area

The cost of Field Engineering will be markedly reduced in maintaining both software and hardware. The simpler programs will be easier to maintain. Self describing data will allow errors to be detected earlier and a simple error will less often grow into massive confusion. Regardless of the complexity of a crash, the information lying around afterward will be more easily and quickly identified.

System control will be accomplished more efficiently by a command language which is simply a facet of the high level language of the system. HLS will thus extend to the full range of problems and to background as well as foreground control, the kind of unified control that was successfully pioneered in the JOSS system at RAND Corporation.

Transition from the current product line will be aided by three measures. HLS will be able to emulate System/360. It will hopefully be able to execute jobs partly by direct execution and partly by emulation, shifting back and forth dynamically. PL/I, FORTRAN, COBOL, APL, and RPG will be treated almost like high level machine language and require such minimal processing as to not deserve to be called compilation. HLS will accept and process System/360 data sets.

The new system will form a sound base for growth and we can foresee that the higher level interface with the user will lead to new methods of programming. In particular, there will be greater use of macro-instructions (i.e., building blocks of program) selected or modified to meet the needs of particular fields, industries or kinds of application. Users will more easily be able to create a variety of specialized languages extending the basic system language.

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."

1. Converging Evidence for Need of New Machine Architecture in the 1970s

There is now increasing concern that we are nearing the limit of exploitation of the instruction concept in computer system design. The evidence derives from several sources:

a) In the marketplace, if LSI is used in the obvious way to reduce CPU logic costs, it cannot reduce user costs by more than a few percent. The only reasonable way to maintain the profitable growth of the Corporation and to leapfrog the competition, is to adapt the new flexibility of future hardware to human users' environment. By serving the user better and reducing the user installation overhead, a larger fraction of the users costs will be channeled into IBM revenue.

b) Human-machine mismatch. The majority of human users will use procedural languages or use application packages based on the procedural languages. The current machine system either compiles procedural code into instructions, which consumes a costly extra pass and produces imperfect emulation (hex dump), or interprets the procedural code inefficiently with loss of memory capacity. The unmistakable trend is towards interpretation with or without the shortcomings.

c) New software systems. Large software systems will be virtually undebuggable and unmanageable unless a level of systematicity is enforced from the start. This can be supplied by the interpretive handling of resources.

d) Communicability. The future systems will interface with a significantly larger portion of the human community and will handle complex communications among man, program, software, and hardware. These communications must be meaning-preserving in nature, and featureless bits without self-description would be woefully inadequate.

e) Language funnelling. There will continue to be new languages. There is a real need to provide a high-level concise language which adequately interprets the richness of user languages and still be machine-efficient in procedure execution. This language should be a programming language in its own right, but is not the present machine instruction set.

f) Hardware evidence. Very large machines favor the cause-and-effect delineation of procedural languages. Array handling allowing the efficient deployment of pipelined resources will be very efficient. The implied system management of all storage resources in higher languages is a welcome extension of the "common data bus", "storage buffering", look ahead-look aside" features. Very small machines have enough micro-code flexibility to interpret procedure statements directly, without the intermediary of an instruction set.

Hardware will be inexpensive and memory-like. This promotes "variable-field length" data handling in general. Associative techniques (such as FM) will be a powerful assist in adapting machines to procedural computation.

g) New hardware/software complexes. The data-base systems badly need codifiable meaning-preserving interpretive actions. Large shared systems, to average out peak loads of many medium users, will become rewarding if the system has meaningful RAS, security, and automatic storage hierarchy.

All these point to the need for a directly executable procedural language using self-describing information as a tool for generalized interpretation. The language-oriented machine system shall be tentatively called HLS (Higher Level System). The key characteristics are:

1. statement orientation
2. meaning preservation
3. interpretive prowess
4. storage independence
5. user/program/hardware/software efficiency

On current architecture items, (1-3) are obtainable at high cost, to the detriment of the last two requirements. We are of the opinion that a major departure in architecture is necessary to achieve global efficiency. With it computing will be put on a completely new basis.

2. Instructions are Inadequate for Future Needs

Instructions were devised 25 years ago. The original computing environment can be contrasted with the present.

a) Memory size was small, circuitry was slow, both were expensive

(now both are fast, inexpensive, small in size, numerous)

b) Activities outside the CPU were infrequent

(now CPU usage is low mainly because of "I/O")

c) Computing was mainly numerical

(now numerical work is only a small part of computing; even numerical problems have a large data processing load)

d) Machine time was the most precious item

(now cost of human time more important)

e) Users were willing, able to conform to machine rigidity

(now few users are willing or able to perform this contortion)

f) Programming was mainly done in absolute binary (octal)

(now use of procedure languages is widespread)

By far the greatest unanticipated change was the spectacular growth of programming systems serving as a cushion between man and machine. This cushion is sufficiently human-attuned that users will tend to insist on procedure language programming, and OS control of the system. It must be said that this cushion rests on the weakest features of the machine. There is hardly any use of "floating multiply" in OS, for instance.

The user will continue with procedure languages and expect the machine to be a procedure interpreter or a virtual machine. He is frustrated whenever the system fails to behave as a proper emulator of his virtual machine. The poor emulation is practically axiomatic with instructions, as context is deformed in the mapping into machine language. As a result, compiling cannot promise source language debugging, even with a large processing overhead (500 executions to provide one compiled instruction). Interpreters try to maintain the original context but are slow. The expected future growth towards interpretation means that machines should (a) try to handle procedure code more directly, and, (b) generalized interpretive mechanisms should be built in.

It turns out that the facilities in OS are format driven, hence are also interpretive in nature. This means interpretie penalties often pile up exponentially, and further tie up the CPU, the latter having to do most of the interpretations alone.

