

SIG MICRO

NEWSLETTER

A Quarterly Publication of the
Special Interest Group on Microprogramming

SEPTEMBER 1976

VOLUME 7

NUMBER 3


CONTENTS:



NINTH
ANNUAL
WORKSHOP ON

MICROPROGRAMMING

SEPTEMBER 27-29, 1976
NEW ORLEANS, LOUISIANA

Sponsored by
acm Special Interest Group on Microprogramming
 IEEE COMPUTER SOCIETY Technical Committee
on Microprogramming

SIGMICRO - ACM SPECIAL INTEREST GROUP ON MICROPROGRAMMING

Chairperson

Dr. Stanley Habib
Polytechnic Institute of New York
333 Jay Street
Brooklyn, New York 11201
212-643-3908

Vice-Chairperson

Dr. Louise Jones
Information Systems Department
E.I. Dupont DeNemours & Co., Inc.
Wilmington, Delaware 19898
302-774-5389

Secretary-Treasure

Ted G. Lewis
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331
503-754-3273

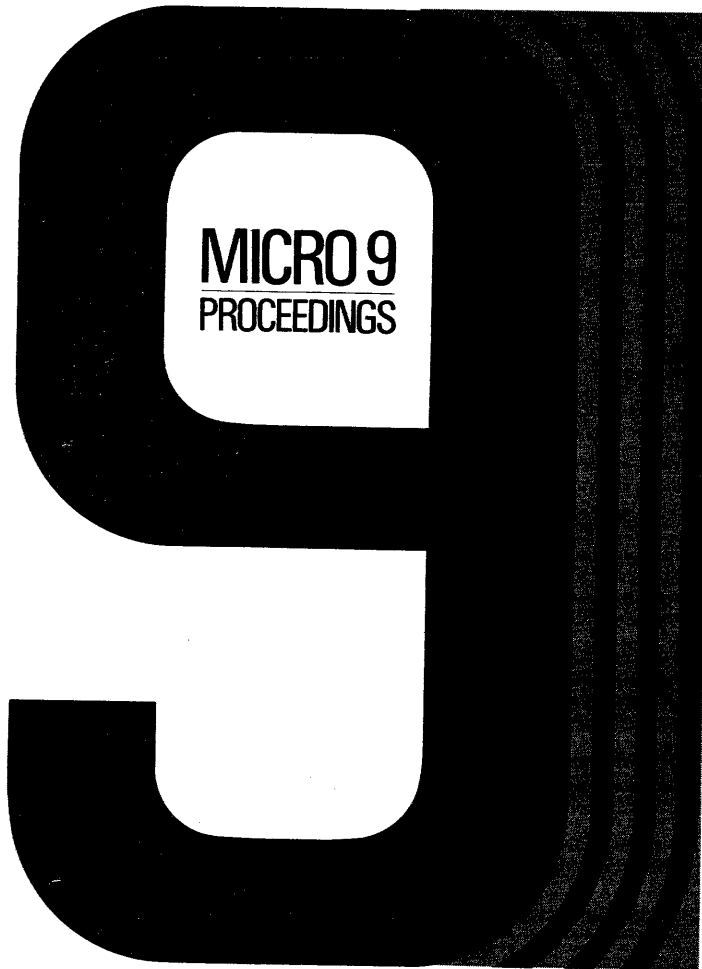
Newsletter Editor

Bruce D. Shriver
Computer Science Department
University of Southwestern Louisiana
Box 4-43380 USL
Lafayette, Louisiana 70504
318-233-3850

SIGMICRO Newsletter is an informal quarterly publication of the ACM Special Interest Group on Microprogramming whose scope of interest includes: Theory of Microprogramming; Applications of Microprogramming to Computer Control Systems; Use of Microprogramming by Computer Science Departments; Simulation based upon Microprogramming; Support Systems and other topics of interest to practitioners in the field of Microprogramming.

Membership in SIGMICRO is open to ACM members, associate members, and student members at a cost of \$4.00 per annum and non-ACM members, who are members of other professional societies whose major allegiance lies elsewhere, may also join at a cost of \$6.00 per annum. All members of SIGMICRO receive the Newsletter. A membership application including more details about joining SIGMICRO is included at the end of this publication.


Contributions to the SIGMICRO Newsletter should be sent to the Editor. Special camera-ready paper for typing manuscripts can be obtained from the Editor. All letters to the Editor will be considered for publication unless accompanied by a request to the contrary. Except for editorial items, all sources of material appearing in the SIGMICRO Newsletter will be clearly identified. Items attributed to individuals will ordinarily be interpreted as personal rather than organizational opinions.



NINTH
ANNUAL
WORKSHOP ON

MICROPROGRAMMING

SEPTEMBER 27-29, 1976
NEW ORLEANS, LOUISIANA

Sponsored by
acm Special Interest Group on Microprogramming
 IEEE COMPUTER SOCIETY Technical Committee
on Microprogramming

 Institute of Electrical and Electronics Engineers, Inc.

IEEE Catalog No. 76CH1148-6C

Copyright 1976 by

The Association for Computing Machinery
1133 Avenue of the Americas
New York, New York 10036

And

The Institute of Electrical and
Electronics Engineers, Inc.
345 East 47th Street
New York, New York 10017

IEEE Catalog No. 76CH1148-6C

Copies available from:

ACM
1133 Avenue of the Americas
New York, New York 10036

IEEE Computer Society
5855 Naples Plaza, Suite 301
Long Beach, California 90803

IEEE Service Center
445 Hoes Lane
Piscataway, New Jersey 08854

Manufactured in the U.S.A.

TABLE OF CONTENTS

	PAGE
A Bit Slice Architecture for Microprogrammable Machines by Michael Andrews	5
Certification of Microprograms by an Algebraic Method by A. Blikle and S. Budkowski	9
Microprogrammed Implementation of a Scheduler by Rahul Chattergy	15
An Insight into PDP-11 Emulation by J. C. Demco and T. A. Marsland	20
Design Problems In Emulating the MIX Computer on the Microdata 1600 by T. Don Dennis and O. G. Johnson	27
Extensibility - a New Approach for Designing Machine Independent Microprogramming Languages by David J. DeWitt	33
Realizing a Virtual Machine by B. Forbes, T. Weidner, R. Yoder, and T. Pitchford	42
The FORTRAN Project - A Multifaceted Approach to Software - Firmware High Level Language Support by Gideon Frieder	47
Automated Proofs of Microprogram Correctness by W. H. Joyner, W. C. Carter, and G. B. Leeman	51
A Microprogrammed Machine Architecture for Efficient Matrix Multiplication by Robert Nowlin and Donald Gustafson	56

A BIT SLICE ARCHITECTURE FOR MICROPROGRAMMABLE MACHINES*

Michael Andrews

Department of Electrical Engineering, Colorado State University
Ft. Collins, Colorado 80523

The use of a bit slice architecture offers an attractive alternative to the design of microprogrammable computing architectures. By bit slice we mean typically a LSI chip containing either a 2 bit, 4 bit where n is very small. This paper describes these particular architectures to design in horizontal microprogramming. Hardware design implementations for both fixed point and floating point arithmetic algorithms are proposed. Experiences with these numerical hardware are discussed, along with problems encountered.

1. INTRODUCTION

Until recently, computing hardware for original design has been based mainly upon LSI (large scale integration) chips which, at a minimum, were 8 bits wide. Generally, configurations have been implemented with 16 bit chips. Now it is possible to design architectures with chips that are very short word width, of the order 2 or 4 bits. This is made possible with the introduction of several industrial chips, [1-4], and others. These short width LSI packages offer some versatility in the design and control of computing architectures for variable word length scenarios. Typically such applications are found in the process control industry, numerical control of milling machines, etc., and other on-line control applications. In this paper we propose some architectural designs using these bit-slice chips. It is found that an additional vehicle for horizontal microprogramming is now possible.

