The Effects of Emerging Technology and Emulation Requirements on Microprogramming

SAMUEL H. FULLER, MEMBER, IEEE, VICTOR R. LESSER, C. GORDON BELL, FELLOW, IEEE, AND CHARLES H. KAMAN, MEMBER, IEEE

Abstract—**The structure of microprogrammed processors is largely determined by the state of (semiconductor) technology and the requirements of the task of emulation. We discuss the impact of LSI components on microprogrammable processors and in particular, the effect of large memory arrays, LSI microprocessors (bit-slices), programmable logic arrays, and high-speed shifters.**

A secondary theme of this article is that microprogramming differs very little from "regular" programming. We argue that the right approach to understanding microprogramming is to recognize that it is primarily applied to the task of emulation. We review the requirements of the emulation (interpretation) task and indicate what capabilities a microprogrammable processor needs to have in order to make the process of emulation efficient. We conclude with a taxonomy of microprogrammable processors.

Index Terms—**Emulation, interpretation, microprocessor, microprogramming, semiconductor technology.**

I. INTRODUCTION

THE STRUCTURE of microprogrammed processors, and microprogramming in general, is largely determined by two factors: the state of (semiconductor) technology and the task of emulation. Therefore, this article first reviews those technological advances as well as those constraints and demands imposed by the emulation process that have shaped the evolution of microprogramming. We then use these observations to put the past developments of microprogramming in perspective and forecast the major developments in the years ahead.

The other main theme of this article is that trying to characterize and understand microprogramming in terms of how it differs from "regular" programming is a fruitless exercise. The futility of this approach can be seen by the numerous, contradictory definitions on microprogramming in the literature (Rosin [14], Wilkes [21], Mallach [10]). Attempts to base a definition on features of a processor's architecture, such as horizontal instruction formats, lack of an explicit program counter, or visibility of real registers and data paths, or on features of a processor's realization, such as the speed of main memory to that of the control (micro) memory, are easily rejected on the basis of existing processors that are commonly recognized to be microprogrammed processors yet do not possess the required features.

Most of this confusion in alternative definitions comes from the fact that microprogramming has been used in two very different ways: 1) as a hardware implementation technique to economically implement a complex instruction set or a small number of different instruction sets on a single processor, and 2) as a software technique to provide programmers with an extra degree of representational freedom, i.e., develop multiple instruction sets, each one appropriate for a particular task domain. The technological use of microprogramming has been the dominant justification for the development of the vast majority of microprogrammable processors in the past decade. But as the cost of software began to become the major cost of a computer system, the use of microprograming as a technique for making a computer more convenient to program has and will continue to become the more important application.

The most direct approach to understanding microprogramming is to recognize that it is primarily applied to the task of emulation (interpretation). Through this approach it is possible to understand and predict the evolution of microprogramming independent of a particular technology and type of instruction set being emulated.

The process of emulation will be taken up in considerably more depth in Section III, but it will be useful here to briefly look at the different processors used to emulate a Basic machine. On the one hand there are the Hewlett-Packard 2100, DEC PDP-11, and PDP-8 that have time-sharing systems supporting Basic. The only language available to the user is Basic and the architecture of the processor is hidden from him. On the other hand there are the Basic programmable calculators available from IBM (i.e., the IBM 5100), Hewlett-Packard (Spagler [15]), and Wang Laboratories that operate as Basic machines: their input keys and displays are tailored to the Basic language. It is difficult to insist that the HP-2100, PDP-11, and PDP-8 are not microprogrammed processors while the "hidden" processors in the IBM, HP, and Wang Basic calculators are microprogrammed. The only characteristic all these processors have in common is that they are emulating Basic and a good case can be made for dropping the

Manuscript received June 16, 1975; revised March 29, 1976. This work was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense under Contract F44620-73-C-0074 and was monitored by the Air Force Office of Scientific Research.

S. H. Fuller and V. R. Lesser are with the Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213.

C. G. Bell was on leave at the Departments of Engineering and Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213. He is with the Digital Equipment Corporation, Maynard, MA.

C. H. Kaman is with the Digital Equipment Corporation, Maynard, MA.

term "microprogramming" altogether and simply using "emulation" in its place. However, we will continue to use the term "microprogramming" here since it is so widely used and it is a convenient way to indicate that we are discussing programming as it applies to emulation (and interpretation) rather than programming in general.

Following our discussion of technology and emulation, we then discuss specific hardware and software techniques for emulation. A number of different types of microprogrammed processors are also included as examples.

II. Semiconductor Technology

The state of the art in semiconductor electronics has had a profound effect on the feasibility of microprogramming. Prior to the 1960's the only effective means of implementing a high-speed control store was to use a diode matrix. This was the technology used by Whirlwind I (Everett [3]) and suggested by Wilkes in his original paper on microprogramming [22]. Figs. 1 and 2 show the structure of these control units. As long as these diodes were discrete components, a control store of any reasonable size was too expensive to compete with alternate implementations using random logic (e.g., over 30 000 bits of control storage are required to implement the full PDP-11 architecture on the DEC LSI-11, while the Whirlwind I had only 4800 "bits" in its control store). It is important to realize that both structures are just the control parts of their processors and are alternatives to conventional sequential control circuits as shown in Fig. 3. It was not until the middle and late 1960's that integrated-circuit technology advanced to the level that economic read-only memories (ROM's) and read–write memories (RAM's) became a practical reality. It stands to the credit of IBM's engineers that they were able to develop the IBM System/360 series of machines via microprogramming in the early 1960's; every model in the early IBM 360 line used a different, nonsemiconductor technique to implement its control store. These ingenious, but admittedly cumbersome and costly, techniques could be laid aside when the IBM S/370 series of machines were implemented since integrated-circuit technology had advanced to the stage that semiconductor control stores were reliable. Fig. 4 illustrates the basic structure of current microprogrammed control units.

