Retrospective on High-Level Language Computer Architecture

David R. Ditzel

Bell Laboratories
Computing Science Research Center
Murray Hill, New Jersey

David A. Patterson

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, California

Introduction

High-level language computers (HLLC) have attracted interest in the architectural and programming community during the last 15 years; proposals have been made for machines directed towards the execution of various languages such as ALGOL,[1,2] APL,[3,4,5] BASIC,[6,7] COBOL,[8,9] FORTRAN,[10,11] LISP,[12,13] PASCAL,[14] PL/I,[15,16,17] SNOBOL,[18,19] and a host of specialized languages. Though numerous designs have been proposed, only a handful of high-level language computers have actually been implemented.[4,7,9,20,21] In examining the goals and successes of high-level language computers, the authors have found that most designs suffer from fundamental problems stemming from a misunderstanding of the issues involved in the design, use, and implementation of cost-effective computer systems. It is the intent of this paper to identify and discuss several issues applicable to high-level language computer architecture, to provide a more concrete definition of high-level language computers, and to suggest a direction for high-level language computer architectures of the future.

Review of Architectural Justifications

All too often the raison d'être for high-level language computer architecture is taken to be obvious, as shown by numerous proposals without justifications. We feel that the justifications are in no way obvious, and that a review of existing justifications reveals many shortcomings. An analysis of proposed HLLCs found that the most commonly cited justifications were as follows:

- Reduce the difficulty of writing compilers.
- Reduce the total system costs.
- Reduce software development costs.
- Eliminate or drastically reduce system software.
- Reduce the semantic gap between programming and machine languages.
- Make programs written in a HLL run more efficiently.
- Improve code compaction.
- Ease debugging.
- Investigate new architectures.
- Esoteric: Aesthetics or no stated advantages.

An almost universal justification for high-level language computers is the view that

> "the prime motivation for developing such a machine is to reduce system costs, for while hardware logic is becoming much cheaper, software is consuming a greater proportion of total system costs. A tremendous savings can be obtained by designing computer hardware that is oriented to aiding the programmer rather than to simplifying the computer designer's job."[22]

The solution to the software problem has appeared to be an increased use of "inexpensive" hardware. According to this viewpoint, the way to use this extra hardware is to raise the level of the machine language, so that in most cases there exists a one-to-one mapping between the source language and the internal machine language. One high-level instruction is intended to perform the task of several lower level instructions, potentially allowing faster execution. Higher-level instructions are believed to imply that a compiler should be smaller, simpler to implement, and should run faster than a compiler for a lower-level language machine. In many cases, mapping from the source to the internal form is a simple enough task to be done by software or hardware with the complexity of a preprocessor. High-level instructions are predicted to lower overall system development costs due to simpler compiler development and an improvement in debugging time, since the machine instructions reflect the operations in the user's source program.

Though not always stated as such, the above arguments for HLLCs are properly focused on the goal of achieving a more cost-effective computing system than is available from existing architectures. The primary means used to achieve a more cost-effective system are: (1) Faster computing through a new architecture and (2) Lower cost computing through reduced software costs. The desirability of these goals is not disputed, but many claims as to how these goals might be reached appear questionable.

CH1494-4/80/0000-0097 $00.75 © 1980 IEEE

Examination of Some "Axioms"

If paper designs of new computers are to be seriously discussed as advances, then the justifications must be valid. Implementations lend evidence for judging the success of the justifications, but often fall short of being able to prove them. There exists a number of "axioms" used to justify high-level language computers which we feel are either misdirected or not cost effective; several of these are now discussed.

Axiom 1: *High-level language computers are needed because efficient compilers are too complex and difficult to implement on conventional machines.*

Response: Part of the initial impetus for high-level language machines was the poor state of the art of compiler technology. Implementing compilers on conventional register oriented machines was often a brute force task. In 1967, McKeeman observed that "...compilers and operating systems are getting less reliable, consuming more memory, taking more time and systems programmers to develop..." and that attempts to use high-level languages for implementing systems programs had often failed because too much memory was required.[23] Because there often exists a one-to-one translation between the tokens of a HLL and a HLLC, and the algorithm for generating postfix instructions from an infix expression is simple, HLLCs are aesthetically appealing to those not familiar with modern compiler writing technology. It is acknowledged that code generation may be simpler for a high-level language computer. What needs to be made more fully understood is that a high-level language instruction set does not eliminate the need for compilers, nor does it greatly simplify them. The need and complexity of compilers extends far beyond code generation. The amount of code necessary for preprocessing, lexical analysis, syntax analysis, assembly, optimization, loading, error detection, error recovery and diagnostics often dwarfs the part of the compiler concerned with code generation. The level of the target computer does not seem to have enough of an effect on the size of a compiler to warrant a totally new architecture.