2.1 A critique of instructions

a) Information in the machine does not have 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 had been, at one time, a necessity when explicit address modification was the only means to program a loop. The same freedom has now been identified as a major source of programming bugs, and is conscientiously avoided by programmers. Therefore, the modern machine should prevent unintentional misuse of information, by explicitly attaching meaning to information.

A characteristic of most procedural languages is indeed this attachment of meaning to data, yet the machine still persists in attaching meaning to actions. So to speak, the machine favors the use of adverbs (modifiers on 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 lack of communication between man and machine.

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 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. Thus true physical address assignments can best be left to real-time hardware.

c) Linearly addressed memory, of limited size
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 the users tend to claim "enough territory" so that insertions are rare, at the cost of poor memory usage. The inflated claims, in turn, make multiprogramming unrewarding.

d) Unnatural register assignments
The compiler management of registers apparently is not a perfected art. 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 seven: seven bytes, seven half-words, seven words, seven double words are four distinct things each calling for an index register for equivalent action.)

For GPR and FLR, the Model 91 (now 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), bur rather to execute the procedural language statement directly.

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, as seldom-used ones can be placed in large memory.

e) Loss of causality information. The instructions have "context freedom" in that each forms a distinct unit of machine processing. The causal chain, as contained in procedure language statements, is broken up, with each piece now capable of being a branch target. The follower of a piece 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, all predecessors of A.

Also as a consequence of the "context freedom", every instruction is a potential branch point or interruption point, until proven otherwise. Large machines often have to expend hardware for such wild goose hunts. The M91 (M 195 too) also takes the pains to reconstruct causality, literally undoing the compiling process and reverting to the procedural code.

f) 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, too. Pipelining in large machines needs array operations, and small machines can take advantage of the reduced decoding overhead.

The S/360 instruction set does have a few "vector-like" instructions, such as the VFL class, decimal arithmetic, and load and store multiplies. In these cases the length of the vector has to be explicitly stated (not even indexed), a very confining requirement indeed.

g) Inadequacy of instructions. The absence of array instructions has been noted. There are other important operations, rather easy to achieve by new hardware, but absent in current sets, their emulation in terms of current sets tending to be clumsy: these include multiple sum, associative search for match, and even exponential and logarithm.

In sum, instructions and the complete freedom to perform detailed modifications, while perfectly adequate in bygone years, are beginning to be 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, and soon will also be adopted by systems programming. However, the mismatch between the adverb-oriented machine world on the one hand, and the adjective-oriented human/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, such as the large data-base system.

3. Architecture Highlights of HLS

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

Raise the system to user's 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:

a) Statement orientation (not instructions);

b) Referencing by name (not addresses);

Self-description of information (not featureless bits);

Dynamic attribute examination (not op code proliferation and over specialization);

c) Processing of arrays and structured data (not looping element by element);

d) Processing of descriptors (not code modification);

e) Automatic storage hierarchy (not explicit addressing, pre-planned overlays, hardware/software I/O);

f) Semantic (not necessarily syntactic) features of major procedural languages: FORTRAN, PL/I, COBOL, RPG, APL (not incoherent complexity); system to interpret procedure language faithfully (not hex dumps, hex patches, delphic error messages);

g) Structured control (not disunity);

h) Decimal arithmetic emphasis (not conversion errors, uncertainties).

3.1 Statement Orientation

The unit of machine procedure will be a multi-operator statement; e.g. A = B + C + D * E. The statement is concise and comprehensible, has well-defined causality meaning. As a machine procedure it avoids the designation of registers for redundant intermediate storage, and gives the system more freedom to optimize. The occurrence of stores, conditional branch, interruption can all be localized. Interruptions, when performed before storing into A, will be fully recoverable.

Because of self-described data (See Section 3.2), the operators in a statement will be free of "action-modifiers", and will be fewer and more general.

3.2 Self-describing Information

Information will be referred to by names, or machine constructed alter-names. The properties of the information are summarized in descriptors which either contain, or point to, the information.

The descriptor typically contains descriptions of:

type (e.g., floating hex)

structure (e.g., 3 x 15 matrix)

constraints (e.g., read only), etc.

The size of descriptors varies, with a large upper limit. The encoding is such that the most common types of information have short descriptors. There is an "escape hatch" encoding of the short descriptors, which "points to" the longer descriptors. The design is expected roughly to correspond to Huffman encoding.

Operation details are based on the descriptors of the operands. Typically 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. Enforcement of security, as well as error checking, can be done together with the attribute examination.

The descriptors can be manipulated by qualified users for an extra degree of handling.

The self-described information includes programs, procedures, subprograms, even hardware features and possibly branch targets. The immediate consequence is data independence. Information transmittal will include the movement of descriptors, and will be a meaning-preserving operation. The implications on asynchronous processing are profound, and the adjectival attachment of descriptors to data may be the only rational basis for a data-base system.

Descriptors also serve a self-documentation purpose, without the conscious effort by the users.

During computation the following may vary:

the number of data entries

space requirement of each entry

the structure of the information

the number of descriptors

the contents of descriptors

Due to the presence of descriptors, a variety of formats can coexist, with unlimited extensibility. Therefore descriptors not only allow meaning to be preserved during communications, but accommodates diversity. A natural next step is to combine diversely created programming material in a meaningul way for a unified purpose. This would mean the reconciling of formats, and subprogram conventions, and may invoke massive software support. But HLS has the framework needed to achieve this combinability.

Non-numbers can be accommodated using descriptors, without expending storage otherwise. Important ones are "null" (important for vector concatenation), "undefined" (important for debugging).

3.3. The Processing of Arrays and Structured Data

Just 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 system should permit operations involving entire arrays, or at least entire vectors in arrays. The standard array element may be a bit, a character, or a number. Nonstandard elements could be arrays, pointers, "null" or "undefined".