The paper begins with a description of the bit-slice architecture of the INTEL series. The next section discusses a hardware design using a modified INTEL 3000 in a floating point processor. This section is followed by a discussion of the microprogramming sequencing techniques employed in the 3000 series. The paper ends with a discussion of the use of the 3000 as a vehicle for demonstrating microprogramming techniques. At this point, however, we mention that a major modification to the INTEL 3002 CPE chip, for our studies, was made in order to employ horizontal microprogramming. In particular, rather than use commercial functions, our functions were devised on the basis of the availability of control gate connections. These connections are hypothesized to permit elementary microoperations. In this way, it is possible to demonstrate the actual timing requirements internal to any CPU given this control gate accessibility. The discussion to follow focuses briefly upon the 3000 series and our modifications to the CPU chip.

2. BIT SLICE ARCHITECTURES

This section provides a brief overview of a bit

*This work was supported by the AFAL under grant F 33615-75-C-1138

slice architecture, the Intel 3000 Series. The 3000 series is configured around two LSI, a 3002 central processing element (CPE), and a 3001 microprogram control unit (MCU)[5]. With ancillary hardware to enhance sophisticated implementations, they include a carry look-ahead generator, pipeline latches, clocks. The CPE chip is a basic ALU which does primitive boolean functions, excluding left-shifts. This absence of a left-shift indeed penalizes some macroinstruction programming as will be shown later. The ALU has two input sources via 2 buses, an M bus and an I bus. However, the I bus is maskable with another bus, the K bus, which allows for byte manipulation and bit masking. The ALU outputs to two destinations, a memory address register and an AC (accumulator) register. However, it has been found in practice that the AC does not perform as a typical accumulator. Rather, an eleven cell scratch pad serves more suitably as the traditional accumulators. The CPE also contains the necessary function bus decoder logic to control the ALU and register transfers. Lastly, carry-in and-out, left-in, and right-out signal lines are available. When cascaded to form an 8 bit or 16 bit macro-ALU, the configuration is controlled by cascades of the chip, which serves as the microprogram sequencer. No address incrementer is available. Thus, all sequencing is via jump addresses [6].

3. FIXED POINT BIT SLICE ARCHITECTURE

Implementation of a fixed point binary machine, 16 bits wide has been accomplished with eight 2 bit CPE's similar to the Intel 3000 Series. See fig. 1.

Modifications were made to the commercially available CPE chip in order to provide for a horizontal micro control structure [7,8]. This is accomplished by assuming that the individual control gates internal to the CPE chip are available to the microprogrammer as a field in the microinstruction. There are 5 fields in each microinstruction: the control field for the control gates, C_1 through C_{19} , a K field of 16 bits, a function field of 4 bits labeled F_6 through F_3 , an address field and a T-field (for branch testing). Each microinstruction word was 42 bits wide not including the next address field which, of course, depends on the total microprogram content for any machine. The structure in the microprogram control was assumed as a horizontal machine. Thus simultaneous register activity could result in the CPE chips. However, because the chip accumulator did not serve as such in the traditional sense, and because the scratch-pad registers were used as accumulators, this particular configuration placed some penalty upon the microprogram size. As a result, for most of the fixed point add and subtract, and all of the fixed point multiply and divide routines, extra microinstructions were required to maintain an accumulating partial product or partial remainder. For a typical nonrestoring division routine by successive additions 30 microinstructions were required. This assumed that the AC held the divisor, and the scratch-pad registers contained the partial remainder, dividend, and shift-right bit counters. Although the 3000 series microprogram format assumes an R group concatenated with an F group for the on-chip function decoder, our configuration assumed the function decoder was simply 4 bits wide instead of 7. Scratch-pad control gates C_3 through C_{13} , both enabled the input and output gates of each scratch-pad cell respectively. This then required that any scratch-pad cell use become both a source and destination for micro operations when called. Without this modification it would have been necessary to add at least 10 more bits to the microinstruction field length for additional control gate signals.

4. FLOATING POINT BIT-SLICE ARCHITECTURE

Floating point architecture again assumes an Intel 3000 CPE chip modified for horizontal microprogramming. However, it was desirable to speed up floating point manipulations. Thus, additional hardware was included to perform some parallel processing. The word format for this machine assumed a sign magnitude notation number system with 16 binary bits for the fraction and 8 binary bits for the characteristic (with offset or bias). For this configuration, then, 8 CPE chips were cascaded for the fraction portion of the algorithm and 4 CPE chips were cascaded for the characteristic portion. Horizontal control, however, allowed for arithmetic functions in both the characteristic and fraction ALUs. This form of parallel processing decreased the execution time of typical floating point manipulations, at the cost of a wider microinstruction width. The hardware for the floating point design is shown in Fig. 2.

Here, like the fixed point design, a T-field was included in the microinstruction word for microinstruction branching during the second phase of our 3 phase clock. In addition, some divide overflow and characteristic underflow indicators were available and reserved in the flag set field of the microinstruction. It was found desirable to include also a CPE group enable field to allow inhibiting of either the fraction of characteristic ALU when so desired. The additional gating required to implement our floating point processor is also shown. These include some miscellaneous control gates which are enabled by the test bits from the CPE chips.

The timing sequence for the processor and microprogram control includes a 3-phase clock with phase 3 designated for fetch cycle, phase 1 for the execute cycle, and phase 2 for branch decision. This timing reflects solely the microprogram control. Additional clocking is required, of course, for microprogram and main-memory cycles, not shown. A typical parallel processing algorithm is shown for the floating point multiply routine of Fig. 3.

Here, initialization and characteristic alignment precede the prenormalization check as with most floating point algorithms. A decision for zero product is made next. This is followed by the parallel processing of the characteristic addition and the fraction multiply. Because an ALU exists for both operations, these arithmetic sequences can be performed in parallel. Upon completion of the sequence, a postnormalization decision is made and the procedure terminates after the normalization is required. A similar flow chart follows for the floating point divide. For our floating point multiply routine and the floating point architecture shown, it was possible to microprogram the floating point multiply in approximately 60 microinstructions. Again, the scratch-pad served as the accumulating partial product and partial remainder while the chip accumulator held the divisor or multiplier during processing.

5. CONCLUSIONS

A hardwired floating-point processor designed about a bit-slice architecture has been proposed. Microprogram implementations for multiplication have been described. It is seen that the for

certain commercially available LSI architecture, modifications to traditional functional element roles were necessary. In particular, the AC in the Intel 3001 CPE functions more like a latch than an accumulating register. Also, further modifications to the CPE control gate structure were introduced to emphasize a horizontal microprogrammable machine. Use of cascaded chips allowed for independent fraction and characteristic ALU blocks. Horizontal control, then, permitted simultaneous parallel activity, though, at the expense of increased microinstruction word width. One unresolved question is the relative comparison of our architecture to others, namely, the Intel 3002 unmodified cascaded configuration.

ACKNOWLEDGMENT

The author wishes to acknowledge the contributions of Robert Emberty, Daniel Eggerding and Greg Lipinsky who analyzed processor designs with floating point microprograms.

