30 Sep 1974

Designing HMO, an Integrated Hardware Microcode Optimizer

James O. Bondi

Paul D. Stigall
Missouri University of Science and Technology, tigall@mst.edu

DESIGNING HMO, AN INTEGRATED HARDWARE MICROCODE OPTIMIZER

James O. Bondi

Department of Electrical Engineering

Paul D. Stigall

Departments of Electrical Engineering and Computer Science

University of Missouri-Rolla
Rolla, Missouri 65401

ABSTRACT

This paper discusses an algorithm for optimizing the density and parallelism of microcoded routines in microprogrammable machines. Besides the algorithm itself, the algorithm's uses, design integration problems, architectural requirements, and adaptability to conventional machine characteristics are also discussed and analyzed. Even though the paper proposes a hardware implementation of the algorithm, the algorithm is viewed as an integral part of the entire microcode generation and usage process, from initial high-level input into a software microcode compiler down to machine-level execution of the resultant microcode on the host machine. It is believed that, by removing much of the traditionally time-consuming and machine-dependent microcode optimization from the software portion of this process, the algorithm can improve the overall process.

INTRODUCTION

Since the advent of microprogrammable machines in recent years, a frenzy of research has occurred on developing good software compilers to generate user-designed microprograms, or microcode, for chosen target machines [1], [2]. The traditional argument against such compilers is that they will never be able to generate the completely compact microcode needed in a typical high-usage microprogram. The traditionalists thus conclude that the tedious and complex task of microprogramming is best left solely to the hardware designers [3], [4], [5], [6]. On the other hand, many machine users have long desired a machine whose instruction repetoire they could tailor to their particular needs [5], [6]. These users argue that a microprogram compiler would drastically reduce microcode production time, thus making even medium-to-low-usage, less highly compact microprograms practical [4].

Two important characteristics usually sought by proponents of such compilers are (1) a powerful, high-level input language and (2) a high degree of target-machine independence for the user. Typical versions of such compilers are structured in two basic phases conducive to these characteristics. The first phase is a complete compiler taking high-level input source into intermediate-level text. The second phase is a simple, direct translator chosen by the user to transform this intermediate text into actual microcode for his target machine [3], [7].

Although microprogram compilers such as those just mentioned have proved quite promising, one particularly annoying problem remains. This problem is the compactness, or degree of optimization, of the microcode output versus the required compilation time. To be feasible, even medium-to-low-usage microprograms require a fair degree of optimization. Furthermore, such microprograms require short compilation times to make them worthwhile producing. These two requirements are inherently conflicting, especially since microprograms and their formats are traditionally highly target-machine-dependent while the compiler attempting to optimize these microprograms is designed to be highly target-machine-independent. In other words, it is extremely difficult to efficiently optimize a machine-dependent process by means of a machine-independent mechanism [2], [7], [8].

One possible solution to this problem is to relieve the microprogram compiler of a large part of its optimization chores. The authors propose moving many local optimization duties out of the compiler and across the software-hardware boundary into the hardware realm of the target machine. The hardware microcode optimizer, HMO, is a simple hardware algorithm capable of condensing a sequence of essentially horizontal microinstructions to increase their bit density and parallelism. It is reasonable to expect that a hardware implementation of such a hardware-dependent process can be both fast and cost-effective [9]. Furthermore, by improving the efficiency of software microprogram compilers, the HMO algorithm can increase the practicality of a truly user-microprogrammable computer system.

It must be stressed that the overall microcode optimization process being proposed in this paper would consist of two basic levels, or phases. The first level, performed by the software microprogram compiler, would be the more complex, global, primarily machine-independent type of optimization procedures. The second level, performed by the HMO algorithm and associated hardware (after receiving the software compiler's generated microcode), would consist ideally of as much as possible of the less complex, local, highly machine-dependent type of optimization.

I. DESCRIPTION OF BASIC HMO ALGORITHM

Consider how the major internal hardware components of a computer are involved with the flow of data, or information, through-out the machine. With respect to the HMO algorithm, the following classification of such components is useful: (1) a fixed source, or data constant (e.g., a pseudo-register which supplies a hardwired constant of 0 or 1 to other components), (2) a data transformer (e.g., an adder, shifter, working register, main memory during a load-from-memory instruction, etc.), or (3) a data sink (e.g., main

memory during a store-into-memory instruction). However, since the production of data constants is a fixed operation, with no inputs on which to perform a function, HMO need not be concerned with such constants. Their control is inherently covered in the control of the transformers and sinks to which they supply inputs.

