Microprogramming revisited*

by MICHAEL J. FLYNN and M. DONALD MacLAREN
Argonne National Laboratory
Argonne, Illinois

INTRODUCTION

Microprogramming was one of the first of the quasi mystical computer term (being joined by multiprocessing, parallel processing, real time, etc.). The term originated in 1951 with M. V. Wilkes [24]* in the context of a machine whose operation code would not be fixed but could rather be chosen at will by the programmer. Wilkes astutely observed then that there was probably no real requirement for such a machine and subsequently [25], in 1958, observed that 'events have confirmed this view,' It is difficult to know whether Wilkes' observation is still true since the term microprogramming lacks (despite many attempts) a universally accepted definition.

It is the objective of this paper to briefly trace the history of the idea and the difficulties involved with defining or implementing it. In doing this, we first consider the general control problem and instruction formats. Next, storage implementations of the control function are considered and a restricted definition of microprogramming is proposed. This is then evaluated from a technological, architectural and programming point of view. We hope to show that our (demanding) definition of microprogramming is now technologically feasible and attractive from systems considerations.

The control problem

Figure 1 shows an idealized processor from the control point of view. The raw resources of the central processor consists of execution logic and immediate or operand storage (registers). The allocation and definition of these resources is appropriately designated as the *control function,* with basic operational control being invested in the "instruction."

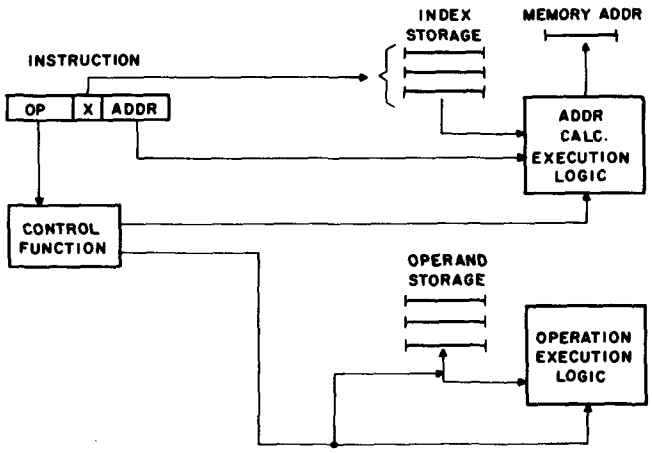


Figure 1 – Idealized processor

The execution logic, consisting of combinatorial circuitry only (at least in this view), represents the ordinary logical and arithmetic microfunctions of the machine: compare operations, shift, primitive "add" operations (or possibly only a part of such operations). The immediate storage represents sufficient storage capacity for the operands of any operation to be performed by the processor. There is usually a minimum of 3 full words (parallel operation being assumed) of operand storage for any reasonable processor (two is a lower bound, three required to collect double size results). The immediate storage units are loaded from, and return their information to, main memory or higher level storage. All transfers to and from the immediate storage, whether to main memory or to the execution logic, are accomplished by opening the logical gates which activate that particular path. Similarly, embedded within the execution logic are the first level gates which specify (completely) the activity of the execution logic on the operands that will be presented to it. Since the execution logic is purely combinatorial (i.e., time independent and without storage function of its own) it

*Work performed under the auspices of the U. S. Atomic Energy Commission.

*Numbers in square brackets refer to the references at the end of this paper.

performs a "truth table like" or fundamental operation on the operand.

The instruction specifies the central function by providing

1. operation specification,
2. operand location specification,
3. next instruction location specification.

These specifications may be implicit or explicit (giving rise to stack machines and 1, 2, and 3 address computers). The location (address) specification (indexing, etc.) function may properly be placed in either the control area or in the execution area. For our purposes we will consider it to be a separate execution function with its own operand storage and execution logic, as well as its own gating requirements. From the point of view of microprogramming the critical part of the control problem is the operation specification. The operation field specifies the interesting functions or vocabulary that can be performed by the processor.

