This guide to the literature on microprogramming is preceded by an exposition intended for the less knowledgeable reader.

Microprogram control is seen as a form of simulation in which primitive operations are combined and sequenced so as to imitate the characteristics of a desired machine. Discussed are such design considerations as microword formats, performance, writable control stores, and the relationship between microprogramming and software reliability.

Readings in microprogramming

by P. M. Davies

Two decades ago, shortly before the first commercial U.S. installation of a general-purpose digital computer, M. V. Wilkes coined the term "microprogramming" and articulated the basic principles in an address at the University of Manchester in England. His intent was to offer a more orderly substitute for the ad hoc process of designing controls in digital computers. Interest spread rapidly in computer development circles, but, except for a few instances of user-microprogrammed machines designed a decade or so ago, exploitation of the technique remained generally within the province of the hardware designers. Within the last few years, however, greater understanding of digital computing processes, combined with manufacturing technologies rendering highspeed changeable controls economically feasible, have led to a resurgence of interest in computers whose controls may be modified during normal use.

The purpose of this paper is to offer a guide to the literature on microprogramming that has developed over the last twenty years. It is intended both for those who have interest but limited knowledge of the subject and those whose knowledge is more than casual. The paper is divided into two parts: an expository section for the less knowledgeable reader and an annotated bibliography.

Many of the articles tacitly assume that the reader possesses a computer engineering or design background, which puts some readers at a disadvantage. The exposition in this paper is orga-

nized as a primer, rather than an exhaustive dissertation, in an attempt to alleviate this disadvantage and to establish some initial perspective. Microprogramming is introduced by considering the concept of simulation, the structure of digital computers, and the need for control of their binary logic. Specific attention is then given to microprogram control and to the format of control words. Cost and performance are considered next, followed by discussions of writable control stores and of the relationship between microprogramming and software reliability.

Simulation and microprogramming

An introduction to microprogramming may best be made through considering the idea of simulation. Suppose that there is at hand some system, say an IBM 7094, which we will call a *host* system. This system has a set of external attributes that define it functionally: an instruction set, storage media (main storage, disks, tapes, etc.), interruption system, channel configuration, and so on. Programs written in this system's language—7094 machine code—can be directly executed. Suppose further that there is a program written for another system, an IBM 7040, perhaps, that we desire to have executed on the 7094. A set of 7094 machine code programs can be written that will accept 7040 machine code and, executed on the 7094, will create the same results that a 7040 system would if it were executing the 7040 code directly. The combination of the 7094 host system and 7094 programs can be considered to constitute a *virtual* system, that is, functionally, a 7040 (time dependencies excepted, of course). We can call the 7040 the *target* system.

Thus a virtual system can be created based on a host system whose external attributes differ from those of the target system through the agency of programs that supplement or transform the host's attributes. The process is generally called simulation of the target system, in this case the 7040 system, on the host system. In general, to create any virtual system, we need only a host containing sufficient facilities for simulation programs to be constructed. The efficiency of the process depends upon the degree of match between host and target system attributes.

The idea of a host system being used to create a virtual system can be extended into the internal workings of a computer. If we were to take the covers off a 7094 central processing unit and delve inside, for example, we could separate its internals into two segments: functional and control. The functional segment includes the facilities that hold, route, and transform information, while the control segment embraces the logic that directs the activities of the functional parts so that, by orderly sequences of internal processes, the entire conglomeration creates the effects

defined by the 7094 CPU specifications. The two segments are linked by control signals flowing to the functional parts from the controls and by information returning to the controls that describes the status of functional activity and the environment.

An imaginary line enclosing all the functional parts but excluding the controls could now be considered the boundary of a host system at the hardware level. The external attributes of this host could be written down (although it might be an arduous task); corresponding to each control signal intersected by the imaginary boundary there is some primitive function in the operational parts, and the set of all such primitives defines the hardware-level language of the host. The control signals are activated so that, through the mediating agency of the control logic networks, the hardware-level host is used to create a virtual 7094 CPU. Note that if the host hardware remains unmodified but the control logic is changed, some virtual CPU other than a 7094 can be created.

In this example, logic networks, rather than simulation programs, are used to manipulate the host to produce a virtual CPU. The important point to note is that the combination and sequence of control signals applied to the host determine the attributes of the virtual CPU; any other adequate agent, not necessarily logic networks, can be used to generate the control signal combinations and sequences without affecting the virtual attributes. If a programmable storage array is used to generate these control signals, the virtual CPU is said to be *microprogrammed.*

Basic control concepts

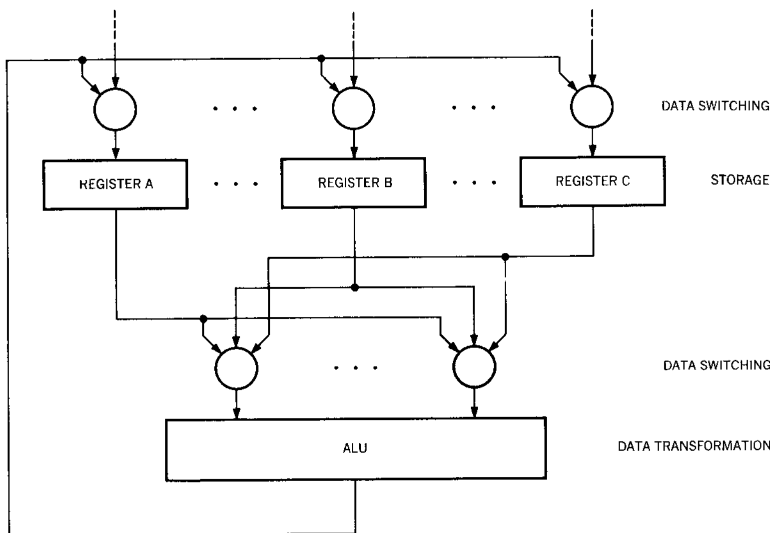
The following paragraphs give a more detailed explanation of what is involved in the control of binary logic.

data-flow section

The working nucleus of a digital computer is the central processing unit and its associated main storage unit. The CPU, as we have seen, can at least conceptually be divided into two parts. To bring our terminology in line with that commonly used, we will now refer to the functional segment as the data-flow section.