Concerning the control of active, functional components such as transformers and sinks, two major areas of interest are the supplying of inputs and the calling for outputs, with only the former area actually being needed for sinks. If we consider now a flexible microprogrammable architecture such as that shown in Fig. 1, these two areas become nothing more than particular groups of horizontal microinstruction bits controlling appropriate register transfers. One other area of interest for both transformers and sinks is timing, or the time interval required for them to complete their respective functions. This timing requirement implies a certain needed minimal distance between some microinstructions, or microwords, in any microinstruction stream. Assume for now that the microcycle time of HM1 in Fig. 1 is such that this needed distance is only one microcycle. This means, for example, that it is acceptable for one microword to excite an adder "input supply" and the microword immediately following to excite the corresponding adder "output call."

Notice that the "latching" type architecture of HM1 affords the microprogrammer virtually complete timewise independence of when inputs are supplied to a data transformer such as the adder. He may, in fact, "latch" in adder inputs during different microcycles. All he must do is make certain all desired inputs are fed at least one microcycle before he calls for the corresponding transformer output. Thus, the HMO algorithm can simply sequence through a stream of microinstructions, condensing (essentially combining) all microinstructions containing "input supply" bits into one instruction, until it reaches the point where the next instruction contains an "output call" bit corresponding to the already condensed "input supplies." At this point, the algorithm must temporarily stop condensing, save (or execute) the newly formed condensed instruction, and then proceed to condense again starting with the next microinstruction in the stream. What all this means is that the HMO algorithm can produce, from a microinstruction stream which exercises HM1's hardware in a purely serial fashion, a corresponding condensed stream which exercises HM1's hardware in a highly parallel fashion.

Unlike data transformers, data sinks, which don't require "output call" bits, make it difficult for the HMO algorithm to spot the point where condensing must temporarily stop. This problem can be solved by requiring that, following the desired sink inputs, a succeeding microinstruction appear containing a "1" bit which actually excites, or causes, the sinking of these preceding inputs. By controlling sinks in this manner, these sinks appear identical to data transformers as far as the HMO algorithm is concerned. It always sees a series of "input supplies" followed at least one microcycle later by a microword containing a control bit which, for transformers, calls for passage of the transformed data to some other point and, for sinks, causes the actual sinking action to be performed. Therefore, the HMO algorithm can now handle transformers and sinks with equal facility. The major hardware needed is a simple set of combinational logic "inhibit" functions which are driven both from the condensed instruction being formed and from the next instruction in the stream. At least one of these functions is activated when the next instruction contains an "output call" corresponding to "input supplies" in the condensed instruction. Further condensing is thus inhibited and the algorithm starts anew on the next instruction.

Note that Fig. 2 allows the option of either saving a condensed result for later use (pre-pass compilation) or executing this result immediately without saving it (interpretive execution). Interpretive execution would be inefficient for all but extremely low-usage microprograms, as it would require repeated condensing of repeatedly executed blocks of microcode. Therefore, all discussion that follows assumes that the HMO algorithm is being used as a pre-pass condensing compiler.

Fig. 3 contains two examples illustrating the algorithm's use. Note that the second example illustrates how the authors would ideally like to handle conditional branch microinstructions. This ideal method would be essentially to allow the HMO algorithm to condense along the "non-branch" path (i.e., the path which is expected to be taken most of the time). Then, later, the algorithm could be restarted separately along the yet untouched "branch" path.

Finally, Fig. 4 depicts one example of the "inhibit" functions which provide the logical signals to control the HMO algorithm.

II. INTEGRATING THE ALGORITHM INTO THE MICROPROGRAMMABLE SYSTEM

While Section I. presented a brief overview of the basic HMO algorithm, this section presents some intricate design problems incurred in evolving the algorithm into a well integrated system component. Since the algorithm is actually the final phase of the overall microcode compilation process, many of these problems involve considerations of whether to allocate a particular function to the software compiler or to the hardware algorithm. However, as will be seen, other problems are not related to such an allocation and must be resolved on other bases.

A. Handling Conditional Branch Microinstructions

As stated in Section I., the second example of Fig. 3 depicts an extreme, idealistic scheme for handling conditional branches, a scheme which allows condensing not only "up to and including" conditional branches but "past" them as well. The astute reader will notice that, in the condensed code, the two transfers "AI1←PGC" and "AI2←0" will always be performed, whereas, in the uncondensed code, they would have been performed only if the "non-branch" path were taken. Obviously, such a situation could result in erroneous results from the condensed code.

This problem can be solved by (1) allowing room in the microinstruction format for not only the normal section of control bits but also for a conditional section of control bits to be executed only if the "branch" path

is taken or by (2) simply prohibiting condensing "past" conditional branches. Although research results tend to favor solution (2), it must be pointed out that the choice between these two solutions is virtually unrelated to the compiler versus algorithm allocation question. Instead the choice here must be made primarily on the basis of the tradeoff between the complex microinstruction format (and related problems) of solution (1) and the slight microprogram condensability loss of solution (2).

