
Doctoral Dissertations

Student Theses and Dissertations

1974

Towards a design of HMO, an integrated hardware microcode optimizer

James Oliver Bondi

Follow this and additional works at: https://scholarsmine.mst.edu/doctoral_dissertations



Part of the [Electrical and Computer Engineering Commons](#)

Department: Electrical and Computer Engineering

Recommended Citation

Bondi, James Oliver, "Towards a design of HMO, an integrated hardware microcode optimizer" (1974). *Doctoral Dissertations*. 285.

https://scholarsmine.mst.edu/doctoral_dissertations/285

This thesis is brought to you by Scholars' Mine, a service of the Missouri S&T Library and Learning Resources. This work is protected by U. S. Copyright Law. Unauthorized use including reproduction for redistribution requires the permission of the copyright holder. For more information, please contact scholarsmine@mst.edu.

TOWARDS A DESIGN OF HMO,
AN INTEGRATED HARDWARE MICROCODE OPTIMIZER

by

JAMES OLIVER BONDI, 1949-

A DISSERTATION

Presented to the Faculty of the Graduate School of the

UNIVERSITY OF MISSOURI - ROLLA

In Partial Fulfillment of the Requirements for the Degree

DOCTOR OF PHILOSOPHY

in

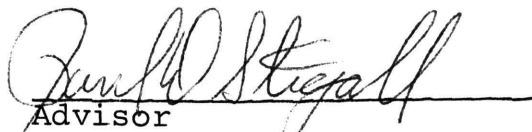
ELECTRICAL ENGINEERING


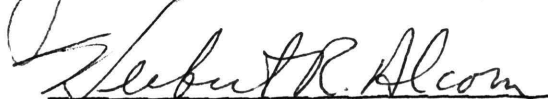
1974

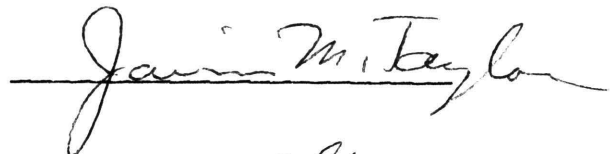
T3035

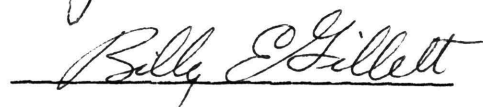
90 pages

c.1


Advisor





ABSTRACT

This paper discusses an algorithm for optimizing the density and parallelism of microcoded routines in micro-programmable machines. Besides presenting the algorithm itself, this research also analyzes the algorithm's uses, design integration problems, architectural requirements, and adaptability to conventional machine characteristics. 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.

ACKNOWLEDGMENTS

The author is very grateful to Paul D. Stigall for his continued advice and assistance in preparing this manuscript. The author further wishes to thank James H. Tracey for his helpful suggestions and comments, and C. V. Ramamoorthy for his gracious critique of basically the main body of this manuscript. Finally, the author acknowledges the willing assistance of Herbert R. Alcorn, Javin M. Taylor, and Billy E. Gillett.

A special note of thanks goes to my wife, Judy, for her patient and repeated proofreadings of this manuscript throughout its development.

TABLE OF CONTENTS

	PAGE
ABSTRACT	ii
ACKNOWLEDGMENTS.	iii
LIST OF FIGURES.	vi
INTRODUCTION	1
I. DESCRIPTION OF <u>BASIC</u> HMO ALGORITHM.	4
II. INTEGRATING THE ALGORITHM INTO THE MICRO- PROGRAMMABLE SYSTEM [10].	8
A. HANDLING CONDITIONAL BRANCH MICROINSTRUCTIONS	8
B. PARALLELING OF COMPLETELY INDEPENDENT TASKS	9
C. REMOVING NONPRODUCTIVE TRANSFERS.	11
III. ARCHITECTURAL REQUIREMENTS [13]	13
A. GENERAL CHARACTERISTICS	13
B. MICROINSTRUCTION FORMATS.	14
C. CONTROL MEMORY CHARACTERISTICS.	15
IV. ADAPTATIONS FOR PERFORMANCE ENHANCEMENT [17].	17
A. USE OF PROGRAMMED WAIT LOOPS.	17
B. INCORPORATION OF ESTABLISHED HARDWARE PERFORMANCE ENHANCEMENT TECHNIQUES.	18
C. USE OF DIFFERENT "FIELDS OF VIEW" FOR DIFFERENT INHIBIT FUNCTIONS	20
CONCLUSION	22

TABLE OF CONTENTS (continued)	PAGE
REFERENCES31
VITA34
APPENDIX A. HMI DETAIL.35
A. OVERALL DESCRIPTION36
B. MCR LAYOUT.38
C. "INHIBIT" FUNCTIONS39
D. CONTROL SECTION ENCODING.40
E. "NEXT ADDRESS" SELECTION.41
APPENDIX B. HMO ALGORITHM DETAIL.45
A. REQUIREMENTS.46
B. USES.46
C. GENERAL CHARACTERISTICS AND ASSUMPTIONS47
D. RESULTANT PROPERTIES.48
E. CONDENSING TECHNIQUE.49
F. CONDENSING LIMITS51
G. SPECTRUM OF POSSIBLE MICROINSTRUCTION FORMATS	.51
H. CONDENSING APPROACH57
I. MORE DETAIL ON PRE-PASS CONDENSING COMPILER USE61
J. SOME IMPLEMENTATION CONSIDERATIONS.63
APPENDIX C. AREAS OF CONCENTRATION FOR FURTHER RESEARCH.73
APPENDIX D. MISCELLANEOUS EXAMPLES.76

LIST OF FIGURES

FIGURE NO.		PAGE
1	Subset of HML (Hypothetical Machine 1)	25
2	Flow Chart of Basic HMO Algorithm. . . .	26
3	Some "Before & After" Examples	27
4	"Inhibit" Function Example	28
5	Paralleling Independent Tasks.	29
6	Nonproductive Transfer Removal	30
A1	HML Working Hardware	42
A2	Microinstruction Format & Addressing . .	43
A3	HML Inhibit Functions.	44
B1	Spectrum of Possible Microinstruction Formats.	67
B2	Potential Problems with Hybrid Single- Length/Double-Length Format.	68
B3	Use of "By-Individual-Bit-Column" Approach	69
B4	Potential Problem with "By-Individual- Bit-Column" Approach	70
B5	Flow Chart of HMO Algorithm as a Pre- Pass Condensing Compiler	71
B6	One Possible Control Memory Layout . . .	72
D1	Problem with Original, One-Step, "Store- Into-Main-Memory" Scheme	77

FIGURE NO.		PAGE
D2	A Peculiarity of the Present Memory Controller	78
D3	Redundant Transfer Removal	79
D4	Appropriate Handling of Functional Unit Direct Feedback Paths.	80
D5	Disguised, Larger-Scale Nonproductive- ness	81
D6	Futility of Microprogramming HMI to Per- form 2 Immediately Successive Output (or Input) Transfers	82
D7	Possible Use of Tomasulo-Type Hardware [18], [19] to Aid HMO Algorithm.	83

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 repertoire 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 author proposes moving many local optimization duties out of the compiler and across the software-hardware boundary into the hardware realm of the target machine. The author's 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.

At this point, the reader may wish to familiarize himself, at least superficially, with the contents and figures of Appendices A, B, C, and D. As he reads the remainder of the main body, he would thus be aware of where, in the appendices, he might refer for more detail. (For example, Figures A2 and A3 of Appendix A may be particularly useful in developing a mental picture of the microinstruction format and associated "inhibit" functions as the main body is read.)

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, throughout 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 HMI 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 HMI 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 HMI's hardware in a

purely serial fashion, a corresponding condensed stream which exercises HMI's hardware in a highly parallel fashion.

Unlike data transformers, data sinks, which do not 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 in the main body 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 author would ideally like to handle conditional branch microinstructions. This ideal method would be essentially to allow the HMO algorithm to condense "past" conditional branches along one of the two available paths (hopefully, the "non-branch" path, or path expected to be taken most of the time). Then, later, the algorithm could be restarted separately along the yet untouched (hopefully "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 [10]

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, in one condensed result, condensing not only "up to and including" conditional branches but "past" them as well, down a selected, "favored" path. 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 "favored" path were taken. Obviously, in general, 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 "favored" path is taken or by (2) simply prohibiting condensing "past" conditional branches. Although present 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 "A11←ACCUM" is inhibited (by an accumulator inhibit function) from condensing, "INDEX←DATA2" could have been brought up past "A11←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, when possible, 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. (Of possible use towards this goal could be techniques for

program segmentation and potential task parallelism detection [11] and allowable code motion [12].) Such paralleling of independent tasks is a relatively machine-independent, global process better suited to the software compiler than the hardware algorithm.

C. REMOVING NONPRODUCTIVE 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 A11. In particular, because the second instruction "writes over" the information supplied to A11 during the first instruction without first using the corresponding added result (by passing adder output A01 somewhere, for example), the transfer to A11 in the first instruction is a "nonproductive" ("negated" [12]) transfer.

The basic HMO algorithm of Section I would, in fact, attempt to condense the two transfers to A11 together. This condensing can be used beneficially to remove the "nonproductive" transfer as long as an appropriate condensing technique is used. This technique necessitates partitioning the control bits of each microword into the mutually exclusive, controlwise independent bit sets controlling each micro-operation (such as the input sets of each hardware register). For example, the A11 input set consists of control bits 8, 9, and 10 (see Fig. 1). The technique

then consists of: (1), for non-zero bit 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 bit 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, "nonproductive" 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 [13]

As expected, easy and efficient support 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 HMI 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 arrangement 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, as a minimal base scheme (from which to build), 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, a scheme better suited to mostly-sequential addressing).

As suggested in Section II, use of the ideal conditional branch condensing philosophy of Fig. 3

necessitates a quite complex microinstruction format. However, 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.

(This minimal format must, of course, be slightly augmented to allow production of, for conditional branches, a second "next address".) However, no matter what overall instruction format is chosen, present 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 (user) control memories. This arrangement, used in varying degrees on the IBM 360/Model 25 [14] and the Burroughs B 1700 [15], 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 of no condensing interest to the algorithm 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 [15] and the Microdata 1600 [16].

IV. ADAPTATIONS FOR PERFORMANCE ENHANCEMENT [17]

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 HMI'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" (or "ready" for the complementary approach) 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 [18], [19] 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 (also possible via a Tomasulo-type routine in the software compiler [12]) 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 [19] 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 A01 register of Fig. 1 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. (Note, however, that more micro-operations/process means not only more required microinstructions/process but also a wider control memory having more bits/microinstruction.)

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.

