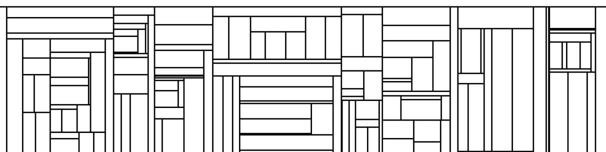
The Significance of Microprogramming

Robert F. Rosin

DAIMI PB - 16
June 1973

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF AARHUS
Ny Munkegade, Bldg. 540
DK-8000 Aarhus C, Denmark



THE SIGNIFICANCE OF MICROPROGRAMMING*

Robert F. Rosin
(University of Aarhus, Aarhus, Denmark,
on leave from SUNY at Buffalo)

Abstract

Interest in the topic of microprogramming appears to have had a great growth in the middle and late 1960's, but since that time it has leveled off or declined somewhat. This paper examines the reasons behind this development and then offers a reconsideration of and a new proposal for the definition of microprogramming.

This result is supported by the consideration of three phenomena. First is the evolution of microprogramming during the past twenty years. Second is the evolution in the use of interpretation as an implementation technique. Third is the set of "rules of thumb" resulting from the system designer having to resolve conflicting forces attempting to influence his activity.

Five possible meanings of the term microprogramming are considered and rejected totally or in part prior to the one finally offered. The suggestion is made that this redefined concept be avoided as much as possible in the future, and two avenues for research alternatives are encouraged instead.

* supported in part by NATO Reserach Grant No. 621,
to be presented at the International Computing Symposium,
Davos, Switzerland, September, 1973.

INTRODUCTION

Although it first arose in 1951 (10), interest in the idea of microprogramming peaked during the 1960's (11), and it seems recently to have declined somewhat. That is not to say that the literature in this area is not growing, for one need only consider the continuance of the ACM SIGMICRO Microprogramming Workshops and issues of the SIGMICRO Newsletter to be convinced that people committed to this concept have produced a reasonable amount of new results. However, there appear to have been no successors to the book by Husson (4), and the word microprogramming seems either to have never caught on or to be of little current interest to many leaders and opinion makers in computer systems research. It is not possible to offer "negative references" to support such a statement. But one can observe that the many papers in the 25th Anniversary Issue of the ACM Communications and the several ACM Turing Lectures * reveal a remarkably infrequent occurrence of this term. In particular, note that the paper by Foster (2) in CACM, which treats "microcomputers" at great length, never even suggests the concept of microprogramming.

Rosen, in the same issue of CACM (6), offers three short paragraphs on microprogramming in which he mentions its use as a replacement for conventional hardware technology, suggests that it is really programming and not hardware, and then predicts that, although "advances in this area have been slower ... than some of the enthusiasts ... have predicted, ... software in microprogramming is bound to be important". Through many of my own activities, including two papers (7, 8), I have clearly expressed the enthusiasm which Rosen refers to. I believe it is only responsible for those of us who have been in that position to offer some explanation as to why our predictions have not proven valid, and that is one of the purposes of this paper.

---

*) The ACM Turing Lectures have been published as follows:
Perlis, JACM vol. 14, No. 1 (Jan. 1967) 1-9.
Wilkes, JACM vol. 15, No. 1 (Jan. 1968) 1-7.
Hamming, JACM vol 16, No. 1 (Jan. 1969) 3-12.
Minsky, JACM vol. 17, No. 2 (April 1970) 190-215.
Wilkinson, JACM vol. 18, No. 2 (April 1971) 137-147.
McCarthy, (unpublished ?)
Dijkstra, CACM vol. 15, No. 10 (Oct. 1972) 859-866.

It is the primary purpose of this paper to offer reorientation and redirection of the common perception of microprogramming. Of course, the reader is cautioned to place no more credence in projections included here than he did in those offered earlier, especially in the light of the shortcomings of earlier enthusiasm. But at the same time it is hoped that he will appreciate the reflection and sincerity which form the basis of what follows.

