MICROPROGRAMMING AND ITS RELATIONSHIP TO EMULATION AND TECHNOLOGY

Samuel H. Fuller and Victor R. Lesser
Carnegie-Mellon University
Pittsburgh, Pennsylvania

C. Gordon Bell and Charles Kaman
Digital Equipment Corporation
Maynard, Massachusetts

May 16, 1974

This report will appear in the 7th Annual Microprogramming Workshop in Palo Alto, California, October 1.

This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (contract F44620-73-C-0074) and is monitored by the Air Force Office of Scientific Research. This document has been approved for public release and sale; its distribution is unlimited.

ABSTRACT

The structure of microprogrammed processors, and microprogramming in general, is largely determined by two facts: the state of (semiconductor) technology and the task of emulation. 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.

The other main theme of this article is that it is a fruitless exercise to try to characterize and understand microprogramming in terms of how it differs from 'regular' programming. The right approach to understanding microprogramming is to recognize that it is primarily applied to the task of emulation (interpretation). Through this approach the evolution of microprogramming independent of a particular technology and type of instruction set being emulated, will be reviewed and future trends indicated.

1. 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. The remainder of this article then uses 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 it is a fruitless exercise to try to characterize and understand microprogramming in terms of how it differs from 'regular' programming. The futility of this approach can be seen by the numerous, contradictory definitions on microprogramming in the literature [Rosin, 1969; Wilkes, 1969; Mallach, 1972]. 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 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 of microprogramming comes from the fact that it has been used in two very different ways: (1) in a technological manner to economically implement a complex instruction set or a small number of different instruction sets on a single processor, and (2) in a software manner 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 was the dominant justification for the development of microprogrammable processors in the 1960's. But as the cost of software began to become the major cost of a computer system, the use of microprogramming as a technique for making a computer more convenient to program has and will continue to become the more important application.

The right 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 evoluton 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 3, but it will be useful here in the introduction 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 he has no way of knowing the architecture of the processor. On the other hand there are the BASIC programmable calculators available from Hewlett-Packard [Spagler, 1972] and Wang Laboratories that operate as BASIC machines: the input keys and the 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 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 term 'microprogramming' altogether and simply use '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, this article then discusses specific hardware and software techniques for emulation. A number of different types of microprogrammed processors are also included as examples.

2. 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, 1951] and by Wilkes in his original paper on microprogramming [Wilkes and Stringer, 1953]. Figures 2.1 and 2.2 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. about 35,000 bits of control storage are required to implement the full PDP-11/40 architecture while the Whirlwind I had only 4,800 'bits' in its control store). It is important to realize that both of these structures are just the control part of the processor and are an alternative to conventional sequential control circuits as shown in Figure 2.3. It was not until the middle and late 1960's that integrated-circuit technology advanced to the point that economic read-only-memories (ROMs) and read-write memories (RAMs) 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, non-semiconductor technique to implement its control store. These ingenuous, but admittedly cumbersome and costly techniques could be laid aside when the IBM 370 series of machines were implemented since integrated circuit technology had advanced to the stage that semiconductor control stores were reliable. Figure 2.4 illustrates the basic structure of current microprogram control units.

Semiconductor memories suitable for control stores in microprogrammed processors are now at the stage where 256 bit/package RAMs and 1K (1024) bit/package ROMs are in wide use in present processors and 1K RAMs and 4K ROMs are being designed into the newer processors. 4K RAMs and 16K ROMs have been announced and are available in limited quantities, but in general they are too slow to be seriously considered for 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 roghly doubled every one to two years. For example, the 4K bit/package RAM (13,000 devices) was introduced roughly two and one half years after the 1K bit (4000) RAM. There is every reason to believe that this trend will continue for at least the next four to six 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.

Memory arrays are not the only development 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 shifters. The basic structure of a PLA is shown in Figure 2.5. It is a two-level combinatorial logic circuit that is 'wired' for a specific application by the masking, or metalization, that is used. 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 input/output terminals [Rockwell, 1973]. A 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 a ROM. Instruction decoding is an example of a combinatorial function amenable to minimization techniques and hence PLAs will be very useful for providing the decoding of instructions that must otherwise be done with random logic or via a sequence of microinstructions.