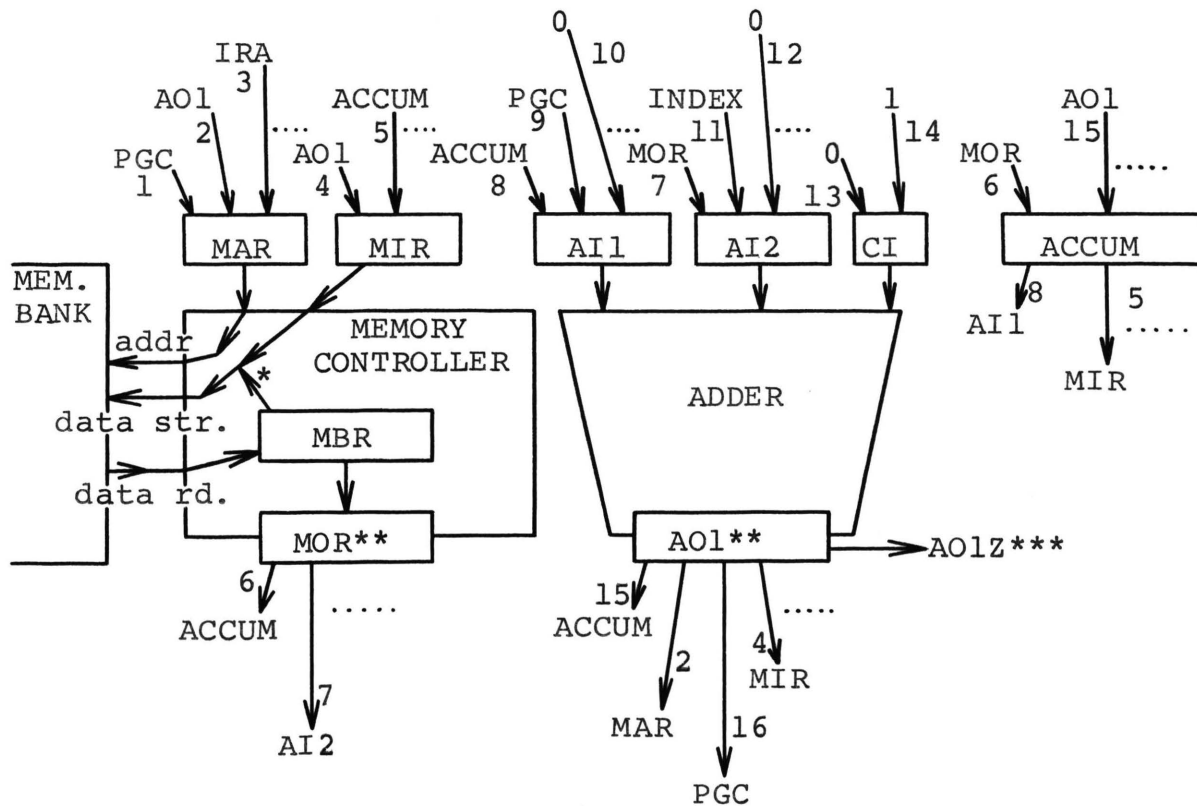
Section I presented the algorithm in very basic form and described its optimization approach of transforming microinstruction streams exhibiting serial machine hardware utilization into equivalent condensed streams exhibiting highly parallel hardware utilization [20], an approach in which the algorithm may accept its input microcode in simple, even purely vertical, form and then produce as output an equivalent, more complex, horizontal stream of microcode [21]. 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 their 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 HMI 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 [22]. 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.

PGC - Program Counter, IRA - Instruction Register
Address Portion,
MIR - Memory Input Register,
MOR - Memory Output Register,
etc.



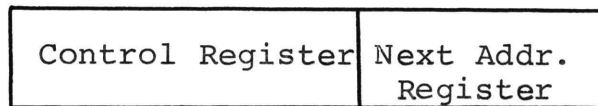
* Write cntrl bit determines gating of either MBR or MIR here.

** These can be real or pseudo registers.

*** This adder cond' code = 1 iff AO1 \neq 0 (cond' code = 0 implies AO1 = 0). The algorithm can treat this cond' code as an adder output.

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

Fig. 1 Subset of HMI (Hypothetical Machine 1)



Master (Control) Register, or MCR
(Contains 1 Microword)

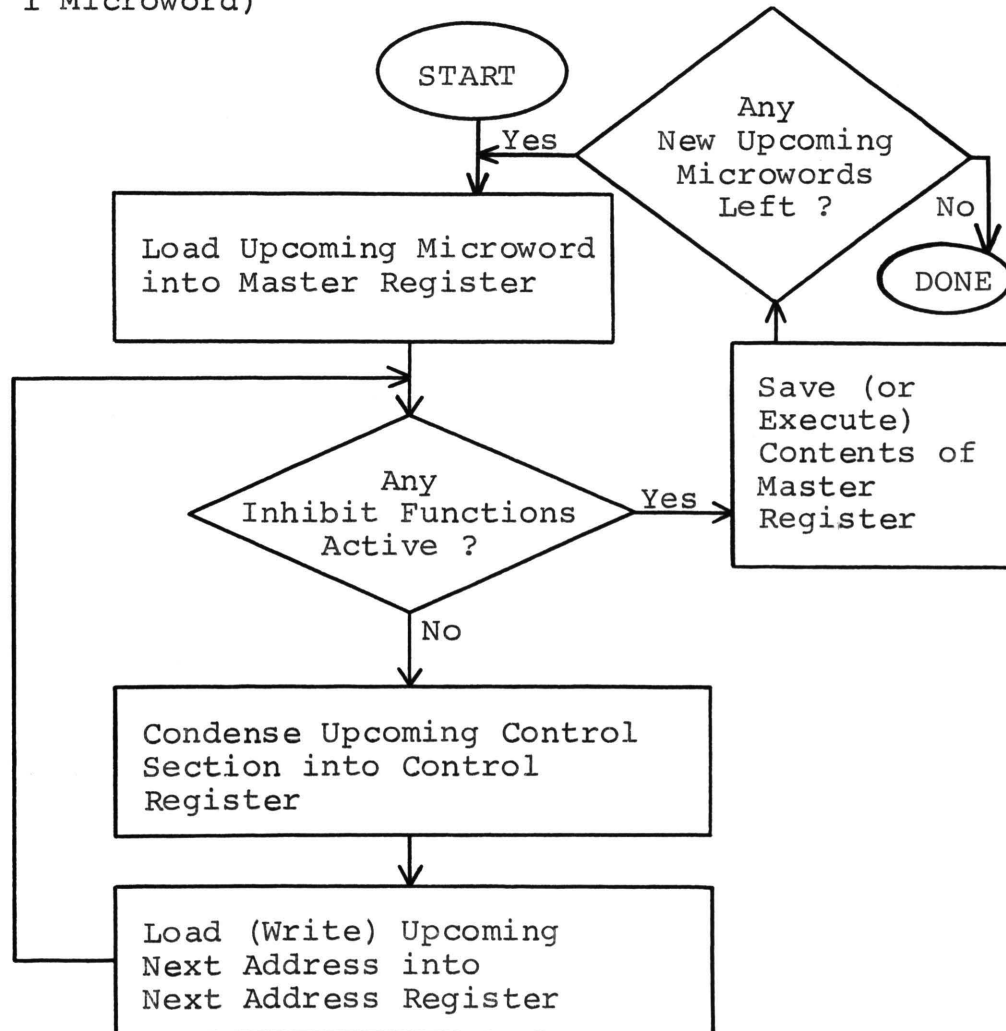


Fig. 2 Flow Chart of Basic HMO Algorithm

The following example illustrates condensing of an "add" with direct address that performs $\text{ACCUM} \leftarrow \text{ACCUM} + \text{MEM}(\text{IRA})$;

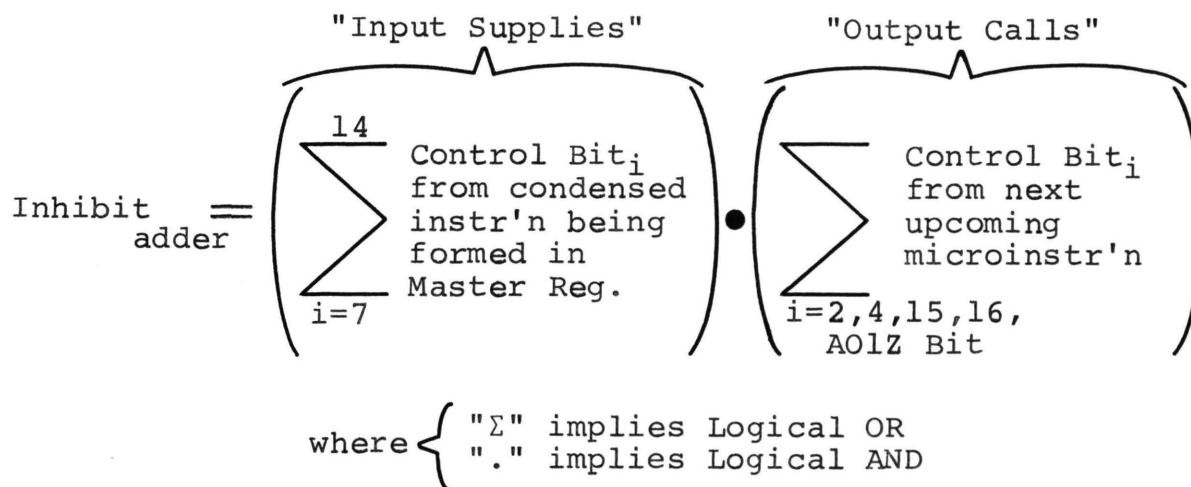
1: MAR←IRA; to 2;		1: MAR←IRA; to 2;
2: AI2←MOR; to 3;	} $\xrightarrow{\text{condense}}$	2: AI2←MOR; AI1←ACCUM; CI←0;
3: AI1←ACCUM; to 4;		to 5;
4: CI←0; to 5;		
5: ACCUM←AO1; to FETCH;		5: ACCUM←AO1; to FETCH;
uncondensed microcode		condensed microcode

NOTE: The label #'s shown above are symbolically representative of control memory addresses and thus, in reality, could correspond to virtually any absolute physical address.

The following example depicts how the author would ideally hope 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.

1: MAR←EFF ADDR; to 2;		1: MAR←EFF ADDR; to 2;
2: AI2←MOR; to 3;	} $\xrightarrow{\text{condense}}$	2: AI2←MOR; AI1←0; CI←1; to 5;
3: AI1←0; to 4;		
4: CI←1; to 5;		
5: MIR←AO1; to 6;		5: MIR←AO1; to 6;
6: WRITE CNTRL=1; to 7;	} $\xrightarrow{\text{condense}}$	6: WRITE CNTRL=1; AI1←PGC;
/* Above implies "MEM←MIR" during data restore */		AI2←0; to(AO1Z) 10,FETCH;
7: to(AO1Z) 8,FETCH;		/* In cond'l branches such as above, parenthesized quantity is a binary- valued cond' code, or CC. If this CC=0, left next address (here "10") is used; if CC=1, right next address (here "FETCH") is used. */
/* No reg. xfers in above, only cond'l branch on cond' code AO1Z */		
8: AI1←PGC; to 9;		
9: AI2←0; to 10;		
10: PGC←AO1; to FETCH;		10: PGC←AO1; to FETCH;
uncondensed microcode		condensed microcode

Fig. 3 Some "Before & After" Examples



NOTE: Refer to Fig.'s 1 & 2 for explanation of "Master Reg.", various control bit #'s, etc. (In above, "AO1Z Bit" refers to the microinstruction bit which performs a cond'l branch based on value of AO1Z.)

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

Fig. 4 "Inhibit" Function Example

1: ACCUM←DATA1; to 2;		1: ACCUM←DATA1; to 2;
2: AI1←ACCUM; to 3;	} $\xrightarrow{\text{con-}} \xrightarrow{\text{dense}}$	2: AI1←ACCUM; INDEX←DATA2;
3: INDEX←DATA2; to 4;		to 4;
4: AI2←INDEX; to NEXT;		4: AI2←INDEX; to NEXT;

/* NEXT is some
"next address"
of no interest
here. */

uncondensed microcode

(a)

condensed microcode

1: ACCUM←DATA1; to 2;	} $\xrightarrow{\text{con-}} \xrightarrow{\text{dense}}$	1: ACCUM←DATA1; INDEX←DATA2;
2: INDEX←DATA2; to 3;		to 3;
3: AI1←ACCUM; to 4;	} $\xrightarrow{\text{con-}} \xrightarrow{\text{dense}}$	3: AI1←ACCUM; AI2←INDEX;
4: AI2←INDEX; to NEXT;		to NEXT;

uncondensed microcode

(b)

condensed microcode

Fig. 5 Paralleling Independent Tasks

<pre> 1: AIL←ACCUM; to 2; /* Above is nonproductive transfer */ 2: AIL←PGC; to 3; </pre>		<p>con- dense</p>	<pre> 1: AIL←PGC; to 3; </pre>
<pre> 3: MAR←A01; to NEXT; /* NEXT is some "next address" of no interest here. */ </pre>			<pre> 3: MAR←A01; to NEXT; </pre>
uncondensed microcode			condensed microcode

Fig. 6 Nonproductive Transfer Removal

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, Vol. 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: Principles 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", MICRO₆

- 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] J. O. Bondi and P. D. Stigall, "HMO, An Integrated Hardware Microcode Optimizer", Proceedings of the Third Annual Texas Conference on Computing Systems, Austin, Texas, Nov., 1974 (Preprints), 12-2-1 - 12-2-8.
- [11] C. V. Ramamoorthy and M. J. Gonzalez, "A Survey of Techniques for Recognizing Parallel Processable Streams in Computer Programs", AFIPS Conference Proceedings, Vol. 35 (FJCC 1969), pp. 1-15.
- [12] 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.
- [13] J. O. Bondi and P. D. Stigall, "HMO, A Hardware Microcode Optimizer", Proceedings of the Second Annual Symposium on Computer Architecture, Houston, Texas, Jan., 1975 (Preprints).
- [14] C. G. Bell and A. Newell, Computer Structures: Readings and Examples, United States of America: McGraw-Hill, Inc., 1971, pp. 567-569, 590-591.
- [15] 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.
- [16] Microprogramming Handbook, Second Edition, Microdata Corporation, Santa Ana, California, 1971, pp. 317-318.