When we look at the state of the art of computers today, we find that the architectures have not changed substantially since 1967; there are no commercial direct or indirect execution high-level language machines. Yet in spite of this, the situation described by McKeeman has noticeably improved. The technology for writing compilers has improved to the point where compilers are typically written in high-level languages and use tools such as compiler-compilers,[24] automatic lexical analyzer generators[25] and parser generators[26] which greatly simplify the programmer's job. Techniques for efficient code generation are available in the literature.[27,28] What is not clear, however, is the optimality of code producible for HLLCs. Machines which have catered to high-level language instruction sets have had problems in generating and optimizing code for languages for which the machine was not specifically designed. For example, machines designed for ALGOL such as the Burroughs B6700[29] and the Manchester University MU5 [30] are good for ALGOL but have difficulty achieving comparable performance for FORTRAN programs.

Axiom 2: *A high-level language machine reduces software costs because programming is easier.*

Response: This would appear to be true only if programming were not done in a high-level language. There are no intrinsic reasons why compiling a program on a low-level computer should appear differently to the user than on a high-level language computer. There may, however, be implications resulting from the level of sophistication of the implementations on the respective computers. For example, the syntax diagnostics from the SYMBOL hardwired translator[31,32] were extremely crude compared to those of common compilers for lower level computers. On the other hand, the execution time diagnostics[33,34] provided through software on SYMBOL were far superior to those found on most computer systems. The differences which exist are due more to the efforts (or lack of efforts) to solve specific problems on a particular system than to the level of the hardware itself. There is nothing inherent in implementing a compiler in hardware which would prohibit excellent compile time diagnostics, nor anything in a low-level language machine which prohibits excellent execution time diagnostics.

Differences in ease of programming are explained by the efficiency and cost to implement a given language and given level of debugging tools. To the designer of a computer system, software costs may be less with a high-level language computer, but this does not necessarily lower total system costs. The goal should be to provide machines that allow the creation of efficient systems with excellent diagnostics. As the current issue is the difficulty of programming seen by the user of a system, not the implementation of the system, the conclusion is that there is no difference inherent in the machine organization itself.

Axiom 3: *High-level language computers are justified by the rising cost of software and falling cost of hardware which mandates the use of hardware in previous software domains.*

Response: There is no disagreement that over the past decade software costs have risen tremendously, while at the same time advances in VLSI technology have made the fabrication of large numbers of gates on a substrate very inexpensive. There are, however, several subtle inferences from this rather common axiom. The first inference is that HLLCs require more logic for their implementation than their contemporary counterparts. Since the primary goal is a cost-effective system, it is necessary to somehow justify this extra hardware; the cliche of the falling cost of hardware is most often used. A second inference is that by moving the implementation of algorithms from software into hardware that costs will be reduced. Assumptions are also made that the complexity of software algorithms are suitable for implementation in hardware.

The problem is that these inferences do not strike effectively against the problems which exist. While hardware fabrication costs are

smaller with integrated circuit technology, the development costs are still very large. There seems to be some confusion with regard to replication versus development costs. Development costs are not reduced by implementing traditional software functions in hardware; replication costs for software are inevitably less than those for hardware. The mere fact that the fabrication cost of gates is inexpensive has to be weighed very carefully against the technological barriers of implementing complex systems in hardware. There is an important distinction to be made between the *feasibility vs. reasonability* of implementing complex algorithms in hardware. Complex systems are rarely implemented without bugs in their initial implementation; this means that any performance gains of a technology have to be weighed very carefully against implementation flexibility. The hardware implemented language and operating system of the SYMBOL system took several years to debug, illustrating some of the dangers of implementing complex algorithms in hardware. Though better documentation and hardware debugging tools might have lessened debugging time, we feel that one of the lessons learned from SYMBOL is that the arbitrary migration of software to hardware will simply result in the exchange of software releases for engineering change orders.

Axiom 4: *A computer system should be oriented towards executing high-level languages more effectively.*

Response: There is very little argument with the basic tenet of this axiom. There is concern, however, that many proposed architectural directions do not serve to create a more efficacious computer system. Support for high-level languages is too often attacked by designing a computer to execute a particular language, with a tailored instruction set which has a high-level one-to-one mapping between the external source language and the internal machine language. This practice is seen as dangerous for two reasons. First, it imposes a global view which must conform to one particular language, and second, it emphasizes language support more than the efficiency of the entire system.