PLAs 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. Associative memories have been toted for some time now as a panacea for many problems but have yet to prove to be a cost effective unit. However, as the number of pins per package becomes more of a limitation than the complexity of the semiionductor circuit itself, associative memories may become viable components, e.g. the SPS-41 used an associative memory to specify sophisticated, programmable I/O patterns that will cause an interrupt [SPS,1972].

The other non-memory semiconductor device that has recently made an important impact on microprocessors is the shifter. For example, the Signetics 8243 takes an eight bit byte as input, shifts it left from zero to seven positions, zeroing out the leftmost bits, and presents the shifted byte on eight 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 technology advances will lead to microprommable 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.

3. 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. 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.

Our present discussion of emulation and microprogramming is especially appropriate given the view a major trend in microprogramming is towards more generalized emulation in terms of both the number and complexity of machine languages capable of being efficiently emulated on a single microprogrammable processor. Recent architectures such as the Burroughs B1700 [Wilner, 1972], which was designed for efficient emulation of algebraic block-structure languages, and SAAB FCPU [Lawson and Malm, 1973], 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 continue to increase.

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 realize 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. 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, 1965], 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, 1970]. 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, i.e. the emulated and execution environment. 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 corresponding aspects of "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, Figure 3.1, represents both static and dynamic aspects of the emulation process. The lefthand side of the diagram represents the effect of executing an instruction of the emulated computer on the state image of the emulated computer. The righthand 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 imbedded state image of 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 hardwired 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 (see Figure 3.2) [Lesser, 1972];

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 templates [Lesser, 1971]. 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 and 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. 1'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. pseudo-interrupt register for embedding control structure of emulated machine (MLP-900, [Lawson and Smith 1971])

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 shed 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 environment. The important point is that they are <u>crucial</u> 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 10 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 machine languages that will be compiled into.

4. HARDWARE AND SOFTWARE INTERPRETATION TECHNIQUES

To predict the future of microprogramming it is necessary to understand how hardware and software techniques are used in effecting interpretation. 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 twenty five years. This is especially true given the trend toward more complex and varied instruction sets which will require writing of many large emulators, each supporting a complex run time environment, e.g. PL/1 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:

1. <u>More high-speed working registers</u>. Efforts to minimize the size of the processor state is not as strong in microprogrammed processors as it is in more conventional processors.

2. <u>Larger control stores</u>. 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. <u>N-way branches (case statements)</u>. 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 ROMs and multiplexors. ROMs provide tables to encode, decode, and sequence control. Multiplexors 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 microcode (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, PLAs, 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 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 micro-operating 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.

5. 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, 1951], formulated the control part as an encoding in a changeable, diode array memory (see Figure 2.1). From this Wilkes and Stringer extended the encoding, and coined the word "microprogramming" [Wilkes and Stringer, 1953].

5.1 One-Machine, Integrated Control and Data Part

With the availability of fast, read only, random-access memories computer processors with a single, fixed instruction-set were designed. These early designs permitted instruction-sets wih more complex data-operations (e.g. multiply, divide, double precision). The most notable design of this type, the IBM System/360 [Blaauw and Brooks, 1964; Stevens, 1964] was actually a set of about 10 computer models implementing the same instruction set covering a performance range of about 300 and a price range of about 100. Over half of the models were implemented using programmed control interpreters.

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 random access memories.

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 input-output processor instruction-sets, and several models of earlier IBM computers. The design methodology of these computers is not well understood outside IBM. 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, special data path breaks) [Tucker, 1965].

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 [SCC 1968; SCC, 1969]. 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 by direct wiring on a plugboard array. Although such an approach is of academic interest, the mechanical aspects of the plugboarding preclude the machine from being interesting in a production or economic sense. The myriad of details associated with the input-output section (e.g. channels, device state words, and transitions) add to the system definition job more than the central processor.

Currently, there are no commercially viable machines that emulate a set of other conventional type machines 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 a 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. It appears the necessary hardware for this task will be available in the near future and that such systems can exist by 1980.

5.4. 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 hardwired 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 non-microprogrammed counterparts, and the speed improvements hold no surprises; the faster memory of the microcode, together with the small, register transfer primitives, provide the improvement.