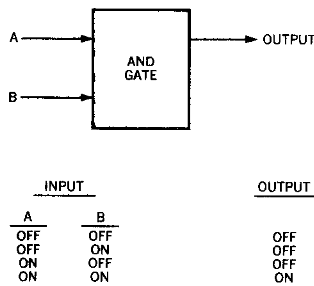
The data-flow section is constructed of binary logic circuits. The detailed characteristics of these circuits may vary from machine to machine, but the basic building blocks are usually AND's, OR's, and INVERT's or simple combinations of these functions, such as NAND's, NOR's, and EXCLUSIVE OR's. Groups of such blocks are interconnected to form storage elements, data-routing switches, and data-transformation networks. The storage part of the data-flow section includes such things as main storage interface registers, operand and address registers, and registers for recording temporary information and ancillary conditions (presence or ab-

Figure 1 CPU data-flow section



sence of arithmetic overflow, pending interrupt requests, and so on). The data-transformation part executes the hardware-level functional repertoire of the CPU. This repertoire may not, and, in fact, usually does not correspond exactly to the machine-code repertoire the CPU is intended to execute, but rather consists of a limited set of basic operations, such as AND, OR, COMPLEMENT, ADD, SHIFT, and so on with which more complex operations can be synthesized. These data-transformation networks are generally called the *arithmetic and logical unit*, or ALU. There may also be networks that perform specialized internal operations, such as incrementing and comparing addresses and counts. The remainder of the data-flow section contains switching networks to route data onto the appropriate paths interconnecting the storage and transformational parts.

Figure 1 shows in abstract form part of a CPU data-flow section. In this example, any of the three registers can be switched onto the ALU input path; the ALU output can be switched onto the input path to any of the registers; and we will assume that the ALU can perform several distinct operations. It is evident that for this data-flow segment to perform a coherent operation, say to ADD the contents of registers A and B with the result deposited in register C, control signals must be applied to the data switches to choose the proper data routing as well as to instruct the ALU to perform the selected operation. Furthermore, since a finite time is required for the circuits to switch and propagate signals through the networks, these control signals must remain in effect long enough to allow completion of the operation. Generally a time is chosen in which all the usual operations can be completed.



INPUT		OUTPUT
A	B	
OFF	OFF	OFF
OFF	ON	OFF
ON	OFF	OFF
ON	ON	ON

Figure 2 AND gate

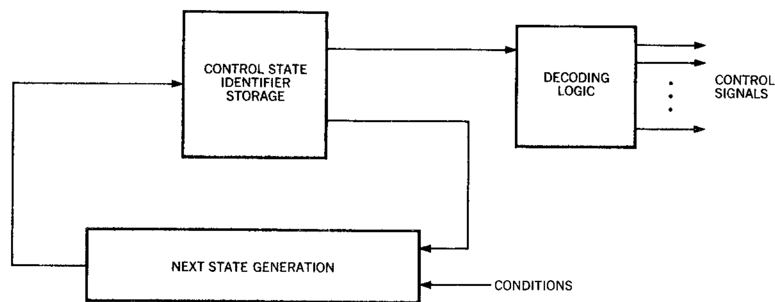


Figure 3 Generalized control section

This time is then used as the basic increment, or *cycle*, of data-flow operation. More complex functions are then implemented in consecutive sequences of this basic cycle.

AND circuits are the fundamental mechanisms by which control is exerted. Consider a two-input AND circuit, such as is shown in Figure 2 with an accompanying truth table. When input A is OFF, the output is also OFF. But when input A is ON, the output is identical to input B. Hence A acts as a *control* signal for B, either blocking it or allowing it to pass unchanged. When used for this purpose, AND circuits are generally referred to as GATES, for obvious reasons.

control section

Referring back to the terminology used earlier, the data-flow section of a CPU provides a hardware host for simulation of a target processor. The simulation is accomplished by the control section, which directs the data-flow activities cycle by cycle to create the desired results. The control task is twofold: first, the control *state* during each cycle must be established; and, second, the correct next sequential control state must be determined. From this statement of the control task, we can infer that the control section must contain at least these facilities: first, storage for the information that identifies the current control state; second, a means of translating this identifier into a pattern of control signal activations; and, third, decision logic to choose the next state identifier. (It can also be inferred that there must be a timing mechanism to synchronize data-flow switching and control state transitions.) A simple schematic of a generalized control section is shown in Figure 3.

In the conceptually simplest possible organization, the storage part of the control section could contain a storage element corresponding to every control line, connected directly to its control terminal in the data-flow section. If this were done, there would be no need for logic to translate the recorded state into

the appropriate control signal pattern; the pattern itself could serve as the state identifier. For efficiency, this is normally not the method used. A typical small-to-medium CPU may have several tens or hundreds of control signals, which, with one storage state recording element per signal, would allow on the order of 2^{100} or more different control states to be recorded. Of these, only some small fraction (say 2^{10} to 2^{15}) represent meaningful combinations. Thus a tradeoff is made between the number and organization of state storage recording elements and the logic networks required to transform an encoded state into a set of control line signals.

In conventional hardware controls, the state encoding can be loosely described as a hierarchical system. At the highest level are storage elements, or registers, whose contents define major modes of operation—instruction stream execution, input/output operation, or interruption handling—that change relatively infrequently. Next are identifiers for specific operations within the major modes, such as the program operation code, for instance, during instruction-stream execution. Then come identifiers for successively finer resolution of activities down to the basic cycle.

The link between the target program in main storage and the control section is provided in the information that flows from the data-flow section into the next state decision logic. The control sequence that fetches an instruction into the data-flow section also switches the operation code into the control section to become part of the state identifier for subsequent execution cycles.

Interpretation is another word often used in connection with the simulation process. In the context of controls, it denotes simulation of a target program, instruction by instruction, so that the effects of a given instruction are completely evaluated before any further instructions in the target program sequence are considered. This is consistent with describing as interpretive those compilers that accept and execute source code statement by statement. (In fact, the distinction between compilation and simulation as outlined above lies primarily in the nature of the target program.)

In the foregoing discussion, we have assumed that the controls are built of the same sort of logic circuits as the data-flow section. What the controls are built of is not important per se; what matters is the efficiency, economy, usability, maintainability, and so on, of the virtual machine. These factors reflect the choices made in the design of the host hardware and control mechanisms. Thus it is really an empty exercise to debate whether or not a particular control implementation is intrinsically better or worse than another; the question is which best achieves the desired attributes in the resultant system.

Figure 4 Microprogram control section

Microprogram controls

At this point we can draw a physical distinction between a logic-network-controlled and a microprogram-controlled unit. In the former, only logic-circuit storage devices (registers) are used to record control information; in the latter, a large part of the control information is stored in some form of regularly organized memory array, called the control store.

In microprogrammed controls, the current control-store address identifies the current control state, while the contents of the addressed array location provide the information required to establish the proper control signal combination and to choose the next address. A simplified diagram is shown in Figure 4 of a read-only microprogram control section, which is quite similar in all its essentials to the generalized control section in Figure 3.

