INFORMATION PROCESSING 77, B. GILCHRIST, EDITOR

DIRECT-EXECUTION COMPUTER ARCHITECTURE

YAOHAN CHU
University of Maryland
College Park, Maryland

The Direct-Execution Architecture is a language-directed computer architecture. It can accept a high-level-language program and directly executes it without compilation, assembly, linkage editing or loading. It offers a means to eliminate compilers, loaders etc. and attacks the problem of mounting software cost. In addition, the advent of microprocessors has demonstrated that highly complex digital hardware can be built reliably and inexpensively. Using this hardware to implement the Direct-Execution Architecture redistributes apportionment of costs between the hardware and software. The paper surveys the Direct-Execution Architecture, presents the relationship between language and architecture, and explains how a Direct-Execution system works. It also brings up the use of Direct-Execution for a highly interactive program writing, debugging, execution system. With this system, program writing could proceed like English composition. This paper then discusses the issue of a single high-level machine language, and the potential role of the interpreters. Finally, it attempts to fortell what could happen to the Direct-Execution Architecture in the next five to 10 years.

1. INTRODUCTION

W.M. McKeeman [14] pointed out at a Joint Computer Conference that it was an accident that the digital computer was organized like a desk calculator. As a result, the digital computer of today requires a large amount of software. The software consumes more memory, needs more time to debug, takes more programmers, spends more overhead computer time, and becomes less reliable than necessary. The progress of computer utilization and application is now limited by and tied to the development of software. Yet, the progress of software development is handicapped by the desk-calculator-like organization.

Barry W. Boehm, [3] stated in the Symposium on the High Cost of Software that the annual expenditure of the U.S. Air Force on software in the fiscal year 1972 was somewhere between 1 and 1.5 billion dollars, which amounts to about 4 to 5 percent of the total Air Force budget. In comparison, the cost of the hardware was something between 300 and 400 million dollars. He also showed that the cost of software exceeded the cost of hardware during 1968 and, if the present trend persists, the cost of software would become 90% of the total military hardware and software cost by 1985.

The advent of microprocessors in 1971 demonstrated that semi-conductor device technology has reached the point where highly complex hardware can be built reliably and inexpensively. It could be rewarding if the semiconductor technology could be used to solve the problem of software cost. This could be feasible if the computer architects would re-examine the desk-calculator-like organization and create new architecture that would help solve the problem of software cost.

2. DIRECT-EXECUTION ARCHITECTURE

The Direct-Execution Architecture is a language-directed computer architecture. [14] It can directly accept a high-level language program and directly execute it without any code conversation. There is no assembly language, no relocatable code, and no absolute code. The high-level programming language is the machine language.

The advent of high-level programming languages during the middle 1950's allowed computer programming to be greatly eased and non-professional programmers (or users) could then readily use the computer. However, in the process of providing high-level languages to the users and simplifying computer operations to the operator, multiple layers of software have been created.

One of the motivations for using the Direct-Execution Architecture is to remove the multiple layers of software and thus reduce the mounting software cost. With a Direct-Execution Architecture, the programmer can follow the program execution directly as the computer hardware executes each symbol, each clause, and each statement of the high-level-language program. There is no need for compilers, assemblers, linkage editors, or loaders. The entire operation of the computer could be controlled by an adequately designed high-level language which is an embodiment of the language-directed-computer architecture.

In contrast to the Direct-Execution Architecture, an indirect execution architecture first translates this high-level language program into an intermediate language and then executes this intermediate language code. It is a compiler-directed process where the compiler can be hardware or software.

3. A SURVEY

The direct-execution architecture has a rather limited history. Anderson [1] proposed a computer organization that could directly execute an Algol-60 program. This proposal probably was the first direct-execution architecture ever reported. This architecture was basically an extension of Burroughs B5500 architecture.

Mullery et al. [16] and Mullery [15] designed a problem-oriented symbol processor called ADAM and concluded that a high-level language could be implemented with a reasonable amount of hardware. A high-level language was designed and a machine organization was proposed to implement this language directly. The most significant feature of the ADAM machine was the way in which the variable-length data were structured. Special symbols were actually placed within the data for describing the data structure. This concept was extended and actually implemented in hardware in the Symbol Computer System.[21]

A number of APL interpretive processors have been proposed or implemented in software or firmware. A recent implementation is the MCM/70 (Micro Computer Machines). Another recent implementation, presumably

in software, is the desk-top IBM 5100 which allows the use of either APL or Basic language. It should be pointed out that an APL statement is just scanned from the left to the right end and is then interpreted from the right to the left end because of APL's right-to-left operator precedence; this mode of interpretation differs from that of a direct-execution architecture which attempts to interpret when it receives each token for immediate interaction.