5.5 Interpreting Many Languages Directly with a Single Machine

To date, only the Burroughs' B1700 [Wilner, 1972] has been built with the goal of either the direct interpretation or compile 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, the execution time is not clear; one would expect a factor of two increase in the execution of the object code, too. 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.

5.6 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 evaluation (e.g., fast fourier transformation). Although there were several early laboratory processors, IBM's 2938 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 [SPS, 1972]. 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.

6. 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. It is our view that these constraints more strongly influence the direction future of microprogramming per se. In fact, as we stated in the introduction, there is a good case for dropping the term microprogramming altogether and simply realize 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. However, the emergence of programmable logic arrays and fast shifters is also bound to have a significant effect on microprocessors. It is relatively unclear at this point exactly what effect the processor-on-a-chip will have. Implementing the entire processor on a single semiconductor chip eliminates much of the flexibility available in constructing processors from MSI/LSI components, but, on the other hand, provides a complete processor at a very low cost. If processors-on-a-chip become sufficiently popular, emulation will come into heavy use as these primitive processors are surrounded by emulation routines to transform them into processors with which we are comfortable working.

----

*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.

Our review of the requirements of the emulation task pointed to a number of central concepts that are required for efficient emulation.

Table 6.1 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

Blaauw, G. A. and Brooks, F. P., "The Structure of System/360," IBM System J. 3,2 (1964), 119-135.

Everett, R. R., "The Whirlwind I Computer," AIEE-IRE Conf., (1951), 70-74.

Lawson, H. W., Jr. and Smith, B. K., "Functonal Characteristics of a Multilingual Processor," IEEE Trans. Comput., C-20, July 1971, 732-743.

Lawson, H. W., Jr. and Malm, B., "The DATASAAB Flexible Central Processing Unit (FCPU): Background, Concepts, Basic Design, and Applications," Data SAAB, Linkoping, Sweden, 1973.

Lesser, V. R., "An Introduction to the Direct Emulaton of Control Structures by a Parallel Micro-Computer," Transactons on Computers (special issue on Micro-programming), IEEE, July 1971.

Lesser, V. R., Dynamic Control Structures and Their Use in Emulation, Ph.D. thesis, Report No. CS 309, Computer Science Department, Stanford University, Stanford, Calif., September, 1972.

Mallach, E. G., "Emulation: A Survey," Honeywell Computer Journal, 6,4 (1973), 287-297.

Mitchell, J. G., The Design and Constructon of Flexible and Efficient Interactive Programming Systems, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pa., June 1970.

Rockwell Programmable Logic Array (PLA), Pub. No. 15900N11, Rockwell Device Division, Rockwell International, Anaheim, Calif., August, 1973.

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

Spagler, R. M., "BASIC-Language Model 30 Can Be a Calculator, Computer, or Term," Hewlett-Packard Journal, December, 1972.

SCC, Inner Compuer--Model 9, Principles of Operation, Standard Computer Corp., Los Angeles, Calif., 1968.

SCC, IC-9000 Processor Functonal Description, Form No. 9001-3, Standard Computer Corp., Los Angeles, Calif., 1969.

SPS-41 User's Manual, Signal Processing Systems, Inc., Waltham, Mass., 1972.