The remainder of this paper is divided into two major sections. In the first we examine three phenomena which are used later to support our reconsideration of microprogramming. The three phenomena are the evolution of microprogramming, the corresponding evolution of the use of interpretation in the implementation of systems, and the practice of computer architecture in which the previous two phenomena play an important role as tools.

In the second major section,microprogramming is reexamined. We first consider and then reject, at least partially, five generally accepted views of microprogramming. This leads to the conclusion which embodies first a new definition, constructed out of what remains from these five ideas, and second the realization that this newly defined interpretation of microprogramming should to a very great extent be avoided by the community of computer scientists. Two alternative avenues for research and development in system architecture are proposed as a direct replacement.

THREE INFLUENTIAL PHENOMENA

The Evolution of Microprogramming

In his paper "The Growth of Interest in Microprogramming", Wilkes (11) presented an excellent picture of the history and development which took place in the area from 1951 to 1969. There is no necessity to repeat that discussion here. However, as a programmer, I attempted in (7) to offer a rather different perspective of the impact of that evolution in terms of the potential application of microprogrammed computers rather than their implementation. What follows is a very brief summary of both of these points of view, and then an attempt to extend that discussion into a more recent period. References to the material of historical interest can be found in each of the papers cited above.

In 1951 Wilkes first suggested that the control unit of computers should be organized in a systematic rather than ad hoc manner, and he called this approach "microprogramming". This concept had almost no impact on the thinking of computer users or even programmers. Even as late as the mid 1950's the idea that such persons could have any influence on the organization of computer hardware was considered quite radical and was carefully discouraged by most hardware designers throughout the next ten years.

In the mid 1960's there were two events which had significant effect in changing the status quo: the development of the IBM System/360 based on a somewhat flexible basis for the realization of microprogram store, the read-only memory; and a suggestion by Opler (5), among others, that read-only store would eventually be replaced by so-called fast/read-slow/write stores. Only then did the potential of "rewirable" machines, user designed instruction sets, "hardware" support for higher level languages, etc. become a matter of popular interest. By the late 1960's there was considerable attention being paid to the idea of using microprogrammed systems to interpret (or "emulate") a wide variety of previously uneconomical "machines". Furthermore, it was suggested, at least among some programmers and users, that one could expect about a factor of ten improvement for each level of perhaps unnecessary interpretation intervening between a computer application and an actual hardware system.

Research projects blossomed in attempts to harvest the fruit of the new era. Consider the work of the author and his group as reported in (8). However, not everyone shared this enthusiasm. In particular, the hardware community suggested that such attempts were bound to yield disappointment if not failure. They said that microprogramming is not ordinary programming; that one must pay particular attention to timing problems; that one must learn to cope with parallelism; that at least some of the existing machines were designed for particular purposes and would not be suited as hosts for microprogrammed support of higher level languages, and, in particular, that microprogramming required special skills in order to use the facilities of the hardware efficiently. It was suggested that the word "programming" never should have been included in the name of the concept, and that perhaps "microcontrol" would have been more appropriate. At least such a choice of nomenclature might have kept the programmers out of the way.

With only a few exceptions there have been in fact no true successes in the quest for microprogrammed support of other than the line of successors to the von Neumann Machine. The hardware community seems to have made their case. Were the programmers so completely wrong? Was this analogous to the usual case of a programmer promising to meet an unmeetable deadline? These questions will be considered later. First let us look briefly at the history of the use of interpretation in system implementation, for it is only in this way that microprogramming has any applicability.

The Use of Interpretation

By the mid 1960's it was generally believed that interpretation was usually less efficient than translation as a technique for implementing higher level languages. Because most programs have been written in higher level languages since that time, one could conclude that interpreters are seldom if ever used, and that they might even disappear from the scene in the near future. In fact, however, almost the opposite is the case. We consider here four motivations for the use of interpreters, from among the many which might be offered, to explain this apparent paradox.