Semiconductor memories suitable for control stores in microprogrammed processors are now at the stage where 1024 bit/package RAM's and 4K (K = 1024) bit/package ROM's are in wide use in present processors and 2K RAM's and 8K ROM's are being designed into some newer processors. 4K RAM's and 16K ROM's have been announced and are available, but at present they are too slow to be seriously considered for most control stores.

For well over 10 years now semiconductor manufacturers have set a pace where the commercially feasible chip complexity (i.e., number of devices per chip) has roughly doubled every one to two years. For example, the 4 kbit/package RAM (13 000 devices) was introduced roughly two and one-half years after the 1 kbit (4000) RAM. Recent

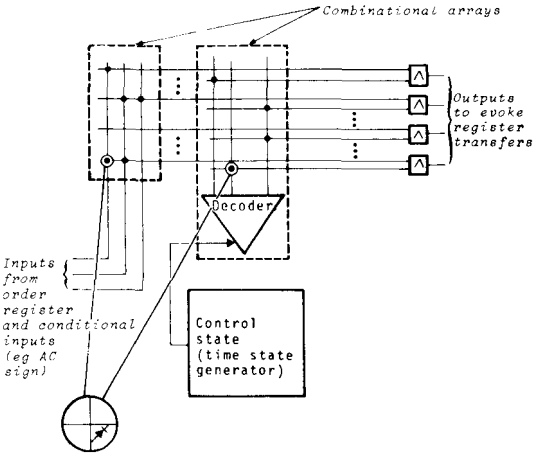

Fig. 1. M.I.T. Whirlwind control unit.

announcements of 16 kbit/package RAM's continue this trend. There is every reason to believe that this trend will continue for at least the next 4 to 6 years. Hence we face a situation where we can expect to see the size of control stores growing as technology encourages designers to use more control storage to cut costs in other areas, improve the performance of the microprocessors, or add additional capabilities.

The other LSI components that are having a major impact on the evolution of microprogramming are the LSI microprocessor chip sets.[1] The Intel 3000 2-bit processor bit-slice,[2] the AMD 2901 4-bit processor bit-slice, and the Western Digital microprocessor set are the most popular versions of this type of component. These LSI microprocessor components implement the major registers, data paths, and arithmetic unit in LSI packages. They all rely on microprogramming to specialize their behavior to the appropriate "target" architecture (e.g., PDP-11, HP-2100, disk channel, communications processor). Since it will be much more cost-effective to implement the next generation of small computers using these LSI microprocessor components, microprogramming will take on an even more central role in the implementation of small computers.

Memory arrays and LSI processors are not the only

[1] By now, the term *microprocessor* has become a terribly overworked word. It is commonly used in at least three ways.

1) A microprogrammable processor. In other words, the processor that fetches and executes the microinstruction sequence which is used to emulate the "target" machine. We will continue to call this processor a *microprogrammable processor.*

2) A processor with very small data types (e.g., 8-bit words), a relatively limited instruction set, and usually implemented in one or a small number of LSI packages. The most common examples are the Intel 4040, Motorola 6800, and the Intel 8080. The character (or digit) oriented processor is not considered in this paper.

3) Any processor whose registers, data paths, and arithmetic unit are implemented in a small number of LSI components. For example, the DEC LSI-11 and the CMU-11, a PDP-11 built with Intel 3000 components (McWilliams *et al.* [11]). Here "micro" refers to the small numbers of integrated circuits needed to implement the basic processor, not the small size of the data types. We use the term *LSI microprocessor* to refer to these types of processors.

[2] An LSI component is characterized as a bit-slice component if multiple copies of the component can be connected together so as to form an arbitrary-width processing element. For example, by connecting together 3, 4, or 8 ADM2901 4-bit processor bit-slices, respectively, a 12-, 16-, or 32-bit processing element can be constructed.

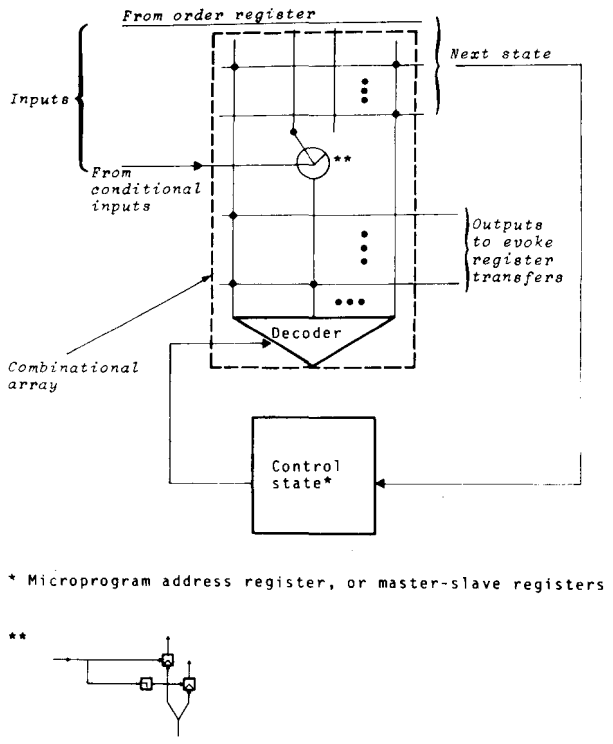

Fig. 2. Wilkes 22 microcontrol unit.