- [17] J. O. Bondi and P. D. Stigall, "Designing HMO, An Integrated Hardware Microcode Optimizer", MICRO₇ The Seventh Annual Workshop on Microprogramming, Palo Alto, California, Sept., 1974 (Preprints), pp. 268-276.
- [18] 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.
- [19] M. J. Flynn, "Very High-Speed Computing Systems", Proceedings of the IEEE, Vol. 54, No. 12, pp. 1901-1909, Dec., 1966.
- [20] 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.
- [21] C. V. Ramamoorthy and M. Tsuchiya, "A High-Level Language for Horizontal Microprogramming", IEEE Transactions on Computers, Vol. C-23, No. 8, pp. 791-801, Aug., 1974.
- [22] W. T. Wilner, "Design of the Burroughs B 1700", AFIPS Conference Proceedings, Vol. 41 (FJCC 1972), pp. 489-497.
- [23] T. L. Dollhoff, "Microprogrammed Control for Small Computers", Computer Design, Vol. 12, No. 5, pp. 91-97, May, 1973.

VITA

James Oliver Bondi was born on May 29, 1949 in St. Louis, Missouri, where he received both his primary and secondary education. He began his college education in September, 1967 at the University of Missouri - Rolla at Rolla, Missouri, later receiving his Bachelor of Science degree in Electrical Engineering from this institution in May, 1971.

He then enrolled in the Graduate School of the University of Missouri - Rolla in September, 1971, later completing his Master of Science degree in Electrical Engineering in July, 1972. A three-year NDEA Fellowship awarded to him has largely financed his graduate studies to date, including most of his current Ph.D. endeavors.

APPENDIX A

HM1 DETAIL

Hypothetical Machine 1, or HML, is a reasonably practical machine developed both to support the HMO algorithm and to facilitate comprehension of algorithmic working details, condensing examples, design considerations, etc. The version of HML here presented is not claimed to be the ultimate version, but rather a basic, yet sufficiently detailed, version usable as a base for initial design analyses.

A. OVERALL DESCRIPTION

HML is a high-speed, general purpose, stored-program, machine-instruction-driven computer. HML control is obtained via a horizontal, microprogrammable, writable control store. Control signals are supplied to HML's working hardware (Fig. A1) from the MCR (Fig. A2), subsequently exciting synchronous transfers via a "major cycle" clock pulse train. The next microinstruction being addressed through the CMAD (control memory address decoder) is always placed in the MCR at the next "major cycle" clock time. Interleaved between this "major cycle" pulse train is a "minor cycle" pulse train used to "mark" the intermediate point of the overall main core memory read-write cycle. Specifically, "MBR←MEM(MAR)" at "minor cycle" times while "MEM(MAR)←either MBR or MIR (depending on WRITE CNTRL bit)" at "major cycle" times.

In Fig. A1, the numbers indicate the MCR bit(s) controlling a particular transfer or gating. If the indicated

function of MCR bits is "true", the indicated transfer or gating occurs; otherwise, it does not. Generally, both the source and destination of all indicated transfers are obvious, the one exception being the combinational logic CMAD of Fig. A2. The CMAD does not "latch onto" the various gated addresses, but merely decodes them to select a particular microword. The resultant transfer of interest is the "major cycle" clocking of the selected microword into the MCR.

To retain flexibility, the MOR, AOl, ESO, and SOl (Fig. A1) may be either real or pseudo registers, but they will always contain the outputs of their respective functional units (without special microprogrammed attention). All other registers shown are real, physical latches. The seven CC's (condition codes) shown in Fig. A1 have the following definitions: AOlZ=1 iff AOl \neq 0, AOlN=1 iff AOl<0, AOlOF=1 iff AOl overflow exists, AOlCO=1 iff carry out of AOl's most significant bit exists, ACCUMLSBZ=1 iff ACCUM's least significant bit \neq 0, KBDRDY=1 iff KBD (keyboard buffer) is ready with some input, PTRRDY=1 iff PTR (printer buffer) is ready for some output.

The shifter unit (Fig. A1) is capable of essentially shifting SI1 one bit to the right or left according to the contents of the SCNTRL register. Additionally, to form the SOl output, the one-bit contents of ESI are shifted into the "leading" (depends on direction) bit position vacated by shifting SI1, and the one-bit ESO output is the bit that

would otherwise be "lost" by shifting S11. This simple shifter arrangement can be microprogrammed to perform various types of shifts and circulates of one or more registers.

B. MCR LAYOUT

Fig. A2 illustrates the specific layout of the MCR, with the control bits basically grouped, for convenience, into register input sets. Generally, when the "all-zero state" exists in a given register set (e.g., bits 1,2,3 all zero for the MAR), the corresponding register will remain unchanged at the next "major cycle" pulse. Similarly, if bits 45-52 are all zero, "normal" microword addressing will occur with the NAR contents being used unconditionally as the next address. Finally, if bit 18 is zero, the main memory will operate in the "read-then-rewrite what was read" mode. Overall, these various types of "zero-state" control were chosen as the "most natural state of affairs" (e.g., for registers, no change). By using "all-zero states" for these "natural, inactive control modes", these inactive modes are readily distinguishable from (and subordinated with respect to) the corresponding "unnatural, active (non-zero) control modes", thus making a wide range of logical condensing techniques (e.g., even logical ORing) usable by the HMO algorithm (as it condenses a microword onto the condensed result being formed).

C. "INHIBIT" FUNCTIONS

Fig. A3 lists the "inhibit" functions which, singly or ORed together, can be used to control the HMO algorithm on HMI. Note that although these inhibits are generally used to ensure proper "one microcycle" timing gaps between functional unit "input supplies" and "output calls", they are flexible enough to be used for special purposes, such as prohibiting condensing "past" conditional branches (last inhibit in Fig. A3). Further note that all inhibits in Fig. A3 (except the last one) are the boolean product of two boolean sum terms, the first term, consisting of possible functional process "starting" steps, being driven from the MCR (condensed result being formed) and the second term, consisting of possible corresponding functional process "finishing" steps, being driven from the control memory output lines (next upcoming microinstruction). However, the last, special purpose, conditional branch inhibit of Fig. A3 consists of only one term driven solely from the MCR. Finally note that the inhibits treat the ACNTRL and SCNTRL modes as functional unit inputs, treat CC usages as functional unit outputs, and treat direct feedback data paths (e.g., bit 20 path in Fig. A1) as both functional unit inputs and outputs.

D. CONTROL SECTION ENCODING

The reader will note that, in general, the control register of Fig. A2 is arranged in an unencoded (or, at best, collection of "1-of-n" coded sets) format. This format is used because using coding, such as binary coding of register input sets, indiscriminately throughout bits 1-54, would complicate formation of the "output calls" term (second term) of the inhibit functions. In other words, encoding according to register input sets disguises this "output calls" information so that at least partial decoding is first required in order to drive the second term of the inhibits. For example, consider the "READ-FROM-MEMORY" inhibit of Fig. A3. Detection of bit 6 in the second term would require some decoding of the encoded ACCUM input set, not to mention bits 7 and 29 and similar bits for other inhibits. The implied complexity becomes evident when one realizes that this decoding (to drive such second terms) needs to be done off of the control memory output lines (or, for the fancy scheme of Section IV.C, off of many positions of a stack)! Note, however, that binary encoding can be readily employed in situations where this encoding does not hinder driving the second term of some inhibit. Such encoding was thus used, for example, in bits 24-26 and bits 39-40 for the ACNTRL and SCNTRL registers respectively. Similarly, encoding could be used for registers such as the CI, which receives only hardwired constants not used as outputs from any other functional unit.

E. "NEXT ADDRESS" SELECTION

Bits 45-54 (Fig. A2) control the selection of the next microword. When these bits are used as intended, only one bit of 45-52 should be "on" in a given microword. If one of the CC selection bits 45-51 is "on", bits 53-54 then allow (encoded) selection of one of four possible conditional branch modes (which involve picking between the NAR and one of two optional hardwired next addresses). Specifically, bit 54=0 activates "FETCH" while bit 54=1 activates "SKIP&-FETCH" (microroutine which increments the PGC by 1 and then goes to "FETCH") as the optional NA (next address). Furthermore, bit 53=0 causes a "1" value of the selected CC to pick the optional NA (and a "0" CC value to pick the NAR) while bit 53=1 causes a "0" value of the selected CC to pick the optional NA (and a "1" CC value to pick the NAR). A study of the CMAD address gating functions shown in Fig. A2 will verify the use of bits 45-54 as just described.

IRO - Instruction Register Op Code Portion,
ESI - Extended Shifter Input, ESO - Extended Shifter Output,
CMAD - Control Memory Address Decoder,
etc.

(See Fig. 1 for other abbreviations.)