1. Limited amounts of fast storage

Interpreters were used on the very earliest computer systems. One particular instance of wide spread interest was the Bell Interpretive System (12) which was implemented for the IBM 650 computer in about 1956. The attempt here was to provide a means to help the ordinary user, who was accustomed to desk calculators, to cope with "certain problems not encountered in desk computing", as the report states. This goal is merely an early, specific statement of the desire to provide application-oriented systems for computer users. Somewhat generalized, the languages of most hardware systems are not directly suited for the applications which must be run on them. This has been the, perhaps unmentioned, justification behind the development of every higher-level language and software system before or since that time.

But that fact does not justify interpretation as opposed to translation as a technique for solving this problem. The report on the Bell Interpretive System goes on to say, "Limitations in storage capacity may necessitate the choice of an interpretive system rather than a system of the once-for-all type in the case of small or medium size computers" (where once-for-all means translation). This happens not only because compilers are often large, but because object programs frequently are also, and an appropriately designed interpreter can lead to meaningful reductions in their size. The problems of large programs and small main stores have not yet disappeared.

2. To support a family of computers

The trend after 1956 was to larger stores, and interpreters seemed to fall into disuse for system implementation. But in fact, this was only because their use became hidden. When IBM made its far-

reaching decision to abandon the development and production of incompatible machines in favor of a family of machines, interpreting of programs came to the forefront again, although not in a way that most people recognized it. The problem was to implement a family of machines which had precisely the same instruction set and data constructs, but across a wide spectrum of cost-performance ratios. While it was economically justified to build a very fast direct hardware implementation of such a system which would cost a correspondingly large amount of money, it was not clear that this was the best approach for implementing a smaller, slower version of the same system. A suitable conclusion was reached when it was demonstrated that the latter version of the system could be implemented by writing an interpreter to be run on a fast but less complicated machine: one which had a small amount of fast memory for holding the interpreter itself, but slow memory for holding the program to be interpreted. This machine could have a narrow data path and functional units which could be programmed to operate serially to simulate the wider parallel operations of the machine to be realized. This scheme, the heart of contemporary microprogramming, was used, for example, in the implementation of the majority of the 360 systems in the field, and is now employed in all of the 370 product line.

3. The trend toward fast, inexpensive hardware components

If the superficial reason behind the decision to use the interpretive technique in implementing some models of 360 was to achieve a family of systems,on closer examination it was again also based on the relatively high cost of fast storage, and the desire to economize on this portion of the system. However, also implicit in this decision is the fact that hardware was becoming cheaper and faster in the middle 1960's, a trend which does not seem to have abated. The most obvious way to employ such devices in low cost systems is to assemble them into very simple host machines which are then used to interpret the languages of slower but far more complex virtual machines.

4. Higher level languages

As much as the hardware world has provided a rapidly changing set of parameters, the world of the user has also forced us constantly

to evaluate our practices. Higher level languages have developed and evolved considerably during the past two decades, along with processors to support them on whatever machines are available. As stores became larger the motivation for interpreters declined, but this was only a temporary phenomenon. Newer languages require very complex support, not only at compile time, but also at run time, in order to be used on available hardware systems. The typical (but not unique) approach to providing this support is to implement a library of routines which are invoked at run time to support various functions required. This is, in effect, processing a somewhat massaged portion of the original code through an interpreter. Perhaps the most obvious example of this application of interpretation is seen in the execution of FORMAT statements in such languages as Fortran. The block structure, dynamic storage allocation and string manipulation facilities many languages have often led to complex interpreters for their realization.

It is reasonable to conclude that interpretation has not disappeared as a tool for system implementation, and, except for the motivation provided by inadequate fast store, the pressures to use this technique seem to be increasing. That is, languages are becoming more demanding at run time, hardware is becoming cheaper and faster, and families of computers are necessary in order to have some control over the rapidly rising costs of system development. Let us now examine in more detail the effects of these and other forces which are operating on the system architect.