A minimum of array operation may consist of the following:

a) Extend the use of standard arithmetic operators (defined for scalars) to standard vectors. If A = $(a_1, a_2, \ldots)$, B = $(b_1, b_2, \ldots)$, c is a scalar, and P, Q are respectively monadic and dyadic operators then PA = $(Pa_1, Pa_2, \ldots)$, BQA = $(b_1 Q a_1, b_2 Q a_2, \ldots)$. cQA = $(cQa_1\ cQa_2, \ldots)$ AQc = $(a_1 Q c, a_2 Q c, \ldots)$.

b) Permit concatenation, contraction and expansion of vectors. Incidentally concatenation is among the top five most frequently invoked operators in APL, yet it cannot even be stated in instruction form, because of the varying memory demands.

c) Conversion between vectors and multidimensional arrays

d) Extract properties of a named array

e) Extract subsets (notably vectors) of an array

f) Generation of named arrays by arithmetic

Users should be free of the burden of detailed space allotment or dimensional information of arrays. The operation will be descriptor-driven, and all else should be automatic, including the alteration of the result descriptors.

Due to the presence of descriptors, a variety of formats can coexist, with unlimited extensibility. Therefore descriptors not only allow meaning to be preserved during communications, but accommodates diversity. A natural next step is to combine diversely created programming material in a meaningul way for a unified purpose. This would mean the reconciling of formats, and subprogram conventions, and may invoke massive software support. But HLS has the framework needed to achieve this combinability.

Non-numbers can be accommodated using descriptors, without expending storage otherwise. Important ones are "null" (important for vector concatenation), "undefined" (important for debugging).

3.3. The Processing of Arrays and Structured Data

Just 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 system should permit operations involving entire arrays, or at least entire vectors in arrays. The standard array element may be a bit, a character, or a number. Nonstandard elements could be arrays, pointers, "null" or "undefined".

A minimum of array operation may consist of the following:

a) Extend the use of standard arithmetic operators (defined for scalars) to standard vectors. If $A = (a_1, a_2, \ldots)$, $B = (b_1, b_2, \ldots)$, c is a scalar, and P, Q are respectively monadic and dyadic operators then $PA = (Pa_1, Pa_2, \ldots)$, $BQA = (b_1 Q a_1, b_2 Q a_2, \ldots)$. $cQA = (cQa_1\ cQa_2, \ldots)$ $AQc = (a_1 Q c, a_2 Q c, \ldots)$.

b) Permit concatenation, contraction and expansion of vectors. Incidentally concatenation is among the top five most frequently invoked operators in APL, yet it cannot even be stated in instruction form, because of the varying memory demands.

c) Conversion between vectors and multidimensional arrays

d) Extract properties of a named array

e) Extract subsets (notably vectors) of an array

f) Generation of named arrays by arithmetic

Users should be free of the burden of detailed space allotment or dimensional information of arrays. The operation will be descriptor-driven, and all else should be automatic, including the alteration of the result descriptors.

A structure (as in PL/I) is an array of possibly dissimilar elements, each of which could also be a structure. Thus a payroll is a structure (character strings mixed with numeric data) and a tree is a structure. Typical operations would be, to construct a structure from a table or a pointer list, and vice versa; to obtain substructures from a structure; and to graft a structure onto another.

3.4 Descriptor Processing

There are many kinds of descriptor processing:

a) During computation, the descriptor of the result operand is changed to reflect the new structure requirement.

b) Users have the right to inquire about the descriptor contents (e.g., what is the size of 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. Mostly it is done for the purpose of indirection.

   Several descriptors may point to the same object, in principle, to achieve synonymy.

e) A descriptor can be displaced from one place to another, say to become an element of an aggregate.

f) Typically, we "talk about the weather without doing anything to it". Similarly in data-base systems we tend to read and process descriptors for a long while before accessing the data object. Large systems in the past have often created internal descriptors to enhance the processing. In Dr. Philip Abrahams' "APL Machine" a "dragalong" principle is used for performance optimization. This principle postpones the actual data processing and manipulates descriptors as far as possible, often reducing huge computations to trivia.

3.5 Automatic Storage Hierarchy

The concept of storage should be clearly separated from that of ultimate (or source-sink) I/O. The latter is strongly format oriented, and the former should be more concerned with efficiency, RAS, security and capacity.

To preserve the meaning of information, the name/descriptor approach should extend to the entire storage hierarchy, including the deep recesses of archives. Incidentally, in the large machines, the fast cache, registers, lookahead/lookaside, branch anticipation, and internal forwarding mechanisms are

storage resources also, though the descriptor encoding and detailed sequencing may be different. A systematic point of view for the handling of all storage resources can now be applied, based on the Huffman principle:

best service for most frequently encountered requirements

lower quality service for rare occurrences

almost open-ended spectrum of service

Format, location and access methods should not be the main concern, but are automatic system functions. So far as the user is concerned, the system always uses his "external format".

Different members of the storage hierarchy probably will be serviced by powerful yet noncostly controllers. Because of self-description, data movement becomes a meaning-preserving activity. It thus becomes entirely possible to perform simple processing far from the CPU, avoiding the staging penalty.

Another technique is to process the descriptors, indicating the nature of minor changes to be made on he object data.

Still another technique is data compression. If the meaning of the data object is known, excellent reversible compression techniques can be applied. An example would be the recoding of triplets of decimal digits (12 bits) into 10 bits.

3.6 Semantic Features of Major Procedural Languages

The HLS language will contain essences of FORTRAN, COBOL, PL/I, and APL in a "semantic cross-section".

These languages were chosen partly because of their current importance in the computing community, and also partly because of their inherent diversity. It is not a "semantic union" which may be large and unmanageable, but a crisp language into which the four can be mapped with little information loss, and (with the help of tables) with full recoverability.