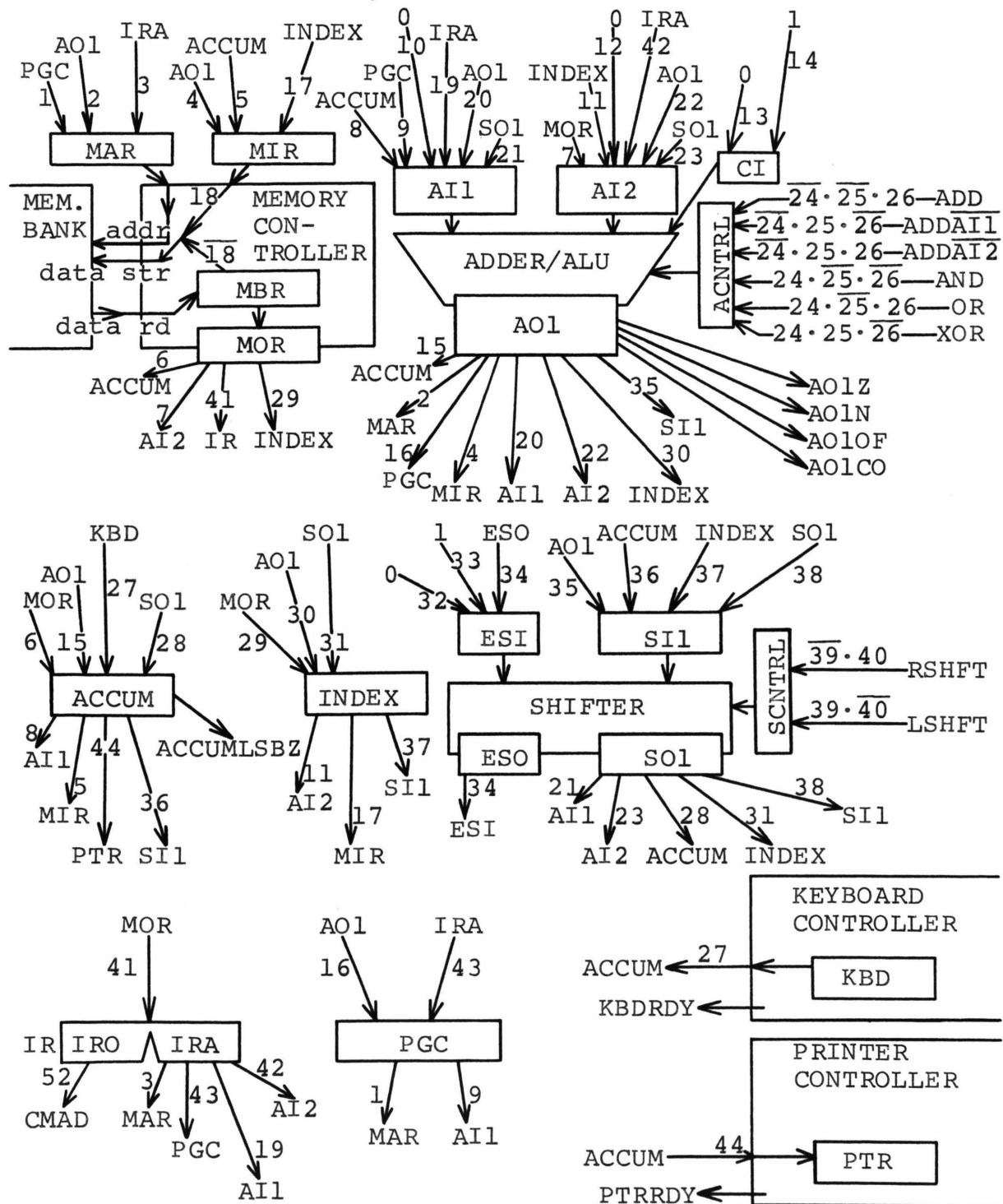


Fig. A1 HMI Working Hardware

NOTE: Bits 1-54 constitute the Control Register (see Fig. 2).

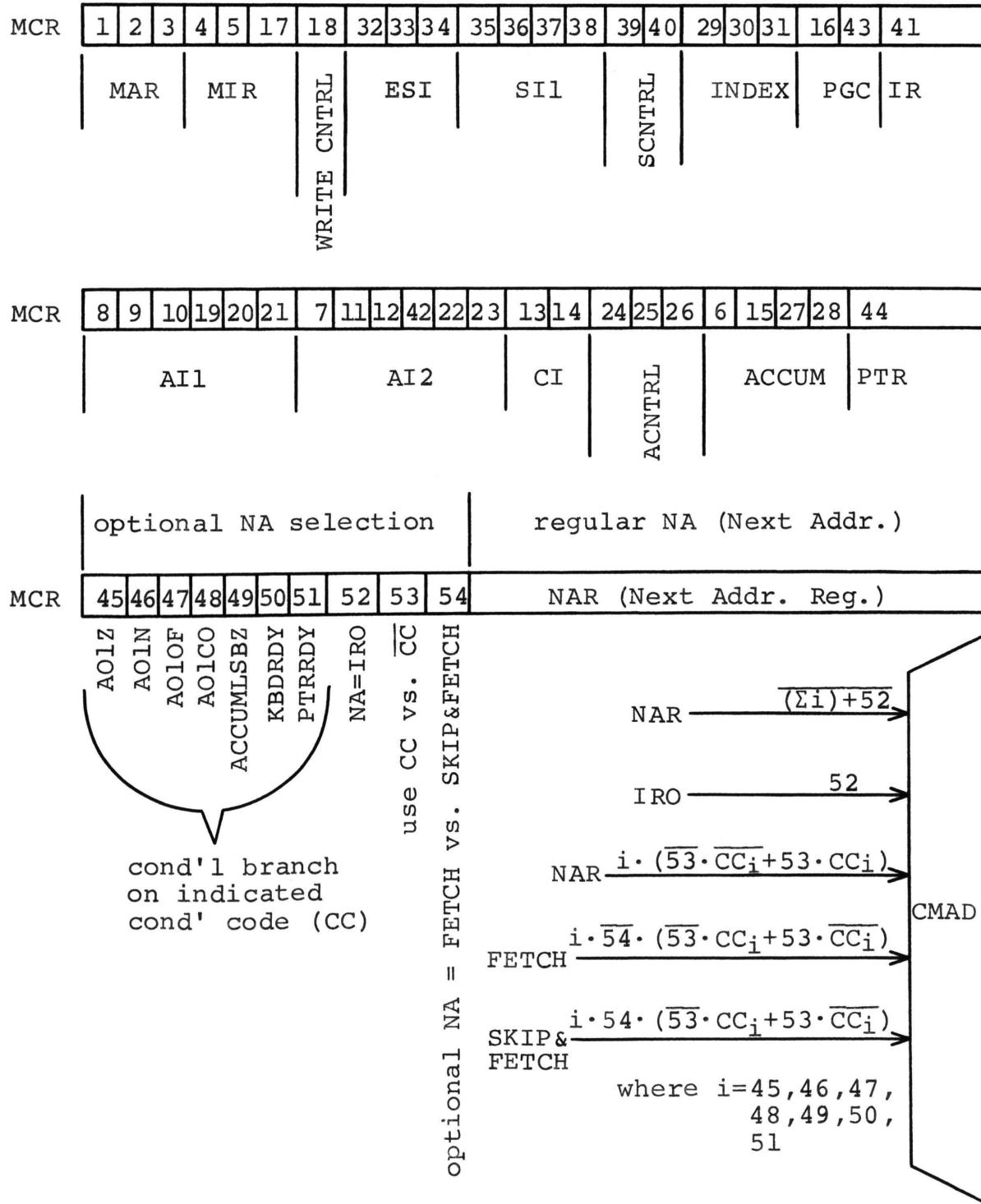


Fig. A2 Microinstruction Format & Addressing

READ (LOAD)-FROM-MEMORY $(\Sigma 1-3, 18) \cdot (\Sigma 6-7, 41, 29)$
 STORE-INTO-MEMORY $(\Sigma 1-5, 17) \cdot (18)$
 ADDER/ALU
 $(\Sigma 7-14, 19-23, 24-26, 42) \cdot (\Sigma 2, 4, 15-16, 20, 22, 30, 35, 45-48)$
 SHIFTER ESO $(\Sigma 35-38, 39-40) \cdot (34)$
 SHIFTER SOL $(\Sigma 32-38, 39-40) \cdot (\Sigma 21, 23, 28, 31, 38)$
 ACCUM $(\Sigma 6, 15, 27-28) \cdot (\Sigma 5, 8, 36, 44, 49)$
 INDEX $(\Sigma 29-31) \cdot (\Sigma 11, 17, 37)$
 IR $(41) \cdot (\Sigma 3, 19, 42-43, 52)$
 PGC $(\Sigma 16, 43) \cdot (\Sigma 1, 9)$
 KBD $(27) \cdot (50)$
 PTR $(44) \cdot (51)$

 COND'L BRANCH (& IRO BRANCH) $(\Sigma 45-51, 52)$

Fig. A3 HMI Inhibit Functions

APPENDIX B
HMO ALGORITHM DETAIL

A. REQUIREMENTS

It is essential that the HMO algorithm meet the following four requirements: (1) the condensed output code must be equivalent to (yield same results as) the uncondensed input code, (2) the output code should be as condensed as possible, (3) the HMO hardware should be as simple as possible, and (4) the algorithm should perform as fast as possible. Obviously, requirement 1 is the highest priority requirement which, if not met, renders the algorithm completely useless. On the other hand, requirements 2, 3, and 4 cannot be rigidly ordered by priority because, as might be expected, they are interrelated by inevitable tradeoffs.

B. USES

Near the end of Section I, two basic uses for the HMO algorithm were mentioned, either interpretive execution (of each condensed result which is then discarded) or pre-pass compilation (of all condensed results which are saved to form an entire condensed microprogram). At that point, interpretive execution was ruled out on the basis that it would require (1) repeated condensing of repeatedly executed blocks of microcode. Other disadvantages associated with interpretive execution are (2) the long-time occupation of control memory space with uncondensed blocks of microcode

and (3) the difficulty of assuring any overall speed increase (control memory would have to be cycled fast enough so that each condensed result could be formulated by the time the associated microcontrolled hardware was ready for it). Indeed, these disadvantages seem to make interpretive execution generally undesirable (with the possible exception of very low-usage microprograms). However, note that, unlike the static approach of pre-pass compilation, interpretive execution is a dynamic type of condensing. This dynamic property, as will be seen later, could be advantageous in helping to realize a more complex condensing approach for the algorithm.

Since, in general, so many inherent disadvantages exist for interpretive execution, this research has concentrated primarily on the use of the HMO algorithm as a pre-pass condensing compiler. Thus, unless otherwise stated, the remainder of this appendix can be assumed as concerned with the pre-pass compiler use.

C. GENERAL CHARACTERISTICS AND ASSUMPTIONS

To facilitate understanding of the various HMO pre-pass compiler design considerations, the following general characteristics and assumptions should be kept in mind: (1) algorithm is "1-pass" (primarily for simplicity), (2) overall optimization approach is a 2-level software-then-hardware approach, (3) uncondensed (partially condensed) microcode received from software compiler is "directly executable" or

"condensable-then-executable" from control memory, (4) algorithm transforms vertical code (serial hardware utilization) into more horizontal code (parallel hardware utilization), (5) algorithm proceeds, generally, under the "1-microcycle assumption" (microcontrolled operations completable in one control memory cycle), (6) algorithm is intended for local optimization.

D. RESULTANT PROPERTIES

The above characteristics and assumptions lead to the definition of many properties for the pre-pass compiler use, two of which are the following: (1) "restoration at the top" and (2) "retention of temporary garbage". Property 1 simply means that a condensed result is always restored (in control memory) at the position initially occupied by the top, or first, microinstruction of the original, uncondensed group of microinstructions (see first example of Fig. 3). This "restoration at the top" allows easiest formation of the "next address" portion of each condensed result (simply use, as implied in Fig. 2, the "next address" portion of the last instruction condensed onto the condensed result) and helps assure that the "temporary garbage" of property 2 is left intact. Property 2 simply means that all instructions between restored condensed results (such as instructions 3 and 4 of the first example of Fig. 3), even though they appear to be useless garbage (by the time instruction 5 is

reached during the single condensing pass), must be retained in original form at least until the condensing pass is entirely finished. Property 2 and property 1 together are necessary and sufficient conditions to ensure proper handling of "loop-backs". For example, it is obvious that instruction 4 must be retained (property 2) in case it is "looped back to" from some later point in the uncondensed microcode. Furthermore, by insisting that the condensed result of instructions 2, 3, and 4 be restored in position 2 (property 1), rather than, say, position 4, it is certain that such a "loop-back" to 4 (during the condensing pass) will find instruction 4 isolated and in its original form (as the uncondensed code intended) rather than finding a condensed combination of instructions 2, 3, and 4 (not intended by uncondensed code). The reader will notice that properties 1 and 2 are direct, but subtle, results mainly of the "1-pass" assumption.

E. CONDENSING TECHNIQUE

Concerning the actual condensing technique used to condense a microinstruction onto the condensed result being formed, Section II.C discussed a particular technique suitable for removing nonproductive transfers (Fig. 6). The reader may wonder why simply "ORing" the entire upcoming control portion onto the control register portion of the condensed result being formed was not suggested as a suitable condensing technique. Note that application of such a technique in Fig. 6

would have produced as the first condensed result an instruction containing not only "AII←PGC" but also "AII←ACCUM". Thus, the condensed code would not even be equivalent to the uncondensed code. To remedy this obviously unacceptable situation, "mutual exclusivity" inhibits could be added to the list of Fig. A3 to prohibit condensing whenever the upcoming instruction and the condensed result being formed both contained input transfers to the same register. Thus, in Fig. 6, for example, an AII input set "mutual exclusivity" inhibit would have been used to prevent instruction 2 from condensing onto instruction 1, the result being that the condensed microcode would then be identical to the original uncondensed microcode. Obviously, then, the simple "ORing" condensing technique would not only necessitate more inhibit functions and associated hardware but also would prevent HMO removal of nonproductive transfers.