For most programming environments, a system must be able to effectively support multiple languages. If the primary language for the machine is not a good systems programming language, then the implementation of the systems language may turn out to be inefficient. Even if not seen by the end user, a systems language is required for implementing the operating system, compilers and other machine dependent software. It is not uncommon for an operating system to consume a third of the processing resources of a machine. Severe performance degradation is likely if the tailored instructions for the user language do not lend themselves to the efficient implementation of the systems language. A single instruction set tailored to one particular language is constrictive, as it can make implementation of other languages difficult and inefficient.

While the implementation of programming languages is important, this is only a partial step to achieving an efficient system. The architecture must make provisions for efficiently supporting operating systems issues such as process handling, memory management, file storage, peripheral interfacing, text processing and program compilation. Paying attention to the execution of the task at hand is more important than the implementation of the language in which the task will be programmed.

Axiom 5: *High-level language machines are desirable because they have very compact code.*

Response: Code compactness is often used as a measure of the quality of a computer; this seems a reasonable measure from an information theoretic point of view. Yet it is highly questionable whether code compaction actually achieves cost or speed goals. Also in question is whether code compaction is due primarily to the high-level nature of the instruction set.

Code compaction is said to help achieve a lower cost computer because less memory is required to run the same algorithm and thus less memory will have to be purchased. There are several weaknesses to this argument. To be significant, memory savings need to be substantial, particularly where a large hardware investment is needed to achieve code compaction. Secondly, very few systems are purchased with the knowledge that only one particular algorithm of a fixed size will be run. Memory savings are directly related to the amount of interpretation inherent in the instruction set. The memory savings on an APL machine is likely to be much greater than on an ALGOL machine. If we take the principle that a HLLC should be able to implement a variety of languages, then it is unlikely that one instruction set could achieve such a large savings. Finally, considering the rapidly falling cost of memory, program size reduction will become less important in total system cost.

Compact programs are also assumed to be desirable because they enhance execution. A common argument is that if a machine has to fetch fewer total bits, then it can fetch them faster, causing an improvement in execution times. This has two fundamental fallacies. The speed of the machine is not entirely limited by the data transfer rate, and the total number of bits fetched is not as important as the number of words fetched. A vivid example of the effect of code compaction was discovered in an informal experiment where the size of programs was reduced 20-30% on an Interdata 8/32 by using a loader which substituted short addresses for long addresses. Though the size of the programs was noticeably different, the execution speed when both versions were run differed by less than one per cent.[35] The paradox is explained by the fact that the computer uses an instruction prefetch which negates any difference in program fetch times. Again, in order to be effective, memory savings must be significant.

It should be noted that many comparisons of code density were made with respect to machines which are known to have inefficient encodings. A serious question which must be confronted is whether a similar or better improvement can be obtained by improving the instruction set of a low-level language machine. Recent studies would appear to make this very likely.[36,37,38] For machines not proposing

significant run time interpretation, the degree of code compaction appears to be comparable to what can be achieved in a traditional architecture.

Axiom 6: *Direct execution machines offer a cost-effective architecture for executing high-level languages.*

Response: Direct execution machines are defined to execute a source program without any form of translation to an intermediate language. Perhaps proposals for such machines stem from the feeling by users that

> "the compilation run of the machine, during which the language translation is accomplished, is a waste of time and money to the user since he must pay for this time, though he gets no problem answers for it."[10]

It is our feeling that direct execution machines, though perhaps technically feasible, will not become viable architectures for executing high-level languages. One of the principle reasons for this is the realization that it is not reasonable to begin executing programs without first checking for syntax or detectable semantic errors. In order to perform syntax analysis the program must be run through a preprocessor, lexical analyzer, parser, and error detection routines; these steps account for most of the phases of a compiler. Once a program has been parsed, it would seem foolish to not generate a more efficiently interpreted internal form for the machine to execute. In generating an internal instruction form, many decisions about how the program can be most efficiently executed can be "bound". With direct execution, every time a source statement is executed, this binding must be done again, causing a loss of execution time.