Assume that gating for operand storage and execution logic represents a total of M individual gates. Let N be the number of similarly defined gates for the addressing function. Then at any one time M + N gates must be completely specified for an elementary operation to occur. Typically, the sum M + N varies between 100 and 6000, with a 7090 class processor having about 1000 gates. However, this specification of the gates is not done independently. Typically, if the i^{th} bit of an operand storage is being transferred or presented to the execution logic, then so is the $(i+1)^{th}$ bit; exceptions might include the highest and lowest positions. The intermediate positions could be treated as a common entry. This allows substantial reduction in M + N. Normally, the 1000 positions are reduced to 100 gating descriptions (as they may now be referred to). In order to conserve space in memory, these 100 bit wide gating descriptions are further encoded into a more closely packed binary code format, since the 100 bit patterns have 2^{100} possible combinations, of which perhaps only 2^{10} have any obvious usefulness.

So far we have merely specified one gating description. Such a description may be termed a micro-operation but it is not usually regarded as a useful member of the processor vocabulary since several of these operations are required on a particular operand set before any familiar logical and/or arithmetic operation can be fully executed. Thus the operation code portion of the instruction format must contain a packed embodiment of the gating descriptors together with appropriate sequencing information to perform some useful function. The sequencing information usually consists of a count. The decoding of the packed gating description is called the *combinatorial logic decoding problem,* while the sequencing of gating descriptors to perform one instruction is called *the sequential control problem.* The operation code is clearly a composite of combinatorial and sequential information which is decomposed matrix fashion, with each count operating on the residue of the operation code field to produce new gating descriptions Typical processors have between 20 and 3000 instructions in their repertoire, with between 4 and 10 gating sequences or micro-operations (each equivalent to one clock pulse) performing one instruction.

Storage implementation of the control function (microprogramming)

Putting practical considerations such as cost and performance aside for the moment, coordinate address storage is an excellent vehicle for the implementation of the control function. In proceeding from the coordinate address of the retrieved data word a natural combinatorial decoding is achieved; further sequencing is merely a progression of memory addresses. Figure 2 shows several possible control

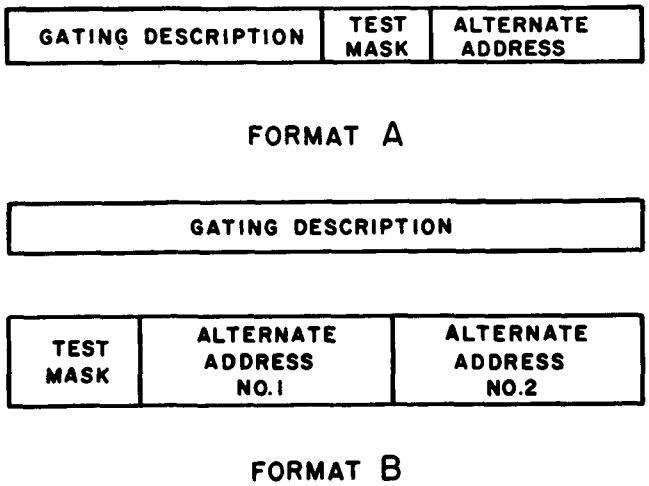


Figure 2 – Micro instruction formats

memory word formats. The gating description forms the bulk of the word, if no further decoding is to be allowed. The remainder of the word must include test specification information and (either explicitly or implicitly) primary and alternate next address information for sequencing. Format A might be considered a typical format. The test portion of the work contains explicit identification of each of the possible test combinations that may be performed in the machine. While this may be a packed format, for purposes of our discussion we will consider this to be an expanded format: i.e. one bit per gate (or chain of gates) to be

tested. If only one test is allowed to be made in any clock pulse, then sequencing can be handled by knowledge of the primary next address if the test proves *zero,* and alternately, the alternate next address if the test proves *one.* Since one of these is normally implied as being simply the contents of the next address in the sequence, the other is explicitly stated in the word as shown. Alternately, this address field might contain merely an increment to be added to the present contents of the micro instruction-counter or present address. B is an alternate format showing only a gating description in a word, that description being followed by a test word. The test word must contain the test specification (or test mask) plus the alternate addresses (or alternate increments), and the last gating descriptor must remain active until the test is performed. The selection between format A and format B or some other alternative is principally dependent upon the relative costs of storage (word size vs. number of words). If test conditions are exclusive they can be mapped directly to one of the alternate address fields. If they are non-exclusive, multiple test words must be used.

