1982

The execution of high level languages

B. G. Hill
University of Wollongong

HE EXECUTION OF HIGH LEVEL LANGUAGES

by

B.G. HILL

Preprint No. 82-16

P.O. Box 1144, WOLLONGONG N.S.W. 2500, AUSTRALIA
tel (042)-282-981
telex AA29022

CONTENTS

CHAPTER 1

INTRODUCTION

Most computer programmers use a high-level language (HLL) to code their solutions to stated problems. Such languages (FORTRAN, COBOL, Algol, Pascal, C) enable the programmer to code in a more efficient, cost-effective manner than in lower-level (assembler) languages. Further, such programs are "cleaner", less likely to contain "bugs", and provide a degree of readability difficult to achieve in assembly languages.

Additionally, most programmers (as distinct from computer engineers) have a grasp, but not detailed knowledge, of the processes that occur after completion of program coding. These processes include the translation stage, any linking and loading necessary, and the actual execution stage. In particular, most programmers would claim that the compiler translation stage, which takes a source program, coded in a HLL, as input, and produces a machine code program, suitable for execution by the electronic circuitry of the computer, as output, is well understood.

Unfortunately, this process in now only rarely the one that actually takes place. The use of emulators, interpreters (implemented in both hardware and software) and microcode is now widespread, and the firm dividing line between hardware and software has been decidedly blurred for many years. Chapters 2 and 3 shall endeavour to define the terms introduced above, and provide a clear basis for future chapters.

As the microprocessor "revolution" has shown, the price of selected components has decreased dramatically over a period, and certain functions which would, only recently, have been prohibitively expensive to build into hardware are becoming feasible. In particular, the "compilation" function, being a general requirement of computing, is subject to attack, with aspects of the translation process being implemented in hardware. These aspects range from better defined, microcoded "basic" machine instructions, through hardware interpreters and read-only memory (ROM) language implementations to actual direct execution language (DEL) machines. Chapter 4 will canvas the scope of change in this area, describing models of the implementations to be

considered, along with the relative advantages and disadvantages of each. Chapter 5 will give some practical examples, categorising them in terms of the above models.

One model of extreme interest is that of a computer which directly executes a high level language. The concept of a computer using a HLL as its machine language, with no intermediate code of any sort, and no translator available, or needed, is still a little startling. There are tremendous advantages to be gained and, naturally, some costs to be paid. Chapter 6 will outline the internal details of such a computer, and describe its working features.

CHAPTER 2

TERMINOLOGY

1. Translation and Interpretation

Computers are capable of executing certain discrete very basic primitive instructions. Historically, these base primitive instructions were at the very lowest level of man - machine communication, and consisted of strings of bits (that is, zeros and ones). The language comprising these primitive instructions was termed the machine language of the machine, and was directly executable by the machine (that is, such instructions were implemented directly in the electronic circuitry of the machine).

When a computer designer is about to implement a new computer, one of the factors to be considered is the composition of the new machine's machine language. In the past, it has been of primary importance to keep the cost of implementing this language to reasonable limits. This has meant keeping the machine language as simple as possible, given that it must be capable of coping with the expected performance of the machine. Thus machine languages have been, and still are, poorly designed from a human interaction viewpoint, and this has led to a realisation that programming in such languages is economically unjustifiable.

The only realistic way to overcome this problem is to design a new set of programming instructions which are more convenient for humans to use. In all probability such a new set of instructions could not be implemented as electronic circuitry, and thus some sort of translation process is necessary to convert these instructions into statements of the machine language.

To achieve this requires a replacement of each statement in the new user language (call it U) with (one or more) statements in the machine language (call it M), such that the final product is equivalent to the original sequence of statements of U. The final product is thus a sequence of statements in M which can be directly executed by the computer. This process is known as translation. Once done, the program can be executed repeatedly without further translation.

Alternatively, one could write a program in M which accepts programs written in U as data, and executes them by sequentially

a. examining each U instruction,

b. preparing a series of equivalent instructions in M, and

c. then executing these instructions.

This process is known as interpretation. The program is executed in a series of small translation, execution steps, which must be undertaken on every execution of the program.

It can be seen that these two processes are similar. Both accept statements in the U language and convert these to statements in the M language. The difference lies in scale and timing. In translation, the U program is converted into an M program, after which the U program can be forgotten. The M program can be executed, in its entirety, many times. In interpretation, this process takes place on a user instruction by user instruction basis, and no final M program is produced.

2. Virtual Machines

It is desirable to consider the above processes as the rationale of an imaginary (virtual) machine. This machine takes U as its machine language. Users can write programs in U and have this virtual machine execute them. That machines of this nature have been kept hypothetical, rather than constructed, has been merely a function of cost. This is not a limiting factor in practice, as users can write programs for such virtual machines as though they really existed, and allow the machine to execute them. There is no conceptual requirement to know the manner in which the machine actually executes the programs.

3. A Hierarchy of Virtual Machines

Once the "existence" of a virtual machine such as that proposed above is accepted, there is no difficulty in accepting a hierarchy of virtual machines, each of which has its own "machine language", and each of which is based upon the virtual machine and its language one step lower in the hierarchy. Most programmers are very familiar with this environment, being accustomed to working on one level with a machine that uses a mnemonic assembler as its machine language and, further up the hierarchy, with a machine that uses a HLL such as Pascal as its machine language. Each program written in any given level of the hierarchy is then either

a. translated into the machine language of a virtual machine at a lower level in the hierarchy, or

b. interpreted by an interpreter running on a virtual machine at a lower level in the hierarchy.

However, the user at the given level need have no knowledge of the translators and interpreters at the levels lower in the hierarchy. The only point of interest to such a user is that his virtual machine executes his programs.

It should be noted that the hierarchy is bound at the lower level by the requirement that programs at the very lowest level must be directly executable by the electronic circuitry. However, there is no upper bound, and the virtual machines at this level have been, and continue to be, expanding in properties and in number. This process could conceivably continue until a "perfect" language is created although, given the nature of humans, it is unlikely that this will ever be agreed upon, even if it is created. Virtual machines have already been created at a higher level than that of the currently accepted HLL - preprocessors and data base managers are two examples.

4. Microcode

Computers currently available vary greatly in apparent structure. Their assembly languages have a rich content, offering powerful functions which would have been considered unrealistic only a few years ago. The tool which has enabled this expansion at the conventional machine level has been microprogramming.