REFERENCES

- [1] INTEL Series 3000 Microprogramming Manual, Intel Corporation, Santa Clara, Calif., 1975, 1-10.
- [2] AM 2901, AM 2909 Technical Data Manual, Advanced Microdevices, Incorporated, Calif., 1975.
- [3] J. Mick, "AM 2900 Bipolar Microprocessor Family," Electronics, 8th Annual Workshop on Microprogramming, Chicago, Ill., Sept. 1975, 56-63.
- [4] J. Rattner, J. Cornet, and M. E. Hoff, "Bipolar LSI Computing Elements Usher in New Era of Digital Design," Electronics, vol. 47, no. 18, Sept. 5, 1974.
- [5] Central Processor Design Using the INTEL Series 3000 Computing Elements, Application Note AP-16, Intel Corp., Santa Clara, Calif., 1975.
- [6] J. F. Wakerly, C. P. Hollander, and D. Davies, "Placement of Microinstructions in a Two-Dimensional Address Space," 8th Annual Workshop on Microprogramming, Chicago, Illinois, Sept. 1975, 46-51.
- [7] G. Reyling, "Considerations in Choosing a Microprogrammable Bit Sliced Architecture," Computer, vol. 7, no. 7, July 1974, 26-29.
- [8] I. Lee, "LSI Microprocessors and Microprograms for User Oriented Machines," MICRO-7 Annual Workshop on Microprogramming, Palo Alto, Calif., Oct. 1974, S-1 through S-9.

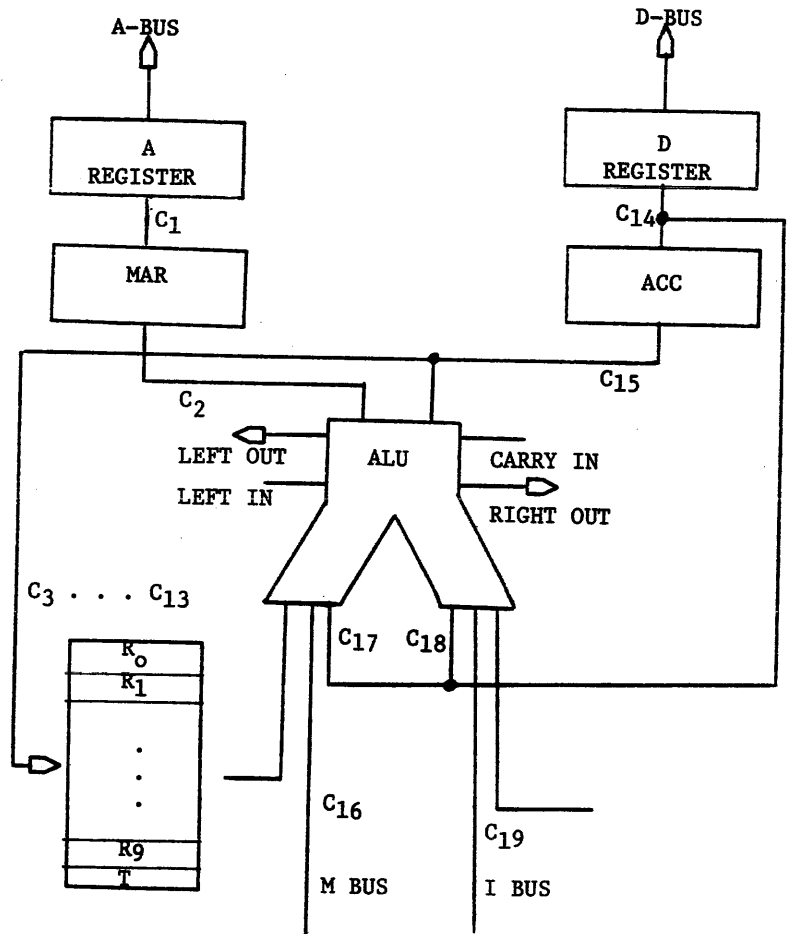


Fig. 1 - Modified Processor Chip

Fig. 2 Floating Point Processor

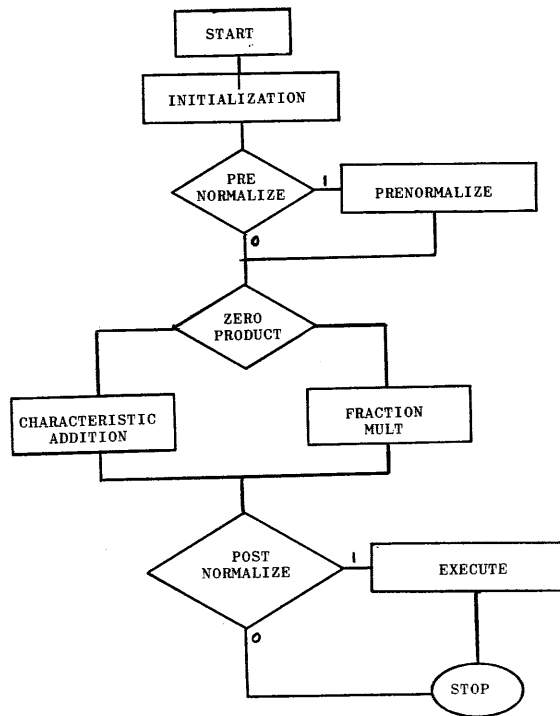
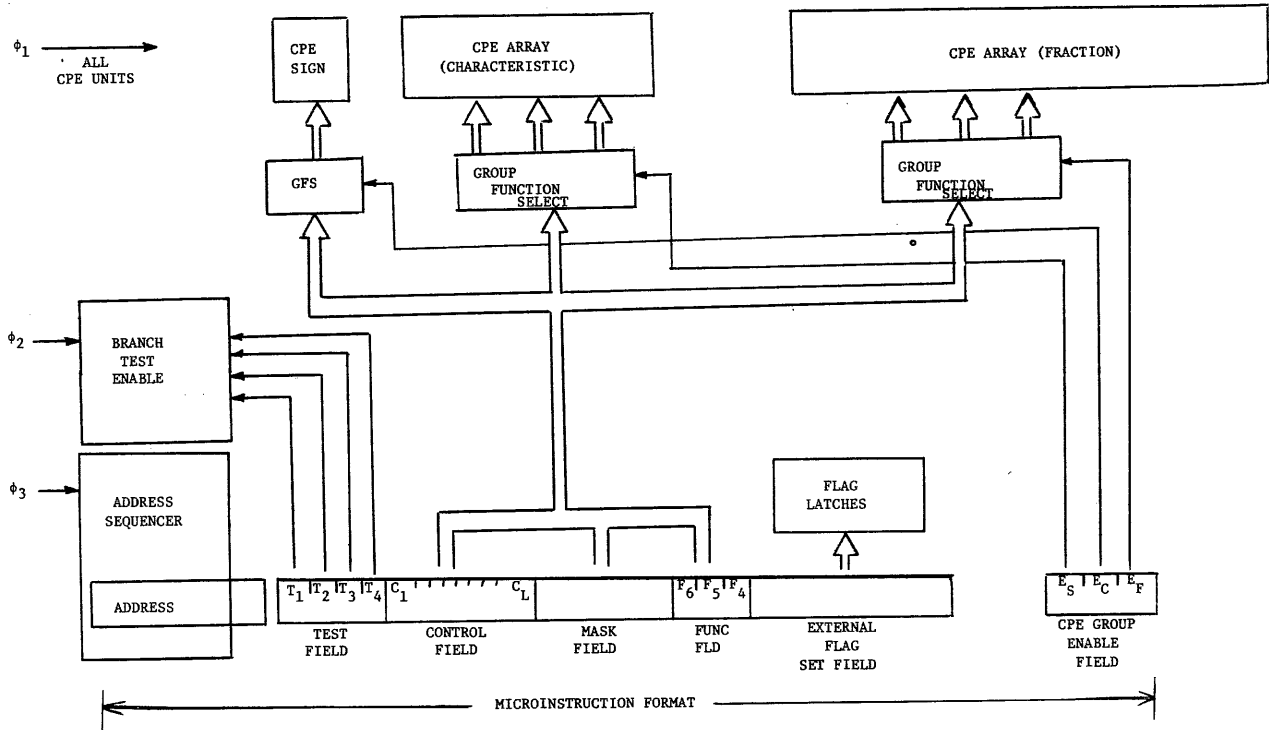


Fig. 3 Floating Point Mult

CERTIFICATION OF MICROPROGRAMS BY AN ALGEBRAIC METHOD

A. Blikle*
 Computation Center
 Polish Academy of Sciences
 00-901 Warsaw, Poland

S. Budkowski**
 Dept. of Computer Science
 Warsaw Technical University
 00-665 Warsaw, Poland

This algebraic method provides a mathematical framework for proving input-output properties (such as partial and total correctness) of iterative programs. Technically it uses a calculus of binary relations extended with fixed-point equations. The method has been tested on several microprograms of a computer's arithmetical unit. One example of such a microprogram and its correctness proof is discussed in the paper.