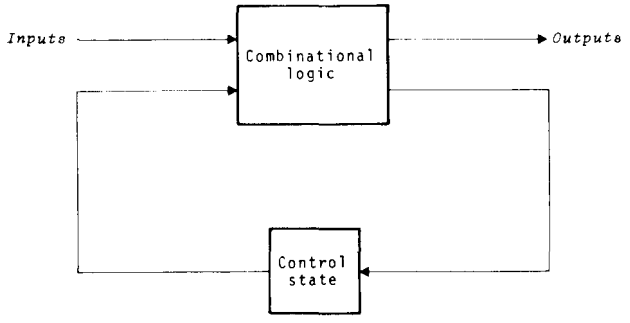

Fig. 3. Conventional sequential control circuit.

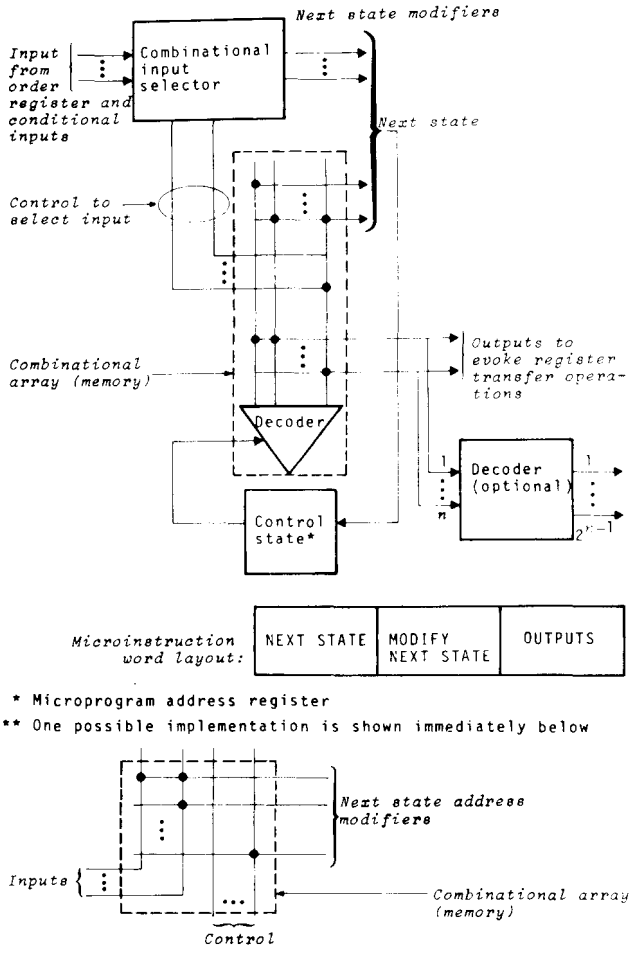

Fig. 4. Current microprogram control units.

developments in semiconductor technology that are having a significant effect on the structure of microprogrammed processors. Two other very important developments are the programmable logic array (PLA) and shifter. The basic structure of a PLA is shown in Fig. 5. It is a two-level combinatorial logic circuit that is "wired" for a specific application through masking, or metalization. The PLA has the same outward characteristics of a ROM except that it would take a ROM with several orders of magnitude more devices to match the function of the PLA in many applications. For example, a common PLA is a Rockwell Corporation package with 48 I/O terminals [13]. ROM that would be equivalent to this PLA in many applications would require two orders of magnitude more bits. A PLA uses the same techniques that designers of digital circuits used a decade ago to minimize the number of gates required to realize a combinatorial function. However, if the function to be implemented is sufficiently ill-conditioned (e.g., a parity tester), the PLA offers no advantage over an ROM. Instruction decoding is an example of a combinatorial function amenable to minimization techniques and hence PLA's will be very useful for providing the decoding of instructions that must otherwise be done with random logic or via a sequence of microinstructions.

PLA's do not lend themselves to dynamic alternations; there is no natural addressing mechanism for each of the make-or-break points in the PLA structure. A dynamically alterable component that could be used much like a PLA is an associative memory (Gardner [5]). The associative memory can be used, for example, in emulation to assist in the decoding of a target machine language instruction. This could be accomplished by loading into each input match field of the associative memory the bit pattern corresponding to one type of instruction format; its corresponding output response field is then loaded with the particular microcode subroutine and its appropriate calling parameters to be executed when its input match field is matched. Since the associative memory is alterable, it could be dynamically reloaded for each different machine language that is being emulated. Another use of an associative memory in emulation would be to use it to specify sophisticated, programmable I/O patterns that will cause an interrupt, e.g., SPS-41 ([18], [2]), thus making it easy

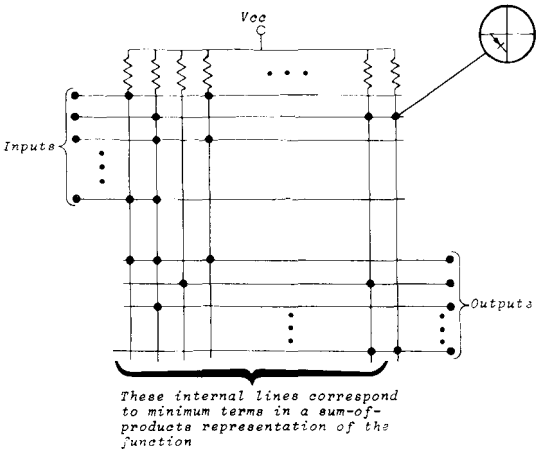

Fig. 5. Organization of a PLA array,

to emulate the interrupt structures of a wide variety of machine languages. However, associative memories have been touted for some time now as a panacea for many problems but have yet to be proven a cost-effective unit.