All control stores must be writable in some manner so that control information can be introduced. The quest for speed often leads to the design of arrays that can be loaded only by mechanical or electromechanical means, and these are designated as *read-only* to differentiate them from arrays whose contents can be changed during normal use, as, for instance, main storage arrays. Where normally writable arrays are used for control, their logical and operational context may render them also effectively read-only except under special circumstances.

The question of encoding the state identification vanishes with a microprogrammed control section. Each control state corresponds to some addressed storage location. The number of unique control states needed thus defines (for a read-only device) the minimum number of words of control store that must be provided.

microwords

The output of the control store is a microword. In the following paragraphs, we discuss what different elements the microword

can contain and some of the considerations involved in choosing a microword format.

First, the microword must contain sufficient information to establish the settings of the control lines for each cycle. The simplest technique would be to assign a bit in the microword for every control line, but this is generally shunned for efficiency reasons. Only a small fraction of all the possible combinations of control signals represent meaningful functions. Thus a word containing a bit for every control line would contain many bit combinations (code points) that are never used, giving low information efficiency. A common procedure is to examine the control signals for groups that are logically mutually exclusive—that is, groups in which only *one* line at a time is activated in meaningful control states—and to assign such groups to encoded *fields* in the microword. Transformational logic is then interposed between the control word and the control line groups to decode the microword field values and to activate the appropriate line in each group corresponding to the encoded value. The number of such encoded data flow control fields in a microword is a rough measure of the parallelism (number of operations that can be done concurrently) in the data flow section.

(In some cases, variables external to the microword may be used to modify the group assignment and decoding of a field.)

A second function that must be accomplished through every microword is to establish the microprogram address (control state) that is to succeed the current one. At first glance, it might seem that simple ways to do this would be either to increment the current address to point to the next word or to store a successor address in each control word and replace the current address with its successor during the execution of each control cycle. Neither of these schemes, however, permits any variation of the sequence of execution of microprograms, since each word uniquely defines its successor without reference to any external conditions that might occur during execution. Since the ability to vary the sequence of execution (to branch) is essential to any useful program, means must be included to vary the choice of next address as a function of previously executed states and of ancillary conditions.

A common technique is to construct the next address from the current address by providing fields in the microword that control modification of the current address as a function of environmental conditions, data-flow contents, and constants (address fragments) stored in the microword.

A third useful function of most microwords is control of the action of temporary condition-recording registers, which can store

information for establishing control store addresses. In engineering jargon, such registers are often called "stats."

A fourth type of field is commonly provided: a literal, or constant, field (in the jargon, the "emit" field), which may be used by the microprogrammer to introduce numerical values into the data-flow section from the control store (for instance, in updating addresses), to set the "stat" registers for microroutine linkages, and to supplement the next-address selection fields.

format variations

So far we have assumed that one microword is executed for each internal CPU time increment (i.e., that there is a one-to-one correspondence between the basic CPU cycle and the control store cycle). Moreover, we have noted that the simplest control word organization (not necessarily the most efficient) assigns bits to control lines on a one-to-one basis. This particular combination represents one end of a continuous spectrum, at the other end of which lies the conventional machine instruction. One moves across this spectrum both by compressing the microword in size (reducing the number of bits at the expense of more intermediate decoding logic and more sophisticated hardware functions) and by expanding it in time (increasing the number of CPU cycles executed per control word, trading control words for intermediate control logic and still more complex hardware functions). This progression moves explicit control information out of the microword and imbeds it instead in logic networks.

Somewhere between the two extremes lies the "miniword," an appellation attached to fairly small control word organizations that generally control multiple rather than single CPU cycles. Miniwords do not directly activate primitive control signals but logic-controlled subfunctions instead. Miniwords provide only a small portion of the range and complexity of normal machine instructions.

microword design

Since the data-flow section of a CPU is the "calculating engine" that does the useful work, one might expect microword characteristics—the microinstruction set—to be strongly influenced by data-flow design. This is indeed often the case.

One of the important objectives in processor design is optimization of the cost-performance ratio. A processor's raw speed is largely determined by the main storage speed and word width (data bandwidth). (Any buffering schemes used to enhance effective data storage rates are logically part of the storage system, although they may reside physically in the processor.) Once a performance level is established and a main storage is chosen, the next-level task is to organize the data-flow and controls. Logic circuit quantity has historically been a major cost-contributing variable, so data-flow organization has been aimed toward at-

taining a best match between hardware host facilities and virtual attributes to minimize the number of circuits needed. There has been as much truth as humor in the statement that the microword is defined by where the logic designer quits.

The balance of power is not all on the side of the data-flow designer, of course. The organization of an efficient processor involves complex trade-offs between control store speed, capacity, format, decision (addressing) logic, and data-flow facilities under the constraints of the target instruction set. An intuitive appreciation of this trade-off process may be conveyed by the following example.

Suppose that one of the operations critical to performance is the computation of an effective address from its base, index, and displacement components. This computation must take place during the interval between availability from main storage of an instruction and the beginning of the next main storage cycle. (This interval depends on the delay from the initiation of a storage cycle until data becomes available and on the length of the cycle itself.) At one extreme, a three-input parallel adder of adequate speed could be provided in the data-flow section solely to compute addresses, permanently connected between the address component source registers and the storage address register by dedicated data switches. This configuration could be controlled by just one or two bits in a single control word, but would require substantial logic circuitry. At another extreme, a single one-byte adder might be provided in the data-flow section that is to be shared by all operations calling for addition. In this case, the address components would have to be switched byte by byte into the adder input registers and the address accumulated in a series of partial sums. At least five or six control cycles would be required in the same length of time as a single cycle in the first case; more data switches would be active and require more bits in the control word; but very little logic circuitry would have to be supplied solely for address computation. The best configuration depends upon the specific relationships between control store speeds and capacities, the logic required, and the costs.

The rapid decrease in logic costs promised by advances in integrated-circuit manufacturing technology is now definitely loosening logic minimization constraints on data-flow section design (hence microinstruction set definitions); and interest is increasing in more generalized organizations that could make a single hardware configuration a reasonable match to several distinctly different virtual machine definitions.

writing microprograms

The discussion so far has probably given the impression that writing microprograms is, as far as logic and information content is concerned, a nontrivial task. This is very often the case. The

performance of a processor is dependent upon the number of control cycles executed per function or instruction, and its cost is a function of the amount of control store needed. There are benefits to be derived from tight, "clever" microcode, which contrasts with the case of conventional software where clarity and maintainability are generally more important. An intimate knowledge of data-flow facilities, microinstruction specifications, and machine timing is prerequisite for writing efficient, tightly packed microcode.