Of course, additonal features could be considered, such as micro indexing, etc. With the addition of such features one might question the requirement for the conventional instruction stream at all. This point will be taken up in a later section. Even without micro indexing, nested subroutines still could be considered as part of the micro program storage possibilities. The addition of several dynamically alterable addresses or pointer words might be required to keep track of present status. As one left successive stages of nesting, there would be a push up replacement of micro instruction counter contents.

What we have discussed thus far is microprogramming in a narrow sense. While it satisfies all the requirements proposed for microprogramming (e.g. see [8,23]) it represents an ideal implementation that in the past has not been considered a practical reality. Early discussions limited storage implementations to the read only [6,7,10,14,20,21] or non-alterable variety. Normally diode arrays were considered for the function (see Wilkes [24,26,27] and Mercer [17]), although other "read-only" memory implementations followed. In its most degenerate form microprogramming became nothing more than a diode decoder. There were attempts to define the term as being "stored logic" [1,20] or a similar notion but without success. [22] The great promise of altering the "structure" of the computer was never realized (in any general sense) since the contents of such storage were difficult or impossible to change. Microprogramming thus became more a mystical than practical abstraction. Several attempts at implementing a quasi-alterable storage for the control function were attempted. Grasselli [33] in particular produced an interesting variation in which a permanent or non-alterable storage contained what we term "gating descriptors" and a dynamically changeable "path finder memory" which traced through string sequences. However, he did not consider the gating descriptors to be already decoded but rather to be packed and decoded on retrieval. Other microprogramming attempts [15] were microprogrammed in the sense of the instruction operation code performing little or no sequencing (for some or all of the instructions). Thus the instruction stream could become a micro-instruction stream. While this allows flexibility, there would be a basic system inefficiency caused by the mismatch between the instruction and data access time and the execution time. Viz., execution would be such a small fraction of time used that the machine generally would be in a state waiting for data or next instructions. Also, such schemes require more instructions to be performed for a given "useful" function (except for special purpose situations [2]). As a practical matter such restrictive implementations of "microprogramming" became merely exercises in semantics, as pointed out by Wilkes [25] and Teager [22].

In order to avoid some of these difficulties we will use the word in the remainder of the paper to mean only the use of *dynamically alterable storage (comparable read & write performance) to perform the combinatorial and sequential decoding functions of machine control.* While this definition may be an improvement in defining the nature of microprogramming, it also remains inexact because the term "gate" to which the combinatorial output is referenced is inexact. In other words, when is the gate doing a control function (controlling information) and when is it actively engaged in a sub part of the decoding operation? To this question we have no quick answer.

Technological implications

In order for our previous definition to be meaningful, practical implementation must exist. While implementations in the past were restricted to read only storage; integrated micro-electronics or monolithic circuitry promises to favorably alter the cost performance equation so as to make our definition workable. With monolithic circuitry, costs are not determined by the number of circuits on a physical logical unit, but rather [24] upon the number of different types of circuit arrays used and [27] the num-

ber of external connections required to be made to each array [11,19]. Presently, monolithic circuit arrays have upwards of 20 to 50 circuits. At least two orders of magnitude improvement are seen in this number over the next decade. Notice that a conventional implementation using a decoding network is particularly unsuited to monlithic implementation because of its lack of regularity for array replication. Exactly the opposite is the case with the storage implementation. Also connectivity to the array goes up only logarithmically as the number of circuits is increased, satisfying the second constraint. Other advantages of the monolithic implementation include the alterability of storage, speeds competitive with the execution logic, and physical compatability (packaging, voltage distribution, etc.) with the rest of the machine.