The HLS language will be a good programming language in its own right, and will be a powerful base to construct new languages. It should be easily decodable and efficiently executable by machine.

It would be nice if one of the current languages would fill the need. Unfortunately we have found none. This is partly due to the other HLS characteristics we demand, but most importantly because control specification has not been included in procedural languages in any significant way.

Nevertheless, we are of the opinion that the correct language can be designed, and this is a subject of the highest urgency. Although we aim mainly at the above languages (and control), other languages naturally fall out. Significant examples are: LISP and ALGOL.

HLS as a machine system is to be a faithful procedure interpreter of the above languages. The user who submits programs in those languages can expect debug messages, dump and error fixups in source language terms. This, incidentally, means that the source language formats will be honored which is possible only because of data descriptor flexibility.

A simple consequence of the multilingual nature of HLS is that programs in different languages can be linked and executed together when format inconsistencies are resolved. This mechanism will be provided, though not necessarily completely in hardware.

A pertinent question is, beyond the interpretation of certain languages, what are the real capabilities of HLS? We are of the opinion that we have an excellent apparatus for generalized interpretation. Large programming systems, largely format driven or interpretation oriented, will probably run efficiently in the HLS system, especially if the storage hierarchy functions adequately.

3.7 Structured Control

The aim is to devise HLS language for both procedure and control. This has been done successfully in simple language systems and should be extended to this general context. Every (or nearly every) statement should be usable as a part of a procedure specification or as a command.

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 language should include editing statements to control source text, and system control statements such as suspending or restarting program execution, breakpoint control, and control of system response to user action.

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

Hardware units have the behavior of 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-declared processing units.

The activity known as SYSGEN, which establishes the correlation between OS and the machine configuration, in he past has taken hours to complete. With self-declared units, SYSGEN will be trivial and dynamic reconfiguration of the system becomes possible.

The task handling (TASKING) will be system-controlled, rather than hardware-unrelated.

The system is expected to be a generalized interpreter apparatus for asynchronous decentralized control. We believe this to be possible because of the environment interface handling, and because of the preservation of meaning and the ability to transmit entire pieces of self-defined work.

3.8 Decimal Arithmetic

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

This attempt has not proved successful. Practically all procedural languages now permit decimal input and output. Some languages (like COBOL) also demand decimal arithmetic internally; most, however, allow the use of an internal 2 radix. The conversion between two radices creates errors (1/5 is exact in decimal, but not in 2 radices), which become "apparent bugs" in the user's program. (The users' tolerance of these apparent bugs, on the other hand, may leave genuine bugs uncovered.)

The reason for choosing 2 radix was once efficiency, now it is mainly compatibility.

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 thus not a basic economic or system efficiency issue.

Indeed the inplementation of decimal arithmetic or binary circuitry can be faster than 2 -radix and more LSI adapted, by going to redundant arithmetic, using the extra code-points in each digit. In HLS the loss of capacity in decimal can be redressed by mapping into a base 1000 system for archival storage. This mapping is entirely reversible, the result comes to within 2.4% of binary efficiency. This mapping is possible because the data descriptor can indicate the format change (indeed, with data descriptors data compression is seen in a completely new light.)

Therefore in the HLS system "decimal arithmetic" will be emphasized. For compatibility reasons, 2 -radix arithmetic will be provided also, as a major option.

3.9 OTHER LANGUAGES

There are many languages other than the six specially favored by HLS (the HLS language itself, PL/I, FORTRAN, COBOL, RPG, and APL). These languages, like ALGOL, JOVIAL, SNOBOL, LISP, etc., have their special purposes, their enthusiasts, and usually a body of existing debugged programs which the authors are not about to abandon. These will be accommodated, in most

cases, by compilers and, in some cases, by interpreters. In either case the language problem is similar and has two parts: (1) The processor must be written in HLS language and (2) The processor must translate from the source language to HLS language.

The first problem is the easier. A very successful PL/I compiler has been written in PL/I and since successful compilers for smaller languages have been written in APL, it is clear that since HLS language will have the semantic features of PL/I and APL it will be suitable to write compilers.

The second problem may be more difficult. Much experience has shown several problems when compiling from one high level language to another when the two languages allow one to say the same things at the same level but with statements that package ideas differently. This problem will be much less severe and perhaps almost non-existent because the self-describing data of HLS will allow efficient translation. The problems left will be met by compiling, when necessary, to a subset of the full high level language of the machine.

The favored languages must be compiled because among other problems name compression is essential. However, the compilers for these languages are small, operate fast, and do not make the object code incomprehensible to a person thinking in source language.

4. Consequences of the HLS Architecture

4.1 HLS and the User

A major reason for HLS is to allow the system manufacturer to reduce the user's cost of programming and reap an appropriate reward beyond the standard 30% of the user's total dollar outlay.

As a result of the expected intensive competition, price cutting due to LSI, deeper integration of the computer into the human society, and explosive growth of interactive computing, the 1970's will be the decade of the users.

There will be many, many more users, whose average computer training will be low. There being a variety of equipment to choose from, all reasonably priced, the user will tend to choose the system closest to his way of life. HLS, based firmly on procedural language computing, will tend to win in such a contest. The tedium of hex debugging, and memory overlay, will vanish, so the user can concentrate on his problem. Interactive programming will expecially be enhanced. Graphic processing tends to be based on list processing; it will be efficient here, too.

As the computer system takes over increasingly involved clerical tasks, in our increasingly complex society, the need for communicability will increase; so will the need for more meaningful RAS and security. The ensuing sections will show that HLS is a major step forward in these directions.

HLS should alleviate the problem that there are not going to be enough trained programmers to realize the full market potential. Assembly language programming is hard to learn, but a previous necessity. This training phase can now be completely bypassed.