The mechanical aspects of creating microprograms are less formidable. To support the development of System/360 and System/370 microprograms, a set of design aids called the Control Automation System has been developed. This system accepts microprograms in a special flowchart format and performs diagnostic, simulation, assembly, and documentation functions. Its principal outputs are printed flowcharts (control logic diagrams) and a manufacturing interface tape to direct the physical production processes.

The bulk of all microprogrammed computers produced so far has employed read-only control stores. Most microprograms have been produced for these computers in a development environment as essentially one-shot operations. While the need for accuracy and efficiency has led to automation of the checking, verification, and bit-pattern generation tasks by simulators and assemblers, there has been no strong impetus to develop compiler-level microprogramming aids. Assembler-level facilities are a good match for relatively small staffs of highly skilled people writing relatively small volumes of microprograms; compilers are more likely to find economic justification where many people of diverse skills have continual need to generate and maintain large quantities of programs. Since the latter environment seems somewhat remote for microprogramming, microprogram compilation will probably remain a subject of academic interest for the near future.

Other design considerations

cost benefits

In order to achieve the required speeds at reasonable cost, technological constraints have generally dictated that control stores be writable only by mechanical or electromechanical means (often involving a factory-only process). The principal benefit of such stores is the increased number of functions (compared with logic network controls) obtainable for a given cost. Costs of logic controls increase in roughly linear proportion to functional capability, but once the physical installation of a control store is accounted for, the incremental cost of adding functions up to the maximum capacity of the store is small. A plot of cost versus

function for microprogrammed control thus approximates a series of step functions, each step representing the addition of a module of storage, and lies beneath the cost line of logic for significant ranges of function. The cost differential has encouraged inclusion of multiple instruction set controls (emulators) and extensive checking, retry, diagnostic, and verification procedures. Economically feasible control store capacities have been far from generous, however, and have still constrained the number of functions a particular machine can include. Advances in manufacturing technologies are now making it economically reasonable to include useful quantities of high-speed control store the contents of which can be rapidly changed in an operational environment. Such storage, loaded by replacement or overlay methods from inexpensive permanent microprogram residence devices, can greatly expand the effective control capacity and largely remove the capacity constraints, hence manufacturer's cost constraints, on the instruction repertoire with which a machine can be equipped.

performance

During program execution in a conventional digital computer, the CPU communicates across an interface to the main storage unit, fetching instructions and data and storing results. If we assume that the CPU is fast enough to always use every storage cycle available to it (main storage is never waiting for the CPU), then we see that data are transferred across the interface at the maximum rate possible, utilizing the maximum available data bandwidth.

The fact that a machine is running at maximum storage data bandwidth does not necessarily imply that a particular programmed task is being executed at the maximum rate that could be achieved given the freedom to vary the organization and representation of the statement of the task (the program) and its associated data. The formats and sequences of communications across the storage interface are functions of the instruction sequence and must conform to the architectural rules of the target instruction set. There are at least two ways in which the storage bandwidth information efficiency may be decreased from its possible optimum: when a program is not optimally constructed for a given architecture, and when the architecture itself permits only an inefficient statement of the algorithm being executed. We will not consider the first source of inefficiency here but will concentrate upon the second.

Let us take as an example a segment of code whose purpose is to multiply two numerical strings of the same number and size of element pairs. Given an instruction set with a typical scalar-oriented, operation-code, memory-address, register-address format, we see that each element pair requires at least the following program steps to be executed:

1. Load memory to register.
2. Multiply memory to register.
3. Store registers to memory.
4. Update indexes and close loop.

For every element pair in the strings, at least four instructions (and possibly more, depending on the data characteristics and the power of the index manipulation and loop-closing branch instructions) must be fetched across the memory interface, in addition to the two data fetches and one result store.

Compare this with an instruction set that contains provisions for initializing data descriptors and repetitively executing operators over described strings of data. To start the program segment would require several initializing steps:

1. Load string 1 starting address, increment, and extent.
2. Loat string 2 starting address, increment, and extent.
3. Load result starting address, increment, and extent.
4. Define end and branch conditions.

This could then be followed by one step:

5. Execute operator (multiply).

This sequence would produce exactly the same effect as the first example. However, once the initializing fetches are accomplished only source and result data transfers are required across the memory interface, regardless of the extent of the strings. The information efficiency with which the available memory bandwidth is utilized is increased by the elimination of the repetitive instruction fetches per string element shown in the first example. Once initialization is complete, microroutines can accomplish all the necessary updating and testing of string addresses without further reference to main storage, assuming of course that sufficient storage and transformational logic is made available to the microprogrammer in the CPU data-flow section, and that the additional internal functions can be accomplished in the available time.

The foregoing paragraphs illustrate one of the principal situations in which microprogramming can be utilized to enhance CPU performance: where information efficiency, or density, across the memory interface can be increased by revising instruction and data format definitions to substitute microprogrammed functions for logically redundant storage cycles. (Note that logic networks could also be used to control the added functions.)

An extension of this principle occurs in special cases where information normally resident in the instruction stream can be removed and instead implied in special-purpose microroutines. An example of such a special case might be the previous example of a string multiply restricted to fixed-length strings with fixed-size elements. The increment and extent parameters could be stored

in the microroutines as local constants, eliminating the need for transfer of this information in the initializing sequence.

A secondary benefit may accrue when specially designed instructions permit a more compact program representation that requires less memory space than its conventional instruction equivalent.

writable control store

An implicit theme in a number of technical discussions of dynamically changeable controls has been that users will leap at the opportunity to tailor machines to their particular requirements by writing tailored microroutines. Industry experience indicates, however, that this is a naive assumption. Developments in computing hardware have been paralleled by developments in software language processors and operating systems (with strong user impetus) that are designed to remove the users' problem statement and operational interfaces as far as possible from the machine level. The tendency to adopt higher-level interfaces has a basis in programming cost considerations. There is a rough correspondence between the power and sophistication of system-provided services and the language level employed. As machine language is approached, a programmer must do more and more for himself; thus he must exercise greater detailed programming skills while running the risk of reduced overall productivity. Since the machine code level has proved thoroughly distasteful to most users, it is hardly reasonable to expect user enthusiasm to manifest itself at the microcode level, which is yet more complex and intricate.

The exposures of users programming at the microcode level are not limited to the possibility of incurring higher direct coding costs. A family of machines that is compatible at the machine instruction and architectural levels will almost certainly not be alike at the data-flow and control levels. Consideration must be given to the loss of compatibility that may be incurred when an installation is made dependent upon special microroutines. Such features may render it impossible to use standard operating systems, language processors, and the like; it may be either prohibitively expensive or impossible to duplicate the features on other models within the family; and nonobvious side effects may have unexpected ramifications in areas remote from those directly affected by the features.