Though several aspects of direct execution have been examined in existing proposals, many problems still exist which even very complex hardware does not seem to be able to solve. For example, how would a direct execution machine know where to jump for a procedure call? Solutions to such problems are attacked by retaining intermediate information to lessen scanning and retranslation. Retaining intermediate information, however, is tantamount to compiling, thereby transforming an optimized direct execution machine into a more conventional architecture which uses a compiler. There appear to be no convincing reasons for using direct execution as an implementation technique which can not be solved cheaper and faster using indirect execution machines. If anything, direct execution machines are likely to have an exorbitant cost and very poor performance.

A New Look at High-Level Language Architecture

Though many justifications for HLLCs seem to have been misdirected, we do not mean to imply that there are not good reasons for additional research, merely that the motivations must be credible. In fact, we feel that there exists great promise for more cost-effective computer systems by taking a new look at the actual issues. The primary focus however must be on the word **system.** The architecture of the future would appear to be the High-Level Language Computer System (HLLCS). There is no doubt that there exists the need for systems directed exclusively for high-level language use. The research issue is to define and build the most cost-effective architecture for this task.

Definition of a High-Level Language Computer System

One of the difficulties with this subject is the lack of a useful definition of a HLLCS. A definition proposed by Chu is:[39]

> "A high-level language computer system is one that can accept and execute a high-level language program."

This definition is almost useless in distinguishing which computers are and which are not HLLCs, as every computer that has a HLL compiler is considered to be a high-level language computer system. We cannot think of a single commercial computer or microcomputer that does not satisfy this definition. One could even build the software that would classify a Turing Machine as a HLLCS. This definition does, however, serve the useful purpose of focusing on the function rather than the implementation of a HLLCS.

The following description of a High-Level Language Computer System, as it appears to a user of that system, is proposed as a more discriminatory definition.

A High-Level Language Computer System is one that:

(1) *Uses high-level languages for all programming, debugging and other user/system interactions.*

(2) *Discovers and reports syntax and execution time errors in terms of the high-level language source program.*

(3) *Does not have any outward appearance of transformations from the user programming language to any internal languages.*

Any system that claims it is a HLLCS must meet these three requirements. The first simply requires that all programming be done in a high-level language. The second requirement can not be met by omission. Debugging tools for a HLLCS must exist which allow the user to query the system about the status of his program with the same degree of detail which might be expected using a machine language debugger.† In order to meet the third requirement, the transformation between the high-level programming language and any internal machine language must be transparently reversible. Errors will occur at the machine level, not at the user program level, but error diagnostics must be mapped back to the user's high-level source program.

This definition does not restrict the implementation to hardware or software. It does, however, require that any HLLCS be able to detect errors and report these errors in terms that do not rely upon an understanding of the implementation. A user of the system who is

† Such a debugger exists on a commercial computer. The NCR Criterion 8500[40] has a symbolic debugger for COBOL. The COBOL programmer is able to trace or dump any COBOL variables or statements by setting flags which tell the microcode to invoke the debugger when appropriate events occur.[9]

ignorant of the implementation is therefore at no disadvantage.

A Measure of Architectures for High-Level Language Computer Systems

Given our definition of a HLLCS, we could define a high-level language computer architecture as one which allows an efficient implementation of a HLLCS. Perhaps it would be more useful to define some measures of evaluation of architectures in terms of HLLCS. Three architectural measures that indicate efficiency of a HLLCS implementation are proposed. Briefly, for a given machine, a set of programs are compiled and executed under two cases; the first case meets all the requirements of a HLLCS and the second ignores the requirements in an attempt to gain efficiency. The three measures are simply the ratios of the execution time, program sizes and compile times for the two cases. The range of all measures should then be between 0 and 1, with the larger numbers indicating greater efficiency. The next two paragraphs describe the terms more precisely.

Let P be a representative set of syntactically and semantically valid programs for some high-level language. For a given HLLCS, H, on a machine M, let T_H be the total execution time for this set of programs. Let L be a fast execution time system on M that is not constrained to meet the HLLCS requirements of checking and reporting errors; and T_L be the execution time for this same set of programs P. Then the *High-Level Language Execution Support Factor(HLLESF)* is defined as the ratio of T_L to T_H. The HLLESF is then an indication of how well the architecture supports a HLLCS. If a system has a HLLESF close to one, it clearly has an architecture that lends itself to efficient implementation of HLLCSs. SYMBOL is an example of a machine with a HLLESF of one and if the B6700 meets our definition of HLLCS, it also does not gain performance by turning off error checking. Conversely, a machine whose HLLESF is close to zero clearly does not have an architecture amenable to a HLLCS. Since the classic execution ratio of interpreters to compilers is an order of magnitude, we would expect these machines to have a HLLESF of about 0.1.