B. Paralleling of Completely Independent Tasks

Fig. 5 is an abstract example illustrating a possible condensing inefficiency. Note that although the groups of uncondensed code in examples (a) and (b) are equivalent, the condensed code in example (b) is more compact than that in example (a). This variance is a direct, but subtle, result of the HMO algorithm's simple condensing scheme presented in Section I. For example, the alert reader may wonder why, in example (a), the algorithm could not have looked at least two instructions ahead of "ACCUM←DATA1" to recognize that, even though "AI1←ACCUM" is inhibited (by an accumulator inhibit function) from condensing, "INDEX←DATA2" could have been brought up past "AI1←ACCUM" and condensed onto "ACCUM←DATA1." Indeed, it appears that a scheme in which the algorithm, during any given condensing step, is allowed to look far ahead and propagate uninhibited instructions (or parts of instructions) up past inhibited instructions could produce the compact condensed code of example (b) directly from the uncondensed code of example (a). However, suffice it to say that research has demonstrated many intricate problems (hardware complexity, difficulty of assuring condensed code equivalency and proper addressing) with such a scheme.

Rather than resort to such a "messy" scheme, the software compiler can instead be used to pretailor the code it feeds to the HMO algorithm. The basic algorithm works more efficiently when its input (uncondensed) code is ordered so that completely independent tasks do not follow one another in completely serial fashion. Essentially, the code of Fig. 5 is intended to show two such independent tasks, a multistep transfer of DATA1 to AI1 and a multistep transfer of DATA2 to AI2. In example (a) these tasks are arranged entirely sequentially while, in (b), they are overlapped in a slightly more parallel fashion, thus allowing the basic algorithm of Section I. to produce a more compact result. Therefore, it should be the job of the software compiler to search for such completely independent tasks, or code groups, and reorder them as needed to ensure they are not left completely sequential. Such paralleling of independent tasks is a relatively machine-independent, global process better suited to the software compiler than the hardware algorithm.

C. Removing Non-Productive Transfers

Fig. 6 is another abstract example illustrating a possible condensing problem. Note that the first two instructions in the uncondensed code both supply information to adder input AI1. In particular, because the second instruction "writes over" the information supplied to AI1 during the first instruction without first using the corresponding added result (by passing adder output AO1 somewhere, for example), the transfer to AI1 in the first instruction is a "non-productive" ("negated" [10]) transfer.

The basic HMO algorithm of Section I. would, in fact, attempt to condense the two transfers to AI1 together. This condensing can be used beneficially to remove the "non-productive" transfer as long as an appropriate condensing technique is used. This technique necessitates partitioning the control bits of each microword into the mutually exclusive input sets of each hardware register. For example, the AI1 input set consists of control bits 8, 9, and 10 (see Fig. 1). The technique then consists of (1), for non-zero input sets in the upcoming word to be condensed, writing this non-zero set over the corresponding set in the accumulating condensed result and (2), for all-zero input sets in the upcoming word to be condensed, leaving the corresponding set in the accumulating condensed result as is. If such a condensing technique is used (whenever the inhibit functions permit condensing), the basic HMO algorithm can easily produce the condensed result shown on the right of Fig. 6. Thus, "non-productive" transfer removal can be handled adequately, at least on a local scale, by the hardware algorithm, without special help from the software compiler.

III. ARCHITECTURAL REQUIREMENTS

As expected, easy and efficient implementation of the HMO algorithm dictates certain architectural characteristics as desirable. This section presents a summary of the major characteristics so dictated.

A. General Characteristics

The architecture of HM1 must be such that all fundamental operations under microprogrammed control consist of two elementary steps which can be intuitively termed the "starting" and "finishing" steps. As implied in Section I., two such steps are found quite naturally for data transforming units such as the adder. However, much time and care went into the rather unusual main memory controller shown in Fig. 1 so that even the data sinking operation of a "store into memory" consists of the needed two basic steps.

The "latching," or "register transfer," type architecture indicated in Fig. 1 is useful for many reasons, some of which are (1) it readily supports the "two-step" structure mentioned above, (2) it gives the microprogrammer (and the software compiler) much freedom from hardware timing requirements (e.g., freedom to supply the three adder inputs of Fig. 1 in sequential fashion, in parallel fashion, etc.) and (3) it lends itself to pipelining slower microcontrolled functions to various degrees (a technique which research indicates may be useful in the interest of machine speed).

B. Microinstruction Formats

As the control section format, a horizontal, unencoded control section having one bit per register transfer is ideal. This arrange-

ment readily supports a neat, two-level realization of the algorithm's inhibit functions, allowing these functions to be driven directly from the control register (Fig. 2) and from the control memory output lines feeding the control register.