Another semiconductor device that has made an important impact on microprogrammable processors is the shifter. For example, the Signetics 8243 takes an 8-bit byte as input, shifts it left from zero to seven positions, zeroing out the leftmost bits, and presents the shifted byte on 8 output pins. Using a package like the Signetics 8243 as a basic building block, larger shifters can easily be constructed. The ability of cheaply implementing a fast shifter makes variable-length byte extraction, a common process in emulation, a much easier task.

As will be detailed in the next sections, these technological advances will lead to microprogrammable architectures that are more uniform in structure (less ad hoc), easier to program, and can more efficiently emulate a wide variety of different and more complex instruction sets.

III. The Process of Emulation

As we stated in the Introduction, the right approach to understanding microprogramming is to examine the task it must perform: emulation. This view is especially appropriate given that a major trend in the use of microprogramming over the last few years has been towards more generalized emulation; this trend has occurred in terms of both the number and complexity of machine languages capable of being efficiently emulated on a single microprogrammable processor. Architectures such as the Burroughs B1700 (Wilner [23]), which was designed for efficient emulation of algebraic block-structure languages, and SAAB FCPU (Lawson and Smith [6]), which provides general emulation capabilities in a high-speed processor, are examples of this more general approach to emulation. This trend should be heightened in the future as the variety and complexity of tasks being programmed on a single processor continues to increase. Thus this section spells out in detail the task of emulation and through this discussion indicates the appropriate representational framework and associated operations for efficiently performing an emulation (interpretation). In the next section we tie together our observations on emulation and technology to predict the future evolution of microprogramming.

An interpreter can be characterized as a system that carries out the execution of a program in one representational framework by dynamically mapping each statement (instruction), at the point it is to be executed, into an execution sequence of statements in another environment which realizes the semantics of the mapped statement. Given this definition of interpretation, emulation could be defined as the special case in which the interpreter maps into an environment which is directly executed by the hardware (e.g., in a microprogrammable machine, this environment would be microcode instructions). However, this type of distinction between interpretation and emulation is often very fuzzy. For example, consider the interpretation of the IBM 7090 on the IBM 360/65 which involves the use of two environments (Tucker [20]), i.e., 360/65 microcode and 360 machine code which is, in turn, emulated in the microcode.

This example also points up the difference between actions which are done solely for the sake of interpretation control and information (mapping actions) and those which actually cause the interpreted program to be executed (execution actions) (Mitchell [12]). In this example, mapping actions were programmed in a different representation environment than execution actions, respectively 360/65 microcode and 360 machine language. As will be discussed later, the appropriate environments for expressing these different types of actions and the interface between them is one of the keys to understanding the evolution of microprogrammable processors and how the emulation task differs from other computational tasks. For example, the SAAB FCPU explicitly recognizes the distinction between mapping and execution actions by providing separate, asynchronous processing elements for each type of action.

The other key to understanding the emulation process is based on a static view of this process in contrast to the dynamic view in terms of mapping and execution action so far presented. A static view of emulation comes from understanding the relationship between the two environments the emulator operates on (maps between), i.e., the environment to be emulated (machine language) and the environment directly executed by the hardware (microcode language). An environment consists of: 1) a data and control state image which includes, for example, in a conventional processor, its set of working registers (accumulator, index register, program counter, interrupt register, etc.) and its main memory which hold data and program; 2) a set of primitive actions which can be used to modify and test the state image; and 3) a set of control rules which decide, based on the current status of the control state image, the sequence of primitive actions to execute. The ease with which each of these aspects of an environment to be "interpreted" can be imbedded into the corre-

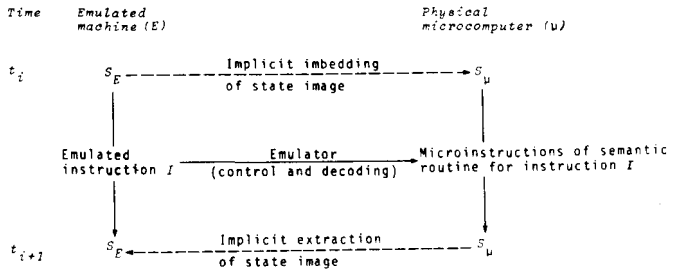

Fig. 6. Commutative state diagram of the conventional emulation process.

sponding aspects of the "execution" environment is one of the main determiners of the efficiency of the interpretation process.

The state diagram of one step in the emulation process, Fig. 6, represents both static and dynamic aspects of the emulation process. The left-hand side of the diagram represents the effect of executing an instruction of the emulated computer on the state image of the emulated computer. The right-hand side represents the sequence of transformations that the microprogrammed processor must perform on its own state image in order to emulate this instruction. In terms of this diagram, efficient emulation occurs when:

1) The data and control state image of target (emulated) machine can be easily imbedded into host (microprogrammed processor) machine.

2) The decoding and control sequencing function can be implemented efficiently. (In conventional instruction sets most of the work involves decoding, but in the emulation of higher level languages, much less of the total effort is spent on decoding.)