The HLLESF may also be an indirect measure of the quality of the software that is developed on a computer. A very small HLLESF would encourage programmers to remove all error checking once the program is "debugged". As there is some doubt whether a large program is ever debugged, disregarding error checking during production runs is certainly an undesirable practice.

Conversely, one would expect systems with a HLLESF close to one to encourage programmers to leave the error checking in their programs, thereby enhancing reliability. Given the same caliber of programmers and computers of the same performance, one would expect that the higher the HLLESF, the greater the reliability of the software produced.

This approach to a measure of execution support can also be generalized to measures for program size and preparation time. The *HLL Size Support Factor (HLLSSF)* is defined to be the ratio of the size of the complete set of programs required to write and execute (i.e., source, object) P using L to the size of the programs of P using H. The *HLL Preparation time Support Factor (HLLPSF)* is defined to be the ratio of the preparation time (i.e.,compilation, linking, loading) for the of the complete set of programs P using L to the program prepartion time of P using H. HLLSSF and HLLPSF are less important than HLLESF, but they are interesting measures.

Designing High-Level Language Computer Systems

High-level language computer systems will be built; it is just a matter of when and how cost effective they will be. In an attempt to push designs along a successful path, we would like to speculate on several attributes which we feel will be part of a high-performance cost-effective HLLCS of the mid 1980's.

Attribute 1. The system will efficiently support a systems programming languages such as BCPL,[41] BLISS,[42] or C,[43] for writing the operating system, compilers, debuggers, editors, and other software which must deal with the low level details of the machine and its peripherals.

Attribute 2. The architecture will be oriented towards the support of operating systems. For example, process handling and context switching must be extremely efficient. It is not uncommon for a third of the CPU resources of current computing systems to be used for the operating system and other utilities not directly concerned with the execution of a user's application program.

Attribute 3. There will be one or more instruction sets which will be output from high-level language compilers. The number of specifically tailored instructions sets will be related to the differences in the level of interpretation inherent in the languages supported by the system. For example, BCPL, BLISS, and C compilers are likely to generate code using the same instruction set, though different instruction sets will be used for highly interpretive languages such as LISP and SNOBOL.

Attribute 4. The instruction set(s) will be optimized for the way programming languages are used. Special purpose hardware will be dedicated only for those functions which are known to occur frequently. There will generally not exist a one-to-one correspondence between the source and object code. The "level" of the instruction set will be raised only when there are specific advantages to be gained.

Attribute 5. The instruction set will be designed to be generated by a compiler. This requires attention to details of orthogonality, and the elimination of complex instructions which compilers are not reasonably able to generate. One of the reasons why high-level languages have been considered inefficient, compared to hand generated code, is that the compilers could not efficiently cope with awkward instruction sets.[44]

Attribute 6. The instruction set will not inhibit well known

implementation techniques such as pipelining and instruction prefetching. Complex instructions will be avoided if they tend to block a pipeline or create difficulties with interrupting and restarting instructions. Besides the instruction set architecture, attention will be paid to optimizing the underlying hardware architecture which will execute the instruction set.

Attribute 7. Details of how the transparent re-mapping of object code back into source will be dealt with from the beginning. This, of necessity, impacts the instruction set and the details of compiler code generation. The architecture may be affected; for example, descriptor-based and tagged architectures[45,46,34] are effective in retaining type information needed for full source level debugging.

Attribute 8. A HLLCS is independent from its implementation; whether the implementation is achieved mainly by hardware or mainly by software has no bearing on meeting the definition of a HLLCS. The relative amounts of hardware and software will not be an issue except in regard to how it relates to execution speed and cost. Experience with the SYMBOL system showed that there is no reason to think that merely increasing the amount of hardware in a system will make it run proportionally faster.[47] A large computer with a great variety of "high-level" instructions and hardware is no more desirable than a very simple but fast computer which can interpret the same higher-level instruction set with the same performance.

Attribute 9. Good compiler technology will have an important role in the overall efficiency of the system. Code generation will not be done in the easiest fashion possible, but rather in the manner that produces the most efficient code. The compilers will use traditional code generation optimization techniques such as constant folding, expression re-ordering, and moving invariant computations out of loops.[48,49] Very high performance machines may use advanced compiler techniques such as common subexpression elimination, replacement of procedure calls by in-line code, removal of statements whose outcome can be determined at compile time by global data-flow analysis, and generating instructions with execution hints to the hardware (such as which way a conditional branch is most likely to jump), provided that these advanced techniques do not impair the achievement of a HLLCS.