On the other hand, it is reasonable to expect manufacturers to seek ways of providing users the performance and efficiency increases that are made technically and economically feasible by large effective control store capacities. We have seen that, for algorithms whose expression in conventional machine instructions entails logically redundant storage cycles, performance increases can be obtained by creating new machine instructions

that permit higher information efficiencies across the memory interface. It appears that many common programmed functions fall into this category. Examples include the table searches and manipulations typical of many operating system and language translation tasks, and computations upon string, vector, and array structured data. However, there is presently little experimental data that can be used to precisely identify such functions and to quantify their frequency of execution.

It is intuitively evident that what comprises an optimal instruction set is an intimate function of the logic and data characteristics of a given program. From an overall system point of view, an instruction set should be judged on efficiency in automating program generation, debugging, and maintenance processes, as well as on execution-time efficiency. In light of the almost infinite variations of programs that exist, it appears a nearly impossible task to choose a small library of instructions—say 200, 500, or even 2000 instructions—for fixed installation on all of a given computer model with the goal of approximating an "optimal" set for a reasonable percentage of all environments. An instruction set of greater flexibility and power than conventional scalar- and register-oriented sets could certainly be provided in the future as a fixed base. But it may prove worth while to also make provision for dynamic optimization of the repertoire as a function of its local program and system environment.

If optimization facilities were to be incorporated into a system, it would be equally important to include a way to measure operations in representative system environments so that optimization choices could be based upon reasonable quantitive estimates, rather than conjecture and trial and error.

In the architectural design of such optimization facilities, at least three questions should be answered: What repertoire parameters should be variable? What should be measured in order that intelligent choices of parameter settings can be made? By what mechanisms should parameter settings and the associated controls be changed? To properly answer these questions, an understanding is required of language, program, and data structures and processes, as well as a knowledge of the potentials and limits of the physical host facilities.

It is fairly clear that, whatever the optimization methods might be, they should not require that users actually microprogram, nor even understand microprogramming. Implementation should be through disciplined and well-controlled combinations of architecture, language processor, and operating system services, with the user's interface as straightforward and as far from the detailed microcode level as possible, and with compatibility maintainable across a complete family of computers.

software reliability

An area that seems to be currently somewhat neglected is the relationship of architecture to software reliability. A program, like hardware, fails when it produces unexpected or incorrect results. Hardware is generally well checked at the functional level, and most current architectures establish some rules of validity for individual machine instructions. These are enforced by hardware checking, so that failures through the machine-code level are relatively well screened. Rules of validity for program representations above the machine-code level are rudimentary, however, and failures due to faulty program structure are usually detectable only through their side effects (unless checking routines are explicitly coded).

Consider as a simple example an architecture that defines some particular operation code as an entry code and establishes the rule that this be the only valid target of branch operations for program control transfers. This rule could be hardware-enforced by a few microinstructions, giving a simple but powerful check on the connectives constructed during execution. The author is of the opinion that extension of architectural discipline to program structure, implemented and checked by microprogrammed controls, may be one of the more rewarding uses of expanded control store capacity.

Summary

The part of a digital computer that performs the useful work is made up of these elements:

- Storage facilities
- Routing and switching facilities
- Data transformation facilities

They are connected together in data-flow sections. A data-flow section can be considered a hardware host for simulating a target machine. The simulating agent is the control section. The task of the control section is to generate sequences of signal patterns that direct the data-flow activities to create the effects described by some target machine specification.

One particular class of control mechanisms uses regularly organized storage arrays to contain a large part of the control information. Machines employing such control mechanisms are said to be microprogrammed. Specific control section and data-flow designs evolve from considering architecture, technology, performance, and cost interrelationships. Microinstruction formats can range across a spectrum from a one-to-one correspondence between control gates and bits in the microword to "mini" formats that approximate conventional machine instruction forms.

Control stores have historically, for speed reasons, generally been effectively read-only, have provided more functional capability above some threshold at a given cost than logic networks, but have still been limited in capacity and, for very high performance environments, in speed. The incremental functional capability has been utilized to provide multiple instruction repertoires and extensive checking, recovery, and maintainability aids. Current manufacturing advances are rendering writable control stores more attractive in cost and speed. Very large effective control store capacities can be achieved through combinations of bulk microprogram residence devices and writable control stores.

Raw processor performance can often be increased by enriching instruction repertoires to permit more efficient representations of algorithms and data structures. Efficient repertoire optimization may depend upon detailed analysis of application environments and means for dynamic, local adaptation of the repertoire. Writable control stores offer the technical means of implementing local adaptation, as well as the capacity for implementing richer base instruction sets. It is unrealistic to require users to microprogram; to exploit these potentials adaptive capabilities in future systems will have to be exercised through well-controlled higher-level language facilities.

A goal for future architectural development exploiting writable control store capabilities should be to increase the efficiency of overall system processes—program generation and maintenance, language processing, data handling, and operational control—as well as to increase raw execution efficiency. This will involve consideration of the relationships between language structure, language transformation processes, program forms, and data structures, as well as the physical data-flow and control facilities that will be technologically and economically feasible. Architectural specification of rules of validity for program structure, implemented by microcode, may be one fruitful avenue to increasing overall efficiency by enhancing program testing processes and operational program reliability.

BIBLIOGRAPHY

This section contains a list of selected readings together with comments to assist the reader in choosing material to fit his particular interests. For ease of reference, the articles are grouped under the following headings:

- Basic exposition and overview—a short list of core reading for the nonexpert reader
- Historical trace of microprogramming development—selections that trace the evolution of basic principles from 1951 to the early 1960's
- Implementation and applications—highlights of the use of fixed-store controls from the late 1950's through the 1960's

- Stored logic and dynamically changeable control—examples of early user-microprogrammed machines followed by the evolution toward the "firmware" concept
- Language-oriented systems—selections that illustrate possible higher-level architectures and organizations
- Related architecture and programming topics—leads into topical areas that may strongly influence future exploitation of the potential of microprogrammed systems.

The readings have been chosen as a guide to the literature rather than as an exhaustive listing. Many of the articles contain good bibliographies for those with special interests.

basic exposition

The articles listed below are chosen to establish a technical base and perspective for the nonexpert reader. They provide self-contained coverage for one who desires a working knowledge for a minimum investment of time, and are also a good starting-point for more extensive investigation.