Since 1970, the author and his students at the University of Maryland have been actively engaged in the research of direct-execution architecture. They have also explored the possibility of direct-execution on a Burroughs B1700 system (Chu, Yeh, Cannon [7]). It should be noted that there are two issues in the direct-execution architecture: the architecture itself and the high-level language or languages. The above work has been largely in the area of architecture. (Chu [5], Robinet [17], Yeh [18]).

A more lengthy survey on direct-execution architecture is available in a recent survey by Carlson [4] on high-level language computer architecture.

4. LANGUAGE AND ARCHITECTURE

To each programming language, there is associated an ideal computer architecture which can directly execute the program written in this language. This ideal architecture images the constructs and the primitives of the programming language. It is a virtual architecture, because it may not be possible to fully implement it by a real architecture. Thus, when a programming language is being designed, the designer perhaps unknowingly creates at the same time a virtual architecture.

To each computer architecture, there is associated a programming language which one uses to communicate directly with the computer architecture. The design of a computer architecture implies the design of a programming language; as an example, the instruction set is essentially the programming language of a conventional computer architecture.

A programming language can be spoken of as high-level or low-level. It is low-level if the details about the problem solution have to be described in detail. It is high-level if the problem solution can be described at a high-level of abstraction. Thus, the programming language is high-level or low-level relative to the degree of abstraction in describing the problem solution.

Likewise, a computer architecture can be said to be high-level or low-level. It is a high-level architecture if the architecture implements closely the constructs of a high-level programming language. Otherwise, it is a low-level architecture. It is of course possible to execute a high-level programming language program by a low-level architecture by using a compiler which bridges over the structural differences. However, a high-level architecture can directly execute a high-level programming language program without the use of a compiler.

5. HOW DOES A DIRECT-EXECUTION SYSTEM WORK?

Let a high-level language program be stored in a memory together with its data area as shown in fig. 1. The hardware lexical processor scans the program string, assembles one or more characters into a symbol (or token), and communicates whether it is an operand (an identifier or a number) or an operator (an operator, a delimiter, etc.) to the language processor. The hardware language processor fetches the next symbol from the lexical processor, and executes it accordingly as shown in fig. 2. The language processor interprets each statement of the program until the last statement is reached. It is noted that the lexical and language processors can be two hardware processors. They operate in a parallel but synchronized manner. Because of this parallel operation, there can be no slow down in the interpretation of a loop due to the repeated lexical processing.

For simplicity, consider the high-level language where each statement begins with a keyword. There are the data keywords such as STACK, TABLE, and QUEUE for the data statements, and the control keywords such as IF, CASE, LOOP, and CALL for the control statements.

For interpreting data statements, the language processor has the data interpreter and the data associative memory as shown in fig. 3. The data interpreter has an internal structure which can directly interpret the data storage constructs of the high-level language. It recognizes the storage keywords, interprets the data statement, and then stores in the data associative memory the data description which includes the name, the type, the length, the size and the location pointer. This pointer points to where in the random access memory the actual data is stored. As an example, fig. 4(a) shows a stack data statement with a two level structure. Each stack element has two fields, NAME of 6 characters and VALUE of 6 characters. This stack is located in the random access memory as shown in fig. 4(b). The stack description is formed as three words and stored in the data associative memory by the data interpreter, as shown in fig. 4(c). A more detailed description of interpreting the data statements is presented elsewhere (Chu and Cannon [7]).

The source program as well as the buffer and the directory for the external procedures is stored in a large random access memory, as shown in fig. 5(b). The source program information which includes the program name and the data type is stored in the data associative memory, as shown in fig. 5(a).

For interpreting control statements, the language processor has a control interpreter and a control associative memory, as shown in fig. 3. The control interpreter has an internal structure which can directly interpret the control constructs in the high-level language such as conditional branch, procedure call, and looping. It recognizes the control keyword, interprets the control statement, and then stores in the control associative memory the control information which includes the control statement name, the control statement type, and the pointers. The location of the first character of a control statement is used as the internal name of the control statement. For example, an IF statement is shown in fig. 5(c). The memory location 5 (character I) is the internal name of this control statement. Similarly, the memory location 34 (character E) serves as the Else Pointer. The END pointer points to the character location immediately following the end of the IF statement. The control information in the associative memory can expedite the repeated execution of those statements in a loop without the need for repeated syntactical processing. A detailed exposition of interpreting control statements is presented elsewhere. [7]

The language processor also evaluates an arithmetic expression for an assignment by using stacks and symbol tables whose descriptions can also be stored in the associative memory. For read and write statements, it turns over the execution to an I/O processor.