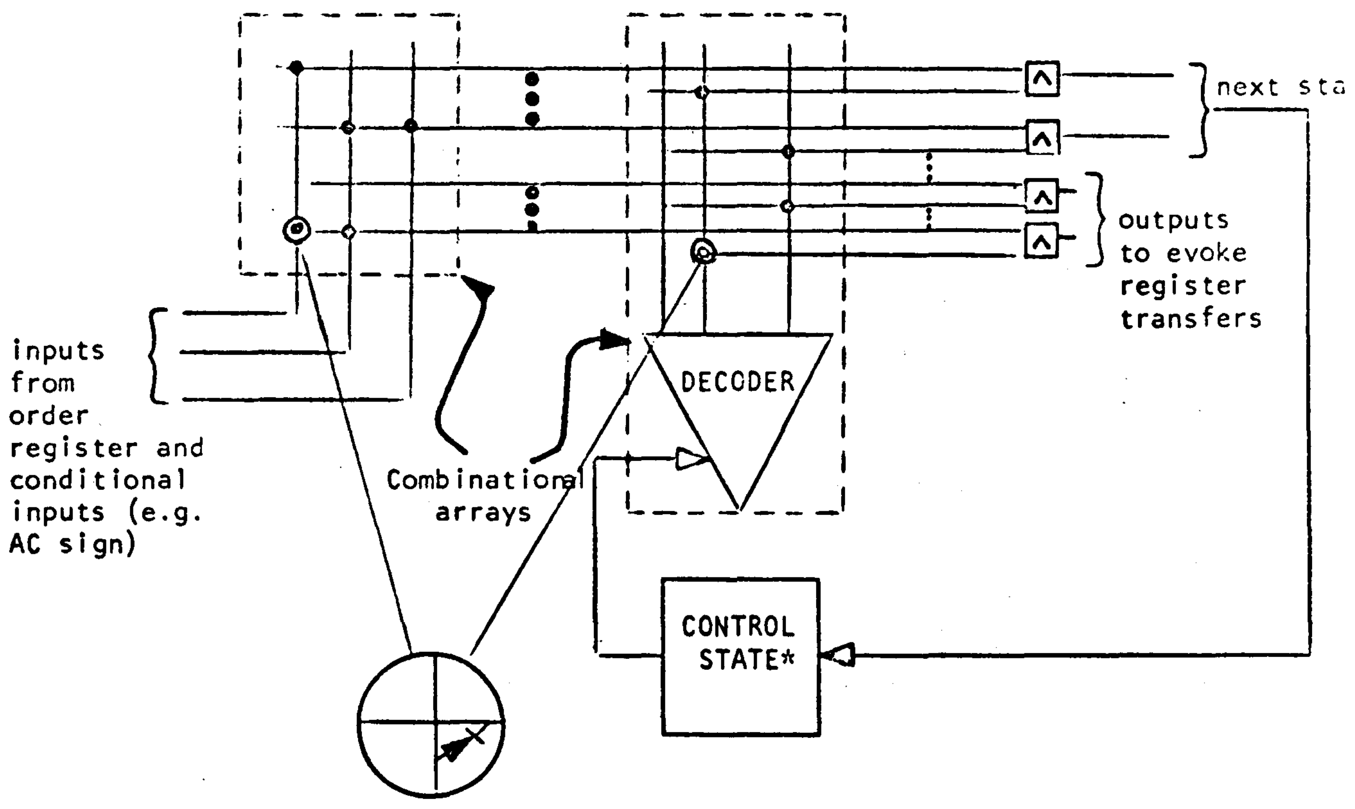
Stevens, W. Y., "The Structre of System/360: Part II--System Implementations," IBM Systems Journal, 3,2, 1964, 136-143.

Tucker, S. G., "Emulation of Large Systems," Comm. ACM, 8,12, December, 1965, 753-761.

Wilkes, M. V., "The Growth of Interest in Microprogramming: A Literature Survey," Computing Surveys, 1,3, (1969), 139-145.