Specifically, to employ the more powerful set-related condensing technique of Section II.C, the following bit sets (of bits 1-54 of Fig. A2) should be condensed according to the two-part rule of Section II.C: bits 1-3, bits 4-5, 17, bits 32-34, bits 35-38, bits 39-40, bits 29-31, bits 16, 43, bit 41, bits 8-10, 19-21, bits 7, 11-12, 42, 22-23, bits 13-14, bits 24-26, bits 6, 15, 27-28, bit 44 (all these groups constituting the various register input sets referred to in Section II.C), bit 18 (the write control set), and bits 45-54 (the optional next address selection set).

F. CONDENSING LIMITS

The reader will recall from Sections II.A and III.B that one condensing limit of interest was whether or not to allow condensing "past" (as well as "up-to-and-including") conditional branch microinstructions. A more detailed discussion of the ramifications of this condensing limit appears in Subsection G of this appendix.

Another condensing limit of interest concerns whether or not to allow the condensing of the beginning of factory-supplied routines (such as "FETCH" and "SKIP&FETCH") onto the tail end of user routines (whenever the inhibit functions would so allow). As will be seen later, use of appropriate control means (such as "condensed" bit markers) for determining the end point of the algorithm's condensing pass could make possible such condensings.

G. SPECTRUM OF POSSIBLE MICROINSTRUCTION FORMATS

The reader will recall from Section III.B that the microinstruction addressing flexibility necessary to accommodate "leap frog" style execution jumps (which circumvent groups of "garbage" instructions remaining from the HMO condensing pass) implies the need for at least one complete "next address" in each microinstruction [13] (Fig. 2, Fig. A2). Furthermore, to accommodate conditional choice of "next addresses" (for conditional branch microinstructions),

some means of producing at least one other "next address" must be incorporated. For example, the microinstruction addressing format of HMI (Fig. A2) allows a choice between the complete "next address" in the NAR and one of two optional hardwired "next addresses". This HMI format is, in fact, a marginally adequate one (as long as condensing "past" conditional branches is prohibited) representing the extreme simple end of the spectrum of possible formats.

On the other hand, if one wishes to ideally allow, in one condensed result, condensing "up to and including and past" conditional branches (Section II.A) down one of the optional paths, then microinstruction formats representing the extreme complex end of this spectrum become mandatory. Specifically, the second example of Fig. 3 demonstrated that condensing "past" CB's (conditional branches) necessitated room in the microinstruction for two sets of control information (essentially so that the collection of transfers to be executed could be "conditionally tuned" to the chosen path). Furthermore, condensing "past" CB's and down one of the paths results in the algorithm automatically updating the NA ("next address") originally pointing to the start of this particular path. Note, however, that the NA pointing to the start of the other path must remain unchanged. (For instance, in the second example of Fig. 3, the NA value of "8" originally in CB instruction 7 was updated to a value of "10" in the condensed result on the right while the other NA value of "FETCH" necessarily remained unchanged.) The conclusion resulting from this requirement is that the two

NA's available to a CB instruction must be completely independent (so that one NA may be changed without changing the other). Thus, one instruction format suitable to allow-
ing condensing "past" CB's (down one of the paths) is a format having essentially two complete control sections and two complete NA's in each microword. (Note, of course, that one does not need to duplicate the bit group of 45-54 of Fig. A2 in the second control section.)

Fig. B1 illustrates the spectrum of possible micro-instruction formats and the position of the two formats just discussed on this spectrum. One thing hinted at in Fig. B1 is the microprogramming flexibility provided by a CC inverting bit such as bit 53 of Fig. A2. For example, consider microprogramming the complex format of Fig. B1. Even though two complete stored NA's are in each microword, a bit such as bit 53 allows the user to microprogram any problem so that, say, the left stored NA of a CB is always the one which points to the "non-branch", or most often used, path. (If such a bit were not used and a particular value of the selected CC always caused use of a particular one of the two available NA's, programming situations would arise in which sometimes the right NA, rather than always the left NA, would be pointing to the "non-branch" path.) Thus, if the left NA always points to the most often used path, it is an easy matter for the hardware algorithm to choose, and thereby "favor", this path as it attempts to

condense "past" a CB, leaving, generally, the other path to be covered later from its beginning.

At this point, the reader may wonder why the complex format on the right of Fig. B1 was not proposed for HMI, since indeed this format appears to be the ultimate one in terms of microprogramming flexibility, compatibility with the ideal CB condensing approach, etc. The obvious answer is that this format, with its essentially "double-length" microwords, would be completely wasting one control section and one stored NA for all non-CB microinstructions. Since non-CB instructions probably account for the majority of most microprograms, such blatantly inefficient bit usage of control memory is a ridiculously high price to pay for the advantages of this format.

One obvious scheme, then, to consider at this point is a hybrid "single-length/double-length" scheme in which either two non-CB instructions or one CB instruction can be stored in each essentially double-length microword. Indeed, such a scheme at first seems feasible, the only obvious hardware requirement being a micromemory single-length/double-length read/write capability. The real problems stem from this scheme's incompatibility with the present simple, unrestricted form of the HMO algorithm. For example, using this scheme, whenever the algorithm restored a condensed, conditional, double-length result, it would generally be destroying one single-length temporary garbage instruction and possibly trying to restore this double-length result starting on an

"odd" boundary, or the midpoint of a double-length microword [13] (an action not always permitted in single-length/double-length addressing schemes, e.g. IBM 360/Model 50 main memory addressing [14]). Fig. B2 is a hypothetical, general example illustrating the problems just mentioned for this hybrid scheme.

As might be anticipated, many other microinstruction formats are capable of bit-efficiently producing, for CB's, an extra set of control information and/or an extra, completely independent NA. For example, the basic format of one CS (control section) and one NA could be augmented to include multiple-use fields so that in CB's a portion of what is normally, say, the CS (for non-CB's) could be "borrowed" to create an extra NA (and/or possibly a partial extra CS). However, such a "borrowing" of bits from some other essential microword section would result in (1) some loss of, in CB's, the potential informational content of that section and, therefore, (2) generally some loss (due to a needed, added "field availability" inhibit function) of CB "upward" condensability (up onto preceding instructions). As a second example, consider a scheme in which an "optional branch register" would always be microinstruction-preloaded with an optional NA so that a CB, when later reached, could choose between its stored NA and the "optional branch register" contents. Although workable, such a preloading scheme would result in potential CB "upward" condensability loss due to the need to ensure appropriate distance (via an

added inhibit) between the preloading instruction and the corresponding CB itself. (Further note that, if, in the interest of bit efficiency, the preloading instruction obtained the optional NA from a self-contained, multiple-use, "borrowed" field, then this preloading instruction would itself suffer problems (1) and (2) mentioned above for a CB employing "borrowed" fields.) The problems incurred, then, in these two example schemes, highlight the general desirability of having the sources of a CB's extra information (extra CS and extra NA) be self-sufficient, with no need to infringe upon other essential informational fields or to depend upon preceding microinstructions.

Not surprisingly, prohibiting condensing "past" CB's makes workable many other members of that myriad of microinstruction format schemes implied in Fig. B1. Indeed, with the elimination of the need for an extra CS and the elimination of the requirement that the two NA's available to a CB be completely independent, the workability of many more instruction schemes is to be expected. For example, IBM's branch set concept [3], [23] could be used to augment the basic "one CS and one stored NA" format, allowing formation of, for CB's, a sequential set of interdependent NA's by CC "injection" into the lower-order bit(s) of the stored NA. However, although allowable when condensing "past" CB's is prohibited, such interdependence of CB NA's is still deemed undesirable. In the interest of user-program loading flexibility (needed in the face of a control memory

conceivably filled with a combination of interlinked condensed instructions and an erratically interspersed residue of garbage instructions), employing completely independent CB NA's avoids the potential difficulty of finding two properly (e.g., sequentially) spaced available (garbage-filled) microwords in which to place the two target instructions of a CB.

Obviously, all the possible microinstruction formats implied by Fig. B1 cannot be discussed in detail in this subsection. However, it should be evident by this point that selection of the "best" format scheme would be a formidable task, involving the complex, but inevitable, tradeoff areas of microprogramming flexibility, complexity of microinstruction handling hardware, efficiency of control memory bit usage, and compatibility with the HMO algorithm in its present, simple, unrestricted form. Although the simple instruction format represented by the left end of the spectrum of Fig. B1 is by no means considered the ultimate format, it was chosen for HMI because it is simple yet more than adequate as an initial design base.

H. CONDENSING APPROACH

As implied in Fig. 2, the present simple form of the HMO algorithm allows the next upcoming microword to condense onto the condensed result being formed only if the entire upcoming CS is condensable (no inhibits active). In other words,

this approach might be described as the condense "by-whole-word-only" (specifically, "by-whole-CS-only") approach, an approach in which uninhibited control bits are automatically prohibited from condensing by any other currently inhibited control bits. Section II.B and Fig. 5 illustrated a possible condensing inefficiency resulting from this simple "by-whole-word-only" condensing approach. Section II.B further ruled out a more sophisticated hardware condensing approach on the basis of several associated, intricate problems.

Specifically, this sophisticated approach would have cycled instructions (to be examined for condensability) up through a multilevel first-in-first-out stack in which individual bit columns were basically independently mobile so that individual columns could be moved upward (until individually inhibited) even though other columns were currently inhibited. Thus, in the example of Fig. 5(a), assuming that instruction 1 is already in the condensing register and that instructions 2, 3, and 4 are in the top three rows of the stack being scrutinized for condensability, the algorithm could look past row 1 (where the column containing "A11←ACCUM" is currently inhibited by the accumulator inhibit) to row 2 to recognize that the independently mobile column containing "INDEX←DATA2" is presently uninhibited and, in fact, capable of being moved upward so that "INDEX←DATA2" enters the condensing register along side of "ACCUM←DATA1". Fig. B3 illustrates the condensing obtainable with this "by-individual-bit-column" approach.

However, the potential problems associated with this more sophisticated approach are many. First, concerning hardware complexity, not only is the column-mobile stack required, but to ensure all columns are inhibitible from all stack levels, multiple copies of the inhibits of Fig. A3 are needed, essentially one copy of each inhibit for each level. Furthermore, the simple inhibits of Fig. A3 would have to be made individually more complex to prevent problems such as the one illustrated in Fig. B4. (In Fig. B4, the simple adder inhibit of Fig. A3 did not prevent instruction 3 from moving up past inhibited instruction 2 into the time frame of the previous addition, and thus changing the results of that addition. Note, however, that with the "by-whole-word-only" scheme of Fig. 2, the inhibiting, via Fig. A3's adder inhibit, of instruction 2 from condensing up onto instruction 1 would have temporarily inhibited all instructions following instruction 2.) Second, a potential difficulty in assuring condensed code equivalency can be demonstrated. If, in Fig. B3, a later "loop-back" occurred to instruction 2 (now condensed as shown on the right), this "loop-back" would no longer subsequently incur the "INDEX+DATA2" transfer of instruction 3 as it would have in the original, uncondensed code. (Obviously, potential "loop-back" equivalency problems also exist for the uncondensed code reordering, or pretailoring, employed in Fig. 5(b). However, if, as suggested in Section II.B, the software compiler is used for this pretailoring, the multiple

passes assumed available should make possible the detection, and thus prevention, of such potential equivalency problems. This is not the case for the HMO algorithm, whose 1-pass simplicity renders impossible the predetection of such potential loop-back unequivalency problems.) Third, the difficulty of determining the NA to be placed in each condensed result is increased. Note that if, in Fig. B3, the NA from the instruction most recently condensed were used as the NA of the condensed result (as implied in Fig. 2), the NA found in condensed instruction 1 on the right would incorrectly be a value of "4". Thus, the "by-individual-bit-column" condensing approach demands a more complex NA determination scheme for condensed results. As can be seen, these nagging problems associated with the sophisticated "by-individual-bit-column" scheme make this scheme generally unsuitable for use by a hopefully simple, straightforward, 1-pass hardware algorithm such as HMO.