3) Microinstruction semantics can operate on the imbedded state image of the emulated machine in the same way the emulated instruction does on its state image.

In the initial use of microprogrammable processors for emulation, each of these aspects that contributes to efficient emulation could be easily attained because the environment(s) to be emulated was known before the design of the processor. This prior knowledge resulted in the design of a microprogrammable processor that had a state image and instruction semantics that were compatible with the emulated environment, and a hard-wired version of the mapping action (control and decoding) between environments. However, as unanticipated and more complex environments began to be emulated a more general approach was needed:

1) a generalized decoding structure;

2) a means of statically reconfiguring, for the duration of an emulation, the state image, control structure, and primitive operation of the execution environment so that these aspects more nearly match those of the emulated environment (Lesser [9]);

3) a means of dynamically modifying the microinstruction semantics based on parameters which are specified in the emulated instruction, i.e., microinstruction as a parameterized template (Lesser [8]). Another way of viewing this requirement is the need for clean, efficient interface between the output of mapping actions and semantics of execution actions.

These requirements for generalized emulation together with the technological advances described in the last section, have led to the following concepts being incorporated into more advanced microprogrammable processors:

1) Flexible bit extraction and manipulation for generalized decoding:
   a) barrel shifter and mask capability (B1700, FCPU);
   b) insertion of data in an arbitrary field of an internal register (FCPU).

2) The concept of residual control as a way of configuring the environment:
   a) set up gating patterns between registers and buses (QM-1);
   b) set up mode of arithmetic, i.e., one's complement, BCD, etc. (B1700, FCPU);
   c) set up word length of data which will be applied to arithmetic operations, memory accesses, and stores (B1700, FCPU);
   d) pseudointerrupt register for embedding control structure of emulated machine (MLP-900, Lawson and Smith [6]).

3) Microinstructions as parameterized templates:
   a) indirect address of general registers, shift count, ALU function (MLP-900);
   b) execute-command (B1700, FCPU).

This list of features when taken as a whole sheds some light on what are the appropriate components of an environment (microprogrammed processor architecture) for general-purpose emulation:

1) a primitive unit of information which is the bit string;

2) a capability for dynamically reconfiguring both the internal and external environment of a microprogrammable processor, i.e., word width, number of general registers, control structures, register bussing connections, arithmetic mode, etc.;

3) a capability for constructing complex-address mapping functions.

These are capabilities that are desirable in almost all types of computer environments. The important point is that they are *crucial* for effective emulation, i.e., these features should be looked at in terms of a matter of degree rather than specific function when comparing with other task domains.

The future of microprogrammable processors will inevitably result in a more generalized version of these concepts as technology permits. However, the aspect of microcomputer architectures that will probably receive the most attention in the next ten years is their control structure. The control structure will play a more important role in future years because one of the dominant trends in programming languages is towards more complex control

structure (i.e., coroutine, data flow models, parallelism, etc.). Inevitably, these more complex control structures in future programming languages will be reflected in the target machine languages.

IV. Hardware and Software Interpretation Techniques

We now consider in this section hardware and software techniques that result from a consideration of the requirements of emulation processes and the possibilities provided by (semiconductor) technology. Then advances in technology can be related to advances in techniques and, hence, to resultant advances in computer systems. Since microprogramming is simply a variation of conventional programming in terms of the desire for generality and ease of coding, advances in microprogramming will likely follow the same pattern already seen in assembly level programming over the last 25 years. This is especially true given the trend towards more complex and varied instruction sets which will require writing of many large emulators, each supporting a complex run-time environment, e.g., PL/I machine, operating systems machine, etc. Since emulation is the major application of microprogramming, specific programming support will be accented. With advances in technology offering more storage capacity and functional processing per unit area (at low cost), hardware structures will become more flexible thus providing a general environment for interpretation and emulation. Since sections of general structures usually go unused in any single application, the cost or cost-performance of generality is rarely acceptable to all. However, the added cost of generality may be borne by improved technology, thus providing the user with more functional capability at a constant cost. In contradistinction, the consumer market for computers requires the lowest possible cost and, so, will trade generality for cost. Here, technology is used to lower cost while keeping the application specific.

In addition to the techniques detailed in the last section for general-purpose emulation, there are also techniques for making it easy to microprogram many large emulators. A list of techniques, in approximate order of increasing generality, include the following.

1) *More high-speed working registers:* Efforts to minimize the size of the processor state is not as strong in microprogrammed processors as it is in more conventional processors.

2) *Larger control stores:* Much of the current involuted character of microprograms is a result of squeezing a complete emulator into a small space (e.g., 256 words) and more reasonable (micro-)programming will be possible with larger control stores.

3) *N-way branches (case statements):* The ability to test several conditions and branch to any of several sections of code which service them.

4) *(Micro)subroutines:* The ability to invoke a function or reference data specified indirectly at a higher level.

5) *Memory management:* Multiprogramming is already a common practice. For example, emulators for central processors, several I/O processors, and microdiagnostics often reside in the same control store. Problems of protection, relocation, and using overlays or paging from backing stores are issues of emerging concern in microprogramming.

6) *(Micro)interrupts:* Useful when multiple emulations are being run on the same processors.