Microprogramming is the technique of using algorithmic methods to define the microscopic state of the computer. Using microprogramming, it is possible to structure the architecture of a computer in such a way as to simplify its design and implementation. The use of microprogramming made it possible to delay final decisions about the actual instruction set until the microprogrammable hardware was nearly finished, allowing greater freedom in implementation detail. Further, it permitted the implementation of features which would have been greatly more expensive had they been implemented in electronic circuitry alone.

This facility rapidly expanded to enable multi-level machines, where the lowest level was the electronic circuitry, which used a set of microprograms as its machine language. Above this level existed the "conventional machine" level, where statements were interpreted by the microprograms. In this sense, only very basic "primitive" operations were actually implemented by use of electronic circuitry. The "conventional machine language" statements became calls to microprogram routines, where the microcode

was an algorithmic combination of the primitives in a way that performed the operation of the given statement. Such microcode enabled various features of the actual computer to be exploited and maximised. For example, some machines stressed high speed instruction sets, others wide breadth of instruction set, and others HLL support instruction sets. In fact, with different microprograms, there is no reason why single examples of variants of a single computer design should not be efficient in each of the above cases.

Unfortunately, the advent of microprocessors confused the issue somewhat, as some believed that microprogramming equated with programming a microprocessor in assembly language. Adding to this confusion is the fact that some microprocessors are microprogrammed (for example, the 8080), whilst others are not (for example, the 6800).

5. Emulator

The complete set of microprograms which act on the electronic circuitry to define the instruction set (that is, the machine language) of a given machine are actually a definition of that machine. This complete set of microprograms is termed the emulator of the machine. Note that the machine which is realised by an emulator is a virtual machine in the sense described above. In effect, an emulator is an interpreter running on the lowest level virtual machine.

6. Memory Access

At a level commensurate with the "conventional machine" level, there exists features of some computers which, though primarily properties of the operating system of the computer, are implemented by microprogramming.This allows the operating system much faster response in commonly occurring situations. Examples include the use of memory address determination in paging and segmentation environments, and process control switching.

It is noted that in the microprogram, conventional machine and operating system levels of the virtual machine hierarchy, the user programmer has no access privileges. These systems run the interpreters and translators needed to support those higher levels which directly support the user.

7. Assembler

The assembler virtual machine uses a mnemonic assembler as its machine language. This enables a user to program in a form which, although maintaining a one to one correspondence with the conventional machine language, is not as painful to use. Most assembly languages are merely symbolic translations of the conventional machine level language on a one to

one basis. Thankfully, the generation of assembly language programs is a dying art.

8. Compiler

At a higher level, a HLL may be translated to a language at a lower level. This translation is normally into an assembly language or a conventional machine language. Such a translator is termed a compiler.

9. Variants

It is most important to realise that the virtual machine hierarchy is not strictly sequential. Translation or interpretation may take place on any level. Many HLLs are implemented by various combinations of compilation and interpretation. This aspect shall be explored more fully in Chapter 4.

CHAPTER 3

THE RELATIONSHIP BETWEEN HARDWARE AND SOFTWARE

1. Historical

Once upon a time, the differentiation between hardware and software was simple - the machine, along with all its peripherals, contributed to the hardware and the programming representation of an algorithm was the software. In the very early days, an operating system description consisted of a wiring diagram!

Later developments, which consisted of the logical combination and interconnection of logic gates to produce some larger structure, such as a register or a memory cell, to fulfill a given design task, constituted the "freezing" of an algorithm in hardware. Such a design has been termed "frozen software". Even at this stage of development, there was no doubt that when a programmer coded an ADD instruction, the electronic circuitry performed the tasks necessary to achieve the desired result. But when he required a multiply operation, the programmer was required to code it explicitly himself.

Further development permitted the implementation of often used "instructions", such as multiplication, division, floating point operations, procedure calls etc., to be provided in the hardware. This was considered a major advancement.

2. Growth in "Primitives"

Such functions are now provided by a computer manufacturer as a matter of routine, and now some very powerful "primitive" operations are available. For example, a single instruction to save the whole register set of the CPU at a given start address of memory is available on some machines. Additionally, such operations as block memory to memory moves (without ALU register change) and string manipulation primitives are sometimes available.

Such primitives are usually implemented via a microprogrammed arrangement of the very basic executable cycles, such as fetch and store. Even the simple ADD instruction is now often implemented as a microprogram of these cycles.

3. Other Developments

One of the major reasons for the growth of the microcomputer industry has been the ability to fix certain software into hardware. In a seemingly trivial, but commercially very important sense, this has been exemplified by the growth of the "intelligent TV games" industry. Another widespread innovation is the placement of HLL interpreters into hardware. This situation is rapidly becoming the norm for variants of the BASIC language, provided in a read only memory (ROM) along with the microcomputer itself. Many operating systems are also ROM based.

In other areas of computing, many algorithms have been implemented in hardware. Those successfully achieved to date include such large products as translators, linkers, loaders, generalised sort routines, paging algorithms and data base managers.

4. Equivalence

Taking the above factors into consideration, many commentators have stated that the boundary between hardware and software is arbitrary, blurred and constantly changing. However, this misses the essential point. When the traditional "primitive" instructions are now implemented via software techniques, and translators can be found in hardware, the dividing line has ceased to exist.

Hardware and software are logically equivalent.

What is today's software is tomorrow's hardware, and vice versa. What may be implemented by one person in hardware may be implemented, exactly equivalently, by another in software. It is no longer possible to rationally distinguish between the two.

Once a given problem has been found to have a correct solution, it is not important how the solution is implemented. Costs alone determine whether a software file or a silicon chip is created. As the relative costs of hardware and software are in a state of flux, the preferred solution may change over time.

Finally, from the point of the hierarchy of virtual machines, the method of implementation of any particular level is of no significance. The user needs a thorough knowledge only of the virtual machine at the level at which he is working, and need not be aware of the messy details of the underlying levels. It is also of interest that the boundaries between levels are in a state of flux and are becoming increasingly blurred.

5. Current Trends

Much of the literature available regarding the continuing crisis in computer utilisation tends to note the rapid fall in hardware prices, and the comparatively small increase in programmer productivity, and conclude that

a. lack of software development improvements is the cause of any slowing in the rate of expansion of computers into society in general, and

b. hardware should be improved still further in order to simplify the massive software task.

These conclusions are basically agreed, but two aspects need to be considered in the context of this paper.