The Practice of Computer Architecture

The following sentence is taken from a proposed encyclopedia entry on computer architecture and is authored by a person of recognized competence in the field: "As normally conceived of, a computer architect accepts from a logical designer units such as adders, stacks, memory blocks, and tape drives, and puts them together so that they form a computer, and turns this over to the systems programmer who then constructs an operating system for the machine". (3) (emphasis added). This quotation is as remarkable in what it does not say as in what it does say. One must ask when, if ever, does the consideration of user applications enter the picture? Is there any reason to believe that the influence of higher level languages will be found in any but a few systems which themselves are often considered eccentric? Is there a place for a system architect in the picture?

Even in the ideal case, where the computer architect is aware of or even responsive to the needs of the user, it is clear that he must contend with what he is obliged to "accept" from the logical designer. Moreover, he is put into the very difficult position of attempting to resolve two basic sets of forces.

There is one set of forces which arise from the evolution of basic hardware and software technology and long standing traditions. They tend to maintain the status quo and are rather well defined; they change, but usually in a predictable fashion. In contrast, the other set of forces, which arise from the needs of society for computer services, is very difficult to define in any precise way, and, furthermore, is subject to rapid and rather unpredictable change. The result is that the system designer establishes a number of "rules of thumb" which, whenever there is a conflict, generally reflect the influence of the first set, and the "acceptance" cited earlier. Consider the following examples.

Rule 1:

"In case of doubt, sacrifice a design concept to preserve cycle time". After all, the user of a system cannot demonstrate that a certain design component is good or bad until the machine is built, and then it is too late. Reconsideration is too expensive for both the user and the producer, and the architect has also usually moved on to his

next machine. In particular, it is essentially impossible to measure the cost of a particular computation on a potential system as a function of various design constraints without first implementing the entire system. On the other hand, one can always estimate cycle time quite accurately. Somehow many a designer's ego is closely connected with that measure. Weinberg (9) discusses the virutes of "ego-less" programming, and there is no reason that this idea cannot and should not be extended to cover the system design process as well.

Rule 2:

"Some facilities are cheap". If an extra capability can be added to a machine at no cost, due to the existence of a set of possible bus connections in hardware which where not intended in the original plan, then a way is often found to realize this function as an added instruction. The fact that this addition may compromise the system by inviting the programmer to bypass a more globally conceived construct,which was purposely intended in the original design,is considered to be of little importance. Following this rule in the era of medium and large scale integration of curcuitry has led designers to add uncalled for register sets and adders, and even to change the word size during design of otherwise "clean" systems.

Rule 3:

"Design constraints don't allow the realization of some otherwise good ideas". This rule usually derives directly from a set of preconceived components being allowed to have undue influence on a design. Examples are found in the precondition of such things as a memory of a certain size and speed, a printed circuit board of some particular dimensions, a committment to a particular field size in an instruction format,etc.

Rule 4:

"If it looks nice, it must be useful". A system is a reflection of the professional ability of the designer; a monument to his cleverness. Therefore, if a particular feature is contrived which happens to illustrate this cleverness, then it is often made prominent in the implementation, even if it compromises the ultimate objectives of the system.

It is our contention that obeying these rules will lead the system architect to specify machines which bear the label "unreasonable".
It is our observation that most hardware systems contain timing idiosyncrasies, gates rather than functions, assymetric bus structures, registers of unsuitable width, preoccupation with a feature of limited scope, etc. The degree to which a machine is considered "unreasonable" is directly proportional to the degree to which it has the above characteristics. Thus a "reasonable" machine tends to be one which is transparent to the user, helps rather than hinders its application to the solution of a given task, and can perform the computation with an acceptable cost/performance ratio, where cost is measured in terms of man-hours as well as CPU-hours.

Based on our understanding of the experience gained in the design and use of the wide variety of computer systems available during the past 25 years, we offer the following guideline for achieving reasonable computer systems.

First, the rules of thumb stated above must be avoided as much as possible in computer system development.