1. S. G. Tucker, "Microprogram control for System/360," *IBM Systems Journal* **6,** No. 4, 222–241 (1967).

   This is a readable and complete description of read-only storage as a direct substitute for logic network controls in digital computers. It requires of the reader only a general familiarity with computer internal organization. Tucker reviews the origins of the technique, develops an abstracted example, and covers the essentials: microword organization, branching, timing, language, and design aids. Comments on purposes and limitations are included. The material is written with a solidly System/360 point of view, but this does not detract from the article's value as a thorough technical primer. Definitely recommended as core reading.

2. R. F. Rosin, "Contemporary concepts of microprogramming and emulation," *Computing Surveys* **1,** No. 4, 197–212 (December 1969).

   The early section illustrates control principles on an obscurely presented fictitious machine, but from the sixth page on, this article develops into an excellent discussion of current state-of-the-art with thought-provoking reflections on microprogramming. Rosin is concerned more with what microprogramming is good for and how it may be exploited than with how microprogramming works. He achieves an unusually even perspective. Also definitely recommended as core reading, either by itself or in conjunction with Tucker's exposition.

3. M. J. Flynn and D. MacLaren, "Microprogramming revisited," *ACM 22nd National Conference Proceedings*, 457–464 (1967).

   Packing a lot of ideas into a small space, Flynn and MacLaren develop the basic principles of stored control, dismiss previous read-only and stored-logic implementations as restrictive and uninteresting, direct their attention to dynamically alterable storage for machine control and consider its technological, architectural, organizational, programming, and usage implications. The paper reflects the authors' preoccupation with technical possibilities. It is recommended as a review of possible variations in computer organization for readers who are willing to go elsewhere for pragmatic and utilitarian considerations. Taken with Tucker and Rosin it rounds out a core selection for the more hurried reader.

4. M. V. Wilkes, "The growth of interest in microprogramming: a literature survey," *Computing Surveys* **1,** No. 3, 139–145 (September 1969).

Wilkes offers a pleasantly written and informative survey of the standard body of microprogramming literature, tracing the world-wide expansion of interest over the last two decades. Although the reader may occasionally wish that there were more in the way of critical comment, this is an excellent compilation of topics, articles, and authors.

5. S. S. Husson, *Microprogramming principles and practice,* Prentice-Hall, New York (1970).

For facts and references, this hard-cover book is a rich source. Nearly two-thirds of the book is devoted to describing in fine detail the microprogramming aspects of the IBM System/360 Models 40 and 50, the RCA Spectra 70/45, and the Honeywell H4200. A good index and an extensive bibliography, arranged both chronologically and alphabetically by author, are appended. The expository material in the first third is comprehensive, including interesting sections on control automation, comparative performance, special applications, and technology, but it is sometimes flawed by an uncritical advocacy of microprogramming. Not a particularly easy introductory text for readers without some prior grasp of computer internal operation and vocabulary, but an excellent engineering-oriented reference volume.

historical trace

These papers trace the early evolution of microprogramming ideas. Since microprogramming developed primarily as an alternative method of control design, most of the articles have a strong engineering orientation. It is interesting to note that awareness existed from the start that programmed controls could greatly facilitate architectural and organizational innovation, although for technological and pragmatic reasons few practical advances have yet been made in these areas.

6. M. V. Wilkes, "The best way to design an automatic calculating machine," *Manchester University Computer Inaugural Conference*, p. 16 (1951).

Wilkes is generally credited with first coining the term "microprogramming" and outlining the basic principles in this presentation in July 1951. Noting the possibility of building writable as well as fixed control stores, he foresaw some of the problems a proliferation of "private order codes" might produce.

7. M. V. Wilkes and J. B. Stringer, "Microprogramming and the design of the control circuits in an electronic digital computer," *Proceedings of the Cambridge Phil. Soc.*, 230–238 (1953).

The principles of microprogrammed control are reiterated and illustrated by a proposed design using a pair of fixed diode matrices, with particular attention given to microprogram branching facilities.

8. M. V. Wilkes, W. Renwick, and D. J. Wheeler, "The design of the control unit of an electronic digital computer," *Proceedings of IEE* 105, 121–128 (1958).

This is a discourse on the logical and engineering design aspects of matrix implementations of control stores, of interest primarily for its snapshot of the engineering state-of-the-art a decade and a half ago.

9. H. T. Glantz, "A note on microprogramming," *Journal of the Association for Computing Machinery* 3, No. 1, 77–84 (1956).

A nicely written consideration of the advantages and drawbacks of designing a computer to allow the interspersion of hand-tailored microprogrammed instructions with standard code. The hardware parts are now obsolete, but he raises questions of pragmatics that are still quite pertinent.

10. R. J. Mercer, "Microprogramming," *Journal of the Association for Computing Machinery* **4,** No. 2, 157–171 (1957).

The concept is explored of a set of matrices, each preprogrammed for some particular function, as a substitute for the conventional arithmetic and logic unit in a digital computer. Definitely dated, although the basic idea is worth noting by those who are seriously concerned with utilizing highly integrated circuit technologies.

11. A. Grasselli, "The design of program-modifiable microprogrammed control units," *IEEE Transactions on Electronic Computers* **EC-11,** No. 3, 336–339 (1962).

Faced with the economic and technological barriers of a decade ago in the way of achieving a sufficiently fast writable control store, Grasselli devises an interesting solution that employs an intermediate, changeable "pathfinder" memory to store strings of addresses that sequence accesses to a fixed, high-speed control store. Worth noting, as is Mercer's article, by those involved with highly integrated circuit technologies.

12. G. B. Gerace, "Microprogrammed control for computing systems," *IEEE Transactions on Electronic Computers* **EC-12,** No. 5, 733–747 (1963).

Detailed considerations of elaborations and expansions of Wilkes' basic matrix principles in the context of large, high-speed computers, based upon the author's work with the CEP machine constructed at Pisa. Laboriously presented and very difficult to read.

13. E. D. Conroy, "Microprogramming," *Preprint ACM 16th National Conference* (1961).
14. R. M. Meade, "A discussion of machine-interpreted macro-instructions," *Preprint, ACM 16th National Conference* (1961).
15. E. D. Conroy and R. M. Meade, "A microinstruction system," *Preprint, ACM 16th National Conference* (1961).

These three papers outline briefly the IBM 7950 instruction system, which makes three levels—macro, standard machine set, and microinstructions—directly available to the programmer.

16. L. J. Boland, "Analysis of read-only memory for control," MEE Thesis, Syracuse University (June 1963).

A development of the general control process in a digital machine is followed by detailed logical and timing analyses of read-only control stores as substitutes for logic networks. Academically lengthy.

implementation and application

17. M. W. Allen, T. Pearcey, J. P. Penny, G. A. Rose, and J. G. Sanderson, "CIRRUS, an economical multiprogram control," *IEEE Transactions on Electronic Computers* **EC-12,** No. 5, 663–671 (1963).