1. INTRODUCTION

In the current literature of the subject one can distinguish two different trends of attacking the problem of the mathematical certification of microprograms. In one approach ([2],[9],[11],[12]) the analyzed microprogram and its expected meaning are described by two abstract machines one of which is defined on the hardware level and the other on the level of architecture. The correctness proofs consist of showing that one of these machines simulates the other in the sense defined by R. Milner [15]. In the other approach the expected meaning of a microprogram is described by more mathematical (or less operational) terms either by verification conditions [16], which involves Floyd's method, or by regular expressions [10] which involves the algebra of events together with the related fixed-point equations. In the latter two cases one refers to some standard mathematical methods of software-program verification.

This paper presents another software-program verification method ([3],[4],[5],[8],[14]) applied to microprograms [6]. The general idea of this method is the following: Given a program Π its meaning is assumed to be a binary input-output relation R (I-O relation) which describes the mapping of the initial values of the vector of variables into the terminal values of this vector.

* Part of this work was done when the author was visiting the Dept. of Comp. Sci., University of Waterloo. This part was supported by the National Research Council of Canada under Grant A-1617.

** Part of this work was done when the author was visiting the Department of Computer Science, University of Maryland. This part was partially supported under grant NSF DCR75-05505.

To establish R explicitly we split the program Π into a finite number of modules Π_1, \dots, Π_n (e.g. assignment statements and tests) which must be simple enough for their I-O relations R_1, \dots, R_n to be obvious. Since the program Π is a combination of Π_1, \dots, Π_n the relation R must be a combination of R_1, \dots, R_n :

$$R = \Psi(R_1, \dots, R_n) \quad (1.1)$$

The function Ψ is defined in the algebra of relations (Sec.2) and describes the control structure of Π . To find the function Ψ we use an algebraic method which consists of writing and solving a set of fixed-point equations in our algebra. Once Ψ has been found, we use (1.1) in the further analysis of Π . This analysis is carried out in the algebra of relations and permits proofs of partial as well as total correctness of programs.

The general method above was applied by the authors to several arithmetical microprograms of a floating-point arithmetical unit designed at the Warsaw Technical University. This application has raised a problem which is usually neglected in the consideration of software programs. Namely, the arithmetical microprograms involve computer arithmetics which is fairly different from the usual (Peano's) arithmetics. E.g. in the computer arithmetics the law of the distributivity of multiplication by 2^{-1} (arithmetical shift right) does not hold. The lack of this property makes the calculations which appear in the verification of the microprogram practically impossible. To solve this problem we consider two programs Π_1 and Π_2 , where Π_1 is the "real" microprogram and Π_2 is an abstract program resulting from Π_1 by the replacement of the machine operations by the corresponding arithmetical ones. We verify the program Π_2 and then we show that Π_2 simulates Π_1 in the following sense: Let F_1 and F_2 denote the I-O functions of Π_1 and Π_2 respectively, let D_1 and D_2 denote the input domains of Π_1 and Π_2 (i.e. the domains of F_1 and F_2) and let $D_1 \subseteq D_2$. There exists a function

$T: D_2 \rightarrow D_1$ such that $T(d) = d$ for all $d \in D_1$ and $F_1(T(d)) = T(F_2(d))$ for all $d \in D_2$. Now all the I-O properties of Π_2 can easily be "translated" into the I-O properties of Π_1 . This concept of simulation coincides, of course, with the algebraic simulation of R. Milner [15] which apparently makes our approach similar to that of A. Birman, B. Leeman and W. Carter. As a matter of fact, however, our program verification is not restricted to the proof of simulation but also provides the proof of the total correctness of Π_2 .

The organization of the paper is the following: First we describe the general Blikle-Mazurkiewicz method which is slightly modified here in regard to the form of fixed-point equations (this permits dealing with programs which have more than one output and proving their local properties - see [7]). Next we show a detailed example of the verification of a software program Π_3 which performs the Booth fixed-point multiplication algorithm and is a simplified version of an abstract program Π_2 which in turn simulates the real microprogram Π_1 . We describe the modifications required in Π_3 to get Π_2 and the modifications of Π_2 which result Π_1 . Referring to the analysis of Π_3 we describe briefly the analysis of Π_2 and show the function of simulation T between Π_1 and Π_2 .

2. THE ALGEBRA OF BINARY RELATIONS

Let D be an arbitrary nonempty set called the domain and interpreted as the set of all possible states of the vector of variables in a program. By $\text{Rel}(D)$ we denote the set of all binary relations in D, i.e. $\text{Rel}(D) = \{R \mid R \subseteq D \times D\}$. For any a, b in D and R in $\text{Rel}(D)$ we shall write aRb for $(a, b) \in R$. By ϕ we shall denote the empty relation, and by I the identity relation, i.e. $I = \{(a, a) \mid a \in D\}$.

Basic operations in the set $\text{Rel}(D)$, which we shall use in the sequel, are defined below. Let $R_1, R_2 \in \text{Rel}(D)$.

$$\begin{aligned} R_1 \cup R_2 &= \{(a, b) \mid aR_1b \quad aR_2b\} && \text{- union} \\ R_1 \circ R_2 &= \{(a, b) \mid (\exists c) aR_1c \ \& \ cR_2b\} && \text{- composition} \\ R_1^0 &= I && \text{- 0-th power} \\ R_1^n &= R_1^{n-1} \circ R_1 && \text{- n-th power} \\ R_1^* &= I \cup R_1 \cup R_1 \circ R_1 \cup \dots = \bigcup_{n=0}^{\infty} R_1^n && \text{- *-iteration} \\ R_1^+ &= R_1 \cup R_1 \circ R_1 \cup \dots = \bigcup_{n=1}^{\infty} R_1^n && \text{- +-iteration} \end{aligned}$$

Interpretation. The operations $\cup, \circ, *$ are used in the descriptions of the I-O relations of programs; precisely speaking they are used to describe explicitly the function Ψ in (1.1). Fig.1 shows the interpretation of the operations defined in this section. Each box in fig.1 represents a module of a flowchart of a microprogram with input-output relation R_i . These modules may be viewed either as elementary (with I-O relations described by assignment statements or tests) or as submicroprograms which consist of a number of elementary boxes. This permits the structuring of analyzed microprograms.

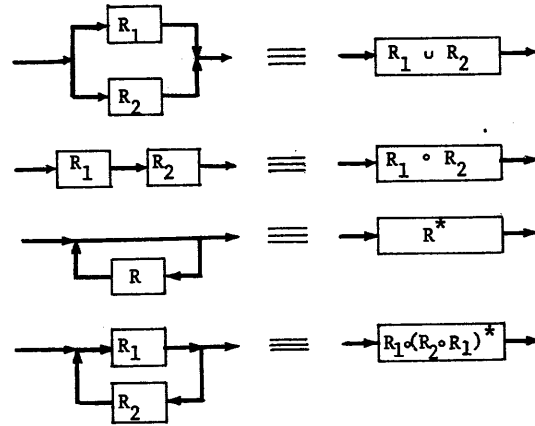


Fig. 1

Below we list the most important properties of these operations. Here and in the sequel we shall omit the symbol " \circ " of composition and write R_1R_2 instead of $R_1 \circ R_2$.