The hardware components which initially supported microprogramming were adequate speed ROM's and multiplexers. ROM's provide tables to encode, decode, and sequence control. Multiplexers extract fields, assemble conditions for testing in parallel, and select control information from registers containing the higher level instructions (indirect control) rather than from the microde (direct control). The next advance came with the availability of high-speed, random-access, alterable memory. With these, microprograms are easily corrected, extended, or swapped for those which provide different functions, for example, machine diagnosis (microdiagnostics). More recent advances in technology have made available low-cost, small-sized shifters, associative memories, PLA's, and decimal arithmetic units. The fast shifter is the most important of these since it easily extracts fields from instructions being interpreted or data from special formats, such as floating point numbers.

To understand the implication of hardware and software techniques it is necessary to consider their application. The next section provides detailed examples. At this point the uses of microprogramming can be decomposed into two dimensions. The first compares designs by the level of language supported. The range includes assembly, intermediate, and high-level languages. The second dimension orders machines by the number of environments supported, typically subdivided in two classes, one and many. Over the last decade the number of environments has increased and their level has risen from the assembly toward the procedure oriented. In the past when several environments were provided, one at a time was selectable from a small, fixed set.

By observing the development of assembly-level programming techniques and by observing the parallel development of microprogramming so far, a reasonable prediction would be the continuation of the trend. If so, the next step will be the generalization and sharing of resources at the microprogram level. First, relocation and protection schemes for alterable microstores will be developed. Then memory management and demand paging (caching) schemes to effect the ability to run large microprograms in comparatively little physical space will be included. The dynamic allocation of microstore address space will probably require a microoperating system with fewer tasks than conventional ones but many similarities with respect to space allocation techniques. To facilitate writing and checkout of so much code, high-level languages designed for microprogramming will be developed, just as

they are now being used more and more as a tool for developing system programs today.

To support these advances in microprogramming software, hardware must be provided. The most important advance on present components is larger microstores made possible by faster and denser memories. As an alternative to a fast, large microstore the cache structure could be used to combine a small, very fast primary microstore with a larger, slower secondary one. Similarly, demand paging requires a fast swapping medium. This might be provided by a high-speed, low-capacity solid-state disk with low latency.

Given the ability to execute so much microcode what use might be found for it? Extrapolating from today's machines and keeping the needs of emulation in mind, one natural application would be to provide multiple programming environments. By this is meant a time-shared computer system whose users divide into classes each requiring the same environment. Some of these would be machine languages for older machines, others would be intermediate, high level (Fortran, PL/I, Cobol), or application oriented. The high-speed shifter is useful in all of these to extract fields. Emulating earlier machines would be made easier by the use of a programmable PLA or associative memory (to replace logic not conveniently embedded in memories due to the large number of inputs). Finally, note that the provision of multiple environments is a problem in multiprogramming and, eventually, as more environments are desired, in time sharing.

V. Machine Species

The various microprogrammed processors can be characterized along evolutionary lines, which in turn roughly correspond to their implementation complexity. One of the earliest computer implementations, Whirlwind I (Everett [3]) formulated the control part as an encoding in a changeable, diode array memory (see Fig. 1). From this Wilkes extended the coding and coined the word "microprogramming" (Wilkes [22]).

A. One-Machine, Integrated Control, and Data Part

With the availability of fast, read only, random access memories, processors with a single, fixed instruction set were designed. These early designs permitted instruction sets with more complex data operations (e.g., multiply, divide, double precision). The most notable design of this type, the IBM System/360 (Blaauw and Brooks [1], Stevens [19]) was actually a set of about 10 computer models implementing the same instruction set covering a performance range of about a factor of 300 and a price range of about a factor of 100. Over half of the models were implemented using microprogrammed control interpreters.

B. A Fixed Group of Conventional Instruction Sets

Given that a single machine instruction set can be implemented in a single processor, the natural extension is to implement several machines. The earliest implementations of multiple instruction sets in a single physical machine used conventional programming. First-generation, cyclic-access, drum-memory computers were "emulated" using higher speed, second- and third-generation computers with RAM's.

An early and extensive use of multiple, fixed machine emulations occurred with the IBM 360 microprogrammed processors as they were used to implement the IBM System/360 instruction set, the 360 I/O processor instruction sets, and several models of earlier IBM computers. However, the design methodology of these computers is not well understood. An approximation to the design process for these machines appears to be: first the primary machine (in this case the 360) is designed; the various other machines to be interpreted are then added to the design by installing their idiosyncrasies (e.g., carry and overflow conditions, state, and special data path breaks) (Tucker [20], Fuller *et al.* [4]), and making it easy to decode their instruction formats.

C. A Variable Group of Conventional Instruction Sets

Given that a single machine can be built that implements several conventional instruction sets (sequentially), can a machine that implements several instruction sets, but on a variable basis, be built? In effect, Standard Computer Corporation attempted such a design in the IC-model 4 and later the MLP 900 [16], [17]. The main goal of the MLP-900 was to implement an IBM 360, together with other undefined machines, e.g., PDP-10, etc. In essence, the machine was designed with much generality using multiple register sets and a two-stage pipeline for instruction fetching and instruction execution. The variable parts, which cannot be emulated easily by sequencing, were brought to a 4-position, multiple-pole, electronic switch, which permitted up to 4 variable parts to be selected. Each variable part consisted of special-purpose logic to assist in a specific emulation. The myriad of details associated with the I/O section (e.g., channels and device state words) add more to the system definition job than the central processor itself.

Currently, there are no commercially viable machines that emulate a set of conventional machines (i.e., architectures) on a variable basis. It appears that the machines to be emulated must be determined *a priori*, in a fixed fashion. Such a machine would permit any one machine to be emulated at a given instant by loading its memory with the information necessary to interpret the target machine. Although this has been done when a large machine interprets another machine, the implication in such a task is that the speed of emulation is essentially that of the target machine. The necessary hardware for this task should be available in the near future and such systems can appear by 1980.