Because of the expected execution efficiency, turnaround in interactive programming will be greatly improved. Making use of the storage hierarchy, smaller machines may now be in a position to handle large problems interpretively. New languages based on HLS will be easier to construct, and programs in different languages can be combined and run. The user's procedures will not be dependent on formats, since the latter information is contained in data descriptors. It, therefore, is entirely possible for the user to rerun his program with altered precision, even with rational or complex arithmetic.

4.2 Communicability

A nagging worry in planning for the future is, how can the computer system handle the complex transactions typical in the human society?

In HLS we recognize that information has meaning in its own right, and data transmittal should be a meaning-preserving process. In this way the man/program/hardware/software communication is put on a new basis.

For man-to-man and man-to-program communication, HSL offers comprehensible code, self-documented data, meaningful debug features, and data-meaningful security enforced through descriptors.

For program-to-program communication, HLS offers combinability without enforcing a unique language convention.

For program-to-machine communication, HLS offers faithful interpretive execution, and reruns with altered formats.

For man-machine interface, HLS has better turnaround for interactive computing, descriptor-based inquires, and better opportunity to use list processing on graphic material.

For machine-machine interface, HLS will try to have generalized interpretive control of asynchronous processes, local autonomy through self-described data, format remapping for data compaction or teleprocessing.

4.3 RAS and Security

In HLS, each piece of named information can be individually protected for security and checked for accuracy. There is the new opportunity to replicate the material for checking and note this fact on the descriptor, without altering the machine code. The protection, checking and redundancy can be redefined dynamically. It is also possible to lock a descriptor so that only the user with proper key-word can use the material.

If the data A is an array of size 3 x 4, then A(1), A(4,3), A(1,2,3) elements do not exist. A (or its subsets) is not a suitable branch target, and (A+B) is not meaningful unless B is also an array of size 3 x 4. Such meaning-dependent checks are trivial with data descriptors, and the user's debugging will become vastly simpler.

A critical resource in the coming decade will be field engineering personnel. To be viable, a system should (a) reduce the need for FE calls (b) make each FE call more effective and (c) simplify FE training.

HLS achieves these aims as follows:

a. Meaningful localization of error (self-checking via descriptors).
b. Meaningful duplication of important material via descriptors.
c. Automatic storage control allows avoidance of areas of known error occurrences.

d. Dynamic reconfiguration means that entire hardware boxes can be installed or disconnected at a moments notice and the workload will be equitably shared by the parts of the reconfigured system.
e. FE education is simplified by the new language interface especially when the control language is an integral part of the HLS language.
f. The diagnostic programming and programmed remedies (especially when hardware units are addressable by name) will be easier to accomplish. The drudgery of using assembly-language hardware debugging will be bypassed.
g. Interactive FE programming allows the machine to describe its own failures using the user's language.
h. HLS tends to create fewer machine error catastrophes. A machine error on branching, for instance, will usually lead to an error halt, rather than further, meaningless executions, compounding the trouble.

4.4 HLS efficiency and large machines

Compared with interpretation using instructions, the direct use of HLS or the interpreting of procedural languages on HLS should naturally be more efficient. There is reason to believe that to the very large machines, instructions are really unneeded understructures; their removal would lead to greater efficiency.

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

There will be no artificial intermediate result register assignments 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 efficiently. For large arrays, memory requirements are not based on access, but bandwidth. It is entirely 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 useful bandwidth when all or most of the lines are needed.

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

The human-oriented causality chain contained in statements removes bottlenecks due to the need to examine every instruction for conditional branch or interruption. Interruptions will be fully recoverable if testing for interrupts precedes the storing of the results.

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. These actions are hard to specify using instructions.

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

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

4.5 HLS efficiency and small machines

At the opposite 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 middle man" so to speak, and enables a high degree of efficiency. Why emulate S/360 which ultimately emulates a virtual procedure-interpreting machine?

Microinstructions, dependent heavily on machine details, are not the adequate basis for an architecture. But rather than normalizing at the instruction level probably paying a normalizing cost for every instruction, it should be far more rewarding to use the HLS procedure language level as architecture definition, and "normalize" at statement boundaries only.

Small machines often have narrow data paths (8, 16 bits), which lend themselves very naturally to character string processing as is typical in procedure languages, especially with the assistance of new LSI hardware. The use of functional memory for "token cracking" at the rate of one character per CPU cycle is an instance.

In small machines the decoding cost is often nontrivial. With array processing, one decoding can allow many arithmetic operations to be performed.

Memory is in critically short supply in a small machine. We expect HLS codes in general to occupy less space than S/360 codes. It is especially pertinent that the users are not encouraged to overclaim territory; dynamic array handling gives the user just the amount he needs in real time. Storage hierarchy allows larger problems to run, at least, and multiprogramming on a small scale can be achieved.

The small machine user of the past had been resigned to slow processing of small jobs. Using mainly compile-go techniques,

most small jobs have been compile-time limited. With HLS the "compiling" cost will be revised, sharply downward, and the quality of computing improves because of the interpretive nature of the system.

4.6 Programming System Implications

HLS will help combat the ever rising cost of creation, revision and maintenance of programming systems in several ways.

Complex operating systems are required by the present market place, and the outlook is for another quantum jump in complexity during the 1970's. With present methods, the complexity of our operating systems tax the intellectual resources and human endurance we can bring to bear on them. Experience within and outside IBM indicates the likelihood that if done in the same old way, data base and large shared systems will be beyond human capabilities.

The key to surmounting the difficulties consists of thorough incorporation of systematic formal and architectural discipline. Only in this manner can we define precisely how an operating system functions establish properties required of each component to insure global well-behavior, and prove that system components posses these required properties. Although our limited knowledge may enable us to only partially attain this goal, HLS provides a better base and more promising opportunities than we have had in the past.