- 1) $R_1(R_2R_3) = (R_1R_2)R_3$ - associativity
- 2) $R_1(R_2 \cup R_3) = R_1R_2 \cup R_1R_3$ - finite distributivity
 $(R_2 \cup R_3)R_1 = R_2R_1 \cup R_3R_1$
- 3) $R_0(\bigcup_{i=1}^{\infty} R_i) = \bigcup_{i=1}^{\infty} R_0R_i$ - infinite distributivity
 $(\bigcup_{i=1}^{\infty} R_i)R_0 = \bigcup_{i=1}^{\infty} R_iR_0$
- 4) $RI = IR = R$ - the unit property of I
- 5) $R\phi = \phi R = \phi$ - the zero property of ϕ
- 6) $R^* = I \cup R^+$
 $R^+ = RR^* = R^*R$
- 7) $R_1(R_2R_1)^*R_2 = (R_1R_2)^*R_1R_2$

To deal with concrete programs (and microprograms) and to carry out their analysis we shall need an explicit notation to specify the I-O relations. Since we are going to deal only with deterministic programs, we can restrict ourselves to the case of partial functions and use the notation introduced by A. Mazurkiewicz [8]. A relation $R \subseteq D \times D$ is a partial function if for any $d_1 \in D$ such that d_1Rd_2 .

Let $f: D \rightarrow D$ be an arbitrary partial function and let $p: D \rightarrow \{\text{true}, \text{false}\}$ be an arbitrary predicate such that if $p(d) = \text{true}$ then $f(d)$ is defined. We denote by

$$\begin{aligned} [p(x) \mid x := f(x)] &= && (2.1) \\ = \{(d_1, d_2) \mid p(d_1) = \text{true} \ \& \ d_2 = f(d_1)\} \end{aligned}$$

Of course, $[p(x)|x := f(x)]$ is a partial function whose domain is $\{d | d \in D \text{ \& } p(d) = \text{true}\}$. For the sake of simplicity we shall also write $[x := f(x)]$ for $[\text{true} | x := f(x)]$ and $[p(x)]$ for $[p(x)|x := x]$. Of course $[p(x)][x := f(x)] = [p(x)|x := f(x)]$. In the sequel we shall use the following equivalences which can be proved easily from (2.1):

- 1) $[p(x)|x := f(x)][q(x)|x := g(x)] = [p(x) \& q(f(x)) | x := g(f(x))]$ (2.2)
- 2) $[p(x)|x := f(x)] \cup [q(x)|x := f(x)] = [p(x) \vee q(x) | x := f(x)]$

3. THE MATHEMATICAL MODELS OF PROGRAMS

In order to define in a rigorous way the concept of the I-O relation, we need here a rigorous concept of a program. To this effect, we shall use the notion of an algorithm introduced by A. Mazurkiewicz [8].

By an algorithm we shall mean any system $A = (D, V, \alpha_1, \mathcal{J})$ where

- D is an arbitrary nonempty set called the domain of the algorithm and is interpreted as in Sec. 2,
- $V = \{\alpha_1, \dots, \alpha_n\}$ is a finite nonempty set of elements called labels of the algorithm,
- α_1 is a distinguished element of V called the initial label of the algorithm,
- $\mathcal{J} = \{\alpha_i, R_{ij}, \alpha_j | R_{ij} \in \text{Rel}(D); i, j \leq n\}$ is a set of $p = n^2$ triples called instructions. Usually many of the $R_{ij} = \emptyset$ which describes the fact that there is no direct trespassing between α_1 and α_j in the program. Given an instruction, the corresponding α_i , R_{ij} and α_j are called the entrance label, the action and the exit label. The α_i 's are interpreted as the control states and the R_{ij} 's define the meaning of "boxes".

We are going to apply our theory to deterministic programs only. Nevertheless, the theory itself will be developed in the general nondeterministic case which makes its presentation much simpler.

Consider an arbitrary algorithm $A = (D, V, \alpha_1, \mathcal{J})$ where $V = \{\alpha_1, \dots, \alpha_n\}$. For any $\alpha_i, \alpha_j \in V$, by an (α_i, α_j) -run we shall mean any sequence of instructions of \mathcal{J} :

$$(\alpha_{i_1}, R_{i_1, \alpha_{j_1}}); \dots; (\alpha_{i_k}, R_{i_k, \alpha_{j_k}}) \quad (3.1)$$

such that $\alpha_{i_1} = \alpha_i$; $\alpha_{j_k} = \alpha_j$ and $\alpha_{j_p} = \alpha_{i_{p+1}}$ for $p \leq k-1$. Of course, an (α_i, α_j) -run is simply a path in the graph of A . The corresponding sequence of actions (R_1, \dots, R_k) will be called an (α_i, α_j) -symbolic execution (s. execution). Let $\text{Exec}(\alpha_i, \alpha_j)$ denote the set of all the (α_i, α_j) -s. executions in A . The (α_i, α_j) -resulting relation is defined as follows:

$$\text{Res}(\alpha_i, \alpha_j) = \cup \{R_1 \circ \dots \circ R_k \mid (R_1, \dots, R_k) \in \text{Exec}(\alpha_i, \alpha_j)\} \quad (3.2)$$

This is of course the I-O relation of A under the assumption that α_1 is the input label and α_j is the output label. Indeed $d_1 \text{Res}(\alpha_i, \alpha_j) d_2$ iff there exists an (α_i, α_j) -s. execution (R_1, \dots, R_k) such that $d_1 R_1 \circ \dots \circ R_k d_2$. Observe that in any (α_i, α_j) -run the control of the algorithm may pass through α_1 and α_j many times.

By the definition of A the label α_1 is assumed to be initial and therefore we shall be interested mostly in the relations $R(\alpha_1, \alpha_j)$ for $j = 1, \dots, n$. Moreover, among these n relations we shall select usually some number of $k \leq n$ relation that correspond to the actual outputs of the program. The particular one-output case corresponds to $k = 1$, but in some applications we may want to consider programs with more than one output (e.g. the successful termination, the overflow and the underflow).

Now suppose we are considering an algorithm A where α_n has been chosen to be the terminal label and suppose that we have proved

$$\text{Res}(\alpha_1, \alpha_n) = [p(x)|x := f(x)]. \quad (3.3)$$

By the definition of $\text{Res}(\alpha_1, \alpha_j)$ this implies the following about A :

- 1) for every initial $d \in D$ the algorithm terminates (stops) if and only if $p(d)$ is satisfied,
- 2) for every initial $d \in D$ if the algorithm terminates, then the terminal value of x is $f(d)$.

Of course 2) is a partial-correctness property of A and 1) defines exactly the domain of termination. Consequently, (3.3) is the strongest total-correctness property of A , since $p(x)$ is not only sufficient but is also a necessary condition of termination (for the concepts of partial and total correctness see [13]).

4. FIXED-POINT EQUATIONS AND PROGRAMS

Dealing with concrete programs we shall attempt to express their resulting relations $\text{Res}(\alpha_1, \alpha_j)$ in terms of the actions of instructions R_{ij} and the operations defined in Sec.2. The definition (3.2) does indicate how to do it (see $\text{Res}(\alpha_2, \alpha_3)$ in Sec.5), but finding $\text{Res}(\alpha_1, \alpha_j)$ this way may be difficult even for a program of middle complexity. Below we present a method of fixed-point equations which permits to find all $\text{Res}(\alpha_1, \alpha_j)$ for programs of any complexity. These equations differ from the ones used previously (see [7] for detailed explanations).