Two notes are of interest concerning this more complex "by-individual-bit-column" condensing approach. First, this approach (with all its problems) is not to be confused with the scheme of Section IV.C which, although also using a "far-look-ahead" stack, is still a "by-whole-word-only" approach (modified to allow adjustment of an inhibit function's "field of view"). Second, the dynamic property of the interpretive execution use of the algorithm (see Subsection B of this appendix) could be of use in helping to alleviate the second and third problems just cited for this more

sophisticated condensing approach. Since interpretive execution does not alter the microcode in control memory, a later return, via some different flow path, to an already passed over block of code (such as a "loop-back") would present no special problem, as the interpreter would then simply flow through the still intact original code in a new manner, dynamically collecting an appropriate condensed result. Thus, no potential condensed code equivalency problems are introduced. Furthermore, since interpretive execution does not restore condensed results, but instead immediately executes such results and then discards them, there is no need to worry about even determining a suitable NA to be restored in each condensed result. The interpretive executer would simply collect a condensed control section result off the top of the stack (which would be kept full, as required, by insertion of upcoming microinstructions at the stack bottom), execute it, and then begin formulating the next condensed result.

I. MORE DETAIL ON PRE-PASS CONDENSING COMPILER USE

Fig. B5 shows more of the detail needed for using the HMO algorithm as a pre-pass condensing compiler. The RAR, or restoration address register, is simply some register in which to hold the address pointing to the control memory position (the "top" position of the original uncondensed code group) where the condensed result will be restored. The use of the "condensed" marker bit is, as the name implies, a means of

marking restored condensed results as the algorithm proceeds through its one and only condensing pass. By using these marker bits to later distinguish between condensed results and yet unchanged code (e.g., "temporary garbage"), the algorithm can spot the point at which to stop its pass rather than, say, getting futilely entrapped in a "loop-back" situation where it might endlessly be reexamining already condensed code. In fact, assuming the factory-supplied-and-condensed routines (such as "FETCH" and "SKIP&FETCH" of Fig. A2) were appropriately marked as "condensed" with these marker bits, the algorithm could attempt to condense the beginnings of such factory-supplied routines, when possible, onto the tail end of user routines (but only to the point where proceeding further would mean nothing but wastefully recycling over nothing but interlinked, already-condensed results). Finally, note that in the "DONE" block of Fig. B5 the possibility of having to go back and cover yet untouched code paths is implied. This possibility results directly from the algorithm choosing, for CB's, one path to work on immediately, thus leaving the other path for later attention. Such a residue of paths yet to be covered would exist, generally, for most microinstruction formats, with the exception of formats like that of Hm1 (leftmost format of Fig. B1). With such a format, assuming the algorithm always chooses the CB's stored NA as the path to work on immediately, the remaining temporarily untouched NA would always point to the beginning of some already condensed,

factory-supplied routine which exits from the user-written microcode. Obviously, there is no need to send the algorithm back to attempt condensing at the beginning of already condensed exit routines, as such attempts would never find any condensability.

J. SOME IMPLEMENTATION CONSIDERATIONS

It should now be evident that the considerations involved in integrating a new component, such as the HMO algorithm, into a system so that this new component works well and smoothly with other system components (e.g., the software microprogram compiler, the other hardware of the host machine, etc.) are many and complex. Since this research is merely the first phase of an overall systems design approach (that would eventually lead to a detailed, physical, microprogrammable system incorporating an HMO algorithm implementation), it has concentrated primarily on HMO algorithm support considerations aimed at developing a system environment suitable for supporting the algorithm (e.g., the algorithm/software compiler cooperation and separation areas of Section II, the microinstruction format tradeoffs of Subsection G of this appendix, etc.). Indeed, such support considerations are the most important first step (as opposed to rushing blindly into a physical algorithm implementation) if the eventual system is to be a smoothly working system (rather than an ad hoc collection of hastily conceived, uncooperative parts). However, the remainder of this subsection will present, in

extreme brevity, some of the actual algorithm implementation considerations deemed relevant at this initial design stage.

One consideration is the type of implementation. For example, although a conventional hardware implementation is certainly possible, a firmware implementation is deemed desirable due to its flexibility (for design changes) and its correctability (for design mistakes).

Another consideration of interest is how to initiate the algorithm. For example, the algorithm could be initiated under strictly user control via machine instruction (by use of a special combination of addressing mode bits available with all operation codes, by use of a separate, unique op code solely for condensing, etc.). However, one quite logical method would be to have the system's microprogram loader itself initiate, if so directed, the algorithm on a microprogram immediately following the microprogram load.

(It must be noted at this point that many techniques employed by the algorithm were chosen, at least in part, because of the flexibility they allowed in the overall picture. For example, rather than insist that some sort of "clean up" routine always follow the algorithm to clean up any residue of "temporary garbage", which is automatically circumvented by the interlinked condensed results anyway, the "condensed" marker bits of Fig. B5 could be further used to help the microprogram loader spot, by the "off" condition of this bit, leftover "garbage" positions which can thus be filled with uncondensed instructions of a new user program. This

is not to say that such marker bits are, in themselves, sufficient means to drive and control the microprogram loader as, for example, following the loading of one yet uncondensed, and thus yet unmarked, user program with the immediate loading of another could cause the first program to incorrectly appear as unmarked "garbage" to the second program. On the contrary, the point here is that it is extremely important, in the initial phase of a design project, to try to make decisions and choose techniques in such a way that other system components are constrained or complicated as little as possible. With HMO, for example, these marker bits, in addition to their use in determining when the algorithm is done, could be useful in helping prevent the restrictive complication that a "garbage clean up" pass be performed either by the algorithm or by some other system component, such as the microprogram loader.)

Another consideration of interest, assuming a firmware implementation is chosen, is how to allot available control memory. For example, rather than have one WCS (writable control store) contain everything, the author's present inclination is to suggest both a WCS (containing at least all user microprograms and other routines of pertinence to the HMO algorithm as it condenses, such as "FETCH" and "SKIP&FETCH" of Fig. A2, non-user routines which the algorithm may be trying to partially condense onto the tail end of user routines) and a separate ROM (containing at least the HMO algorithm itself and other routines with which the algorithm

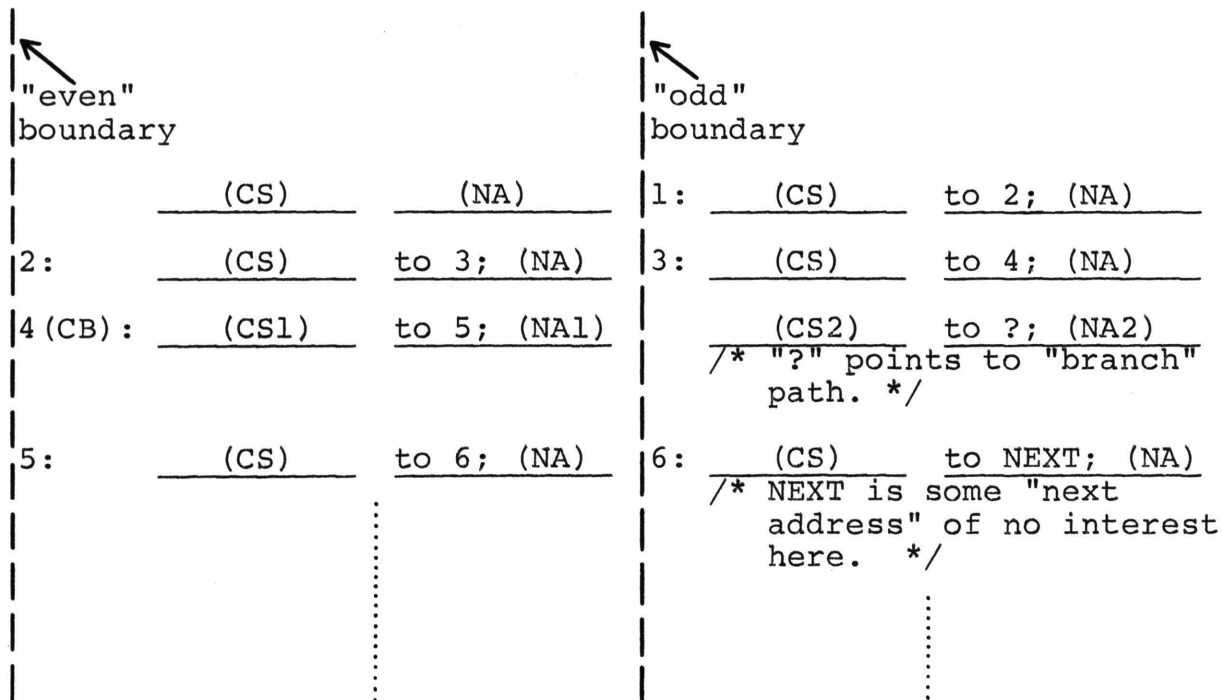
will cooperate, such as the microprogram loader). Such placement of the algorithm in a separate, essentially dedicated ROM not only removes the need to use the more expensive WCS for everything but also allows the algorithm to be viewed more or less as an extra process simply tacked onto the normal host hardware of HML. Fig. B6 is a crude illustration of this suggested control memory structure.

	less complex ←		→ more complex
	less flexible microprogramming- wise		more flexible microprogramming- wise
Format Description (of 1 microword)	1 Cntrl Sect'n, 1 Stored NA + Fixed Option(s)	Myriadof..... Other Schemes	2 Full Cntrl Sect'ns, 2 Full Stored NA's
Total # of NA's available to a CB	$2^n + \# \text{ of}$ fixed options *	:	$2^n + 2^n *$
Achieves complete independence of 2 CB NA's	Yes	:	Yes
Accommodates condensing past CB's	No (not w/o an added 2nd cntrl sect'n)	:	Yes
If condensing "past" CB's used,allows easy algorithmic choice of "non-branch" ** paths	Yes, thanks to flexibility pro- vided by bit # 53 of Fig. A2	:	Yes, if bit like # 53 of Fig. A2 employed
No matter which CB path is chosen for immediate condensing use, would generally need list of yet un- used paths for later condensing coverage	No,assuming stored NA's covered imme- diately, remaining fixed options always point to factory-supplied, condensed routines which exit from user microprogram	:	Yes
Allows flexible enough CB's to directly micro- program any problem within host machine's capabilities	No (assuming small # of fixed options pointing to fac- tory-supplied rou- tines), some prob- lems (e.g., I/O "wait" loops) nec- essarily relegated to machine instr'n level (software)	: : : :	Yes

* "n" is # of bits/stored NA.

** See 2nd-to-last paragraph of Section I.

Fig. B1 Spectrum of Possible Microinstruction Formats



NOTE: Above microcode is shown in uncondensed form;
assume instr'ns 1-6 found condensable.

NOTE: CS - Control Section, NA - Next Address,
CB - Conditional Branch

NOTE: The double-length condensed result would be
restored "at the top" in positions 1 & 2, thus
destroying "temporary garbage" instruction 2.