To quantify the speed requirements for such a storage, Boland [5] has shown that in a conventional parallel machine typically 35% of a clock pulse time is spent decoding and setting up the gates for the operation to be performed. The remaining 65% is spent in executing the elementary operation. Thus an access of 35% of the clock pulse or a ratio of about 1/2 of the execute time of an elemental operation would be sufficient for our microprogram storage. Insofar as size and capacity are concerned successful read only storage implementations use between 1000 and 2000 words with about 100 bits per word, totalling approximately 200,000 bits.

Costs of 10 cents per bit using integrated circuits [14b] appear imminent for the memory ($20,000 total for the memory) and an order of magnitude improvement in the not too distant future appearing assured if present trends continue. Expected microstorage performance using integrated circuits would include access time of 25-50 n sec. A broader cost-performance trade-off can be considered with a film storage technology, but it is doubtful that the performance level offered (100–200 n sec. [16b]) would be interesting for the micro storage application. Film storage would seem more suitable for the main memory application, at least for intermediate and large scale processors.

Architectural implications

Given the existence of such a storage for implementation of the control function, how would it be incorporated in a central processing unit; what would the CPU's characteristics be? Figure 3 shows a possible organization for the CPU of a simple machine, peripheral devices being neglected. The principal components are

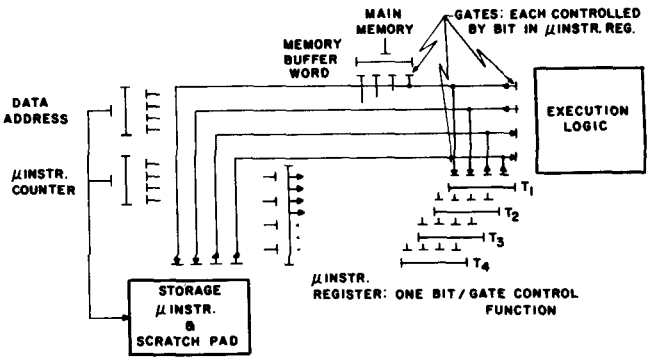


Figure 3 – Simplex microprogrammed processor

1) The execution unit, which carries out arithmetic, logical, and shifting operations.

2) Immediate storage containing four registers T_1, T_2, T_3, T_4, any one of which could be used to hold partially decoded macro-instructions while they are being processed.

3) Main memory buffer register, which holds operands going to, or coming from, the computer's main core memory, and its control, which accepts store and fetch addresses.

4) The control memory, which is also used as a scratch pad memory.

5) The micro-instruction counter, a register containing the address of the next micro-instruction to be executed.

6) The data address register, containing the location of an operand.

7) The micro instruction register, which stores micro instructions from the control memory and sets the gates.

The basic sequential operation of the machine is shown in Figure 4. First the micro instruction counter is incremented, then its contents are fetched. This is overlapped with the execution of the previous micro instruction. The operands are assumed to be in one of the immediate registers.

There are three general classes of instructions:

1) Normal instructions, which control all gates, both to registers and within the execution unit. (Figure 4, example 1.)

2) Control memory fetch and store instructions, which transfer data between a register and a selected location in the control memory, the address being in either in the micro instruction or a register. (Example 2.)

3) Test instructions, which execute a branch in the micro program conditional upon the value of one or more test indicators in the execution unit. (Example 2a.)