Let $A = (D, V, \alpha_1, \mathcal{J})$ be an arbitrary algorithm with $V = \{\alpha_1, \dots, \alpha_n\}$. By the canonical set of equation (CSE) of A we mean the set:

$$\begin{aligned} X_1 &= X_1 R_{11} \cup \dots \cup X_n R_{n1} \cup R_{11} \\ &\dots \\ X_n &= X_1 R_{1n} \cup \dots \cup X_n R_{nn} \cup R_{1n} \end{aligned} \quad (4.1)$$

where every R_{ij} is, of course, the action of the instruction $(\alpha_i, R_{ij}, \alpha_j)$ in \mathcal{J} . The unknowns X_i of

(4.1) range, of course, over the set $\text{Rel}(D)$ of relations. Any vector (P_1, \dots, P_n) of relations which satisfies (4.1) is called a solution of this set. In the general case (4.1) has more than one solution. The solution (P_1, \dots, P_n) is said to be the least solution if for any other solution (Q_1, \dots, Q_n) we have $P_i \subseteq Q_i$ for $i = 1, \dots, n$. It is a well-known fact that the least solution, if any, is unique. In our case we can prove the following:

Theorem 1. For any algorithm A the vector of relations $(\text{Res}(\alpha_1, \alpha_1), \dots, \text{Res}(\alpha_1, \alpha_n))$ is the least solution of the corresponding CSE. \square

The proof is described in [7]. Here we shall show an effective method of solving (4.1). The method consists in the application of two variable-elimination transformations:

1. Substitution: the substitution of $X_1 R_{1i} \cup \dots \cup X_n R_{ni} \cup R_{ii}$ for an arbitrary occurrence of X_i on the right side of (4.1).
2. Iteration: the replacement of the equation $X_i = X_1 R_{1i} \cup \dots \cup X_{i-1} R_{i-1i} \cup \dots \cup X_n R_{ni} \cup R_{ii}$ by the equation

$$X_i = (X_1 R_{1i} \cup \dots \cup X_{i-1} R_{i-1i} \cup \dots \cup X_{i+1} R_{i+1i} \cup \dots \cup X_n R_{ni} \cup R_{ii}) R_{ii}^*$$

Each of these transformations is applicable to any set of equations like (4.1) and yields another set of equations of the same form. As can be proved (see [5] for the references) the new set of equations has exactly the same least solution as the former. To solve a given CSE we keep applying our transformations as long as there are some unknowns (variables) on the right side.

5. AN EXAMPLE OF A SOFTWARE-PROGRAM VERIFICATION

The method described in this paper has been tested on several arithmetical microprograms of a floating-point arithmetical unit designed at the Warsaw Technical University. One of these microprograms, call it Π_1 , performed Booth's algorithm of the fixed-point multiplication of mantissas. We analyzed Π_1 by introducing and verifying an abstract program Π_2 and by proving that Π_2 simulates Π_1 (see Sec.1 and 6). Here we shall investigate a simplified version of Π_2 , call it Π_3 , which differs from Π_2 in neglecting overflows. In spite of this simplification the example still provides an adequate flavor of the method. In Sec.6 we show how to extend these calculations to deal with the real cases of Π_2 and Π_1 .

In our program we shall deal with numbers represented in the 2's complement code: The numbers from the interval $\langle -1, 1 \rangle$ are represented as:

$$\beta = -\beta_0 + \sum_{i=1}^n \beta_i * 2^{-i} \quad (5.1)$$

where $\beta_i \in \{0, 1\}$ for $i = 0, \dots, n$ and $n \geq 1$ is fixed. We shall also use the equation

$$\beta = \sum_{i=0}^n (\beta_{i+1} - \beta_i) * 2^{-i} \quad (5.2)$$

which follows from (5.1) under the condition that $\beta_{n+1} = 0$. The flowchart of our program Π_3 is given in fig. 2.

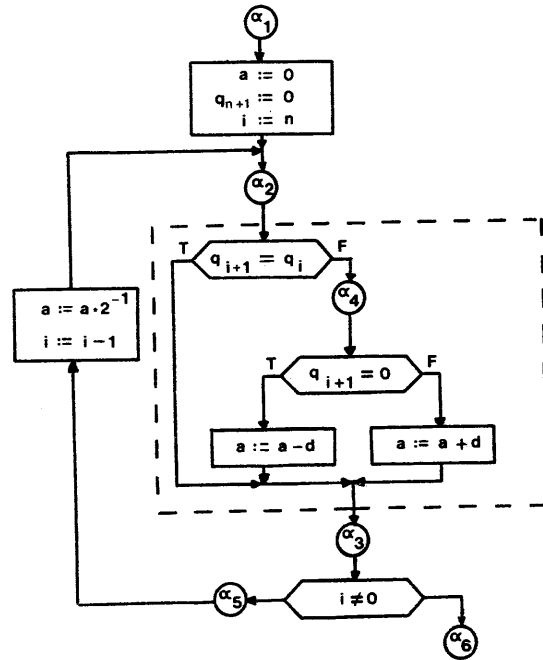


Fig. 2

This program operates on the following variables:

- 1) the real variable a which ranges over an arbitrary interval of real numbers (this is actually the assumption which neglects the overflow),
- 2) the real variable d which ranges over $\langle -1, 1 \rangle$,
- 3) the integer variable i which ranges over $\{0, \dots, n\}$,
- 4) the 0's and 1's array $q[0:n+1]$.

We shall prove that the program performs the multi-

plication $(-q_0 + \sum_{j=1}^n q_j * 2^{-j}) * d$ and stores the result in a .

In the calculations we shall use an extension of the notation introduced in Sec.2. Namely, for any vector of variables (x_1, \dots, x_n) , the function $[(x_1, \dots, x_n) := (f_1(x_1, \dots, x_n), \dots, f_n(x_1, \dots, x_n))]$ will be written as

$$\begin{bmatrix} x_1 := f_1(x_1, \dots, x_n) \\ \dots \\ x_n := f_n(x_1, \dots, x_n) \end{bmatrix} \quad (5.3)$$

Of course all the assignment statements in (5.3) are understood to be performed simultaneously. We assume also to omit in (5.3) all the assignment statements of the form $x_i := x_i$.

We shall prove the following about our program:

$$\text{Res}(\alpha_1, \alpha_6) = \begin{bmatrix} a := (-q_0 + \sum_{j=1}^n q_j * 2^{-j}) * d \\ i := 0 \\ q_{n+1} := 0 \end{bmatrix} \quad (5.4)$$

According to the definition of $\text{Res}(\alpha_1, \alpha_j)$ (see also the remarks by the end of Sec.2) this implies that our program terminates everywhere in its domain and that it performs the multiplication of the number whose representation is stored in $q[0:n]$ by the number which is stored in d .