5.1. Architecture

Firstly, most computers implement the software using them in a horrendously inefficient manner. The development of HLLs has resulted in tools which are substantially easier to use, enable greater programmer productivity and give rise to more reliable programs. Block structured HLLs like Algol, Pascal and C are currently used, and more and better languages are being developed. However, the computer architectures on which programs written in these languages are to be run have changed little, in spirit, since the 1950s. Thus the functional behaviour of the programs reflects more closely the machine language of the computer than the language of the programmer.

Much research is currently being undertaken in an effort to develop instruction sets which more closely reflect the actual primitives of the problem solver, rather than those of the electronic circuitry. If achieved, this will simplify compiler generation, and improve both the size and speed of compiled programs. This research has shown, for example, that it may be possible, by microprogramming, to redefine the architecture of a computer to more closely reflect the particular environment of its use.

5.2. Human Aspirations

Secondly, any great improvement in speed is not going to solve any perceived crisis. Typical timesharing systems have, over the past ten years, increased in speed by a factor of five, increased in primary storage by a factor of ten and increased in secondary storage by a factor of twenty. Yet with the same number of users, the response time has become worse. Why?

As users have become familiar with the attributes of computer systems, they have used them more widely. The users

aspirations have grown even faster than the technology. No longer are they prepared to accept poor service in return for the privilege of using the machine. A typical user now expects the machine to do much more for him than he did ten years ago. And this growth in human aspiration will continue, with the implication that as more sophisticated machinery becomes available, there will be a (possibly faster) growth in the sophistication of the uses to which it is put.

CHAPTER 4

LEVELS OF HARDWARE EXECUTION

1. Traditional Situation

Given any HLL, there exists at least two processes undertaken before a program written in that HLL can produce results. The first is a translation of the program representation into some "image language". The program provided in this image language is then interpreted. Traditionally, the translation has been undertaken by software (compilers), and the interpretation by hardware (electronic circuits). With the increasing use of microcode, even this traditional model of execution is now rarely undertaken in the traditional manner.

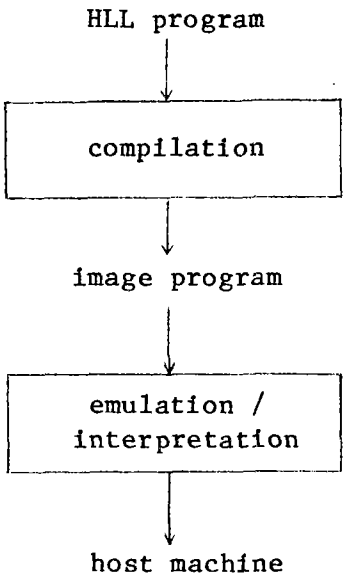

figure 1.
Compilation and interpretation.

With the decreasing cost, and thus importance, of the CPU (in relation to the total system), and the increase in capability of HLLs, particularly those utilising block structure and abstract data types to the full, the structure of the traditional machine, with its close affinity to electronic design, is being increasingly seen to be based on concepts which are no longer in the user interest. This structure is, in itself, adding to the problems of the computing community, rather than assisting in alleviating them. Can this model be changed into something which will provide more assistance to the computing community as a whole, and at a reasonable cost?

2. Models of Execution

In an investigation of this question, four very broad models shall be presented, and the benefits and costs of each indicated.

2.1. Direct Execution of HLL

In this model, the HLL is directly interpreted or emulated by the host computer. This implies that the HLL is the machine language of the host machine. There is no translation phase at all. This implies no compiler, no linker, no loader, no editor, no debugger - just the HLL.

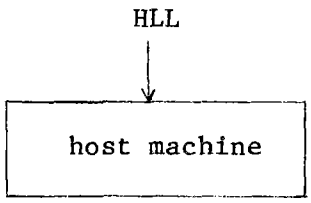

figure 2.
Direct execution.

2.1.1. Advantages

1. There is no compilation stage at all. This is a very large time saving in a developmental environment.

2. The user will have a interactive program execution. Any error can be rectified as it occurs. There is no need for the messy re-edit, re-compile, re-execute sequence. The HLL used should have constructs to facilitate debugging, editing, display and execution control

during interactive operation. For example, since the HLL is the machine language, debugging becomes much simpler. Access to the basic structures of the machine (registers or stacks or ROMs) will be feasible in the HLL, it will be possible to single-step through the program (remember one step is one HLL instruction), and so the user will have all the features of a good current monitor, but utilising a HLL.

3. Whilst entering a program representation (the normal editing phase), syntax checking will be carried out. This is both time saving and of educational value.

4. Type and range checks will automatically be carried out on every occurrence of every statement.

2.1.2. Disadvantages

1. The interpretation is complex and potentially slow. The major problem is that each symbol must be rediscovered on every occurrence of its use. The computer needs to be designed in a manner to minimise this problem.

2. The program representation (the source program) is comparatively large.

3. Obviously, the hardware will be more complex and thus more expensive than an indirect execution computer. The savings in software tend to counterbalance this.

Note that

4. The host machine is restricted to a particular HLL.

is not valid, as every computer has only one machine language. Other languages can be implemented on such a machine by writing (relatively simple) translators (in the machine language, the HLL, of course).

2.2. Language Oriented Translation

This model involves a very simple compilation stage, which translates the HLL into an "image language", called a directly executed language (DEL). Statements in the DEL correspond on a one to one basis with statements in the HLL. The DEL is then directly interpreted by the machine (that is, the DEL is the machine language). Instructions in the DEL correspond closely to the primitives of the HLL, and bear little relationship to conventional machine language. Note that the virtual machine which interprets the DEL is of the type explained in the previous section.

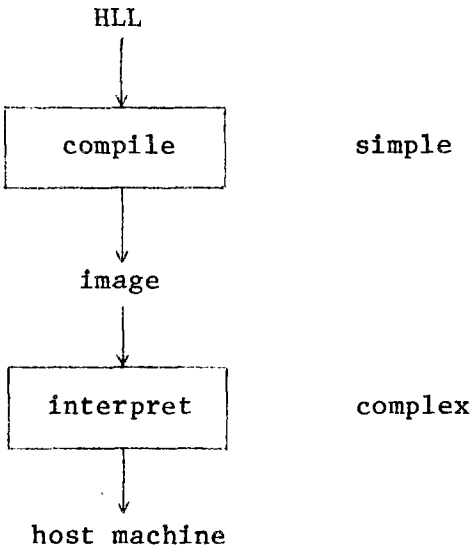

figure 3.
Use of language oriented image.