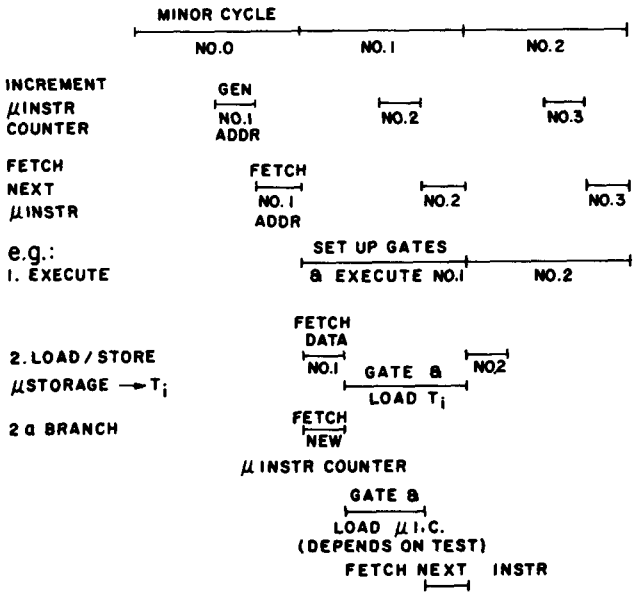


Figure 4 – Simplex timing chart

A 30-n-sec micro-storage access would seem to be well within the present state of the art, thus indicating minor cycles of 100 n sec.

In the simple organization sketched in Figure 3 the only additional synchronization involves operand fetches or stores in main memory. If the main memory is busy (or if the referenced memory module in an overlapped memory is busy), the execution of further micro instructions is simply delayed. Thus, as far as the micro program is concerned, a requested operand is always available in the memory buffer register a fixed time after the address is transmitted to the memory control.

Additional systems' sophistication might include the flexibility of splitting the gates apart so that data path control could be carried out (optionally) on an every-bit basis (chained together), on an every-other-bit basis, or in two-, four-, or eight-bit blocks, etc. While increasing the word size somewhat this would give such desired options as altering the number base of the arithmetic. Further, careful correlation between the gate label and the gate function (by label we mean the corresponding bit location in the control storage word), would allow the execution logic to calculate the various gating descriptions and thus optimize its own gating patterns. This optimization could be done on any of a number of criteria such as precision requirements, significance, or data queues. Of course this is done dynamically now at the instruction level; what is being proposed here allows one further degree of refinement on the process.

Certain features of even the simplex organization are quite remarkable when compared with a conventional machine. First, there are no hidden registers; all may be used as operand sources or for storage of results. Thus, for example, a computed microprogram branch may be effected simply by gating the computed address to the micro-instruction counter register. Second, multiple registers for temporary results are not needed because the control memory is also a scratch pad memory whose speed is like that of the execution logic. Finally, and perhaps most surprising, no distinction is made in the architecture, between macro-instructions and other data in main memory. Whenever appropriate, the micro-program simply fetches a new macro-instruction in the same way any operand is fetched, partially decodes it using the execution unit, and then branches to the section of the microprogram that executes that macro-instruction.

The micro-computer is especially efficient in complicated operations, such as number base conversion, which normally require a loop of several macro-instructions but may be performed by a single microprogram. If certain conversion operations are especially important, the associated tables may be kept in the control memory, which would provide a significant further increase in speed. Notice that although some conventional machines have similar instructions, they are very rigid. With microprogramming, the conversion could easily be from pounds, shillings, and pence to binary dollars. Even a computation as sizeable as a matrix conversion might be performed by a single microprogram. This would eliminate all instruction fetching and decoding, and in many instances the effect would be to replace a macro-instruction by a single micro-instruction.

Software design for microprogramming computers

The process of designing software for a microprogrammed computer will differ significantly from that for a conventional machine. Aspects of design which up to now have been in the province of architecture will come under the programmer's control. The most obvious benefit of this is the software designer's freedom to choose his own macro-instruction repertoire. (Note that there are no "machine instructions" and hence all instruction references to memory are "macro.") For example, he could choose the type of normalization for arithmetic operations and the possible actions to be taken on overflow, underflow, etc. However, the software designer's freedom will go farther than this. He can consider the computer as a stack machine, a one-, two-, or three-address machine, or a variable word length machine. Core addresses within macro instructions can be complete or can be specified by a dis-

placement and a 'register' (control memory word). In the latter case, the size of the displacement might be made dependent on the total size of core memory.