To simplify the calculations let's consider first a module of fig.2 between the control states α_2 and α_3 . For this simple program we can find the $\text{Res}(\alpha_2, \alpha_3)$ directly from the definition (3.2). Namely (using (2.2)):

$$\begin{aligned} \text{Res}(\alpha_2, \alpha_3) &= [q_{i+1} = q_i] \cup \\ &\cup [q_{i+1} \neq q_i] ([q_{i+1} = 0 | a := a-d] \cup \\ &\cup [q_{i+1} = 1 | a := a+d]) = \\ &= [q_{i+1} = q_i | a := a+(q_{i+1} - q_i)*d] \cup \\ &\cup [q_{i+1} = 0 \ \& \ q_i = 1 | a := a+(q_{i+1} - q_i)*d] \cup \\ &\cup [q_{i+1} = 1 \ \& \ q_i = 0 | a := a+(q_{i+1} - q_i)*d] = \\ &= [a := a+(q_{i+1} - q_i)*d]. \end{aligned}$$

In the next step we establish the CSE of the program of fig.2:

$$\begin{aligned} X_1 &= \phi \\ X_2 &= X_5 R_{52} \cup R_{12} \\ X_3 &= X_2 \text{Res}(\alpha_2, \alpha_3) \\ X_5 &= X_3 R_{35} \\ X_6 &= X_6 R_{36} \end{aligned}$$

where

$$R_{52} = \begin{bmatrix} a := a*2^{-1} \\ i := i-1 \end{bmatrix}; \quad R_{12} = \begin{bmatrix} a := 0 \\ q_{n+1} := 0 \\ i := n \end{bmatrix}$$

$$R_{35} = [i \neq 0]; \quad R_{36} = [i = 0]$$

Solving this set with respect to X_6 we get

$$X_6 = R_{12} (\text{Res}(\alpha_2, \alpha_3) R_{35} R_{52})^* \text{Res}(\alpha_2, \alpha_3) R_{36}. \quad (5.5)$$

By Theorem 1 we have, of course, $X_6 = \text{Res}(\alpha_1, \alpha_6)$. For $1 \leq k \leq n$ denote

$$S_k = \begin{bmatrix} a := \sum_{i=1}^k (q_{n+1-k+i} - q_{n-k+i}) * 2^{-i} * d \\ i := n-k \\ q_{n+1} := 0 \end{bmatrix}$$

and for all $k \geq 1$ denote

$$P_k = R_{12} (\text{Res}(\alpha_2, \alpha_3) R_{35} R_{52})^k.$$

We can prove by induction (see [7] for details) that for all $1 \leq k \leq n$ we have $P_k = S_k$. We can prove also [7] that the loop in our program must be performed exactly n times, which formally means (c.f. (5.5)) that

$$\text{Res}(\alpha_1, \alpha_6) = S_n \text{Res}(\alpha_2, \alpha_3) [i=0]$$

By straightforward calculations we get now

$$\text{Res}(\alpha_1, \alpha_6) = \begin{bmatrix} a := \sum_{i=0}^n (q_{i+1} - q_i) * 2^{-i} * d \\ i := 0 \\ q_{n+1} := 0 \end{bmatrix}$$

which by (5.2) gives us the required result (5.4). This terminates the proof of the total correctness of the program Π_3 .

It can be observed [7] that using our method one can also prove some local properties of the program, as for example the estimation of the current value of a at the label α_2 and α_3 .

6. AN EXAMPLE OF A MICROPROGRAM VERIFICATION

Let us consider now the program Π_2 which differs from Π_3 in the following:

- 1) d ranges over numbers representable by (5.1) for $n = 23$
- 2) a ranges over numbers representable by (5.1) for $n = 47$ (generally speaking $n \geq 47$)
- 3) to the assignments $a := 0; q_{n+1} := 0; i := n$ for $n = 23$ the assignment $z := 0$ is added
- 4) the assignments $a := a+d$ are replaced respectively by:

$$\begin{aligned} a &:= a+d - 2*0vf(a+d); \\ z &:= |0vf(a+d)|; \end{aligned}$$

where

$$0vf(x) = \begin{cases} -1; & x < -1 \\ 0; & -1 \leq x < 1 \\ 1; & 1 \leq x \end{cases}$$

- 5) the assignment $a := a*2^{-1}$ is replaced by:

$$\text{if } z = 0 \text{ then } a := a*2^{-1} \text{ else } a := a*2^{-1} + a_0 \\ z := 0;$$

This means that R_{52} is equal now

$$R_{52} = [z=0 | a := a*2^{-1}; i := i-1; z := 0] \cup \\ \cup [z=1 | a := a*2^{-1} + a_0; i := i-1; z := 0]$$

Following the same way as in Sec.5 - just with more calculations - we can prove the following:

$$\text{Res}_2(\alpha_1, \alpha_6) = \begin{bmatrix} a := q*d - 2*0vf(q*d) \\ i := 0 \\ q_{24} := 0 \\ z := 0vf(q*d) \end{bmatrix} \quad (6.1)$$

where $\text{Res}_2(\alpha_1, \alpha_6)$ is the appropriate I-0 relation in

Π_2 , $q = \sum_{i=0}^{23} (q_{i+1} - q_i) * 2^{-i}$ and a is the 48-bit representation of the required result (double precision).

Now consider the original microprogram Π_1 for which the length of a is assumed to be 24-bit only (single precision). Every occurrence of $a*2^{-1}$ in Π_2 must now be replaced by $a \oplus 2^{-1}$ since the computer multiplication \oplus by 2^{-1} (arithmetical shift right of the content of the register storing a) is no longer the usual one. Observe that in general

$$a \oplus 2^{-1} = a*2^{-1} - a_n * 2^{-(n+1)}$$

and only for Π_2 (and Π_3) $a \oplus 2^{-1} = a*2^{-1}$.

Of course, we could use this equation with $n=23$ to replace every occurrence of a $\theta 2^{-1}$ in Π_1 by its right side and to get an equivalent program Π_1 with only usual arithmetical operations. The analysis of such a program, however, would be very cumbersome. It is much easier to observe that Π_2 simulates Π_1 and to verify Π_2 . Let us describe briefly what the simulation of Π_1 by Π_2 looks like.

In the original case of our analysis we proceed, of course from Π_1 to Π_2 . First we establish the algorithm $A_1 = (D_1, V, \alpha_1, \mathbb{J}_1)$ which corresponds to Π_1 . The domain D_1 is the set of all the vectors of the form $(a, d, q_0, \dots, q_{24}, i, z)$ where a and d range over these numbers in $\langle -1, 1 \rangle$ which can be represented by (5.1) with $n = 23$, and the other variables have the ranges as described earlier. Next we establish the algorithm $A_2 = (D_2, V, \alpha_1, \mathbb{J}_2)$ which corresponds to Π_2 and which differs from A_1 only in D_2 and \mathbb{J}_2 . The set D_2 results from D_1 by letting a range over all numbers in $\langle -1, 1 \rangle$. The set \mathbb{J}_2 results from \mathbb{J}_1 by replacing every instruction $(\alpha_i, R_{ij}, \alpha_j)$ by the instruction $(\alpha_i, \hat{R}_{ij}, \alpha_j)$, where \hat{R}_{ij} results from R_{ij} - informally speaking - by replacing all a $\theta 2^{-1}$ by $a*2^{-1}$. Now, for any number $\beta \in \langle -1, 1 \rangle$ whose standard 2's complement representation is

$$\beta = -\beta_0 + \sum_{i=1}^{\infty} \beta_i * 2^{-i}$$

let

$$t(\beta) = -\beta_0 + \sum_{i=1}^{24} \beta_i * 2^{-i} .$$

The function $t(\beta)$ effectuates, of course, the "truncation" necessary in order to store β in a register of 24 bits. The simulation relation between Π_1 and Π_2 is the function $T: D_2 \rightarrow D_1$ defined as follows:

$$\begin{aligned} T(a, d, q_0, \dots, q_{24}, i, z) = \\ = (t(a), d, q_0, \dots, q_{24}, i, z) \end{aligned}$$

Of course $D_1 \subseteq D_2$ and $T(v) = v$ for all $v \in D_1$. Now it is an easy task to check the following [see 7]:

$$TR_{ij} = \hat{R}_{ij}T \quad \text{for all } i, j \leq n \quad (6.2)$$

From (6.2) we infer immediately the equation $TRes_1(\alpha_1, \alpha_6) = Res_2(\alpha_1, \alpha_6)T$ which applied to (6.1) results

$$Res_1(\alpha_1, \alpha_6) = \left[\begin{array}{l} a := t(q*d) - 2*0vf(q*d) \\ i := 0 \\ q_{24} := 0 \\ z := 0vf(q*d) \end{array} \right] \quad (6.3)$$

This equation says that the original microprogram Π_1 always terminates ((6.3) implies that $Res_1(\alpha_1, \alpha_6)$ is a total function) and that it produces the 24-bit representation of the required product. This representation can happen to be modified by an overflow in which case the value of z will become 1 (if $q = d = -1$, then $z = 1$ & $a = -1$).