Concerning microinstruction addressing schemes, flexibility is the key requirement. Research has shown that employment of the algorithm in its simple, one-pass Section I. form yields condensed instructions which are linked together but interspersed with remaining groups of "garbage" instructions. During run time, execution will proceed by "leap frog" style jumps which circumvent these garbage instructions. Thus, at the very minimum, a scheme employing one complete "next address" in each microword (Fig. 2) is needed (as opposed to, say, the sole use of a separate microprogram counter, or pointer, register).

As suggested in Section II., use of the ideal conditional branch condensing philosophy of Fig. 3 necessitates a quite complex microinstruction format, but, if one prohibits condensing "past" conditional branches, many instruction formats between this extremely complex one and the required minimal one of Fig. 2 become possible. However, no matter what overall instruction format is chosen, research indicates it is in all cases desirable, though not always necessary, to have the "branch" path address be completely independent of the "non-branch" path address.

C. Control Memory Characteristics

Although many types of control memory can be used, one arrangement well suited to supporting the HMO algorithm is to use the same memory type (and speed) for both main and control memories. This arrangement, used in varying degrees on the IBM 360/Model 25 [11] and the Burroughs B 1700 [12], helps to achieve realization of the Section I. assumption that one control memory microcycle is sufficient to complete any elemental machine operation.

Of the many possible methods which can be used to actually implement the HMO algorithm, a firmware implementation's flexibility is particularly attractive. A feasible firmware implementation can be realized by using two separate control memories (or, at least, two separate memory sections), one containing the HMO algorithm plus other factory-fixed routines and the other containing the user's microprograms. While condensing, the factory-fixed, restricted-access memory would be operating on the contents of the user-accessible memory. Again, this control memory arrangement employing both fairly-restricted and easily-accessible memories has been used in varying degrees on real production machines like the Burroughs B 1700 [12] and the Microdata 1600 [13].

IV. ADAPTATIONS FOR PERFORMANCE ENHANCEMENT

Up to this point, the simplifying Section I. assumption that one microcycle is sufficient time for all elemental machine operations has not been questioned. Obviously, such an assumption, if adhered to rigidly and inflexibly, could result in a control memory cycle too long to allow acceptable machine performance.

This section presents some techniques which can help prevent such possible performance degradation. Basically, these techniques allow cycling of control memory at a reasonable, chosen speed rather than restricting it to cycling at least as slowly as the slowest elemental operation under its control. While the techniques of the first two subsections are modifications of HM1's execution hardware, the technique of the last subsection is a modification of the basic HMO algorithm itself.

A. Use of Programmed Wait Loops

By incorporating "busy" signal indicators into those operations which are of longer duration than the control memory cycle, conditional branch microinstructions can be made to branch to an "increment-the-PGC-and-then-go-to-FETCH" routine. Thus, conditional machine instructions for such operations can be microprogrammed so as to simply skip the next machine instruction whenever the desired operational facility is still "busy" from some previous use.

For example, consider I/O operations. With such machine instructions available, it is a simple matter to program an I/O "transfer/idle" (or "wait") loop at the machine instruction level. (Note that, given a rich enough addressing scheme for conditional branch microinstructions, there is no real reason why such "wait" loops could not also be implemented at the microinstruction level.)

B. Incorporation of Established Hardware Performance Enhancement Techniques

If control memory is to be cycled at a rate too fast to allow one-cycle completion of some slower elemental operations, then several established hardware techniques can be employed to help avoid the implied timing hazards which could result during execution. For example, "request/reply" control interfacing can be used to ensure that control memory idles while awaiting the results of slower, previously initiated elemental microcontrolled operations.

On the other hand, an adaptation of the Tomasulo algorithm [14], [15] can be employed so that the microprocessor need not often be idled unproductively. Instead of idling the microprocessor can pass appropriate "tags" to the intended destinations of the yet unavailable results and simultaneously mark such destinations as "busy awaiting information." When later available, the actual information itself would then be passed to all appropriately "tagged" units and the associated "busy bits" turned off. This Tomasulo type hardware can permit a rapidly cycled control memory to proceed executing even in the face of temporarily unavailable information, with the possible beneficial side effect of eliminating the use of temporary storage stations called for in the microcode being executed.