The microprogrammed computer's architecture will allow the programmer to, in effect, tailor the machine's structure to his particular problem. The same structure does not have to be imposed on all programs. This flexibility should make software not only easier to write but also more efficient. For example, the IBM 7090 FORTRAN compiler generated object code using only 19 of the more than 200 instructions in the 7090 repertoire. Thus 90% of the instruction repertoire was wasted. In a microprogrammed computer these unused instructions would be eliminated and the control memory space used for scratch storage or microprograms of special value in FORTRAN object code. For example, a single macro instruction might be used for the three-way branch arising from a FORTRAN "IF" statement. Such an instruction would probably take only two more cycles (minor) than a normal branch.

Efficiency of software will also be improved by the capability of implementing complicated functions by a single microprogram. Opler [18], in discussing fast-read-slow-write control memories, has pointed out the value of this in implementing the more important control functions of a computer's operating system. The possibilities in a true microprogrammed computer are much greater. The operating system and principal programming systems may have libraries of microprograms going far beyond what the control memory will hold at one time. The less frequently used microprograms would be loaded and executed only when needed. Thus the microprogrammed part of the entire software system could adjust quickly to changing conditions and could be easily modified in the light of experience.

So far we have avoided one important question: how will the basic software (that is programs normally written in assembly language) be written? Since there is no fixed macro instruction repertoire or even a fixed set of instruction formats, a conventional, assembler would not be of much value. A reasonable approach would be a macro-language, similar in many respects to a meta assembler. Simple macro-definitions would define a single object program-macro instruction; more complicated definitions would involve conditional expressions, sequence of expressions, and nested macro-expressions. With such a language, the macro-instruction set used in a program could be easily modified and extended as the program develops.

Compilers themselves should be organized in a way that makes changing the object code format easy. Fortunately the code generating part of a compiler is not the major part of a compiler and is easily isolated. For a compiler with a proper overall design it should be easy to effect changes, say from three address object code to a stack organization or from compiling array operations as loops of macro instructions to effecting them by special microprograms. In the long run compilers should also be able to optimize inner loops and frequently used subroutines by compiling them directly into microprograms.

Although simple assemblers should disappear as a tool for writing object programs [programs held in core], the assembler still has a use: namely, for writing microprograms. For a computer designed with this in mind it will be easy to construct a microprogram assembler. It would accept instructions like "Add,1,2,1" standing for gate registers 1 and 2 to the adder, set the adder gates for simple addition, and gate the result to register 1.

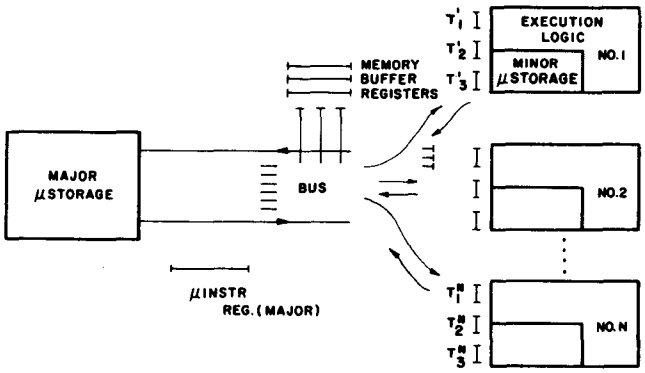
Now it is the authors' view that the macro language, the micro-instruction assembler, and a high level language might all be combined in one system. The micro-language would be a special part of the macro-language, which would also allow macros defined using the high level language. This high level language would be chosen for its usefulness in general (as opposed to numerical or commercial) programming, especially system programming and compiler writing. It might, for example, be a subset of PL/I. The compiler would be written in this language and inplemented first on an existing computer.