It is apparent that these processors can be microprocessors. The associative memory can be built on chips as it is within the state of the semiconductor art, or it can be implemented by using a high-speed

random-access memory and a conventional microprocessor.

6. INTERACTIVE DIRECT-EXECUTION SYSTEM

The interactive high-level language system has become increasingly popular, because it gives an interaction between the user and the system. If one decides to use an interactive high-level language system, there are at least two choices, an APL system or a BASIC system, from currently available systems.

A highly interactive high-level language system could be designed and programmed using the idea of interactive direct-execution. This highly interactive mode of operation could be similar to the manner in which English composition is written. As we write each word, we may consciously or unconsciously check the syntax and semantics of each phrase, each sentence, and each paragraph just written. In other words writing a high-level language could be made similar to writing an English composition. In this case, a highly interactive high-level language system could check the syntax and semantics (by execution) of the high-level language program as each symbol, each expression, each clause, and each statement are being entered at the terminal. Errors could be immediately indicated and correction could be immediately made. When the source program is completely entered at the terminal the program could have been debugged and could have run once. In short, an interactive direct-execution system could give maximum interaction in entering and debugging a high-level language program.

The above system requires an interpreter which can interpret in several ways, depending on the unit of interpretation. For example, an mentioned before, an APL interpreter interprets one APL statement at a time; therefore, the unit of interpretation is one statement. As another example, a Fortran compile and go interpreter immediately executes the compiled program after the entire program is compiled; in this case, the unit of interpretation is the entire program. A direct-execution interpreter is one which executes each symbol (or token); the unit of interpretation is one symbol. The direct-execution interpreter can be in software, firmware or hardware and it can be implemented on a high-level or low-level architecture. A software direct-execution interpreter has been implemented and reported. [8].

7. THE ISSUE OF HIGH-LEVEL MACHINE LANGUAGE

Conventional computers have been built mostly for a single low-level machine language. For example, there is a single machine language for the IBM S/360 family of computers. The single machine language has not limited the usefulness of the computers, because compilers and translators are provided to offer high-level programming languages.

A similar situation may prevail in a high-level-language machine. A direct-execution machine may be built for a single high-level machine language. This should similarly not limit the usefulness of the machines. Nothing prevents one from using software translators for other high-level or very high-level languages. However, in a direct-execution machine, one deals with the high-level language software only. The high-level language software should cost less and take shorter time to develop.

Alternatively, it is possible to build a direct-execution machine for a number of high-level languages by using either microprogramming with a high-level architecture or multiple high-level-language hardware interpreters; in this case, some processors performing the same functions could be shared.

8. COMPILERS OR INTERPRETERS

Since the development of the Fortran Compiler, this technique has made a great contribution to the ease of computer programming and thus to the large increase in computer applications. The use of the compiler has become so popular that it has been almost dominating the computer world for the past 20 years.

However, the compiler has created the need for other software such as linkage editors and loaders. Furthermore, many languages (high-level language, assembly language, relocatable code, and core dump) are involved in a compilation process. A competent Fortran programmer really needs to know these languages to some extent which causes additional difficulty in debugging. Therefore, Fortran programming is not as ideal as was first thought when it was introduced.

Since their arrival, compilers have become very complex, partly because more complex language constructs (which may or may not be genuinely useful to the users) are created. The computer manufacturers are becoming weary of implementing any new and powerful high-level languages because of the cost and time to develop and maintain the compilers and because of the investment in old programs! Yet, programming language is a central issue; more programming languages are bound to come. It appears that the development of compilers and high-level language are at present on a "collision course".

Historically, interpreters came before the arrival of compilers. Interpreters have not become popular because the computer systems of 20 years ago were too slow, and the main memory capacity too small. Since then, there has been a great advance in computer technology. One of the most significant contributions in hardware is the enormous increase in computer speed. The justification for ignoring the interpreter does not exist any more. It is time to look again at the idea of interpreters. After all, computation cannot be achieved by a compiler but must be accomplished by an interpreter (whether the interpreter is software or hardware).