2.2.1. Advantages

1. As the features of compression of compilers are now available, program representation is much more concise.

2. Although there is now a compile time, it is relatively short.

3. The execution time should be reduced.

2.2.2. Disadvantages

1. Loses most of the advantages of the previous model, as the compilation has inserted a degree of displacement from the hardware.

2.3. Host Oriented Translation

The HLL is compiled into a language that closely reflects the architecture of the underlying machine, and then this representation is interpreted. This entails a relatively complex compile stage (the HLL primitives need to be converted into a representation of the machine primitives), and a simpler, but not necessarily simple, interpret stage. This situation describes the traditional approach. It

has advantages in generality, but the translation stage is inherently flawed in that it is based on the lowest common denominator of the technology of the underlying machines.

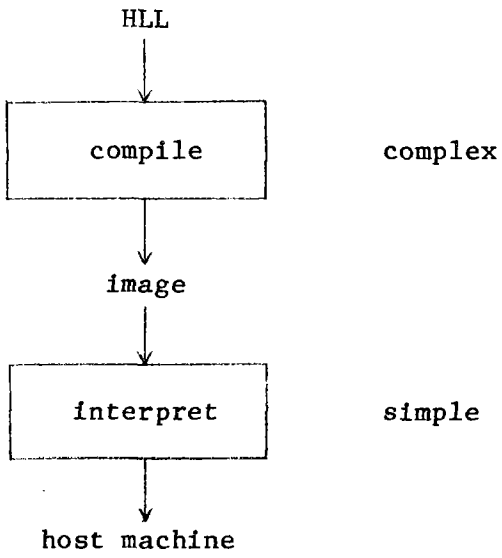

figure 4.
Use of host oriented image.

2.3.1. Advantages

1. General in nature. One program representation can be used on many different machines.

2.3.2. Disadvantages

1. Flawed in its basic design concept.

2. Program size is much too large, as it needs to be able to cater for many instances of a general problem. Thus interpretation must cover many often unused features.

2.4. Direct Compilation

In this situation, the HLL is compiled directly into a program consisting of the microcode of the machine. This program is then directly executed by the machine. No overt interpretation is involved. The microprograms produced tend to very large, and the compilation process is immense.

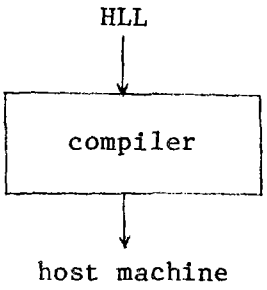

figure 5.
Direct compilation.

2.4.1. Advantages

1. Execution time is minimised.

2.4.2. Disadvantages

1. Compilation is very complex.

2. Microprograms are very large. Algorithms are not very well expressed in microcode.

2.5. Summary

In actual practice, the divisions between these models can not be considered clear cut. In particular, the division between models two and three depends on the relative complexity of the compile and interpret stages. Some recent development has centred on this area with, for example, the compilation of the HLL Pascal into an intermediate p-code, and the software interpretation of that code.

The models are summarised in figure 6.

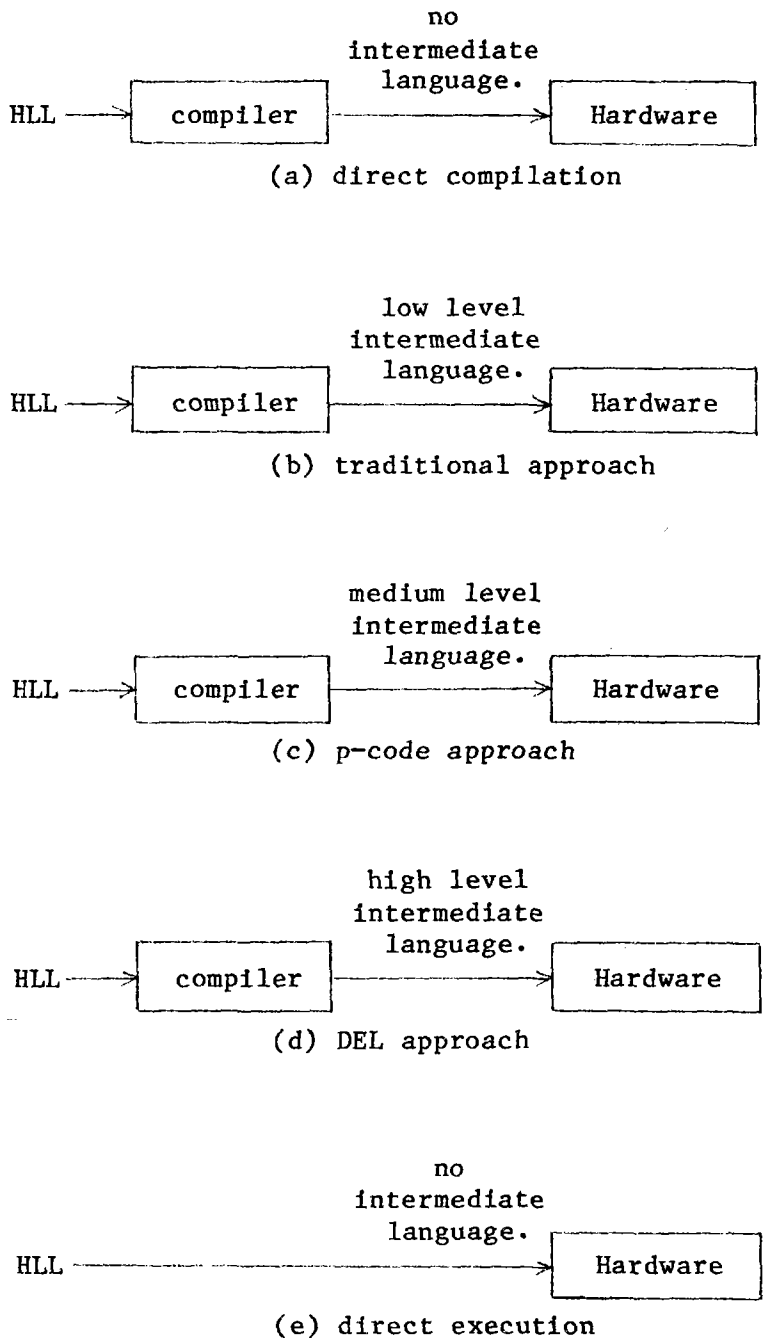

figure 6.
Summary of approaches to HLL execution.