Elaborations

The simplex central processing unit shown in Figure 3 is essentially a von Neumann machine, whose normal memory is the control memory, and which has core memory as fast auxiliary storage. Looked at in this way it appears overly simple by present-day standards, and it should be possible to improve on it while retaining the basic microprogrammed structure. The improvements would be aimed at two things: making the CPU faster through parallelism and other sophistications in its structure, and adapting it to the complicated operating conditions present in modern computing systems, especially multiprogrammed operations.

Figure 5 shows a system variation where multiple execution units are employed. As shown, the micro storage is partitioned among each of the execution units: they are independent processors enslaved to the major micro storage. While one could refer to this arrangement as "parallel" or "concurrent" operation, actually at one time only a single micro-instruction (however complex) is being executed. Notice that the minor storage units could be combined into

the major unit, but with increased total capacity requirements. In order to provide the required main memory band width, multiple units are used, thus necessitating a small number of memory buffers. These buffers serve the "immediate" or way station function since the micro-storage is the true buffer.



MULTIPLEX MICROPROGRAMMED PROCESSOR

Figure 5 – Multiplex microprogrammed processor

The two principal problems in multiprogramming are protection and resource sharing. Protection in a micro computer would be handled by protecting gates rather than instructions. Certain gates could be activated only in the supervisor mode, which ould be entered only by a micro-branch to a fixed location. By keys associated with suitable sized memory blocks and each active program, data within the control memory could be protected against modification. Naturally, keys could be changed only in the monitor mode. Read protection could also be provided if necessary.

Some protection would also have to be provided in connection with interrupt handling. We envision the individual microprograms testing for interrupt conditions at the most convenient points in their execution. A sentinel unit could seize control if an interrupt was not recognized in the maximum acceptable time. Such seizure would completely disrupt the particular program being executed (it might, for example, occur in the middle of a multiply operation), and it would be the individual microprogrammer's responsibility to insure that interrupt testing occurs with the required frequency.

A resource sharing problem might arise in the central processing unit of the micro computer. For example it might be desirable to switch the CPU back and forth between several programs in order to make most efficient use of core storage or the CPU execution logic. Since the control memory contains a large amount of data, completely changing it would not be practical. Therefore the control memory must be shared between programs. Fortunately this is not as severe a problem as the sharing of main memory. A large part of the control memory will be occupied by microprograms used by many programs, and complete switching need occur only in rare instances.

In our opinion a rather simple allocation scheme for the control memory would work well. The first part of the memory would be occupied by macro instructions used by the basic system software (and by all other programs that need them), the second part by macro instructions used by important programming systems (FORTRAN, PL/I, COBOL, etc)., the third by micro programs and operands for individual programs. It would seem reasonable to make the memory large enough to hold the microprograms (macro instructions) for all the large programming systems in use at the installation. Also, if any reasonable coordination exists between the system designers, many macro-instructions and hence segments of the microprograms will be common to two or more systems.

Each object program, while controlling the CPU, would have a block of storage in the third part of the control memory. For short jobs this would probably be a small block used only for temporary results. Large programs in which efficient code is very important might be compiled so as to use almost all the available scratch memory. This block would be treated as being relocatable, with the relocation of addresses perhaps being assisted by special hardware. The operating system would presumably avoid running several large programs simultaneously, aiming for little or no swapping of the control memory. If too much swapping occurred in practice the control memory size would be increased.

Of course if the cost of control memory were low, enough would be provided to accommodate all jobs simultaneously. Status switching in such a system would require only fractions of a microsecond. The possibilities of this sort of computer in a multiprocessor configuration are also very interesting, for the individual processors could be restructured in response to changing job loads and operating conditions.