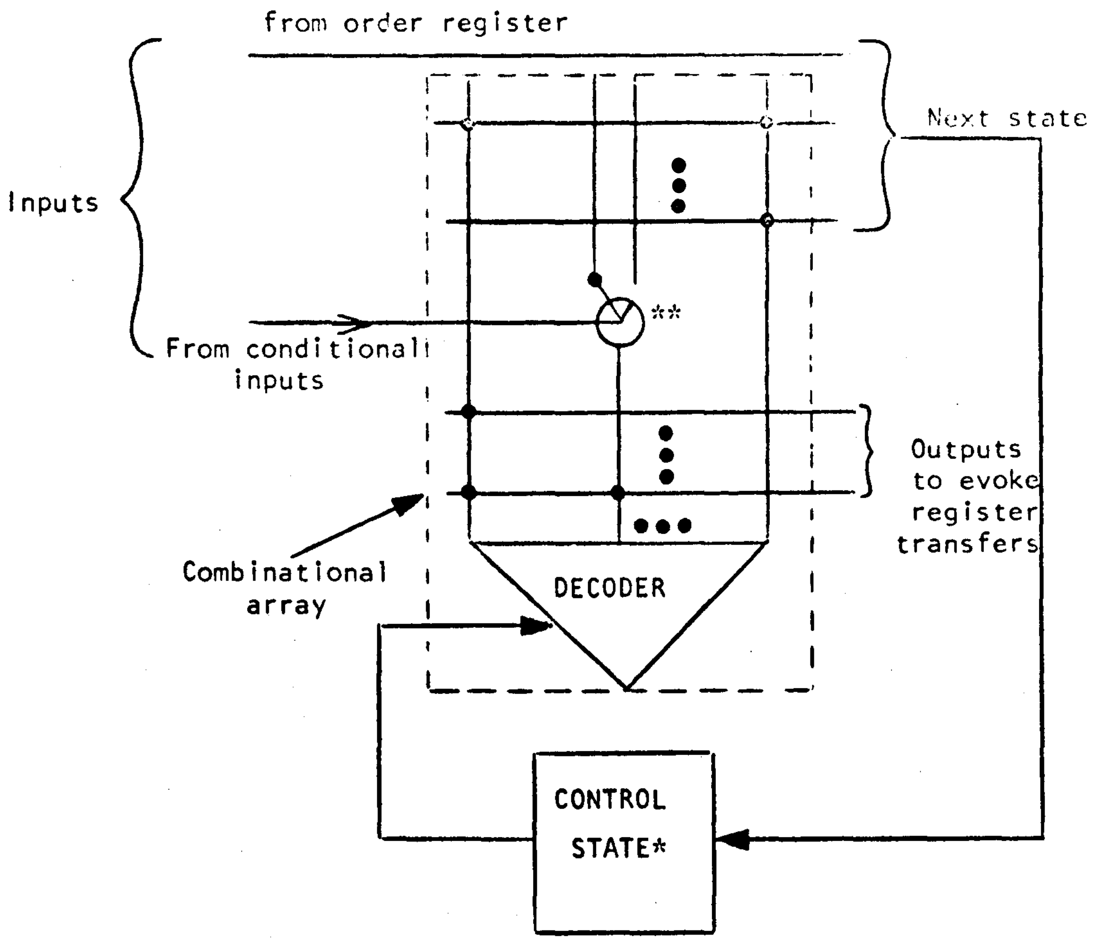
Wilkes, M. V. and Stringer, J. B., "Microprogramming and the Design of the Control Circuits " Proc. Cambridge Phil. Soc., Part 2, 49, April, 1953, 230-238.

Wilner, W. T., "Design of the Burroughs B1700," Proc. of AFIPS FJCC, 41, 489-497.


next sta
outputs to evoke register transfers
inputs from order register and conditional inputs (e.g. AC sign)
Combinational arrays
DECODER
CONTROL STATE*

*Time State Generator.

Figure 2.1 MIT *Whirlwind control unit*.



* Microprogram address register, or master-slave registers.