CHAPTER 5

SOME EXAMPLES

Most presently operating computer systems use the traditional approach to HLL execution, as set out in the previous chapter. Increasingly, however, more aspects of the language translation and execution process is being undertaken by "hardware". This chapter will give a very brief look at some of the innovations in this area. The coverage is not intended to be either historical or exhaustive.

1. SYMBOL

The SYMBOL computer system [6] was designed and constructed around 1970. It has a high level Algol like machine language, called SYMBOL, and accepts statements in this language directly. To the user, there is no compilation stage. The system is multi-user, with access by terminals, and is a multi processor system. This machine has an incredibly fast execution rate of the order of 80000 SYMBOL statements per minute (using late 1960's technology), and was the first to demonstrate the feasibility of a computer having a HLL as its machine language. Internally, there is a translation from SYMBOL into a DEL, so this machine is an example of the DEL approach of the previous chapter.

2. The Microengine

The late 1970's saw the dramatic rise in the use and development of microprocessors. One company has produced a series of hardware chips that directly interprets p-code. Initially, this code was the output of a number of Pascal HLL compilers, and was interpreted in software. The realisation of this interpretive phase in hardware sufficiently cheaply to become available to the consumer market constitutes a major advance. Note that there is still a compilation stage with this chip. Execution is much faster than with a software interpreter.

Subsequently, compilers have been produced to translate a variety of HLLs into p-code, with FORTRAN, BASIC and FORTH being examples of languages available. Recently, a similar compiler has been written for the United States of America Department of Defence sponsored language Ada.

3. LISP

Recently a single chip LISP executing computer (Scheme-79) has been produced [11]. This accepts LISP statements and, to the user, directly executes these. In fact, the chip has a machine language called S-code. This chip is a hardware representation of the p-code approach of the previous chapter. There is a translation from LISP to S-code, and an interpretation of the S-code. The implementation of the chip was a research effort, and the final result is not as promising, in a commercial sense, as the researchers had hoped. The performance of the chip is similar to that of other computers doing similar tasks.

Other LISP machines, of varying hardware implementation, are available. LISP is widely used in the artificial intelligence and computer aided design fields.

4. The Intel iAPX 432

The Intel company have recently announced [9] a new 32 bit microprocessor "chip" (actually a collection of chips). One of the design goals of this chip was that its execution should try to provide a realisation of HLL constructs, and in many ways this has been achieved. Whilst there is a machine language in the chip at a basic level, its instruction set attempts to satisfy the needs of a HLL rather than the needs of the machine. Because the design was begun six years ago, there is a curious mix of trade offs between the design goals and traditional architectures.

The chip is claimed to be able to execute compiled versions of Ada in an extremely efficient manner, and Ada is likely to be the "systems language" for systems based on this chip. Some poorly informed reports have claimed that Ada is the machine language of this chip, but this is not the case. There are many other aspects of this chip of interest (H-MOS technology, tricks with ROMs, operating systems, concurrency), but these are not relevant to this paper. Based on the available reports, this chip would appear to be a major step toward direct execution of HLLs.

5. Lilith

Lilith [14] is a personal (single user) microcomputer developed with the aim of simplifying the software engineering problem. It uses as its only HLL the Modula-2 language. Modula-2 is an extension of the Pascal language, designed with the aims of permitting concurrent operation, and ease of use in a systems environment (along with all the Pascal aims). A source program in Modula-2 is compiled into an intermediate level code called M-code.

The Lilith computer directly executes the M-codes. The

computer and M-codes were both designed with the aim of making the generation of M-code from Modula-2 as simple as possible, which meant retaining as much of the HLL structure as possible. In an effort to measure the success of this goal, a Modula-2 program was run both on Lilith and on a conventional machine (after compilation with a conventional compiler). The generated code for Lilith was four times shorter than the code for the conventional computer, leading to the obvious conclusion that the inappropriate structure of traditional computer instruction sets is a major problem for the computing community. This traditional structure is still basically followed by the modern marvel, the microprocessor.

6. BASIC in ROM

This pseudo direct execution approach has gained wide popularity. A BASIC interpreter is provided in a read only memory with the majority of microcomputer sales. Most dialects of BASIC are line by line interpreters, and perform a lexical analysis as each line is typed. This is immediately followed by the creation of a greatly abbreviated intermediate code that is easier to store and interpret. The interpreter then does syntax processing, code generation and execution when the RUN command is given. Reinterpretation takes place on each pass through a program loop. Because of this, these implementations suffer from slow execution.

Note, however, that some of the dialects of BASIC are becoming very powerful, with elaborate graphics and file manipulation attributes. To the user, these seem to be direct execution machines.

7. Microdare

Microdare [8] is a real time HLL system utilising the BASIC approach to direct execution. There is no external software (compiler, linker, editor) at all. The system is totally interactive, and is proving very useful in laboratory automation, control processing, simulation and other fields where the need for interactive change is high, and where compilation would require a significant portion of computer time.

Microdare is, in effect, a "sophisticated" extension to an advanced dialect of BASIC. Thus all the features of the underlying BASIC dialect are available, plus the real time aspects provided by the extension. Some optimisation has been done, with the claimed result that Microdare executes faster than compiled FORTRAN.

8. Jovial

There are reports that a computer which directly executes the US Army language Jovial is in the testing stage at

the University of Maryland.

9. History

An overview of the development can be found in chapter 3 of [3].

CHAPTER 6

THE ARCHITECTURE OF A DIRECT EXECUTION COMPUTER

The development of computers which directly execute a HLL is one of the very promising areas in computing research. This chapter will describe the architecture of such a computer. More detail can be found in [1], [2], [3], [4], [5], [7] and [12].

1. A Single Cycle

As is usually the case, the representation of the program to be executed is stored in the main memory of the computer. In any given cycle, a single token (word) is fetched, and then directly executed. This cycle continues until the program ends.

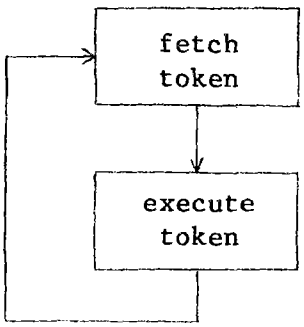

figure 7.