Fig. B2 Potential Problems with Hybrid
Single-Length/Double-Length Format

1: ACCUM←DATA1; to 2;

2: A11←ACCUM; to 3;

3: INDEX←DATA2; to 4;

4: A12←INDEX; to NEXT;
/* NEXT is some
"next address"
of no interest
here. */

uncondensed microcode

1: ACCUM←DATA1; INDEX←DATA2;
to 2;
/* Above formed from
instr'ns 1 & 3. */

2: A11←ACCUM; A12←INDEX;
to NEXT;
/* Above formed from
instr'ns 2 & 4. */

3: /* "Temp' garb'," same as
on left */

4: /* "Temp' garb'," same as
on left */

condensed microcode (via
"by-individual-bit-column"
approach)

Fig. B3 Use of "By-Individual-Bit-Column" Approach

```
1: AI1←DATA1; AI2←DATA2;
   CI←0; to 2;
```

```
2: ACCUM←A01; to 3;
```

```
3: AI2←DATA3; to NEXT;
   /* NEXT is some
      "next address" of
      no interest here.
   */
```

uncondensed microcode

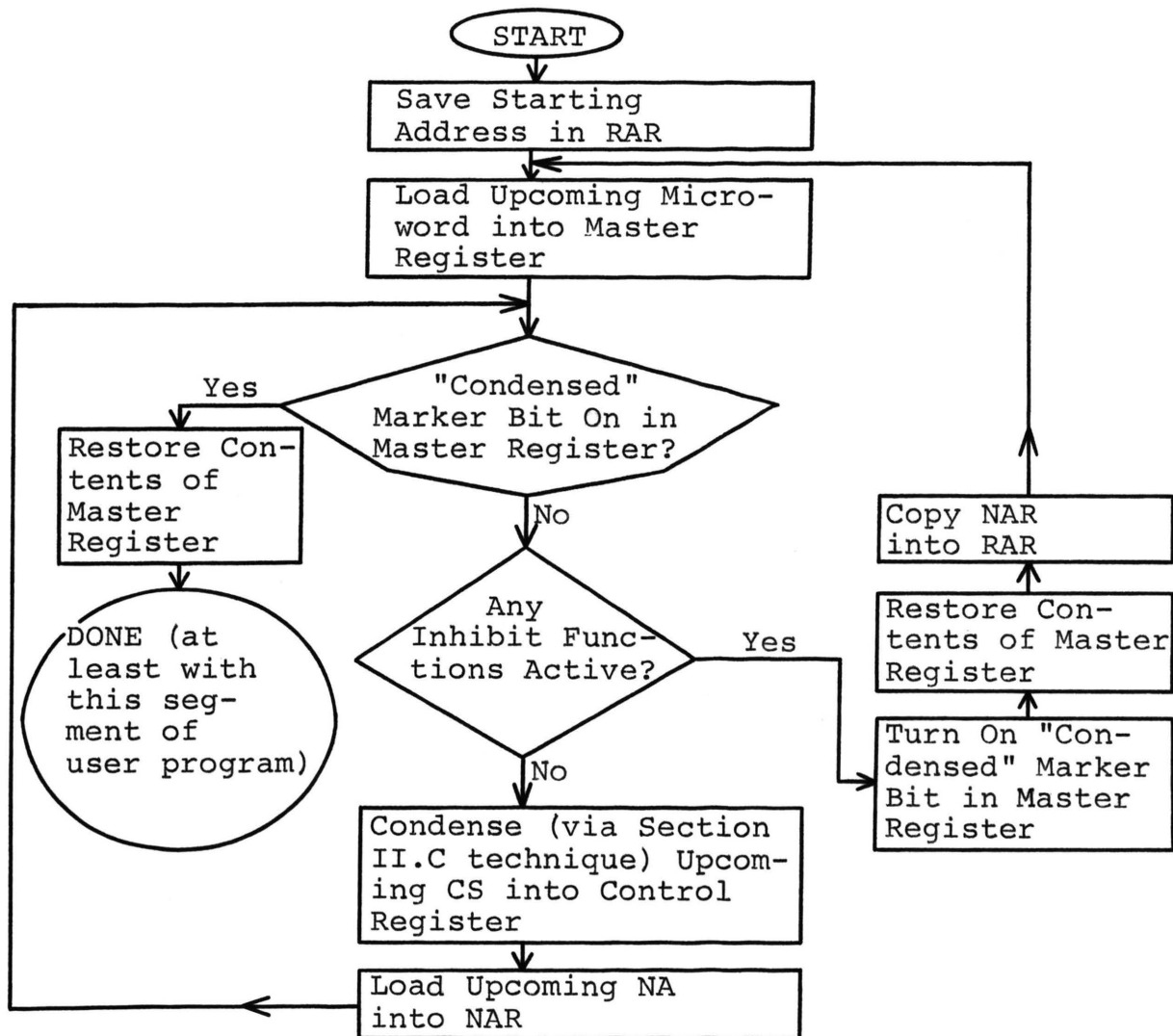
```
1: AI1←DATA1; AI2←DATA3;
   CI←0; to 2;
   /* Note that "AI2←DATA3"
      (instr'n 3 of uncon-
      densed code) has been
      moved up into the time
      frame of this addition,
      thus changing the added
      result transferred by
      the following instruc-
      tion. */
```

```
2: ACCUM←A01; to NEXT;
   /* Above instr'n no long-
      er produces results
      equivalent to uncon-
      densed code. */
```

```
3: /* "Temp' garb'," same as
      on left */
```

condensed, unequivalent
microcode (via "by-individ-
ual-bit-column" approach)

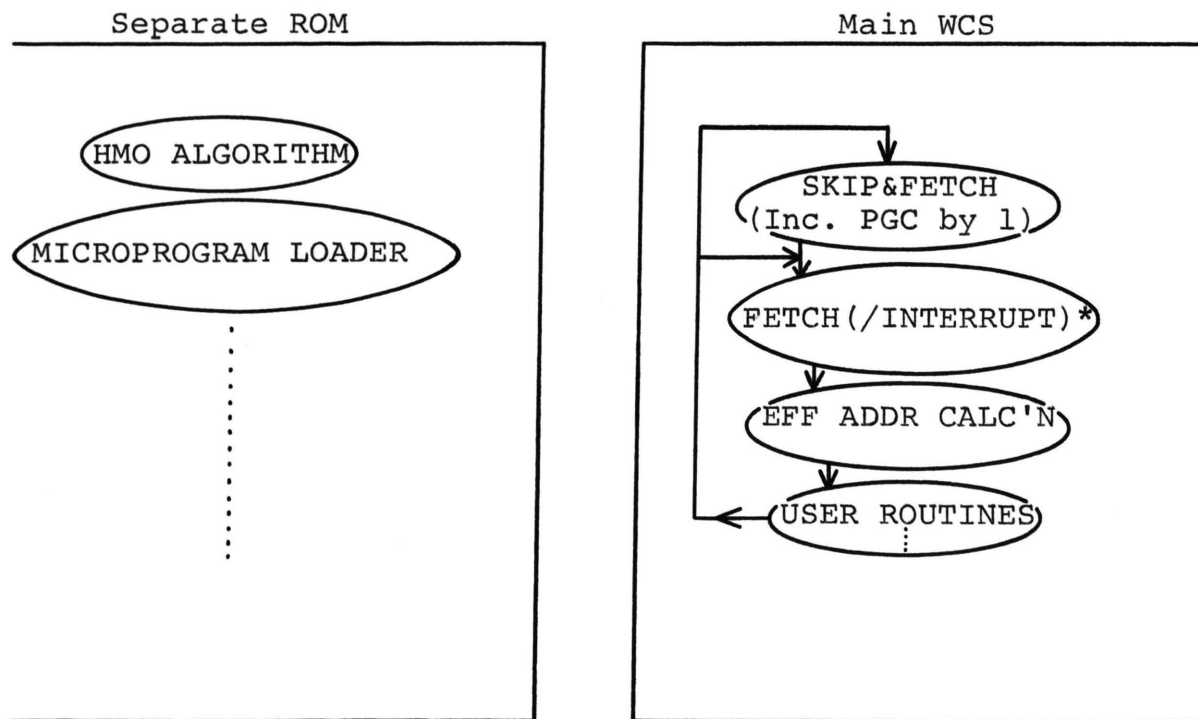
Fig. B4 Potential Problem with
"By-Individual-Bit-Column" Approach



NOTE: See Fig.'s 2 & A2 for explanation of "Master Register," "Control Register," and "NAR".

NOTE: CS - Control Section, NA - Next Address, RAR - Restoration Address Register (any suitable register)

Fig. B5 Flow Chart of HMO Algorithm as a Pre-Pass Condensing Compiler



* Probably need another optional CB NA (besides FETCH and SKIP&FETCH) to implement reasonably efficient interrupts on HMI

Fig. B6 One Possible Control Memory Layout

APPENDIX C

AREAS OF CONCENTRATION FOR FURTHER RESEARCH

This appendix lists areas deemed appropriate for concentrated research in future phases of the overall design of HMO, a hopefully well conceived, orderly, "total-system" design eventually leading to actual physical fruition of a microprogrammable system with HMO algorithm. As this research on hardware microcode optimization has proceeded through its first phase (laying an HMO-suitable, environmental, supporting foundation of algorithm properties and techniques, compatible machine characteristics, etc.), the areas mentioned in the remainder of this appendix have naturally evolved as areas worthy of attention in any further research.

First, a concentrated investigation of microinstruction formats compatible with the ideal approach of condensing "past" CB's (conditional branches) should be performed, the aim being to develop the "ultimate" format which is as flexible and powerful as the "strictly double-length" format (see extreme right end of Fig. B1's spectrum) yet free of its glaring bit inefficiencies.

Second, as opposed to using the algorithm strictly for either interpretive execution or pre-pass compilation, a hybrid "interpretively execute/compile only as needed" use should also be analyzed. Since, for CB's, this hybrid use would compile along the CB path actually being used during execution (going back to cover the other CB path only when and if it is later used), there would never be any need, no matter what the microinstruction format, for

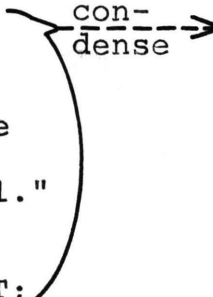
an accumulated list of paths yet to be covered (as there would be, with some formats, for strictly pre-pass compilation).

Third, a skeleton software microprogram compiler should be developed at least to a point permitting simulation of the overall microcode generation process (including both software compiler and hardware algorithm), such simulation hopefully enabling, via various simulation-derived measures, enlightened design decisions.

Fourth, the exact areas of "software compiler/hardware algorithm" cooperation and separation should be further investigated and crystallized, the flexibility of simulation here allowing investigatory variation of where and how a particular optimization chore is handled, whether primarily by software or by hardware or by a combination of both.

Fifth, some variations of the basic algorithm should be examined. For example, rather than allowing the algorithm to choose (blindly in its one pass) the starting instruction of each successive condensed result as being the first instruction inhibited from condensing onto the preceding condensed result, these condensing-step starting points could be adjusted (possibly by appropriate instruction markers planted during a software compiler pass preceding the hardware algorithm) in the hope of assuring an overall maximally condensed program.