Second, the entire history of computer system development has shown us that systems are always used for purposes that their designers never foresaw. Supplying generally applicable, readily accessible, and highly symmetric facilities will go a long way toward ensuring the longevity of a particular system.

Third, although programmers may not have the force of rigorous science to support their beliefs, their convictions about computers are not necessarily invalid. For example, whether or not it can be proven desirable, the ability to allow convenient relocation of code is very useful. The authors know of at least two so-called microprogrammed machines whose implementations of branching disallow effective relocation.

Finally, one should not attempt to apply tools which are inappropriate to a given task. For example, although the various hardware systems on which System 360 has been emulated are possibly good hosts for that purpose, the same hardware is generally unsuited for efforts in the direct or indirect support of higher level languages. Conclusions drawn from such an application of inappropriate hardware could be seriously misleading, if one is interested in the general question of low level support for such languages.

Having considered these three phenomena, that is, the evolutionary development of microprogramming and interpretation along with system architecture, we now are able to make a broad assessment of the significance of microprogramming.

MICROPROGRAMMING REEXAMINED

Microprogramming has been described by many, including this author, as helping to bridge the gap between hardware and software. However, I now feel that microprogramming has actually been used to disguise the gap rather than bridge it. To understand this conclusion requires that we appreciate what people really mean by the term microprogramming.

A Definition of Microprogramming

Let us first consider a set of criteria usually used to describe this concept.

1. "Using systems which are faster than those normally in use is microprogramming". If this aspect were allowed, then the first programs ever run were microprograms, and the programs written for any machine faster than yesterday's machines are microprograms, etc. This is clearly not a valid criterion, although faster execution can lead to the adoption of interpretation as a tool for system implementation. Therefore, as one alternative, one might conclude that:

2. "Support of emulation is microprogramming". As discussed earlier, emulation is merely one aspect of interpretation. There are obviously hundreds of uses of interpretation which could in no way be accepted as microprogramming.

3. "Programming for real hardware machines is microprogramming". That would lead one to believe a machine language program run on a 360/75, which is a system implemented directly in hardware, is somehow different from identically the same program run on the 360/30, which itself is simulated by another program (a "microprogram") run on the 2030 CPU. But of course, except for timing differences, it is not at all different, so we must reject this idea. But one can now consider:

4. "Writing programs in consideration of the timing of various instructions and storage devices is microprogramming". Going back to our previous example, it is possible, but almost an act of herecy, to write a program for one of these 360 systems which depends on its ba-

sic timing to the extent that the program will not run successfully on the other system. Yet few people if anyone would agree that coding for either of these systems is microprogramming.

5. "Microprogramming is programming the direct control of the gates and buses of a computer". This criterion is derived from the idea of "horizontal" or "minimally encoded" microinstructions. But, since this excludes "vertical" or "highly encoded" microinstructions, it does not suffice to include all of microprogramming. Nevertheless, this criterion is one of the cornerstones of our proposed redefinition.

What then is the essence of microprogramming? These five potential criteria, taken all together, after distilling out those aspects which are irrelevant, lead me to propose that:

> "<u>microprogramming is the implementation of hopefully reasonable systems through interpretation on unreasonable machines!</u>"

The concept of reasonability was introduced in the last section. It follows that an unreasonable machine is one which requires the programmer to know about and cope with particular gates, buses, race conditions, split cycle storage, and all other "features".

Given this definition, microprogramming is equivalent to the use of interpretation to <u>hide</u> the nature of unreasonable machines by constructing reasonable systems on top of them. Our earlier comment about its use in disguising the hardware-software gap follows directly.

Furthermore, it is believed by many computer scientists that unreasonable machines are neither a good idea nor are they necessary: In fact, they are to be avoided, and, therefore, microprogramming should also be avoided whenever and wherever possible. On the other hand, if one is faced with an unreasonable machine, then microprogramming, as defined here, is a necessary tool.