Attribute 10. The system will be a refinement of several previous, less ambitious systems. It is not reasonable to expect that a complex system can be designed successfully without actual use and iterative improvement. As the specification of a programming language or operating system is often a moving target, the system will avoid obsolescence by being flexible, allowing for incremental change without total system redesign.

A Scheme for the Transparent Reversibility of the Compilation Process

The one major obstacle preventing the realization of HLLCSs seems to be the difficulty of relating execution errors and debugging information to the source program. Current debugging tools often fail to inform the user of anything more than the line number of the statement in execution. Execution time penalties discourage providing even this crude piece of information; Shustek reports that for a particular PL/I program on an IBM 370/168, 16% of the instructions executed, representing 23% of the execution time, were moves used to record the current statement number from the source program.[50] Such execution time penalties are not required to provide sophisticated debugging. Indeed, to achieve a HLLESF of 1, there should be no execution time penalties. A method is now suggested to allow the implementation of efficient high-level language diagnostics and debugging tools which meet the criteria of a HLLCS.

First, it must be possible to identify the statement containing the execution error. Giving the line number is sufficient only if the source statement can also be printed. This presents many problems in itself, for example, with separate compilation there may more than one set of line numbers to consider, and there is generally no restriction that the user cannot modify the source program after compiling it (in this case obtaining the line number from the object program would not necessarily lead to the correct source statement). The ideal approach would be to "decompile" the object program to reproduce the offending source statement. This approach has been studied[51] and was used successfully on the SYMBOL system.[34] This approach is not recommended for many reasons, the most important of which is that it is not possible to recreate the original source line exactly.

Three data structures are used to obtain the original source statements from the object code. The first is an address map which uniquely identifies a range of addresses in a load module with the corresponding source statement number. The second is a copy of the source program which created the object module. A pointer to the user's source is sufficient until this original source is somehow modified, at which time a separate copy of the source which created the load module must be saved. A third data structure, the symbol table, is needed to print the name and values of operands with each load module. The symbol table contains the external ASCII name and all type information for operands. All of these data structures are permanently stored with every load module and as they are needed infrequently, they can be stored on high latency devices.

More detailed information is required to meet the definition of a HLLCS. It is necessary to identify the particular operator or operand in the source statement which caused the error to occur. To achieve this granularity, pattern matching is used -- a decompiled version of the statement is compared with the original source statement. The decompiled version obtains the operator or operand in question from the state of the machine, which in turn is matched to the original

source statement. An undesirable consequence is that no compile time optimizations, such as the elimination of common subexpressions or program restructuring, can be done if they interfere with the decompilation process. The equivalent of the "imprecise interrupt"[52] at the source program level can not be allowed to happen. The instruction set can affect the ease and success of decompilation, it is felt that stack machines offer instruction sets which are easiest to decompile. Though certainly not the only solution, the above scenario is one approach which can be used to relate machine errors to the high-level source program.

Conclusion

Much of the prior work has not carefully examined the reasons for proposing new architectures. Possibly as a result, few of these proposals went beyond paper designs. One of the faults of HLL Computers is that they ignored the view of the total system; a bare architecture without the surrounding system can no longer be considered as a viable solution to software problems. Hopefully, the realization that almost any computer can be transformed into a HLLCS through the appropriate software will allow incremental growth of present systems into HLLCSs. Adherence to the more demanding definition and architectural evaluation in terms of the given measures should lead to more cost-effective systems for high-level languages. Achieving total high-level language support while increasing performance and lowering costs is the job for designers of High-Level Language Computer Systems.

Acknowledgements

The authors would like to thank R. Carpenter, E. T. Cohen, A. M. Despain, L. I. Dickman, A. G. Fraser, S. C. Johnson, K. Lew, D. L. Presotto, C. H. Séquin, M. Turner, and W. Uejio for their helpful comments and criticisms during the preparation of this paper. This research was sponsored in part by Bell Laboratories and in part by the Defense Advance Research Projects Agency (DoD), ARPA Order No. 3803, and Monitored by Naval Electronic System Command under Contract No. N00039-78-G-0013-004. Support for the preparation of this paper at the University of California at Berkeley was provided in part by the National Science Foundation under grant MCS 7807291.