TABLE I
Emulation Subtasks for each of the Major Machine (Language) Levels

Level	Sequencing	Instruction fetch	Instruction decoding	Operand accessing	Data operations
Machine language	conditional branch, subroutines		fixed format	immediate indirect indexed	add, multiply, or, complement
	medium	high	high	medium	low
BASIC and many system prog. languages	iteration		simple syntax	subscripted data structures	sine, cosine, matrix operations
	medium	medium	low	medium	medium
FORTRAN, PL/1, ALGOL, etc.	block structure recursion, coroutines			non-rectangular data structures	I/O formatting
	high	low	low	medium	medium
LISP, SNOBOL, Simula/67, APL, etc.	parallel processes, synchronization, message sys.			linked lists, associative searching	vector operations garbage collection
	high	low	low	high	high

D. A Single Higher Level Language Interpreter Machine

Since the use of higher level algebraic languages (e.g., Algol, Fortran) and more natural textual languages (e.g., Cobol) there has been a substantial interest in the development of hardware that would interpret the languages directly. To date, several machines have been built for single languages (using directly hard-wired techniques), and a number of machines have been microprogrammed to interpret languages directly. These designs have not resulted in any particular insight about direct language interpretation. The implementations execute the object target language faster than the nonmicroprogrammed counterparts, and the speed improvements hold no surprises; the faster memory of the microcode, together with the small, register transfer primitives, provide the improvement.

E. Interpreting Many Languages Directly with a Single Machine

To date, only the Burroughs B1700 (Wilner [23]) has been built with the goal of either the direct interpretation or the compiling and execution of several higher level languages. In that it is able to interpret the various languages, and encode the object code in a space of roughly one-half that of a conventional small computer (the IBM System 3), it is successful. However, its success as measured by execution time is not clear for one would also expact a factor of 2 increase in the execution of the object code. There has been no attempt to compare the execution time on a technology-normalized basis. The B1700 has also been used in the direct interpretation of several conventional machines (e.g., IBM 1401 and Burroughs B2500). Considering all factors, the B1700 appears to be the most general of the microprogrammed machines in existence.[3]

F. Special-Purpose Machines

An especially interesting evolution of microprogrammed machines has occurred for the interpretation of array data for matrix and vector operations, including time-series

[3] As measured by ability to access any bit in memory, to have arbitrary length microcode in any memory, and to operate on variable length field with both binary and BCD formats.

evaluation (e.g., fast Fourier transform). The IBM 2938 is an example of an early connected processor that performs this function. Most recently, a 3-processor system for these operations has been developed and is attached as a peripheral to a conventional minicomputer [18]. The three processors are functionally separated for: fetching data from the attached computer, collecting analog inputs, and storing the results back; moving data from the local array in the right order for the arithmetic part; and the arithmetic part.

Although less exotic than array processors, microprogrammable processors are finding increasing use as efficient ways to implement I/O processors and I/O controllers. The IBM 3830 Storage Control Unit, which controls 3330 disks, is an excellent example of the use of a microprogrammable processor as I/O control unit.

VI. Conclusions and Forecasts

In this article we have reviewed the most important constraints within which successful microprogrammed processors must operate: semiconductor technology and the task of emulation. We think these two constraints will have the strongest influence on the future direction of microprogramming. In fact, as we have stated in the Introduction, there is a good case for dropping the term microprogramming altogether and simply realizing that many processors are designed to efficiently emulate the instruction set of "target" machine architectures.

The major impact of semiconductor technology on microprogramming is to provide large and fast control storage. Moreover, the emergence of LSI microprocessors, programmable logic arrays, and fast shifters will have a significant effect on microprogramming.

Our review of the requirements of the emulation task pointed to a number of central concepts that are required for efficient emulation. Table I summarizes the major dimensions of emulation for different levels of target machines. In each cell the importance of each subtask is indicated and new concepts or capabilities, not used by a subtask at the previous level, are noted.

References

[1] G. A. Blaauw and F. P. Brooks, "The structure of System/360," *IBM System J.*, vol. 3, pp. 119–135, 1964.

[2] J. D. Erwin and E. D. Jensen, "Interrupt processing with queued content addressable memories," in *Proc. 1972 AFIPS Fall Joint Computer Conf.* Montvale, NJ: AFIPS Press, pp. 621–627.

[3] R. R. Everett, "The Whirlwind I Computer," in *Proc. 1951 AIEE–IRE Conf.*, pp. 70–74.

[4] S. H. Fuller *et al.*, PDP-11/40E Microprogramming Reference Manual, Dep. Computer Science, Carnegie-Mellon Univ., Pittsburgh, PA, Tech. Rep., Jan. 1976.

[5] P. L. Gardner, "Functional memory and its microprogramming implications," *IEEE Trans. Comput.*, vol. C-20, July 1971.

[6] H. W. Lawson, Jr., and B. K. Smith, "Functional characteristics of a multilingual processor," *IEEE Trans. Comput.* vol. C-20, pp. 732–743, July 1971.

[7] H. W. Lawson, Jr., and B. Malm, "The DATASAAB flexible central processing unit (FCPU): Background, concepts, basic design, and applications," Data SAAB, Linkoping, Sweden, 1973.