Alternatives to Microprogramming

It was suggested in the introduction that there has been a relatively poor showing in most attempts to use microprogramming for other than the implementation of conventional machines, which is merely an application of the disguising function referred to in the previous section of this paper. However, given our new understanding of microprogramming, this lack of success is no real surprise. It is due, in large measure, to the fact that these other attempts to use microprogramming precluded by definition the use of reasonable machines as hosts. In fact, in past years, there existed no such reasonable machines. There appear to be two viable alternatives to the pursuit of microprogramming which can be recommended.

The first major alternative is to pursue the development of reasonable machines. To do this it is necessary to analyze the forces acting upon the system architect, such as those suggested earlier, and then to reject totally any rules of thumb which resemble the ones stated. It is possible to design and build machines which have no timing problems, which exhibit an economy of well chosen and well integrated facilities, which have no obtrusive "features" based on some accidentally available componentry or someone's ingenuity, and which are not befouled by constraints imposed prior to conceptualization of a whole design. The quest will become more and more important as time and technology progress. In (2) Foster predicts an era of machines which are very inexpensive and yet very powerful. But will they be reasonable? It is obvious that they should be, but, given the present exploitation of microprogramming to hide a multitude of design sins, we cannot be sure.

The second major alternative to microprogramming is to carry out the proposed research programs, for example that of the author in 1969 (7), but on machines intended to support the proposed projects; that is, on reasonable machines. Such activity appears to be underway in a small number of institutions but may become widespread as the availability of somewhat reasonable machines increases.

CONCLUSION

Given the ideas expressed above, the significance of microprogramming is presently far out of proportion to its potential return. At least a portion of the vast amount of time being spent on coping with unreasonable machines would be better spent on other tasks. A committment in the direction away from microprogramming can be expected to result in an era with a high number of hopefully reasonable systems implemented through interpretation on reasonable machines.

Rather than basing my new interpretation of microprogramming on prior attempts to define this term, I have concentrated on the characteristics popularly attributed to the term by contemporary practitioners. This analysis is preceded by an examination of the closely related topics of the development of microprogramming, the use of interpretation, and the state-of-the-art in system architecture. At the heart of the latter part of the paper is the concept of unreasonable machines and the claim that their use interferes with productive work in system implementation.

REFERENCES

(1) Flynn, M., and Rosin, R.F. Microprogramming: an introduction and viewpoint. IEEE Trans. Comput. C-20, 7, (July 1971), 727-731.

(2) Foster, C., A view of computer architecture, Comm. ACM, 15, 7 (July 1972) 557-565.

(3) Foster, C., Computer Architecture, Computer Architecture News, 2, 1, ACM, (Jan. 1973).

(4) Husson, S., Microprogramming: Practices and Principles, Prentice-Hall, Englewood Cliffs, N.J., 1970.

(5) Opler, A., Fourth generation software, Datamation, vol. 13, No. 1 (Jan. 1967) 22-24.

(6) Rosen, S., Programming systems and languages, Comm. ACM, 15, 7, (July 1972) 591-600.

(7) Rosin, R.F., Contemporary concepts of microprogramming and emulation. Computer Surveys 1, 4 (Dec. 1969), 197-212.

(8) Rosin, R.F., Frieder, G., and Eckhouse, R., An environment for research in microprogramming and emulation, Comm. ACM, 15, 8 (August 1972) 197-212.

(9) Weinberg, G., The Psychology of Computer Programming, van Nostrand Reinhold, New York, 1971.

(10) Wilkes, M.V. The best way to design an automatic calculating machine. Manchester U. Computer Inaugural Conf., 1951, 16-21.

(11) Wilkes, M.V. The growth of interest in microprogramming, Comp. Surveys, 1, 3 (Sept. 1969) 139-145.

(12) Wolontis, V.M., A complete floating-decimal interpretive system for the IBM 650 magnetic drum calculator, Technical Newsletter No. 11, Applied Science Div., IBM, March,1956.