As has been argued in previous chapters, the architecture of a computer to undertake this task must closely mirror the available constructs in HLLs, rather than the requirements of the traditional architecture. For this reason, there are three processors - the lexical processor, the control processor and the data processor. Conceptually, there are also different memory areas for programs and for data, and associative stores for two of the processors. See figure 8.

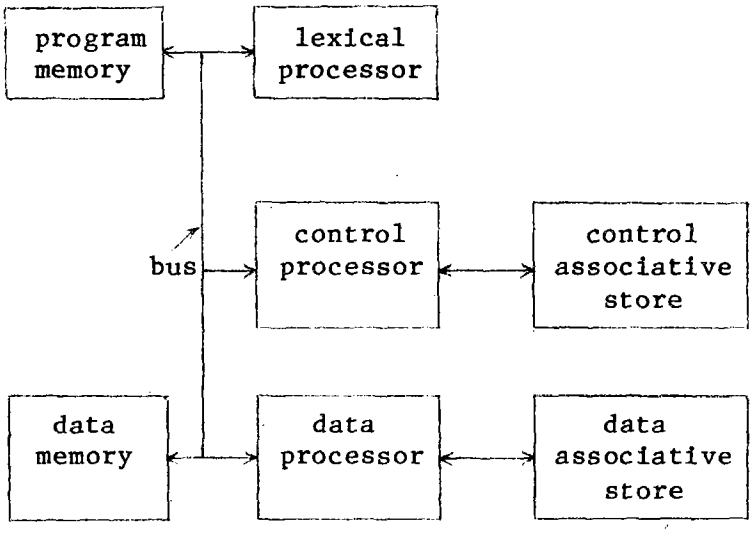

figure 8.
Organisation of a direct execution computer.

and their interrelationship is:-
The lexical processor fetches tokens (at this level, these are source program characters) from the program memory, and organises these into other tokens (comprising operators, reserved words, data types, names and numbers), and passes them on.
The control processor accepts tokens of the control category, and executes these.
The data processor accepts tokens of the data category, and executes these.

The control and data processors and their associative stores are conceptually referred to as the language processor. A clearer statement of one cycle of execution of the direct execution computer can now be made. The lexical processor fetches the characters from the program memory, assembles these into tokens, and delivers these to the language processor for execution. Tokens are assembled and executed in this manner until the token indicating the program end is assembled.

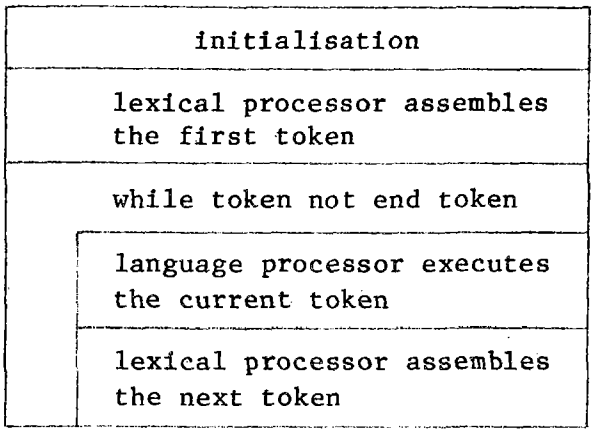

figure 9.
Execution sequence in a direct execution computer.

As can be seen, there is a degree of parallelism possible, in that the "execute current token" and "assemble next token" processes are undertaken by different processors.

2. The lexical processor

The lexical processor accepts the character strings of the representation of the program in the HLL, checks them for legality and assembles them into tokens. The action of the lexical processor could be compared to that of the instruction decoding sequence of the CPU of a conventional machine. Just as the CPU fetches one, or more, words to produce a variable length instruction, the lexical processor fetches one or more characters to form a token. The lexical process is inherently complex, and so is deserving of a dedicated processor.

The lexical processor needs to interface both with the language processor and with the program memory. To facilitate this, there are two registers. The first is the program memory location register (PML) which contains a pointer to the current character position of the high level program in memory. In effect, the PML controls the point of program execution, as the program counter does in conventional machines. The second is the token register, which contains both the token itself and a type indicator. The type information will indicate whether the token is a control token, or the type of the data.

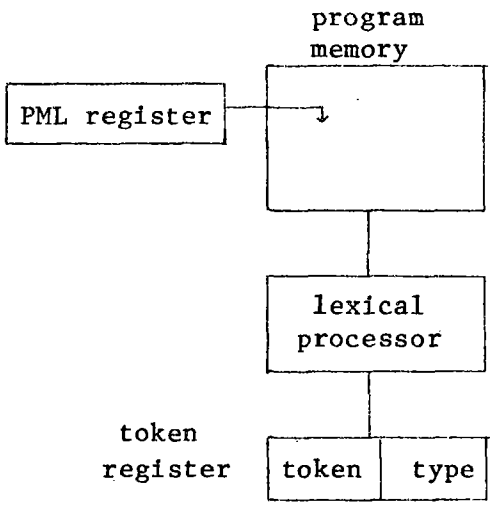

figure 10.
Operation of the lexical processor.

Note that the control processor must also have access to the PML register, as the pointer will need to be updated as a result of program determined values (for example, as a result of the test of a boolean condition in a control structure).

The lexical processor and the language processor operate in a parallel, but synchronised, manner. In particular, this means that there is no slowing of execution due to the necessity for repeated lexical processing during the execution of a program loop.

3. The control processor

The task of the control processor is to organise and execute the control structures of the program, such as while.....do and if.....then constructs. The control processor recognises the control keywords, and manipulates the PML register so that the lexical processor will interpret the appropriate sections of the program memory.

The control processor communicates with the program memory via the PML register, with the lexical processor via the token register, and with the data processor via the result register. This result register is necessary as the control processor needs to know the results of some calculations and tests to determine the setting for the PML

register (for example, in the case of an if.....then.....else statement). Additionally, the control processor uses a control associative store (CAS) in which to store the needed details of control constructs. It can thus directly interpret high level language control constructs.