APPENDIX D
MISCELLANEOUS EXAMPLES

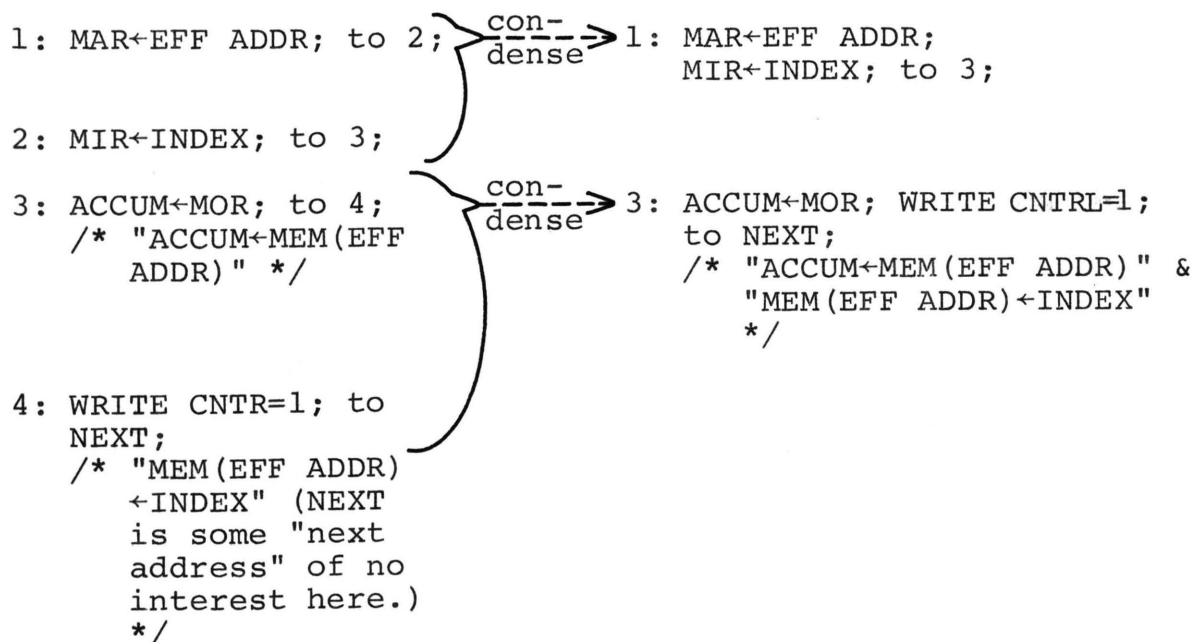
<pre> 1: MAR←ADDR1; READ FF←0; WRITE FF←1; CI←0; to 2; /* Set up main mem' con- trol for an upcoming "store into MEM(ADDR1); "finish supplying adder inputs. */ 2: MBR←A01; to 3; /* During next microcycle following above xfer, "MEM(ADDR1)←A01." */ 3: MAR←ADDR2; to NEXT; /* Begin setting up main mem' con- trol to work on another address. (NEXT is some "next address" of no interest here.) */ </pre>		<pre> 1: MAR←ADDR1; READ FF←0; WRITE FF←1; CI←0; to 2; 2: MBR←A01; MAR←ADDR2; to NEXT; /* Here, however, follow- ing microcycle will result in "MEM(ADDR2)←A01." Ob- viously, results here no longer equivalent to those on left. The lack of an obvious "finishing" step left algorithm unaware (even with the MAR and MBR input "mutual ex- clusivity" inhibits in original scheme) that instr'n 3 should not be condensed onto instr'n 2 (into the time frame of the pre- vious store-into- memory process). */ </pre>
--	---	--

uncondensed microcode

condensed, unequivalent
microcode

NOTE: This original scheme [13] used the MBR itself (no MIR existed) to accept data to be stored. In fact, a "store-into-memory" process really had only one step (consisting of supplying the storage address, storage data, and appropriate storage control information), the actual memory store being handled invisibly by the memory controller during the following microcycle. Thus, although the elements of this one step could be spread out over several microinstructions, no succeeding instruction was required to contain some sort of "finishing" step, as with the "WRITE CNTRL" bit of the present main memory scheme.

Fig. D1 Problem with Original, One-Step,
"Store-Into-Main-Memory" Scheme



uncondensed microcode

condensed microcode

NOTE: The present memory controller (with both MBR and MIR) allows both a main memory read (load) and write (store) to occur in the same instruction (when possible, as in instruction 3 of above condensed code). (Specifically, when the condensed code is executing, the "MCR←instr'n 3" via a major cycle pulse, "MBR(or MOR)←MEM(MAR)" at next minor cycle pulse, and "MEM(MAR)←MIR" at next major cycle pulse.) However, if instructions 3 & 4 of the uncondensed code had appeared in reverse order, the "read-from-memory" inhibit of Fig. A3 would correctly have inhibited their condensing together. This inhibiting would be necessary since the write (store), then occurring first rather than last as above, would directly affect the results of the following read (load).

Fig. D2 A Peculiarity of the Present Memory Controller

<pre> 1: AIL←ACCUM; to 2; 2: AIL←ACCUM; ACCUM←MOR; to 3; /* Since instr'n 1 above did not alter the ac- cumulator's con- tents, the "AIL← ACCUM" xfer of instr'n 2 above is "redundant" in that it ac- complishes noth- ing not already accomplished by this same xfer in instr'n 1. However, since instr'n 2 is not inhibited by instr'n 1 (from any Fig. A3 in- hibit), the con- densing technique of Section II.C can be used to remove this re- dundancy. */ 3: AIL←ACCUM; to NEXT; /* Due to the "ACCUM←MOR" ac- cumulator change of instr'n 2 above, the "AIL←ACCUM" xfer of instr'n 3 above is not "re- dundant." (NEXT is some "next address" of no interest here.) */ </pre>		<pre> 1: AIL←ACCUM; ACCUM←MOR; to 3; 3: AIL←ACCUM; to NEXT; </pre>
---	--	--

uncondensed microcode

condensed microcode

Fig. D3 Redundant Transfer Removal

```

1: SI1←ACCUM; ESI←0;
  SCNTRL←RSHFT; to 2;
  /* Above will produce,
    at S01,
    (0→ACCUM→lost)1. */

2: SI1←S01; to 3;
  /* Above will produce,
    at S01,
    (0→ACCUM→lost)2. */

3: SI1←S01; to 4;
  /* Above will produce,
    at S01,
    (0→ACCUM→lost)3. */

4: SI1←S01; to 5;
  /* Above will produce,
    at S01,
    (0→ACCUM→lost)4. */

5: ACCUM←S01; to NEXT:
  /* That is, ACCUM gets
    (0→ACCUM→lost)4.
    (NEXT is some "next
    address" of no
    interest here.) */

```

uncondensed microcode

NOTE: The microcode on the left is not condensable. Note, for example, that although the consecutive string of "SI1←S01" transfers (instr'ns 2-4) may appear to contain redundancy or non-productiveness, it does not. Each such transfer is a productive transfer of transformed shifter output data back to the shifter input for further transformation. The HMO algorithm recognizes the general nonremovability of such direct feedback transfers by having the associated inhibit function treat them as both a functional unit input and output. (Specifically, in this case, bit 38 appears in both terms of the "SHIFTER S01" inhibit of Fig. A3.)

NOTE: "(0→ACCUM→lost)ⁱ" refers to an i-times-repeated one-bit accumulator right shift during which the left-most bit receives a "0" and the right-most bit is lost.

Fig. D4 Appropriate Handling of Functional Unit Direct Feedback Paths

```

1: AI1←ACCUM; AI2←MOR; CI←0;
  to 2;
  /* Nonproductiveness of
    "AI2←MOR" in above is
    disguised by
    "ACCUM←A01" in follow-
    ing instruction # 2. */

2: AI2←INDEX; ACCUM←A01; to 3;
  /* Above "ACCUM←A01" is
    nonproductive. */

3: ACCUM←A01; to NEXT;
  /* NEXT is some "next
    address" of no
    interest here. */

```

NOTE: HMO algorithm (as presented) would find code on left uncondensable. Below, "ACCUM←A01" has been removed from instruction # 2 so that code is condensable.

uncondensed microcode

(a)

<pre> 1: AI1←ACCUM; AI2←MOR; CI←0; to 2; /* Nonproductiveness of "AI2←MOR" in above no longer disguised since "ACCUM←A01" no longer in follow- ing instr'n # 2 */ 2: AI2←INDEX; to 3; 3: ACCUM←A01; to NEXT; </pre>	<p>con- dense →</p>	<pre> 1: AI1←ACCUM; AI2←INDEX; CI←0; to 3; 3: ACCUM←A01; to NEXT; </pre>
---	-------------------------	---

uncondensed microcode

condensed microcode

(b)

Fig. D5 Disguised, Larger-Scale Nonproductiveness

```

1: to (PTRRDY) 2, SKIP&FETCH;
  /* If PTRRDY=0 (i.e.,
    PTRRDY=1, or printer
    ready), then go to 2;
    else go to SKIP&FETCH
    (and possibly link
    into a "wait" loop at
    machine instr'n level).
  */

2: PTR←ACCUM; to 3;
  /* Execute output trans-
    fer. (PTR inhibit of
    Fig. A3 treats above
    xfer as "starting"
    step of output
    process.) */

3: to (PTRRDY) 4, SKIP&FETCH;
  /* Test for availability
    of output channel for
    another output. (PTR
    inhibit necessarily
    treats above PTRRDY
    test as "finishing"
    step of output process
    begun in instr'n 2.) */

4: PTR←ACCUM; to NEXT;
  /* Interestingly, the
    second output xfer in 4
    above will never be
    performed, since instr'n
    3, if reached, will al-
    ways find the printer
    still busy from the first
    output xfer in instr'n 2.
    (NEXT is some "next
    address" of no interest
    here.) */

```

NOTE: The microcode on left is not condensable (due to noncondensability "past" CB's with HMI's microinstruction format and to the noncondensability of instr'ns 2 and 3 caused by the PTR inhibit of Fig. A3). Note that if instr'ns 2 and 3 had been condensed together, then this condensed result, when reached from instr'n 1, would perform the first "PTR←ACCUM" and simultaneously find the PTRRDY CC still indicating the printer as ready, thereby causing instr'n 4 to be performed next. (In other words, whenever the first output xfer was found performable, the second output transfer would immediately follow one micro-cycle later, an obviously incorrect situation.)

uncondensed microcode

Fig. D6 Futility of Microprogramming HMI to Perform 2 Immediately Successive Output (or Input) Transfers

```

1: A11←ACCUM; A12←INDEX;
  CI←0; ACNTRL←ADD;
  to 2;
  /* Prepare to add
    ACCUM & INDEX.
    (Assume addition
    will take 3 micro-
    cycles after this
    instr'n to
    complete.) */
2: ACCUM←A01; to 3;
  /* Place added result
    in ACCUM. */
3: S11←ACCUM; ESI←0;
  SCNTRL←RSHFT; to 4;
  /* Place added result
    in S11 and prepare
    to right-shift it
    once. (Note,
    w.r.t. getting
    added result from
    A01 to S11, ACCUM
    here appears as
    intermediate (tem-
    porary) storage
    station.) */
4: INDEX←S01; to NEXT;
  /* Put right-shifted,
    added result in
    INDEX. (NEXT is
    some "next
    address" of no
    interest here.) */

```

condensed microcode
(under "1-microcycle
assumption")

NOTE: The 1-microcycle assumption not valid here because example assumes addition takes 3 micro-cycles. However, Tomasulo hardware could allow execution to proceed as on right.

```

@t1 /* Begin addition */
      A11←ACCUM; A12←INDEX; CI←0;
      ACNTRL←ADD; (MCR←instr'n 2;)

@t2 /* Added result not ready,
      ACCUM input not ready */
      ACCUMTAGREG←A01TAG;
      ACCUMBB←1;
      /* Tag ACCUM "busy awaiting
        A01" */
      (MCR←instr'n 3;)

@t3 /* Added result not ready,
      ACCUM busy, S11 input
      not ready */
      S11TAGREG←ACCUMTAGREG(=A01-
      TAG); S11BB←1;
      /* Mark S11 "busy awaiting
        whatever ACCUM is
        awaiting (A01)" */
      /* Supply available shifter
        inputs */
      ESI←0; SCNTRL←RSHFT;
      (MCR←instr'n 4;)

@t4 /* Added result ready */
      ACCUM←A01; ACCUMBB←0;
      /* Added result ready, but
        S11 (& thus S01) still
        marked "busy awaiting
        A01" @ start of t4 */
      INDEXTAGREG←S01TAG;
      INDEXBB←1; S11←A01; S11BB←0;
      /* Here, S11 gets adder
        output directly */

@t5 /* Shifter output ready */
      INDEX←S01; INDEXBB←0;

```

corresponding execution sequence
with Tomasulo-type hardware

NOTE: BB - Busy Bit,
TAGREG - Tag Register (for
holding tags),
TAG - Tag (unique # asso-
ciated with a
particular
hardware unit)

Fig. D7 Possible Use of Tomasulo-Type
Hardware [18], [19] to Aid HMO Algorithm