[8] V. R. Lesser, "An introduction to the direct emulation of control structures by a parallel micro-computer," *IEEE, Trans. Comput. (Special Issue on Microprogramming)*, July 1971.

[9] ——, "Dynamic control structures and their use in emulation," Ph.D. dissertation, Rep. CS 309, Computer Science Dep., Stanford Univ., Stanford, CA, Sept. 1972.

[10] E. G. Mallach, "Emulation: A survey," *Honeywell Computer J.*, vol. 6, pp. 287–297, 1973.

[11] T. McWilliams, S. H. Fuller, and W. Sherwood, "Designing a PDP-11 with Intel 3000 bit slices," Dep. Computer Science, Carnegie-Mellon Univ., Pittsburgh, PA, Tech. Rep., June 1976.

[12] J. G. Mitchell, "The design and construction of flexible and efficient interactive programming systems," Computer Science Dep., Carnegie-Mellon Univ., Pittsburgh, PA, June 1970.

[13] Rockwell Programmable Logic Array (PLA), Rockwell Device Division, Rockwell International, Anaheim, CA, Pub. 15900N11, Aug. 1973.

[14] R. F. Rosin, "Contemporary concepts of microprogramming and emulation," *Computing Surveys*, vol. 1, pp. 197–212, 1969.

[15] R. M. Spagler, "BASIC-language model 30 can be a calculator, computer, or terminal," *Hewlett-Packard J.*, Dec. 1972.

[16] Inner Computer—Model 9, Principles of Operation, Standard Computer Corp., Los Angeles, CA, 1968.

[17] IC-9000 Processor Functional Description, Standard Computer Corp., Los Angeles, CA, Form 9001-3, 1969.

[18] SPS-41 User's Manual, Signal Processing Systems, Inc., Waltham, MA, 1972.

[19] W. Y. Stevens, "The structure of System/360: Part II System implementations," *IBM Systems J.*, vol. 3, pp. 136–143, 1964.

[20] S. G. Tucker, "Emulation of large systems," *Comm. ACM*, vol. 8, pp. 753–761, Dec. 1965.

[21] M. V. Wilkes, "The growth of interest in microprogramming: A literature survey," *Computing Surveys*, vol. 1, pp. 139–145, 1969.

[22] ——, "The best way to design an automatic machine," in *Proc. Manchester Univ. Computer Inaugural Conf.*, July 1951, London, England: Ferrante, 1951.

[23] W. T. Wilner, "Design of the Burroughs B1700," *Proc. AFIPS FJCC*, vol. 41, pp. 489–497, 1972.

Samuel H. Fuller (S'67–M'72) received the B.S.E. degree in electrical engineering from the University of Michigan, Ann Arbor, in 1968, the M.S. and Ph.D degrees from Stanford University, Stanford, CA, in 1969 and 1972, respectively.

He is an Associate Professor of Computer Science and Electrical Engineering at Carnegie-Mellon University, Pittsburgh, PA. His research interests include topics in computer architecture and the performance evaluation of computer systems. He is currently involved in the measurement and evaluation of C.mmp, a multiminiprocessor computer system, and the design of new multiprocessors based on the emerging microcomputer technology. He is the author of *Analysis of Drum and Disk Storage Units* (Springer-Verlag) and a coauthor of *Introduction to Computer Architecture* (SRA). He is an Editor of the Computer Systems Department of CACM and has served as a member of the IEEE Computer Society Task Force on Computer Architecture.

Victor R. Lesser was born in New York, NY, on November 21, 1944. He received the A.B. degree in mathematics from Cornell University, Ithaca, NY in 1966, and the M.S. and Ph.D degrees in computer science from Stanford University, Stanford, CA, in 1969 and 1972, respectively.

He was a Research Associate in the Computer Science Department at Carnegie-Mellon University, Pittsburgh, PA, from 1972 to 1974. He currently holds the position of Research Computer Scientist there. In addition to work on system organizations for speech understanding, his research interests include computer architecture, particularly multiprocessor systems and microprogramming.

C. Gordon Bell (S'54–SM'67–F'74) is Vice-President of Engineering for Digital Equipment Corporation, Maynard, MA. He has been on leave as Professor of Electrical Engineering and Computer Science at Carnegie-Mellon University, Pittsburgh, PA. He was previously Manager of Computer Design for Digital from 1960–1966. During that time he was responsible for DEC's PDP-4, -5, and -6 computers. He consulted for Digital in 1966–1972 while at Carnegie-Mellon University, working on various computers and products including the PDP-11. He has worked in the computer field on computer architecture, modularity of design, multiprocessors, and applications. His publications include *Computer Structures* (McGraw-Hill), coauthored with Allen Newell; *Designing Computers and Digital Systems, Using PDP-16 Register Transfer Modules* (Digital Press) with John Grason and Allen Newell; and several papers.

In addition to his industrial interests, he has served on the U.S. Government as a member of three COSINE committees of the National Academy of Sciences for computer engineering education, and the National Science Foundation, Office of Computing Activities. He is a Department Editor for the *CACM*.

Charles H. Kaman (M'69) received the B.A. degree in mathematics from Harvard University, Cambridge, MA, in 1965, and the M.S. and Ph.D. degrees in system sciences from the Polytechnic Institute of Brooklyn, New York, NY, in 1967 and 1974, respectively.

In 1969 he joined the Research and Development Group of the Digital Equipment Corporation, Maynard MA. He has worked in the areas of processors and peripheral controller diagnostics.

Mr. Kaman is a member of the ACM and SIAM.