While the other techniques of Section IV. are essentially means of compensating for (during execution) microprograms which were condensed under the "one-microcycle assumption" even in situations where this assumption is not completely valid, pipelining [15] can be a useful technique in increasing the validity and practicality of the "one-microcycle assumption." That is, rather than simply shortening the control memory cycle, pipelining can be used in conjunction with such shortening to simultaneously shorten the

required time of slower microcontrolled operations. For example, by insisting that the AOl register of Fig. l be a real physical latching register (which has not been assumed thus far), the overall process of addition (from operand source registers to result destination registers) would then consist of three elemental stages instead of the present two stages. Thus, pipelining yields more, but shorter, elemental micro-operations for a given process, making the "one-microcycle assumption" easier to meet even if the control memory cycle is shortened.

C. Use of Different "Fields of View" for Different Inhibit Functions

Unlike the other techniques already presented, the following technique proposes dropping the "one-microcycle assumption" of the basic HMO algorithm and giving the algorithm the capability to ensure different length "timing gaps" (in its output stream of condensed microcode) for different length elemental microcontrolled operations. By setting each inhibit function's "field of view" equal to the number of microcycles needed to complete the machine operation scrutinized by that inhibit function, appropriate "timing gaps" for all such operations can be produced (where "field of view" is the number of microinstructions an inhibit function can look ahead from the condensed result being formed in the condensing register).

Specifically, by employing a first-in-first-out stack (through which microinstructions are sequenced up to the condensing register), inhibit functions could be driven both from the condensing register and from a particular stack position appropriate to the desired "field of view." For example, the second position in the stack would be used to create a "field of view" of two for those operations requiring two control memory cycles for completion.

CONCLUSION

This paper has proposed a hardware algorithm which could enable a microprogrammable machine to do its own local, machine-dependent optimization of user-written microprograms, leaving the global, machine-independent optimization to an associated software compiler. In fact, one software microprogram compiler could efficiently serve a group of logically different, but architecturally similar, machines, each possessing an implementation of the HMO algorithm enabling it to do its own machine-dependent condensing and "cycle squeezing." Such a system should be the ideal environment for a software compiler which can efficiently serve several different machines but still present the user with a maximum degree of machine independence as he writes a microprogram for a particular, chosen machine.

Section I. presented the algorithm in very basic form and described its optimization approach of transforming microinstruction streams exhibiting serial machine hardware utilization into equivalent condensed streams exhibiting highly parallel hardware utilization [16]. Then, Section II. discussed some of the subtle design details involved in evolving the algorithm into a true system component that works well with other system components. Next, Section III. presented some architectural characteristics suitable to the algorithm's implementation. It is encouraging to note that these characteristics are not exotic ones. On the contrary, many are found on real production machines, thus implying cost effectiveness. Finally, Section IV. discussed both possible modification of the basic algorithm and also incorporation of existing, established hardware algorithms and control techniques as useful means of ensuring an acceptable level of machine performance.

Since the algorithm presented in this paper is new and untried, many practical questions still remain unanswered. For example, since the algorithm itself and the horizontally microcontrolled architecture of HMl were developed jointly to complement each other, the algorithm's usefulness in direct application to significantly different hardware layouts (such as a strictly vertically microprogrammable machine) is uncertain at this time. Similarly, until the HMO algorithm and an associated software compiler are actually built and implemented so that the exact areas of software/hardware cooperation and separation in the overall microcode optimization process can be specifically determined, it would be extremely difficult, if not futile, to attempt to derive meaningful, precise numerical evaluation measures of the algorithm's efficiency or performance. Indeed, the lack of appropriate, precise evaluation measures to guide the design of novel developments is more often the case than not [17]. As a result, the designer must often rely, at least initially, on less precise, more subjective tradeoffs and decisions (such as those of Section II.) to guide his work.

ACKNOWLEDGEMENT

The authors wish to thank C. V. Ramamoorthy for his helpful comments and suggestions offered to aid in the preparation of this paper.

REFERENCES

[1] R. K. Clark, "Mirager, the "Best-Yet" Approach for Horizontal Microprogramming," Proceedings of ACM '72, Association for Computing Machinery, New York, 1972, pp. 554-560.

[2] M. Hattori, M. Yano, and K. Fujino, "MPGS: A High-Level Language for Microprogram Generating System," Proceedings of ACM '72, Association for Computing Machinery, New York, 1972, pp. 572-581.

[3] S. G. Tucker, "Microprogram Control for System/360," IBM Systems Journal, Vol. 6, No. 4, pp. 222-241, 1967.

[4] R. H. Eckhouse, Jr., "A High-Level Microprogramming Language (MPL)," AFIPS Conference Proceedings, 38 (SJCC 1971), pp. 169-177.

[5] R. F. Rosin, "Contemporary Concepts of Microprogramming and Emulation," Computing Surveys, Vol. 1, No. 4, pp. 197-212, Dec., 1969.

[6] M. J. Flynn and R. F. Rosin, "Microprogramming: An Introduction and a Viewpoint," IEEE Transactions on Computers, Vol. C-20, No. 7, pp. 727-731, July, 1971.