**

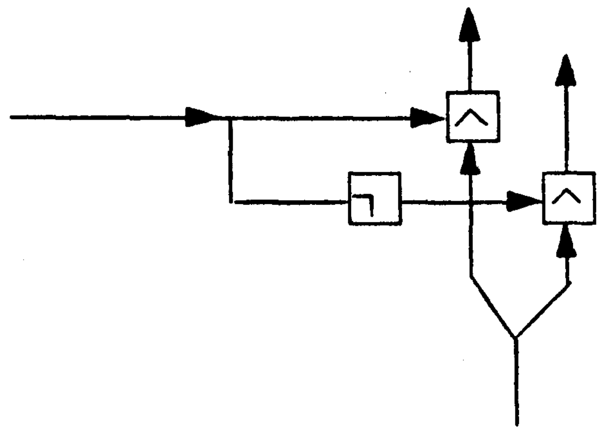


Figure 2.2 Wilkes-Stringer Micro-control unit

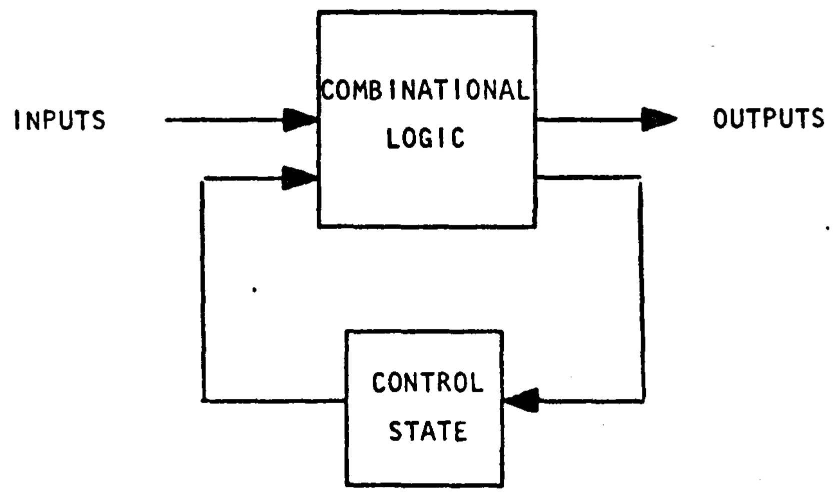
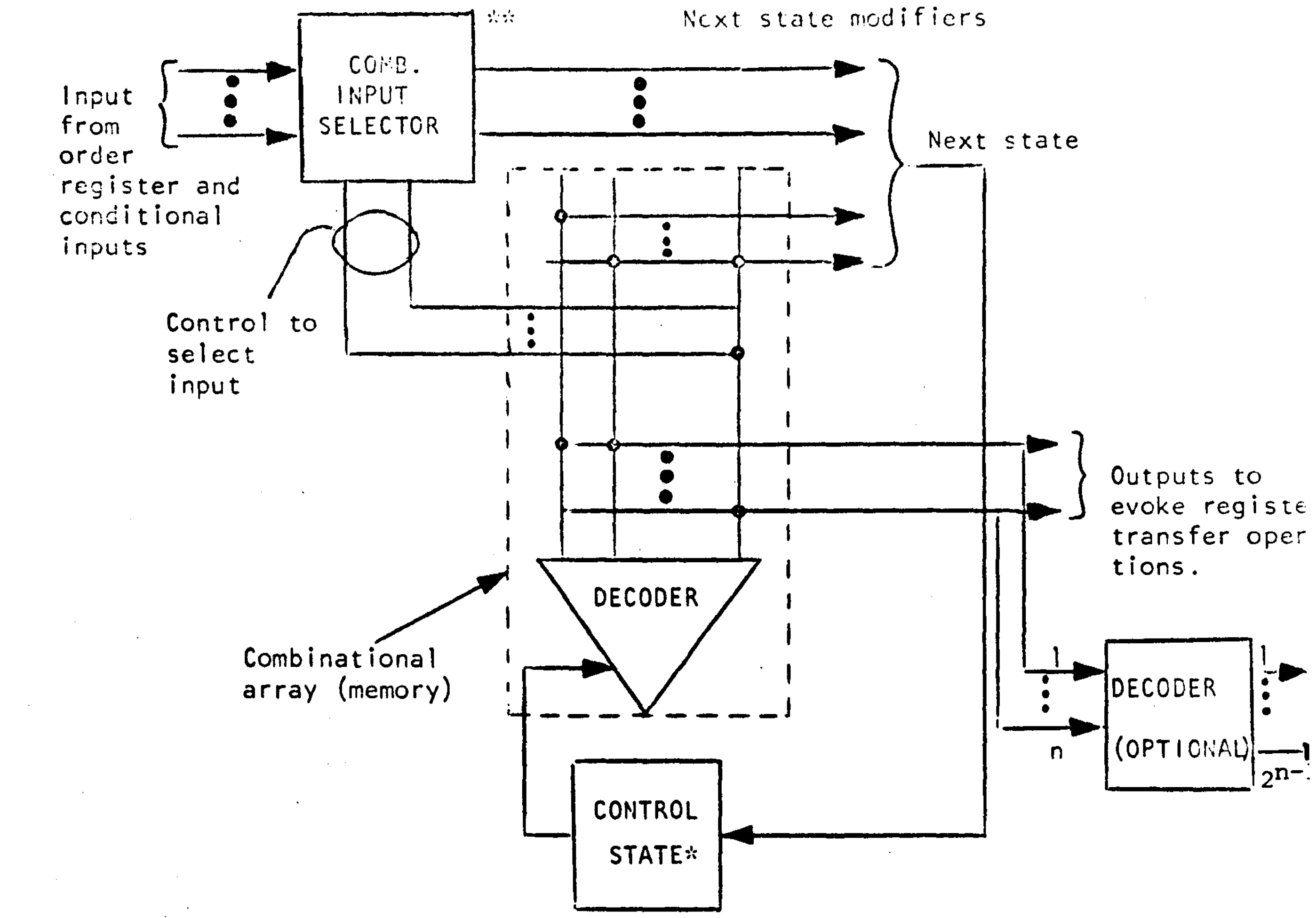