References

1. H. J. Lane, "An Algol 68 Machine and Translator," Computer Languages Group Report UCLA-ENG-7369, Computer Science Department, University of California (1973). Ph.D. Dissertation
2. A. S. Tanenbaum, *Design and Implementation of an Algol 68 Virtual Machine*, AFDELING INFORMATICA, Amsterdam (June 1973).
3. P. S. Abrams, *An APL Machine*, Stanford University (1970). Ph.D. Dissertation
4. A. Hassitt, J. W. Lageschulte, and L. E. Lyon, "Implementation of a High Level Language Machine," *Communications of the ACM* **16**(4), pp. 199-212 (April 1973).
5. S. C. Schroeder and L. E. Vaughn, "A High Order Language Optimal Execution Processor (FIRST)," *Proceedings of the ACM-IEEE Symposium on High-Level-Language Computer Architecture*, pp. 109-116 (November 1973).
6. J. J. Burkle, A. Frick, and C. Schlier, "Hardware Structures for the Interpretation of High Level Languages," *Proceedings of a Conference on Struktur und Betrieb von Rechensystemen* , Munich (1978).
7. H. J. Burkle, A. Frick, and C. Schlier, "High Level Language Oriented Hardware and the Post-von Neumann Era," *Proceedings of the Fifth Annual Symposium on Computer Architecture*, pp. 60-65 (April 1978).
8. R. J. Chevance, "A Cobol Machine," *ACM SIGPLAN/SIGMICRO Interface Meeting*, New York, pp. 139-144 (1973).
9. M. D. Shapiro, "The Criterion COBOL System," *Proceedings of the 1978 AFIPS Conference*, pp. 1049-1054, AFIPS Press (1978).
10. T. R. Bashkow, A. Sasson, and A. Kronfeld, "System Design of a FORTRAN Machine," *IEEE Transactions on Computers* **EC-16**(4), pp. 485-499 (August 1967).
11. A. J. Melbourne and J. M. Pugmire, "A Small Computer for the Direct Processing of FORTRAN Statements," *Computer Journal*, pp. 24-27 (April 1965).
12. R. Greenblatt, T. Knight, J. Holloway, and D. Moon, *The LISP Machine*, In preparation.
13. L. P. Deutsch, *Experience with a Microprogrammed Interlisp System*, In preparation.
14. Western Digital Corporation, *PASCAL Microengine Reference Manual*, March 1979.
15. M. Sugimoto, "PL/I Reducer and Direct Processor," *Proceedings of the 24th ACM National Conference*, New York, pp. 519-538, ACM (1969).
16. D. B. Wortman, *A Study of Language Directed Computer Design*, Stanford University (1972). Ph.D. Dissertation
17. G. J. Myers, *Advances in Computer Architecture*, John Wiley & Sons (1978).
18. M. D. Shapiro, "A SNOBOL Machine: A Higher-Level Language Processor in a Conventional Hardware Framework," *Digest of the Sixth Annual IEEE Computer Society International Conference*, pp. 41-44 (1972).
19. M. D. Shapiro, *A SNOBOL Machine: Functional Architectural Concepts of a String Processor*, Purdue University (June 1972). Ph.D. Dissertation
20. R. Rice and W. R. Smith, "SYMBOL -- A Major Departure from Classic Software Dominated von Neumann Computing Systems," *Proceedings of the AFIPS 1971 Spring Joint Computer Conference*, Montvale, N.J., pp. 575-587, AFIPS Press (1971).
21. H. Weber, "A Microprogrammed Implementation of EULER on IBM System/360 Model 30," *Communications of the ACM* **10**(9), pp. 549-558 (September 1967).
22. W. C. Nielsen, "Design of an Aerospace Computer for Direct HOL Execution," *Proceedings of the ACM-IEEE Symposium on High-Level-Language Computer Architecture*, pp. 34-42 (November 1973).

23. W. M. McKeeman, "Language Directed Computer Design," *AFIPS 1967 Fall Joint Computer Conference*, pp. 413-417 (1967).

24. B. W. Leverett, R. D. G. Cattell, S. O. Hobbs, J. M. Newcomer, A. H. Reiner, B. R. Schatz, and W. A. Wulf, "An Overview of the Production Quality Compiler-Compiler Project," Report CMU-CS-79-105, Carnegie-Mellon University (February 1979).

25. M. E. Lesk, "Lex — A Lexical Analyzer Generator," Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey (October 1975).