[7] S. S. Husson, Microprogramming: Princi-

ples and Practices, Englewood Cliffs, N. J.: Prentice-Hall, Inc., 1970, pp. 125-144.

[8] C. V. Ramamoorthy, M. Tabandeh, and M. Tsuchiya, "A Higher Level Language for Microprogramming," $MICRO_6$ The Sixth Annual Workshop on Microprogramming, College Park, Maryland, Sept., 1973 (Preprints), pp. 139-144.

[9] H. Falk, "Hard-Soft Tradeoffs," IEEE Spectrum, Vol. 11, No. 2, pp. 34-39, Feb., 1974.

[10] R. L. Kleir and C. V. Ramamoorthy, "Optimization Strategies for Microprograms," IEEE Transactions on Computers, Vol. C-20, No. 7, pp. 783-794, July, 1971.

[11] C. G. Bell and A. Newell, Computer Structures: Readings and Examples, United States of America: McGraw-Hill, Inc., 1971, pp. 567-569.

[12] Burroughs B 1700 Systems Reference Manual, Preliminary Edition, Burroughs Corporation, Systems Documentation, Technical Information Organization, TIC-Central, Detroit, Michigan, 1972, pp. 1.7-1.8, 1.10, 3.1.

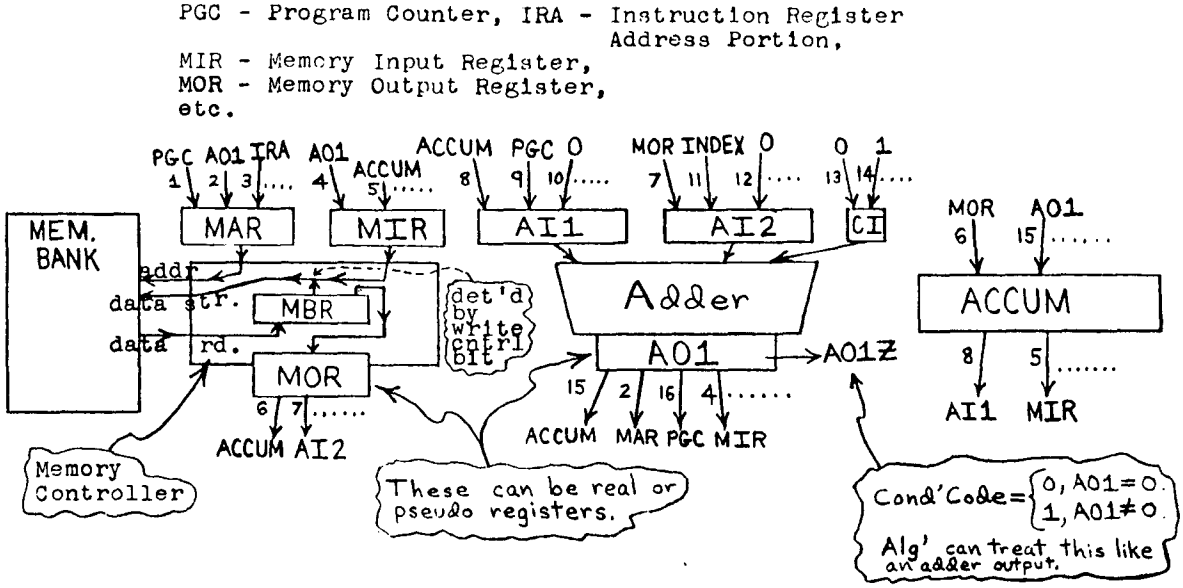
[13] Microprogramming Handbook, Second Edition, Microdata Corporation, Santa Ana, California, 1971, pp. 317-318.

[14] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," IBM J. of Res. and Dev., Vol. 11, No. 1, pp. 25-33, Jan., 1967.

[15] M. J. Flynn, "Very High-Speed Computing Systems," Proceedings of the IEEE, Vol. 54, No. 12, pp. 1901-1909, Dec., 1966.

[16] A. K. Tirrell, "A Study of the Application of Compiler Techniques to the Generation of Micro-Code," Proc. of ACM SIGPLAN-SIGMICRO Interface Meeting, Harriman, New York, May, 1973 (Preprints), pp. 67-85.

[17] W. T. Wilner, "Design of the Burroughs B 1700," AFIPS Conference Proceedings, 41 (FJCC 1972), pp. 489-497.



NOTE: The #'s indicate the microinstruction bit controlling a transfer.