REFERENCES

1 D AMDAHL
Microprogramming and stored logic
Datamation 10 2 February 1964 24-26

2 L BECK and F KEELER
The C-8401 data processor
Datamation 10 2 February 1964 33-35

3 H BILLING and W HOPMANN
Mikroprogramm-Steuerwerk
Electronische Rundschau Berlin-Borsigwalde Vol 9 1955 pp 349-53

4 J V BLANKENBAKER
Logically micro-programmed computers
Transactions Professional Group on Electronic Computers Institute of Radio Engineers Vol EC-7 June 1958 pp 103-09

5 L BOLAND
Control memory
Thesis for MSEE degree Dept of Elect Engr Syracuse Univ Syracuse N Y Submitted June 1963

6 E O BOUTWELL JR
The PB 440 computer
Datamation 10 2 February 1964 30-32

7 E BOUTWELL and E HOSKINSON
The logical organization of the PB 440 microprogrammable computer
Proc. AFIPS Vol 25 FJCC November 1963 pp 201-213

8 E D CONROY
Microprogramming
Preprints of papers presented at the 16th National Meeting of the ACM Los Angeles September 5-8 1961 ACM New York N.Y.

9 E D CONROY and R M MEADE
A microinstruction system
Preprints of papers presented at the 16th National Meeting of the ACM Los Angeles September 5-8 1961 New York N. Y.

10 C H DEVONALD and J A FOTHERINGHAM
The atlas computer
Datamation 7 5 May 1961 23-27

11 J J FLYNN
A prospectus on integrated electronics and computer architecture
Proc AFIPS Vol 29 FJCC 1966 pp 97-103

12 H T GLANTZ
A note on microprogramming
Journal of the association for computing machinery Vol 3 April 1956 pp 78-84

13 A GRASSELLI
The design of program-modifiable micro-programmed control units
IRE Transactions on Electronic Computers June 1962 p 336-339

14 H HAGIWARA
The KT Pilot computer-a microprogrammed computer with a phototransistor fixed memory
Proc IFIPS Cong 62 Munich 1962 North Holland Pub Co Amsterdam 318-321

14b R A HENLE and L O HILL
Integrated computer circuits—past present and future
Proc IEEE Vol 54 No 12 (Dec 1966) pp 1849-1860

15 W C McBEE
The TRW-133 computer
Datamation 10 2 February 1964 pp 27-29

16 R M MEADE
a discussion of machine-interpreted macroinstructions
Preprints of papers presented at the 16th National Meeting of the ACM Los Angeles September 5-8 1961 ACM New York N Y

16b S A MEDDAUGH and K L PEARSON
A 200-manosecond thin film main memory system
AFIPS Vol 29 FJCC-66 pp 281-292

17 R J MERCER
Micro-programming
Journal of the Association for Computing Machinery Vol 4 April 1957 pp 157-171

18 A OPLER
Fourth Generation software
Datamation 13 1 January 1967

19 R L PETRITZ
Technological foundations & future directions of large-scale integrated electronics
Proc AFIPS Vol 29 FJCC 1966 pp 65-87

20 H M SEMARNE and R E PORTER
A stored logic computer
Datamation 7 5 May 1961 33-36

21 F H SUMMER
The central processing unit of the atlas computer
Proc IFIPS August 1962 Munich North Holland Pub Co Amsterdam pp 291-296

22 H M TEAGER
A discussion of machine-interpreted macroinstructions
Preprints of papers presented at the 16th National Meeting

23 W L VAN DER POEL
Microprogramming and trickology
Digitale Informations-wanderler, Hoffman Ed. F Viewag Publisher 1962

24 M V WILKES
The best way to design an automatic calculating machine
Manchester University Computer Inaugural Conference Manchester July 1951 pp 16-18

25 M V WILKES
Microprogramming
Proc AFIPS Vol 12 EJCC 1958 pp 18-19

26 M V WILKES W RENWICK and D J WHEELER
The design of the control unit of an electronic digital computer
Proceedings of the Institution of Electrical Engingeers London Vol 105 pt b March 1958 pp 121-28

27 M V WILKES and J B STRINGER
Micro-programming and the design of the control circuits in an electronic digital computer
Proceedings of the Institution of Electrical Engineers London Vol 105 pt B March 1958 pp 121-28