The aptness of HLS architecture for language processes is evident. What is not obvious is that the scheme of interpretive operation via descriptors also embraces operating systems in a natural way. Operating systems and language interpreters have much in common. In both cases practice has evolved rather simple methods to complex interpretive schemes employing descriptor-like objects. For languages the objects are called symbol table entries, or dictionary entries. These descriptors contain the name and all attributes of simple or structured language variables. For operating systems the objects are called control blocks, and they describe the states and inter-relationships of system variables (logical resources, physical resources, units of control, etc.). In both cases the processes may be characterized as interpretive operation employing descriptors.

With this approach the following simplifications can be achieved:

a. Operating system descriptors and operations can be classified, organized, and implemented directly in microcode or hardware. This will provide extremely efficient primitive operations for operating system construction, within a consistent framework.

b. Uniform symbolic addressing within a storage hierarchy will entirely eliminate many current SCP functions. It will also relieve the user of concern

Another basic concept of HLS is that the user must be able to program in high level language and not be forced to think in machine language. With System/360 the high level programmer is confronted with the need to understand memory dumps that he gets in hex, diagnostic messages that involve lower language levels, and if he is good he is expected to understand the translation of his program to machine language. It seem to be inevitable that the machine language will creep out to where it doesn't belong, and the only way to keep it from reducing programmer productivity is to move to a high language level.

5. Appendix: 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 committee was as follows:

John C. McPherson (CHQ, Armonk), Chairman

Tien Chi Chen (Research San Jose)

Carl J. Conti (Poughkeepsie)

Claud M. Davis (Poughkeepsie)

Albin D. Kolwicz (Boulder)

John C. Laffan (Poughkeepsie)

Albert A. Magdall (Endicott)

Anthony Peacock (Poughkeepsie)

Anthony Proudman (Hursley-Poughkeepsie)

Nathaniel Rochester (Boston)

David Sayre (Research Yorktown)

Ralph F. Schauer (CD Poughkeepsie)

William S. Worley, Jr. (Time-Life)

5.1 What we have done

a) We have identified the need for HLS as a new architecture basis

b) We have examined the major procedural languages as candidates for the HLS language, and found them all deficient in some respects, especially regarding control.

c) We have agreed that a coherent HLS language and architecture can be developed. We have listed highlights of this language, and supplied much detail.

d) We have become convinced of the feasibility and basic soundness of concept.

e) We have left open some important detailed choices, as deeper studies with simulation verification is clearly required.

f) We recommend exploratory implementation for small and large systems.

g) We note 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 rather broad scale will ultimately be needed, if full value is to be quickly extracted from the concept. Thought should be given during the exploratory implementation as to how other IBM divisions can assist in this process.

5.2 Resolution

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."

320-3271
January 1970

A PROCEDURE-ORIENTED MACHINE LANGUAGE - PART I

A. Hassitt
J. W. Lageschulte
H. F. Smith

ABSTRACT

It is convenient and inefficient to program in a procedure-oriented language on existing computers. It is inconvenient and not always efficient to code in machine language. We discuss the requirement for a convenient and efficient machine language and we claim that APL fills these requirements. We have implemented an APL machine in microcode using the hardware of an IBM 360 Model 25. We describe the implementation and compare its performance with the performance of the IBM 360.

Index Terms for IBM Subject Index

Microprogramming
Machine Language
Compilers
APL
Performance
IBM System 360-25
07 - Computers
21 - Programming

Introduction

1. The complex computing systems of today have evolved from the simple machines of the first generation. Machines have become bigger and faster; new features such as floating point arithmetic and interrupt schemes have been added. Despite this evolution there is an increasing dichotomy between the way in which programmers should wish to use the machine, and the way in which the machine actually works. The higher level languages such as PL/I represent the way in which programmers should be writing programs. The low-level language, namely assembly language, represents the way the machine works. The compiler is a method of bridging the gap between the high-level and low-level languages. Although compilers have been widely and successfully used, it is obvious that they are not the complete answer. The fact is that machines and programmers are working in opposite directions and the result is an inefficient use of both machines and people. Compilers use a large amount of machine time; they usually generate code which is inefficient in space and time. What is needed is a machine language which facilitates the use of high-level languages. Even when the compiler produces efficient machine language code, this does not necessarily give the best use of the hardware; the machine language is sometimes a poor interface between the computing algorithm and the hardware. What is needed is a machine language which facilitates the efficient use of the hardware.

Requirements for a Machine Language

2. With existing technology it is quite possible to build very complex machines and it is obviously feasible to build a machine which

would directly execute the statements of a high-level language. There is no reason to believe that such a machine would be too expensive: let us consider two examples of such a machine. First consider a small or intermediate size machine. The IBM/360 model 25 is a good example of a modern machine of this class. The model 25 has hardware which is very fast and very simple. This hardware does not and cannot execute 360 instructions. There is a microcode control program which drives the hardware in such a way that it emulates the behavior of a 360; each 360 instruction requires from ten to a thousand microcode instructions. Writing microprograms for the model 25 is in many ways like writing conventional programs for a small simple machine. Any program which is written for a conventional computer could be written in microcode on the model 25; it would probably take two or three times longer to write the microcode, but it certainly could be done. Using these techniques it is possible to write a FORTRAN emulator, or a PL/I emulator, or an emulator for any high-level language. It might be contested that an emulator is not hardware; in reply we would point out that the model 25 is considered to be an "IBM 360 machine" even though it consists of hardware plus an emulator. A "FORTRAN machine" would consist of some hardware plus a FORTRAN emulator. As we have pointed out, microprogramming the model 25 is not too difficult. Microprogramming larger machines (for example the model 50) is difficult; however, we believe that in the future there will be a trend towards faster circuits and simpler microcodes in the larger machines. Let us now consider the problem of building a very fast machine. The IBM/360 model 195 is a recent example of such a machine. In order to achieve rapid execution speeds, the 195 consists of hardware which executes 360 instructions plus some very complex

hardware which analyzes and manipulates the instruction stream. This analysis is very involved and in our view is no more difficult than the analysis required in the execution of a high-level language.