Fig. 1 Subset of HM1 (Hypothetical Machine 1)

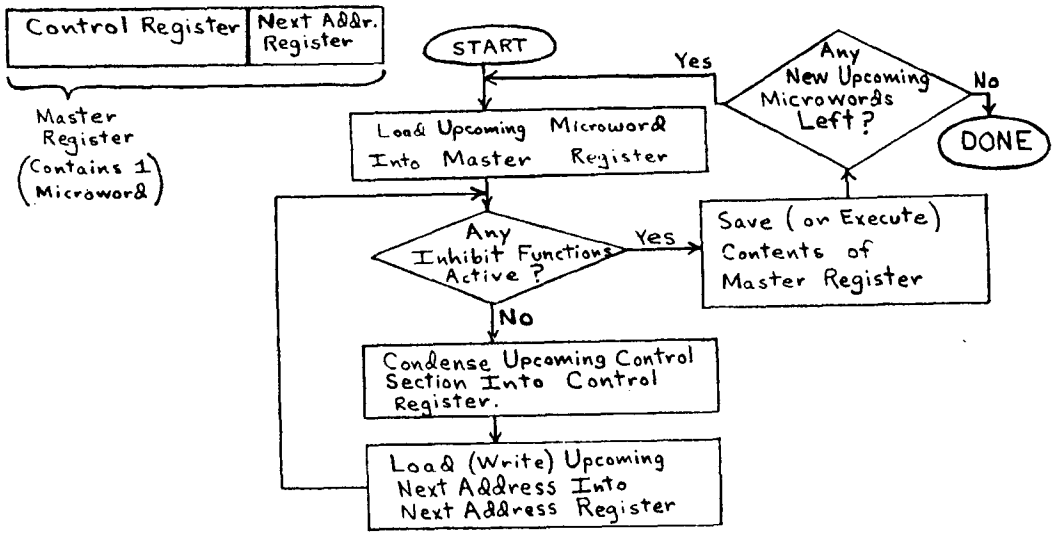
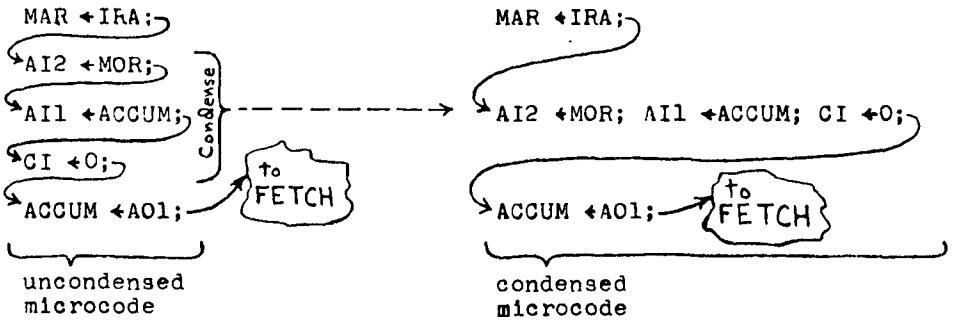


Fig. 2 Flow Chart of Basic HMO Algorithm

The following example illustrates condensing of an "add" with direct address that performs ACCUM←ACCUM + MEM(IRA);



The following example illustrates essentially how the author hopes to handle conditional branch microwords. The example is a "mem. increment and skip next instr. if result is 0" instruction. Note that "EFF ADDR" means Effective Address.

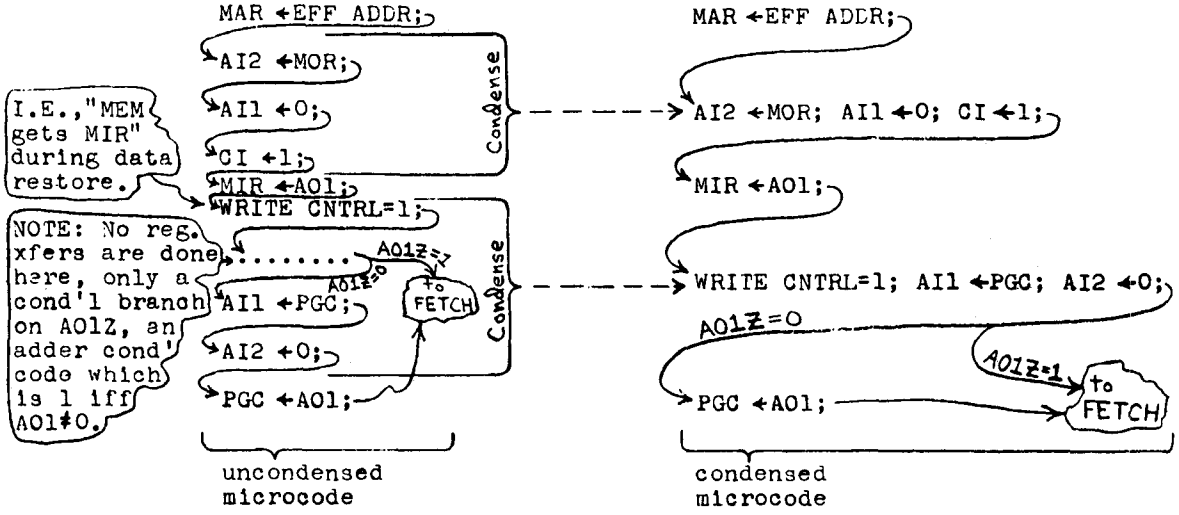
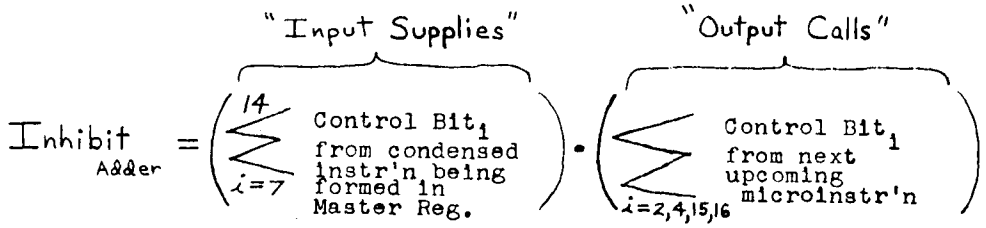