Designed and built in Australia, the CIRRUS machine is an early (1959) example of general-purpose fixed-microprogram organization and of the international interest in microprogrammed techniques (see also Gerace).

18. I. T. Hawryszkiewyxz, "Microprogrammed control in problem-oriented languages," *IEEE Transactions on Electronic Computers* **EC-16,** No. 5, 652–658 (1967).

The CIRRUS system is adapted for analog system simulation by microprogramming functional equivalents of analog processes. Although a large part

of this paper is devoted to discussing details of analog processes, it is worth scanning as an illustration of the functional adaptability made possible by easily changeable controls.

19. P. Fagg, J. L. Brown, J. A. Hipp, D. T. Doody, J. W. Fairclough, and J. E. Greene, "IBM System/360 engineering," *AFIPS Conference Proceedings* **26,** 205–231 (1964).

An overall summary of the System/360 CPU organizations and technology, of interest for its mention of the read-only store systems used in the different models.

20. D. L. Schnabel, "The design of processor controls using a read-only storage," Technical Report TR00.1318, IBM Systems Development Division, Poughkeepsie, New York (August 1965).

A fairly detailed explanation of the IBM System/360 Model 50 control word organization and functions. Quite readable if one is interested in digging into the specifics of read-only control design, and a good follow-on to Tucker's article.

21. B. R. S. Buckingham, W. C. Carter, W. R. Crawford, and G. A. Nowell, "The control automation system," *6th Annual Symposium on Switching Circuit Theory and Logical Design* (October 1965).

A description of the general flow and user language facilities of the original microprogramming automation system supporting System/360 development.

22. W. C. McGee and H. E. Petersen, "Microprogram control for the experimental sciences," *AFIPS Conference Proceedings, Fall Joint Computer Conference* **27,** Part 1, 77–91 (1965).

Good general discourse upon microprogrammed control unit concepts. The 2841 storage control unit is cited as an example and a film-scanning control unit design is described in some detail. The application details are very difficult to wade through, but the idea is a nice illustration of the flexibility of microprogrammed control outside the central processing unit.

23. G. A. Rose, "'Intergraphic,' a microprogrammed graphical-interface computer," *IEEE Transactions on Electronic Computers* **EC-16,** No. 6, 773–784 (December 1967).

A fairly general-purpose microprogrammed "front-end" computer is proposed as the interface controller between a System/360 Model 50 with large capacity storage and a group of video terminals. An example of the diverse applications potential of microprogrammed digital machinery, but the paper is strongly video-oriented and not particularly easy to read.

24. H. J. White and E. K. C. Yu, "Use of read-only memory in Illiac IV," *AFIPS Conference Proceedings, Spring Joint Computer Conference* **36,** 197–205 (1970).

An outline of the method employed to control each quadrant of the Illiac, written from a hardware-engineering point of view.

25. C. R. Campbell and D. A. Neilson, "Microprogramming the Spectra 70/35," *Datamation* **12,** No. 9, 64–67 (September 1966).

An outline of the internal organization and microprogram controls (a mini-instruction format is used) of the 70/35 is given. The ability to execute microprogramming steps obtained from main memory is noted.

26. N. Bartow and R. McGuire, "System/360 Model 85 Microdiagnostics," *AFIPS Conference Proceedings, Spring Joint Computer Conference* **36,** 191–197 (1970).

An illustration of ancillary uses for microprograms other than for normal machine code stream interpretation.

27. S. G. Tucker, "Emulation of large systems," *Communications of the ACM* **8,** No. 12, 753–761 (December 1965).

Tucker offers a definition of emulation, comments on design choices, and describes packages developed for the IBM System/360 Model 65 as a host machine.

28. M. A. McCormack, T. T. Schansman, and K. K. Womack, "1401 compatibility feature on the IBM System/360 Model 30," *Communications of the ACM* **8,** No. 12, 773–776 (December 1965).

An outline of the 1401 compatibility feature, which allows the Model 30 to directly interpret the 1401 instruction repertoire under microprogram control.

29. R. I. Benjamin, "The Spectra 70/45 emulator for the RCA 301," *Communications of the ACM* **8,** No. 12, 748–752 (December 1965).

The concept of emulation is discussed, highlights of the 301 and 70/45 hardware are noted, and the emulation process is described.

30. G. R. Allred, "System/370 integrated emulation under OS and DOS," *AFIPS Conference Proceedings, Spring Joint Computer Conference* **38,** 163–168 (1971).

A summary of emulator history is offered followed by a discussion of the development and special features of System/370 emulators, which can run as problem programs in a multiprogramming environment.

dynamically changeable control

31. H. M. Semarne and R. E. Porter, "A stored logic computer," *Datamation* **7,** No. 5, 33–36 (May 1961).
32. W. C. McGee, "The TRW-133 Computer," *Datamation* **10,** No. 2, 27–29 (February 1964).

These two papers discuss the organization and logical characteristics of essentially the same machine. The stored logic concept is essentially that of "mini-programming"; the basic machine language comprises a set of relatively primitive operations that are combined in short sequences to interpret conventional instructions. Only one memory is employed.

33. L. Beck and F. Keeler, "The C-8401 Data Processor," *Datamation* **10,** No. 2, 33–35 (February 1964).

The C-8401 is controlled by an instruction memory, writable under operator control, containing routines that interpret conventional instructions stored in main memory. Compare the internal organization, in which arithmetic and logical functions are wired to certain exchange registers, with Mercer's multiple T-matrix proposal. The control word's basic function is to choose the appropriate "exchange register" transfer paths.

34. E. O. Boutwell, Jr., "The PB 440 Computer," *Datamation* **10,** No. 2, 30–32 (February 1964).

Another variation on the "mini-program" concept. Primitives can be executed either from control memory or from main store.

35. L. D. Amdahl, "Microprogramming and stored logic," *Datamation* **10,** No. 2, 24–26 (February 1964).
36. R. H. Hill, "Stored logic programming and applications," *Datamation* **10,** No. 2, 36–39 (February 1964).

These two articles are introduction and summary, respectively, for the papers on the TRW-133, C-8401, and PB-440. In spite of their vintage, they are still good reading (Hill's, particularly) for anyone speculating on the role of writable control store in a general-purpose commercial marketplace.