3. Although it would be possible to build a FORTRAN machine or a PL/I machine, this would not necessarily be a good thing to do. What we need is a machine which satisfies both the programmers and the engineers. On the one hand, it should be easy and foolproof to program and it should make compiling easier. On the other hand it should be possible to build it with an economical amount of hardware. To be precise, we need a machine language with the following properties:

a) The language should be both powerful and complete. FORTRAN will not suffice. FORTRAN is powerful but not complete: consider the difficulty of writing a compiler or assembler or loader in FORTRAN.

b) The language should be simple. As a measure of simplicity we can use either the number of instructions in a program which simulates the language, or the number of microcode instructions in an emulator or the number of components in a hardware implementation. The size of the current PL/I compiler suggests that PL/I is too complex. There is a proposal by Sugimoto (1969) for a PL/I-like machine but there is no information on its implementation.

c) The language should be concise and should not ask the programmer to state the same thing repeatedly. On the IBM/360 (and almost all other machines) every time you wish to add two floating point numbers you have to repeat that it's floating point addition. In a higher level language, all you need say is '+' since the compiler can

easily discover the type of arithmetic to use.

d) The machine should check for errors at the place where they are likely to occur. The statements

```
DIMENSION A(100)
A(101) = 9.25
```

should produce a subscript range error. With most compilers on most computers the execution of these statements will not cause an immediate error, but it will usually generate some catastrophe at a later time. By the time the error is detected, all trace of the original cause is lost. Some compilers (for example, PL/I level F) have an option to produce code which checks for errors, but the cost, in terms of program space and execution time is so high that many programmers will not use these checks.

e) The machine should recognize that programmers use subroutines and that procedure-oriented languages use statements. It should therefore keep track of subroutine names and statement numbers; this can be done by compiler-generated code but it is easier, faster, and foolproof if the machine does it.

f) The machine should not force the programmer or compiler to throw away useful information. To take a simple example, if B and C are matrices then the APL statement

```
A ← B + C
```

will add all the elements of B and C and put the result in A. The equivalent FORTRAN program

```
DO 100 J = 1, M
```

```
      DO 100 I = 1, N
100   A(I,J) = B(I,J) + C(I,J)
```

loses the fact that we want all the elements of B and C and we are not really interested in I and J. The IBM/360 model 195 is forced to use its elaborate hardware to try and detect this type of loop and to organize the calculation more efficiently.

g) Programmers, particularly when working in high-level languages, use memory in a dynamic way. The machine should support the dynamic use of memory.

4. It would be possible to design a language with the above properties. However, there is an existing language, namely APL, which we believe has most of these properties. It is easy to see that APL has the virtues of a high-level language. What is perhaps surprising is that APL has the virtues of a machine language, namely, APL can be implemented in a reasonably small number of microcode instructions and APL can be used without needing a compiler. We will discuss an implementation of such a machine. First of all we will look at it from a user's point of view and then we will look at the underlying structure. Finally we will discuss the size, effort, and problems of implementation and we will compare performance of the APL machine with the IBM/360.

PERTINENT REFERENCES

CARPENTER, P. F., IBM Hursley, "Functional Memory Programming Studies - PL/I Incremental Compiler/Interpreter", January 16, 1970

DIJKSTRA, Edsger W., "The Multiprogramming System", Communications of the ACM, Vol. 11, #5, May 1968

HASSITT, A., Lageschule, J. W., Smith, H. F., "A Procedure Oriented Machine Language-Part I", TR 320-3271, IBM Confidential, Abstract and Introduction

ILIFFE, J. K., "Basic Machine Principles"

IRONS, Edgar T., "Experience with an Extensible Language", Communications of the ACM, Vol. 13, No. 1, January 1970

KILBURN, T., Morris, D., Rohl, J. S., Sumner, F. H., "A System Design Proposal", 1968 IFIP Proceedings

MC DONNELL, E. E., "A Formal Description of JCL"

MC KEEMAN, W. M., "Language Directed Computer Design", Fall Joint Computer Conference 1967

MORGAN, Howard L., "Spelling Correction in Systems Programs", Communications of the ACM, Vol. 13, No. 2, February 1970

WIRTH, Niklaus, "On Multiprogramming Machine Coding, and Computer Organization", Communications of the ACM, Vol. 12, No. 9, September 1969

IBM International Business Machines Corporation

1000 Westchester Avenue, White Plains, N.Y. 10604

Office of the President
Systems Development Division

May 7, 1970

Mr. R. B. Talmadge
IBM United Kingdom
Laboratories Limited
Hursley House, Hursley Park
Winchester, Hants, England

Dick, John is fighting back and that's healthy, at least to a point.

Thought you would be particularly interested in Markstein's letter.

B. O. Evans

BOE:dm
Attachment

Copies sent to Haddad and Humphrey for info.

5/1

SDD PRESIDENT
APR 30 9 41 AM '70

April 28, 1970

MEMORANDUM TO: Mr. B. O. Evans

After rereading the attached letter from Peter Markstein after talking to Dick Talmadge, I am bringing it to your attention because I think it fairly states the case for the opposition to our study group proposal from our most expert programmers. They are today's conservatives who fear change and are more comfortable with the original machine instruction base for their work.

There is one anomaly in this situation. The powerful interpreter needed to "interpret" HLS may not be very large. As Lathwell's memo pointed out the PL/I compiler is approximately 500,000 bytes, the APL/360 interpreter is 60,000 bytes, and a Model 25 APL emulater is 7,000 bytes.