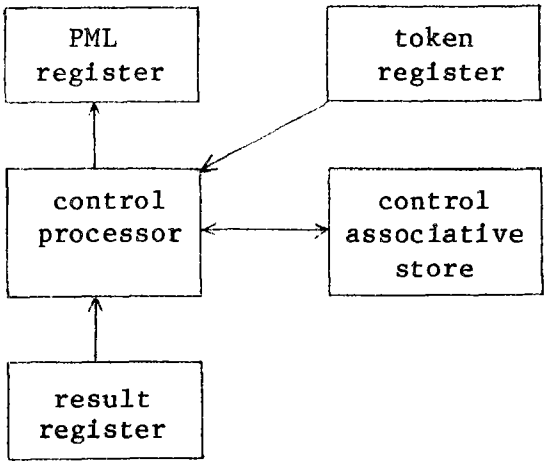

figure 11.
Organisation of the control processor.

As an example, consider the case of the following IF statement.

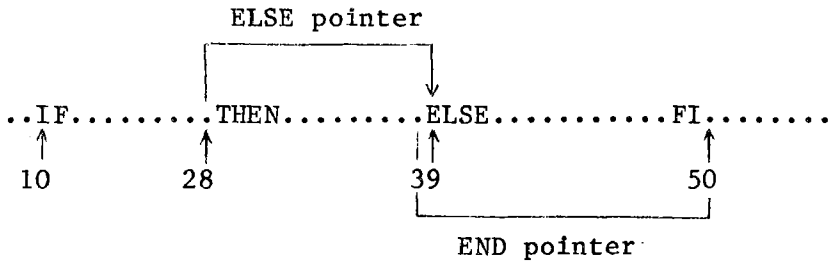

figure 12.
An IF control statement.

The control information is coded and formulated into a control descriptor, which contains all the information needed to control the program interpretation flow for the particular control structure. In the case of the above IF construct, the control descriptor will be as shown in figure 13.

control name	control type	end pointer	else exists	else pointer
10	IF	50	yes	39

figure 13.
Control descriptor for the IF construct.

In this example, memory location 10 (the character I) is the internal name for this control statement. Memory location 39 (the character E) serves as the else pointer. The END pointer points to the character immediately following the end of the IF statement. The control descriptor is stored in the CAS, the PML register is set and, depending upon the value returned in the result register, the appropriate clause of the IF can be executed.

Other control structures have different control descriptors. Each entry in the CAS describes one control statement in the source program. Each descriptor contains several fields, each containing information needed to execute the control statement described. This information can expedite the repeated execution of statements in a repeated loop, by restricting the need for continued lexical interpretation of the control keywords.

4. The data processor

The data processor needs to be able to recognise data declarations (for example, integer, real, character, string, and user defined structures based on these predefined primitive types) and interpret data statements (for example, assignments, arithmetic statements and text-editing statements). Some data statements will be embedded in control statements, in order to enable the control processor to determine the flow of execution. The organisation of the data processor is shown in figure 14.

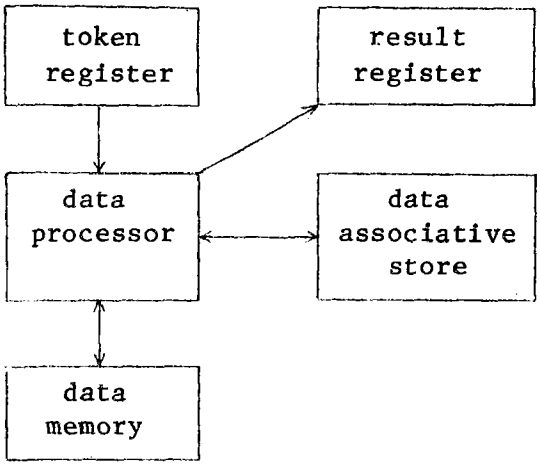

figure 14.
Organisation of the data processor.

The data processor communicates with the data memory directly, with the lexical processor via the token register, and with the control processor via the result register. Additionally, the data processor uses a data associative store (DAS) in which to store the needed details of data constructs. It recognises data declaration keywords, creates data descriptors, and stores these descriptors in the DAS. It can thus directly interpret high level language data constructs.

The data descriptors are formulated in much the same way as the control descriptors, and they serve a similar purpose. Each entry in the DAS contains a descriptor for one program variable. As in the case of control descriptors, each entry contains fields to describe the variable adequately. In the case of a DAS entry, there must be at least fields for the variable name, the variable type, and a pointer to an address in the data memory at which the value of the variable is to be held. In the case of user defined data structures, this information will also be needed for each of the subfields of the variable.

5. The associative stores.

There are two further points of interest with respect to the associative stores. Firstly, the CAS and the DAS entries are created during program execution, in response to tokens found and interpreted by the lexical processor. The associative stores do not store any source program or data. They store information needed to control that source program and data.

Secondly, the time taken to search the CAS or DAS for an entry and, if not present, to insert a new one, is very short. This process can be considered similar to that of using an associative store, in an operating system paging environment, to enable rapid access to memory locations. The time taken is roughly equivalent to that of a register to register operation (that is, faster even than a basic store instruction in conventional architectures).

6. Overall organisation.

The overall organisation is shown in figure 15. Note that the program memory is directly accessible by all processors.

After initialisation, the lexical processor generates a new token. The control processor inspects the token register and, if the token is a data token (found by inspection of the type field), directs the data processor to execute it. Otherwise the control processor executes the token. As the data or control processors execute this token, the lexical processor is preparing the next token. This cycle continues until the end of program token is accessed.

The program and data are entered by the user at a terminal, permitting totally interactive operation. Syntax is checked on entry, the program is executed (as far as possible) whilst it is being entered, and so debugging of syntax and logic errors should be much simplified over the current pattern. Direct execution computers have characteristics that make them ideal for developmental and educational facilities.

The organisation of the machine reflects closely the organisation of the HLL itself. The lexical processor recognises legal characters, keywords and punctuation, and thus reflects the lexicality of the HLL. The control processor interprets the control statements, and sequences the order of execution of program statements, and thus reflects the structure of the HLL. The data processor references symbolic names and executes data operations, and so reflects the data constructs of the HLL. Thus the hardware executes the HLL in exactly the way that the designer of the HLL visualised.

Additionally, because the lexical processor operates in a parallel but synchronised manner with the control and data processors, the need for repeated lexical analysis does not impede the speed of the machine. Further, the use of control descriptors in the CAS negates the need for repetitive syntactical processing of the statements in a program loop. Thus the speed of program execution is not affected by the lexical task.