26. S. C. Johnson, *Yacc: Yet Another Compiler-Compiler*, Bell Laboratories internal memorandum (1978).

27. A. V. Aho and S. C. Johnson, "Optimal Code Generation for Expression Trees," *J. Assoc. Comp. Mach.* **23**(3), pp. 488-501 (1975). Also in *Proc. ACM Symp. on Theory of Computing*, pp. 207-217, 1975.

28. J. L. Bruno and T. Lassagne, "The Generation of Optimal Code for Stack Machines," *Journal of the ACM* **22**(3), pp. 382-396 (July 1975).

29. E. I. Organick, *Computer System Organization:The B5700/B6700 Series*, Academic Press (1973).

30. R. N. Ibbett and P. C. Capon, "The Development of the MU5 Computer System," *Communications of the ACM* **21**(1), pp. 13-24 (January 1978).

31. T. A. Laliotis, "Implementation Aspects of the SYMBOL Hardware Compiler," *Proceedings of the First Annual Symposium on Computer Architecture*, pp. 111-115 (1973).

32. J. W. Anderberg, "Source Program Analysis and Object String Generation Algorithms and their Implementation in the SYMBOL 2R Translator," Report NSF-OCA-GJ33097-CL7410, Cyclone Computer Laboratory, Iowa State University, Ames, Iowa (1974). NTIS number PB-230 614/AS

33. D. R. Ditzel, "Interactive Debugging Tools for a Block Structured Programming Language," Report MCS72-03642-CL7802, Cyclone Computer Laboratory, Iowa State University, Ames, Iowa (1978).

34. D. R. Ditzel, "High Level Language Debugging Tools on the SYMBOL Computer System," *Submitted to the 1980 Workshop on High-Level Language Computer Architecture* (January 1980).

35. D. M. Ritchie and S.C. Johnson, *Private Communication*, Bell Laboratories (1979).

36. A. S. Tanenbaum, "Implications of Structured Programming on Machine Architecture," *Communications of the ACM*, pp. 237-246 (March 1978).

37. E. C. R. Hehner, "Matching Program and Data Representations to a Computing Environment," Technical Report CSRG-44, Computer Systems Research Group, University of Toronto, Toronto, Canada (November 1974). Ph.D. Dissertation

38. E. C. R. Hehner, "Computer Design to Minimize Memory Requirements," *Computer* **9**(8), pp. 65-70 (1976).

39. Y. Chu, "Concepts of High-Level Language Computer Architecture," in *High Level Language Computer Architecture*, ed. Y. Chu, Academic Press, New York (1975).

40. T. Tang and K. O'Flaugherty , "Virtual Machines and the NCR Criterion," *Datamation* , pp. 129-134 (April 1978).

41. M. Richards, "BCPL: A Tool for Compiler Writing and Structured Programming," *Proceedings of the AFIPS 1969 SJCC* (1969).

42. W. A. Wulf, D. B. Russell, and A. N. Haberman, "BLISS: A Language for Systems Programming," *Communications of the ACM* **14**(12), pp. 780-790 (December 1971).

43. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey (1978).

44. N. Wirth, "On 'PASCAL', Code Generation, and the CDC 6000 Computer," Technical Report TR-257, Stanford University, Stanford, California.

45. E. A. Feustal, "On the Advantages of Tagged Architecture," *IEEE Transactions on Computers* **C-22**(7), pp. 644-656 (July 1973).

46. R. J. Zingg and H. Richards, Jr., "SYMBOL: A System Tailored to the Structure of Data," *Proceedings of the National Electronics Conference*, Oak Brook, Illinois **27**, pp. 306-311, National Electronics Conference, Inc. (1972).

47. D. R. Ditzel and W. A. Kwinn, "Reflections on a High Level Language Computer System or Parting Thoughts on the SYMBOL Project," *Submitted to the 1980 Workshop on High-Level Language Computer Architecture* (January 1980).

48. W. Wulf, R. K. Johnsson, C. B. Weinstock, S. O. Hobbs, and C. M. Geschke, *The Design of an Optimizing Compiler*, American Elsevier, New York, N. Y. (1975).

49. A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, Mass. (1977).

50. L. J. Shustek, "Analysis and Performance of Computer Instruction Sets," STAN-CS-78-658, Stanford University (May 1978).

51. C. R. Hollander, "Decompilation of Object Programs," Technical Report No. 54, Stanford University (January 1973). Ph.D. Dissertation

52. D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo, "The IBM System/360 Model 91: Machine Philosophy and Instruction Handling," *IBM Journal of Reserch and Development* **11**(1), pp. 8-24 (January 1967).