Please note that Peter's position as stated in the final paragraph, that an "effort should be set up immediately to build a prototype and prove that HLS can work" is almost identical to the study group's position.

John

John C. McPherson

JCM:gp

att.

Date: March 11, 1970

From (location and address):
Thomas J. Watson Research Center
Dept. & Bldg: Dept. 470, Bldg. 801
Ext. & Tel. Ext.: 1065

IBM

Subject: HLS (Higher Level System) Proposal

Reference: Peacock-Conti Presentations of February 26 and 27, 1970

To: H. G. Cohen

The major theme of the HLS Proposal is to break away from the von Neumann architecture, since the underlying assumptions behind today's architecture are held by the proposal to be invalid.

The HLS system operates on statements (consider an APL statement as an example) rather then on instructions, it references objects by name rather than by machine address, the objects are self-describing, its operators are valid over wide classes of objects, its operators determine the nature of their operands dynamically, and it automatically manages its storage hierarchy, thus giving the impression of a uniformly referenced storage medium.

The goals of this proposal are laudable. Benefits claimed include easier implementation of applications and systems, good exploitation of LSI technology, and improved man-machine interface. A wealth of other benefits are claimed but these do not seem to depend as strongly on the radical architectural change as do the above.

The principal departure, of course, is to have the computer offer to the user as the lowest level of language, the equivalent of what is considered to be a higher level language today. There are some examples of similar approaches. The most notable of these is the Burroughs 5500 type machine. This series was intended to be programmed in Algol at the lowest user level, and the hardware was strongly influenced by this objective. While the basic machine is still of the von Neumann type, it has been designed to make the compilation of Algol particularly efficient, and indeed, users do not generally have an assembly language made available to them by Burroughs. HLS goes further by eliminating assembly language altogether.

Since HLS "interprets" a more ambitious instruction set than does System/360, it relies on a powerful interpreter built with or interpreted by LSI components. My reservation here is the following: We would be building in hardware, an

MAR 11 1970

interpreter of a language with which we have had insufficient experience, namely, a higher level language which is sufficiently powerful and rich to eliminate the user's need to lapse into a lower level language. I venture to say that the first attempts to implement this language in software will not be error free, and that several rounds of debugging and perhaps even redesign would be necessary. The production of software, unfortunately is not yet very systematized. Yet the hardware which such a language requires is merely physical embodiment of such a software interpreter. Thus, the methodology for producing the hardware may be set back to the methodology used for producing software. My fear is that at this time, hardware based on an interpreter will be subject to all the difficulties that software experiences today. The ease with which System/360 hardware was produced relative to the software, makes it unattractive to put the next hardware design on the same methodological footing as today's software.

A second item which would be moved out of software is the control of the storage hierarchy. Again, this control of a hierarchy of widely varying performance by any technique is not a completely solved problem. Automatic control of the cache in the Models 85 and 195 is successful, but the two storage media involved only differ in performance by one order of magnitude. In cases where the difference is more pronounced (e.g., paging systems as run on the Model 67), the best means of managing transfer of information between levels of storage is far from being a closed question.

To be sure, the atuomatic paging and the higher level language interpreter will not be built directly in hardware, but will be implemented in micro-code. Writable control stores will make reworking of these components easier than repairing a microprogramming error today. But firmware changes with the frequency of software changes would cause customers great anxiety, since they should view firmware as an extension of the hardware.

In short, the goals of HLS are noble, but the need to move into hardware functions which are traditionally done by software (but not well understood) compromises the stability of the hardware. Rather than shoot for making HLS the FS series, an ad tech effort should be set up immediately to build a prototype and prove that HLS can work. Then, for the next line after FS, the Company can base its decision about an HLS architecture on hard evidence.

P.W. Markstein
ld
cc: D.N. Streeter
cc: J. Mc Pherson (3/16/70)

SDD POUGHKEEPSIE
Dept. B58 - Bldg. 931
Extension 59900
April 22, 1970

Memorandum for Dr. R. B. Talmadge

Subject: High Level System Interim Report

Reference: Your memo to Mr. B. O. Evans of March 26, 1970

I would be very interested in receiving a copy of your detailed critique when it is available. Thank you very much.

R P Case

R. P. Case

RPC:mw

SDD - HARRISON
April 16, 1970

Memorandum to: Dr. R. B. Talmadge

Subject: High Level System Interim Report

Reference: Your memorandum of March 26, 1970.

I hate to keep troubling you but am very much interested in a detailed critique from you on the High Level System Interim Report and do want you to personally stay close to programming and advanced systems plans. Therefore, at your earliest convenience, please send me the detailed critique.

Thanks.

B. O. Evans

BOE:mr

Mr. B.O. Evans,
Harrison

26th March 1970

Subject: Higher Level System Interim Report

Reference: Your letter of 5th March 1970

I have read the subject report with interest and regret to say that I can find very little in it with which to agree. It appears to me to be naive where it is not erroneous, both in the justification arguments advanced and the conclusion drawn.

I am leaving tomorrow on holiday, returning the middle of April. I shall be glad to prepare a detailed critique at that time, if it will be of any use to you.

R.B. Talmadge
RBT/lae

B. O. Evans
1000 Westchester Avenue, White Plains, N.Y. 10604

March 5, 1970

Dear Dick,

You may have heard that we had a small task force studying the possibility of a higher level language system. John McPherson has led this effort which now reports enthusiastically about the potential of such an approach. I am forwarding my copy of the report for your study and consideration, and am very much interested in your conclusions and recommendations.

Sincerely,

B O Evans

Dr. R. P. Talmadge
IBM United Kingdom Lab., Ltd.
Hursley House, Hursley Park
Winchester, Hampshire,
England