Figure 2.3 Conventional sequential **control** *circuit*


**
Next state modifiers
Input from order register and conditional inputs
COMB. INPUT SELECTOR
Next state
Control to select input
Outputs to evoke registe transfer oper tions.
DECODER
Combinational array (memory)
1
n
DECODER (OPTIONAL)
1
2n-
CONTROL STATE*

Microinstruction word layout

NEXT STATE	MODIFY NEXT STATE	OUTPUTS

* Microprogram address register.
** One possible implementation.

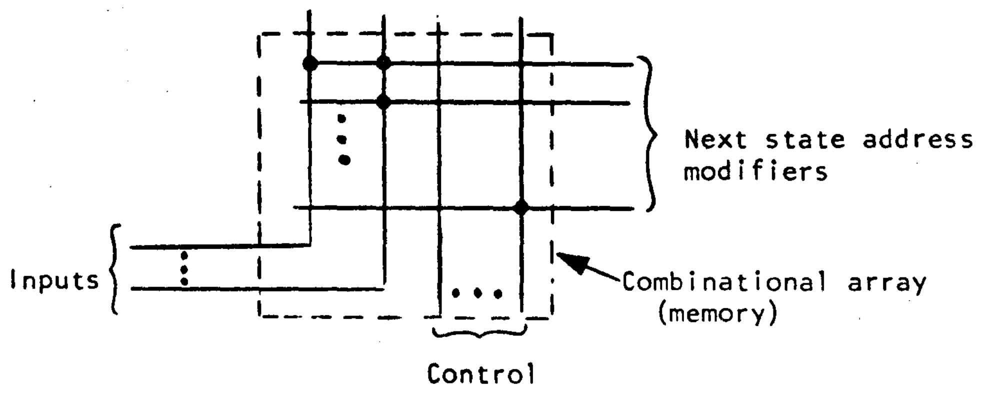


Figure 2.4 Current microprogram control units

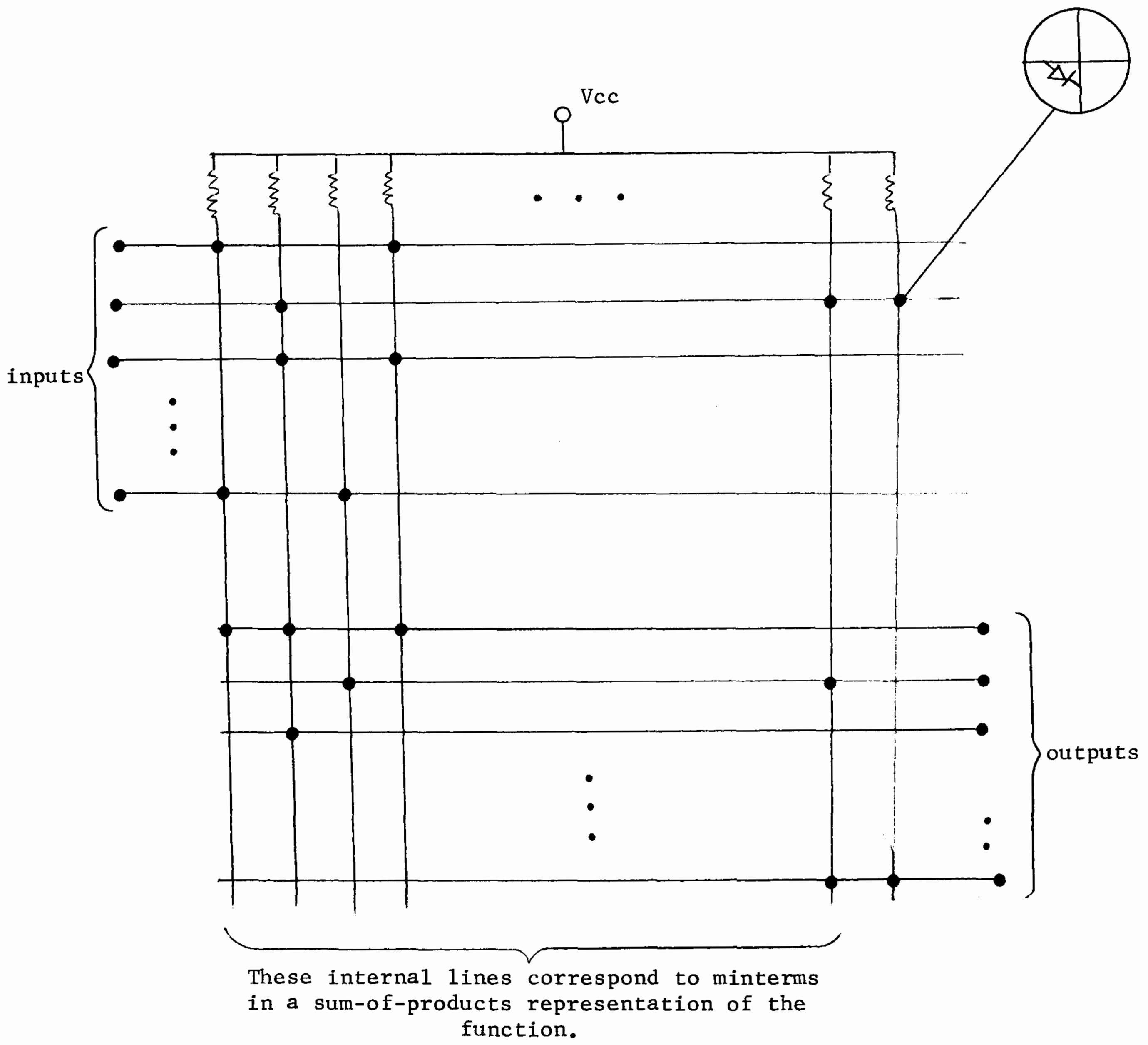

Vcc
inputs
outputs
These internal lines correspond to minterms in a sum-of-products representation of the function.

Figure 2.5. A programmable logic array (PLA).

$$\begin{tikzcd}[column sep=huge]
\underline{\text{Emulated Machine}} & \underline{\text{Virtual Microcomputer System}} & \underline{\text{Physical Microcomputer System}} \\
S^{I}_{\text{Emulated machine}} \arrow[r, dashed, "\text{implicit imbedding}", "\text{of state image}"'] \arrow[d, no head] & S^{I}_{V_{\mu}} \arrow[r, "\text{mapping}", "\text{hardware}"'] \arrow[d, no head] & S^{I}_{\mu} \arrow[d, no head] \\