37. A. Opler, "Fourth-generation software," *Datamation* **13,** No. 1, 22–24 (January 1967).

In this article, Opler speculates on the possible nature of "fourth generation" computers and coins the term "firmware." He gives a quick but comprehensive review of the ways in which changeable microprograms—"firmware"—could be used to enhance the performance of software, although he doesn't assess what level of demand (hence investment in firmware) he expects for brand-new fourth-generation functions. The responsibility for generating firmware is clearly placed at the manufacturer's level for all but application programs, and the complexity of microprogramming is recognized in the contemplation of a "new generation of specialists."

38. R. W. Cook and M. J. Flynn, "System design of a dynamic microprocessor," *IEEE Transactions on Electronic Computers* **C-19,** No. 3, 213–222 (March 1970).

Positing a fast read-write micromemory, the authors propose an organization for a "dynamic micro-processor" employing a three-level memory hierarchy and having no formal instruction repertoire above the "micro" level. Examples are given of microroutines that interpret higher-level functions. Long on technical enthusiasm. Of interest as an example of an organization unconstrained by optimization toward a conventional machine instruction set.

39. C. V. Ramamoorthy and M. Tsuchiya, "A study of user-microprogrammable computers," *AFIPS Conference Proceedings, Spring Joint Computer Conference* **36,** 165–181 (1970).

This long, often superficial survey of many ways in which writable control store *might* be exploited is followed by a mathematical model purporting to describe the parameters of a memory hierarchy that minimizes average access time to any information block within some given total cost. The thesis that users, rather than manufacturers, will utilize writable controls is posited but not supported. Short on perspective.

language oriented systems

40. J. P. Anderson, "A computer for direct execution of algorithmic languages," *AFIPS Conference Proceedings, Fall Joint Computer Conference* **18,** 184–193 (1961).

Anderson gives a brief review of the structure of ALGOL and then develops the block design of a stack-oriented machine that could directly execute ALGOL-like programs. Although implementation means are not discussed, the system diagram he develops could be the base for a microprogrammed machine.

41. J. E. Meggitt, "A character computer for high-level language interpretation," *IBM Systems Journal* **3,** No. 1, 68–78 (1964).

A fairly complete description of a proposed microprogrammed, character-oriented machine that could interpretively execute programs stored in forms similar to the proposed form of Mullery and Schauer.

42. A. J. Melbourne and J. M. Pugmire, "A small computer for the direct processing of FORTRAN statements," *Computing Journal*, No. 8, 24–27 (1965).

The proposed design of a small microprogrammed machine is described that accepts FORTRAN from a keyboard, translates statement by statement to a FORTRAN-resembling internal representation, executes completed programs, and provides debugging facilities, all under control of microprogram routines. Some rather general comparisons with a similarly sized commercially available machine are included.

43. T. R. Bashkow, A Kronfeld, and A. Sasson, "System design of a FORTRAN machine," *IEEE Transactions on Electronic Computers* **EC-16,** No. 4, 485–499 (August 1967).

A machine is described to process a FORTRAN subset in a load and an execute phase. While the description is not explicitly for a microprogrammed system, the clear flow design could easily be the skeleton for a microprogrammed implementation.

44. H. Weber, "A microprogrammed implementation of EULER on IBM System/360 Model 30," *Communications of the ACM* **10,** No. 9, 549–558 (September 1967).

The introduction gives a good outline of the process of decomposing source language into intermediate text, which is then translated into some machine-executable form. The body of the paper describes some of the attributes of the EULER language (similar to ALGOL), an EULER processing system written both in System/360 Model 30 microcode and System/360 machine code, and some details of Model 30 internal organization and microcode structure. An interesting experimental demonstration of the facility of directly interpreting functions more complex than conventional machine code.

related topics

45. L. L. Constantine, "Integral hardware/software design," *Modern Data Systems* (April 1968 through February 1969).

This series of nine articles offers an interesting and leisurely review of problems that should be tackled in designing a complete system, both hardware and programming, rather than a hardware configuration with subsequent programming support as is generally the case. Of particular interest are comments on program structure, hardware architecture, and software reliability. Although not all of Constantine's conclusions are above controversy, the series is a good perspective-broadener, especially for nonprogrammers.

46. J. Green, "Microprogramming, emulators, and programming languages," *Communications of the ACM* **9,** No. 3, 230–232 (March 1966).

A brief but excellent treatment of the relationships between language syntax, semantics, interpreters, and automata, followed by a short panel discussion. Well worth a careful reading for those only passingly familiar with the terms.

47. H. W. Lawson, Jr., "Programming-language-oriented instruction streams," *IEEE Transactions on Electronic Computers* **C-17,** No. 5 476–485 (May 1968).

A good primer on instruction stream structure, interpretation, and execution, with several examples of intermediate language forms. Not a how-to-do-it microprogramming article, but it definitely should be on the reading list of nonprogramming people concerned with extension of machine architecture by "firmware."

48. A. P. Mullery and R. F. Schauer, "ADAM—a problem-oriented symbol processor," *AFIPS Conference Proceedings, Spring Joint Computer Conference* **23,** 367–380 (1963).

Clearly written although not always easy to follow, this paper describes a string-oriented language and data structure for a proposed experimental machine. The systematic emphasis upon information processing rather than mathematical computation is worth noting.

49. G. Radin, "A note on the concept of binding," IBM Thomas J. Watson Research Report RC 3287, Yorktown Heights, New York (March 1971).

A readable and excellent treatise on the relationships among program form, information content, evaluation procedures, and automaton functions.

50. W. M. McKeeman, "Language-directed computer design," *AFIPS Conference Proceedings, Fall Joint Computer Conference* **31,** 413–417 (1967).

This article prefaces some serious suggestions for architectural consideration with a most entertaining sketch of the development of an imaginary Turing-based architecture. Combines light reading and good pointers.

51. T. C. Chen, "Unconventional superspeed computer systems," IBM Thomas J. Watson Research Report RC 782, Yorktown Heights (November 1970).

In a brief but concentrated paper, Chen reviews the fundamentals of computer parallelism, pipelining, and efficiency and then suggests language-directed organizations that enhance both internal optimization and programmer effectiveness.

52. R. W. Floyd, "The syntax of programming languages—a survey," *IEEE Transactions on Electronic Computers* **EC-13,** No. 4, 346–353 (August 1965).

53. W. M. Waite, "A language-independent macro processor," *Communications of the ACM* **10,** No. 7, 433–440 (July 1967).

Neither of the above papers is connected directly with microprogramming and neither is particularly easy to read. Floyd reviews the principles of formal grammar structure and analysis while Waite describes a preprocessor for expansion of source-language macros. They are included here as leaders into the fields of language construction and manipulation for those contemplating future architectures exploiting writable controls and "firmware."