For the last 20 years, more time and effort have been spent on the research and development of compilers than on interpreters. Thus, it is fair to say that more knowledge exists about compilers than interpreters. However, it is known that interpreters are easier to write, partly because they need no code generation, and generation of efficient code is a most difficult task in compiler writing. Furthermore, most of today's high-level languages are compiler-oriented languages; they are not the most suitable for writing interpreters for. Since the users of an interpreter need know no language other than the high-level language itself, the interpreter is much easier to use and is more amenable to an interactive system. The direct-execution interpreter whether software or hardware can make the system highly interactive with the users.

9. WHAT COULD HAPPEN IN THE NEXT 5 TO 10 YEARS?

We now have two computer industries. The first computer industry is old. It is centered around major computer centers and major computer manufacturers. The center of gravity of this first computer industry is the "200 billion dollars" of software. Because of this enormous investment, the first computer industry has an enormous resistance to changes and innovations, unless hardware is built to fit the existing software as it is developed now.

The second computer industry is new. It is represented by microprocessor manufacturers. It is called the second computer industry, because the

hardware manufacturers are different from the first computer industry. The markets are different. The applications are different. The system producers are different. The second computer industry is receptive to new ideas.

Although the second computer industry is different in almost every respect as mentioned above, their approach to the software follows that of the first computer industry, particularly the use of compilers and assemblers. It has been widely recognized that software cost is becoming the bottleneck to the large-scale usage of the microcomputer systems. Something new in software should be conceived and developed to overcome this difficulty. Why not use interpreters instead of compilers? Why not build the microprocessors with an architecture for interpreters instead of for compilers? Why not develop simple high-level languages for particular applications instead of "general-purpose languages?" Why not abandon the approach of "one-system-and-one-language" for many applications that has prevailed in the first computer industry? Why not re-examine the balance and the trade-offs among the hardware, software, and language instead of traditionally considering software alone?

There is little doubt that microprocessor systems will have the fastest growth ever experienced in the computer industry in the next 5 to 10 years, particularly if this software bottleneck could be removed. The use of the direct-execution idea combined with microprocessor technology could offer a solution to the problem of software cost, and the second computer industry could become even larger than the first one.

ACKNOWLEDGMENT

This research is supported by Grant NSF DCR75-05505 from the National Science Foundation of U.S.A. to the Department of Computer Science of University of Maryland.

REFERENCES

[1] P.P. Anderson, A computer for direct execution of algorithmic languages, Proc. of EJCC, 1961, 184-193.

[2] H.M. Bloom, Conceptual design of a direct high-level-language processor, Technical Report TR-239, Department of Computer Science, University of Maryland, April 1973. (NITS PS-224098/AS)

[3] B.W. Boehm, The high cost of software, Proceedings of a Symposium on the High Cost of Software, Monterey, California, September 17-19, 1973. Standord Research Institute.

[4] C.R. Carlson, A survey of high-level language computer architecture, High Level Language Computer Architecture, Academic Press, Inc., 1975.

[5] Yaohan Chu, Introducing the high-level-language computer architecture, Technical Report TR-227, Department of Computer Science, University of Maryland, January 1973. (NITS PS-224398/AS).

[6] Yaohan Chu, (editor), High-Level Language Computer Architecture, Academic Press, Inc., New York, 1975.

[7] Yaohan Chu, Bloom and Cannon, High-level language hardware control structure, Technical Report TR-412, Department of Computer Science, Univ. of Maryland, October 1975.

[8] Yaohan Chu, and E.R. Cannon, Design of an interactive direct-execution system, Technical Report TR-385, Department of Computer Science, Univ. of Maryland, June, 1975.

[9] Yaohan Chu, and E.R. Cannon, High-level language memory structure, Technical Report TR-409, Department of Computer Science, University of Maryland, September 1975.

[10] Yaohan Chu, and E.R. Cannon, Interactive high-level language direct-execution microprocessor system, IEEE Transactions on Software Engineering, June 1976.

[11] Yaohan Chu, J.C. Yeh and E.R. Cannon, Direct-Execution on the Burroughs B1700 system, Technical Report TR-335, Department of Computer Science, University of Maryland, October, 1974. (NITS PB-238052/AS)

[12] L.H. Cooke, Programming time vs running time, Datamation, December, 1974, 56-58.

[13] A. Hassit, J.W. Lageschulte, and L.E. Lyon, Implementation of a high-level language machine, Commun. of the ACM, 1973, 199-212.

[14] W.M. McKeeman, Language directed-computer design, Proceedings of AFIPS FJCC, 1967, 413-417.

[15] A.P. Mullery, A procedure-oriented machine language, IEEE Transactions on Electronic Computer C-13, 1964, 449-455.

[16] A.P. Mullery, R.F. Schauer, and R. Rice., ADAM--A proglem-oriented symbol processor, AFIPS SJCC, 1963, 367-380.