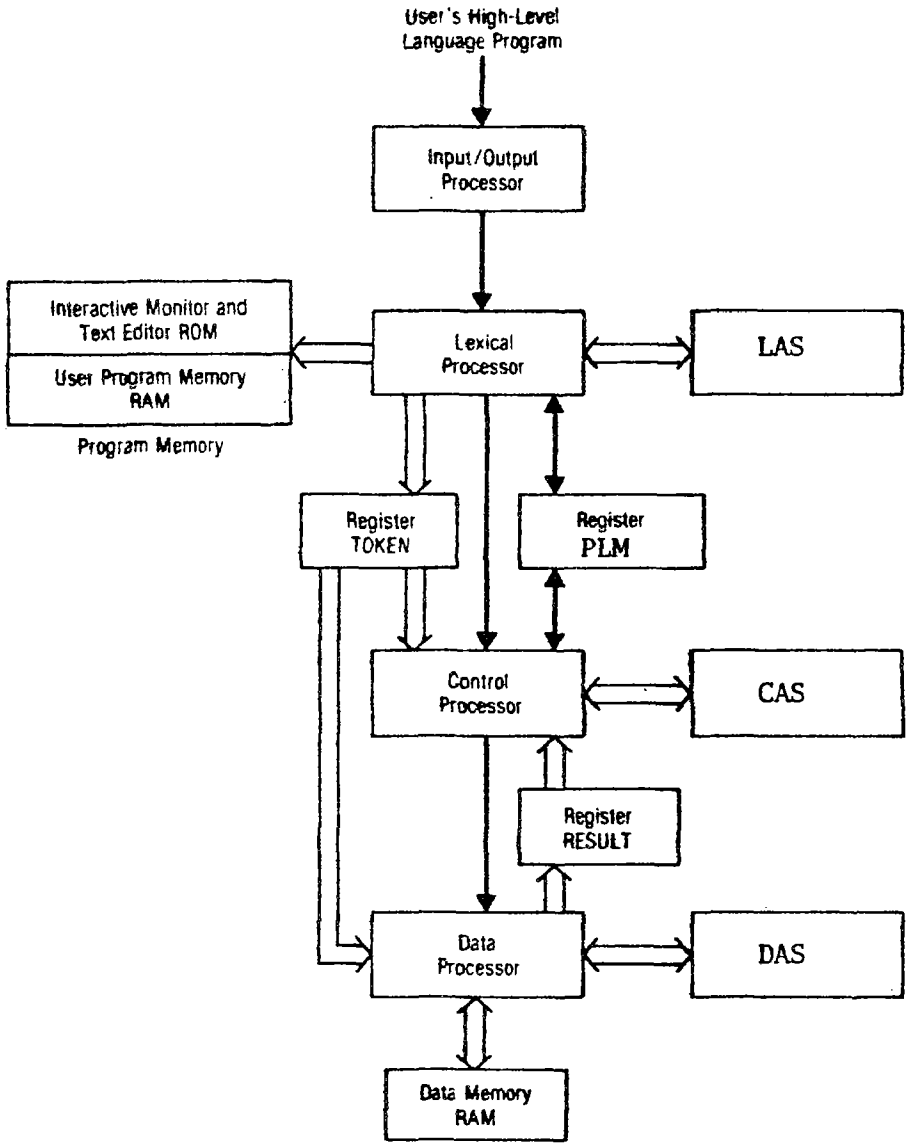

figure 15.
Organisation of a direct execution computer.

CHAPTER 7

CONCLUSION

The relationship between HLLs and the architecture of computers has been heavily biased in favour of the hardware. This is understandable as, until very recently, the cost of the computer hardware was of prime concern, and programmer productivity, and hence software costs, were relatively unimportant.

The rapid decrease in the real cost of selected items of hardware, coupled with the equally rapid rise in the capabilities of that hardware, has forced a growing realisation that the software area is now the most costly portion of the life cycle of a computing system. Unfortunately, there appears to be a software versus hardware ideology conflict developing, instead of the joint approach which is necessary to produce any real gains.

As has been shown in this paper, development of hardware to provide a friendlier face to HLLs is being undertaken, and there are indications that some future computer architectures will enable HLLs to perform much better than at present. One of the areas showing promise is the development of HLL direct execution computers, and computers of this type should be of particular benefit in developmental and educational environments.

BIBLIOGRAPHY

1. CHU, Y. An LSI modular direct execution computer organisation. Computer 11, 7 (July 1978), 69-86.

2. CHU, Y. Architecture of a hardware data interpreter. IEEE Transactions on Computers C-28, 2 (February 1979), 101-108.

3. CHU, Y. (ed) High-Level Language Computer Architecture. Academic Press, New York, 1975.

4. CHU, Y., AND ABRAMS, M. Programming languages and direct-execution computer architecture. Computer 14, 7 (July 1981), 22-40.

5. CHU, Y., AND CANNON, E.R. Interactive high-level language direct-execution microprocessor systems. IEEE Transactions on Software Engineering SE-2, 2 (June 1976), 126-134.

6. DITZEL, D.R. Reflections on the high-level language Symbol computer system. Computer 14, 7 (July 1981), 55-67.

7. FLYNN, M.J. Directions and issues in architecture and language. Computer 13, 10 (October 1980), 5-22.

8. KORN, G.A. Microdare : A fast, direct-executing high-level language system for small computers. Computer 12, 10 (October 1979), 61-71.

9. RATTNER, J., AND LATTIN, W.W. Ada determines architecture of 32-bit microprocessor. Electronics 4, 54 (February 24, 1981), 119-126.

10. RAUSCHER, T.G., AND AGRAWALA, A.K. Dynamic problem-oriented redefinition of computer architecture via microprogramming. IEEE Transactions on Computers C-27, 11 (November 1978), 1006-1014.

11. SUSSMAN, G.J., HOLLOWAY, J., STEELE, G.L., AND BELL, A. Scheme-79 - Lisp on a Chip. Computer 14, 7 (July 1981), 10-21.

12. TANENBAUM, A.S. Implications of structured programming for machine architecture. CACM 21, 3 ,237-246.

13. TANENBAUM, A.S. Structured Computer Organisation. Prentice-Hall, Englewood Cliffs, 1976.

14. WIRTH, N. The personal computer LILITH. Institute for Information (ETH), paper 40 (April 1981).

15. WULF, W.A. Compilers and computer architecture. Computer 14, 7 (July 1981), 41-47.

16. YAMAMOTO, M. A survey of high-level language machines in Japan. Computer 14, 7 (July 1981), 68-78.