Fig. 3 Some "Before & After" Examples



where { "Σ" ⇒ Logical OR
"•" ⇒ Logical AND

NOTE: Refer to Fig.'s 1 & 2 for explanation of "Master Reg.", various control bit #'s, etc.

NOTE: "Inhibit" functions for other components in HM1 are formed in a similar manner to the one shown above for the adder.

Fig. 4 "Inhibit" Function Example

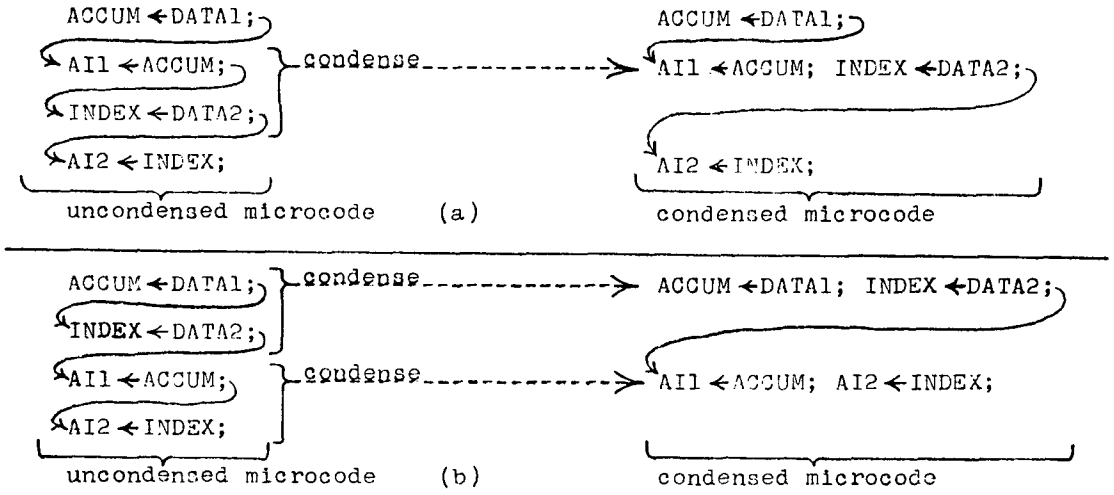


Fig. 5 Paralleling Independent Tasks

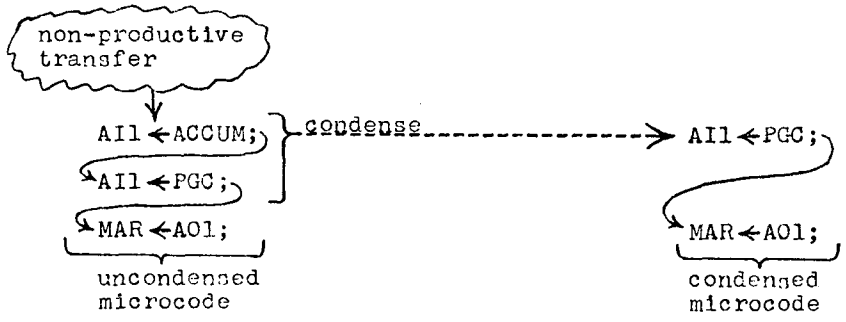


Fig. 6 Non-Productive Transfer Removal