[17] B.J. Robinet, Architectural design of a directly-executed APL processor, Technical Report, TR-320, Department of Computer Science, University of Maryland, August, 1974. (NTS PB-235775/AS).

[18] John T.C. Yeh, Architectural design of a L6/M language processor, Technical Report TR-279, Department of Computer Science, University of Maryland, December 1973. (NITS PB-226548/AS).

[19] R. Zaks, Microprogrammed APL, Proc. of IEEE Computer Society Conference, 1971, 193-094.

[20] R. Zaks, D. Steingart, J. Moore, A firmware APL time-sharing system, Proc. of AFIPS SJCC, 1971, 179-190.

[21] R. Rice and W.R. Smith, SYMBOL--A major departure from classic software dominated von Neumann computing systems, Proc. of Spring Joint Computer Conference, AFIPS Press, 1971, 575-587.

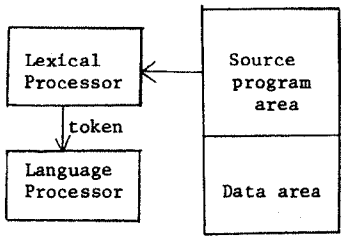


Fig. 1. Program storage and lexical and language processors

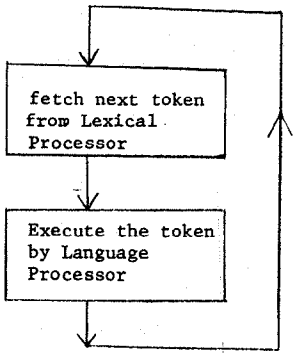


Fig. 2. Interpretation cycle

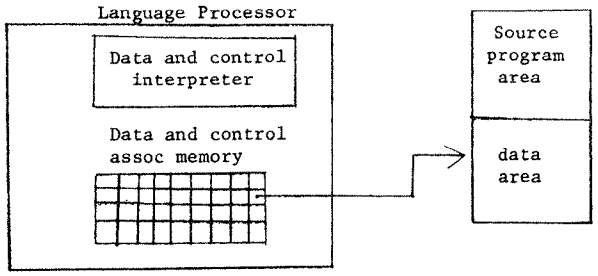


Fig. 3. The data and control interpreter and the data and control associative memory in the language processor

```
Stack 01 S, CHAR=10, MAXSIZE=8,
         02 NAME, CHAR=6,
         02 VALUE,CHAR=4;
```

(a) A stack data statement

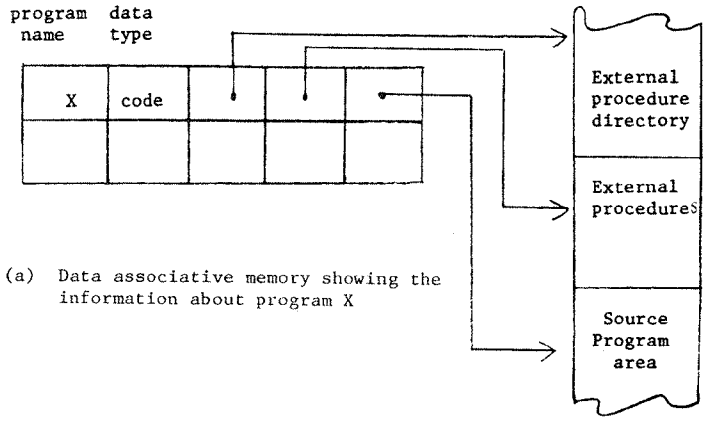
stack name	structure type	date type	data length	max size	cur. size	location pointer
S	stack	char	10	8	4	
S.NAME	stack	char	6	--	--	
S.VALUE	stack	char	4	--	--	

JOE	101
A	1456
JOHN	4123
C	7612

(b) Data memory

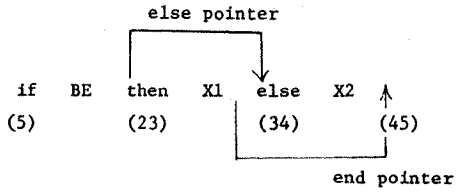
(c) Data associative memory showing the information about the stack statement

Fig. 4. Interpretation of the stack statement by the data interpreter



(a) Data associative memory showing the information about program X

(b) Program memory



(c) An IF control statement

control stmt name	control stmt type	end pointer	else-clause indicator	else pointer
5	C	45	yes	34

(d) Control associative memory showing the control information for the IF statement

Fig. 5. Interpretation of the IF statement by the control interpreter