\text{Emulated instructions} \arrow[r, "\text{emulator}"] \arrow[d] & \mu\ \text{instructions} \arrow[r, "\text{mapping hardware}", "\text{(residual control, indirect addressing)}"'] \arrow[d] & {} \arrow[d] \\
S^{I+1}_{\text{Emulated machine}} & S^{I+1}_{V_{\mu}} \arrow[l, dashed, "\text{implicit extraction}"', "\text{of state image}"] & S^{I+1}_{\mu} \arrow[l, "\text{mapping}"', "\text{hardware}"]
\end{tikzcd}$$

Figure 3.2. A New View of Commutative State Diagram of Emulation Process

Table 6.1. Emulation subtasks for each of the major machine (language) levels.

Level	Sequencing	Instruction fetch	Instruction Decoding	Operand Accessing	Data Operations
Machine Language	conditional branch sub-routines		fixed format	immediate indirect indexed	add, multiply, or, comple-ment
	medium	high	high	medium	low
Basic	iteration		simple syntax	subscripted	sine, cosine, matrix ops.
	medium	medium	low	medium	medium
Fortran, PL/I, Algol,etc.	block struc-ture recur-sion, co-routines			non-rectangu-lar data structures	
	high	low	low	medium	medium
Lisp, Snobol, etc.				linked lists	
	high	low	low	high	high