The final remarks are the following. This method: -is different from the regular expression method, -permits the structuring of analyzed programs, -applies in exactly the same way to programs where the number of iterations in their loops depends on the input data, -may be automated using simple symbol-manipulation programs.

REFERENCES

- [1] J.W. de Bakker and W.P. de Roever, A Calculus for recursive program schemes, in *Automata Languages and Programming* (M. Nivat, ed.) pp.167-196, North Holland, Amsterdam, 1973.
- [2] A. Birman, On proving correctness of microprograms, *IBM J. Res. Develop.*, vol.18, pp.250-266, May 1974.
- [3] A. Blikle, Iterative systems; an algebraic approach, *Bull. Acad. Polon. Sci., Ser. Sci. Math. Astronom. Phys.*, pp.51-55, vol.20, 1972.
- [4] A. Blikle, Complex iterative systems, *ibid.*, pp.57-61, vol.20, 1972.
- [5] A. Blikle, Proving programs by δ -relations, in *Formalization of Semantics of Programming Languages and Writing of Compilers* (Proc. Symp. Frankfurt am Oder 1974), *Elektronische Informationsverarbeitung und Kybernetik* (to appear in 1976).
- [6] A. Blikle and S. Budkowski, A general program-verification method applied to microprograms, a short paper in Proc. of the 1976 Int. Conf. on Fault-Tolerant Computing (FTCS-6) Pittsburgh, 1976.
- [7] A. Blikle and S. Budkowski, An algebraic program verification method applied to microprograms, Res. Rep. CS-76-31 Dept. of Comp. Sci., Univ. of Waterloo, Waterloo Ontario, Canada N2L 3G1, June 1976.
- [8] A. Blikle and A. Mazurkiewicz, An algebraic approach to the theory of programs, algorithms, languages and recursiveness, in *Mathematical Foundations of Computer Science* (Proc. Symp. Warsaw-Jablonna 1972) Warsaw, 1972.
- [9] W.C. Carter, W.H. Joyner and G.B. Leeman, Automated experiments in validating microprograms, Digest of Papers of The 1975 Int. Conf. on Fault-Tolerant Computing (FTCS-5), p.247, Paris, 1975.
- [10] T. Ito, A theory of formal microprograms, in Proc. of the Int. Adv. Sum. Inst. on Microprogramming, 1971.
- [11] G.B. Leeman, Some problems in certifying microprograms, *IEEE Trans. on Comp.*, vol.C-24, No.5, pp.545-553, May 1975.
- [12] G.B. Leeman, W.C. Carter and A. Birman, Some techniques for microprogram validation, *Inf. Processing 74*, pp.76-80, North Holland 1974.
- [13] Z. Manna, *Mathematical Theory of Computation*, McGraw-Hill Book Co., New York, 1974.
- [14] A. Mazurkiewicz, Proving algorithms by tail functions, *Inf. Cont.*, vol.18, pp.220-226, 1971.
- [15] R. Milner, An algebraic definition of simulation between programs, 2nd Int. Joint Conf. Artificial Intelligence, London 1971, pp.481-489.
- [16] D. Paterson, The design of a system for the synthesis of correct microprograms, *Micro-8 Proc.*, Eighth Annual Workshop on Microprogramming, Chicago, Sept.1975.

MICROPROGRAMMED IMPLEMENTATION OF A SCHEDULER

R. Chattergy
University of Hawaii
Honolulu, Hawaii

Application of microprogramming to enhance the performance of operating systems has been discussed in the literature in the past [7,1,5]. Two examples of such applications can be found in [4,6]. This paper discusses the philosophy behind the microprogrammed implementation of a scheduler, used in a large, time-shared computer incorporating several processors.

1. INTRODUCTION

This paper describes the activities of a typical microprogrammed scheduler (microscheduler) in a time-shared system with multiple processors. This description is a simplified version of the actual microscheduler in the BCC 500 computer system, designed by W. Lichtenberger, M. Pirtle, B. Lampson, J. Freeman, R. Schultz and R. Van Tuyl in 1969. A functional diagram of the system is shown in figure 1. The general philosophy of scheduling for a multiprocessing system has been discussed at length in [3]. As mentioned in [3], scheduling consists of two activities. The first is the determination of an optimal schedule based on some scheduling criterion. The second is the enforcement of that schedule on the processes in the system.

Clearly the task of selecting a scheduling criterion and determining a schedule by some algorithm is a complex and evolutionary process. The environment within which resources are allocated by scheduling often changes, forcing a change of the scheduling criterion, and in extreme cases a change of the corresponding algorithm. Hence a scheduling algorithm is unsuitable for microprogrammed implementation in a read-only memory. On the other hand, the task of enforcing a schedule, classified as a mid-primitive in [7], consists of more permanent subtasks such as, the creation and maintenance of queues, blocking and awakening of active processes, making calls on the swapper etc. These subtasks are discussed in detail in the later sections. These subtasks are even system independent in the sense that, they must be carried out in one form or another regardless of the system architecture.

The above considerations prompted the design of the scheduler in the following form. The scheduler is

a hardware processor, microprogrammed to execute a set of instructions in a loop. The task of schedule enforcement is directly microprogrammed as part of this loop. Besides schedule enforcement, the processor also executes a machine instruction of an emulated machine in every iteration of the loop. The scheduling algorithm is implemented in software on this emulated machine. Thus the microscheduler carries out both scheduling activities, executing the scheduling algorithm written in a high-level language for flexibility, and enforcing the schedule discipline via firmware for speed of execution.

2. HARDWARE DESCRIPTION

The BCC 500 shown in figure 1, is a large time-shared computer with two processors for executing user-processes, and three special purpose processors for carrying out system management tasks (i.e., executing the operating system). All of the processors operate independently, communicate with each other via main memory, and are microprogrammed. Figure 2 shows the arithmetic-logic unit of a microprogrammed special purpose processor and its bus structure.

All registers shown in figure 2 are twenty four bits wide. M, Q, and Z are the main registers, where M serves as the communication register with the main memory via the main memory interface. The outputs of M and Q are connected to the left Boolbox (LB), and the outputs of Q and Z go into the right Boolbox. Each Boolbox can perform any of sixteen boolean operations on its inputs. The outputs of the Boolboxes are connected to the Adder. The output of the left Boolbox goes into the Cycler.

The outputs of the Adder and the Cycler can be put

into any of the seven Holding Registers, R0,...,R6. The register R0 acts as the memory address register (MAR) when main memory is accessed. The output of any of the Holding Registers can be incremented by one and hence any of these registers can be used as a counter. In addition there are sixty four Scratch Pad registers which are loaded from the X-bus and read onto the Y-bus.

The Control Memory of the microprocessor is a read-only, diode memory containing atmost 2 048, ninty-bit words. Different fields of the 90 bit micro-word control different logic circuits and in case of a branch to a subroutine, the return address is automatically stored in an auxilliary register.

3. MICROWORD

The bits and fields in the 90 bit words in the control memory are coded to generate the controlling signals necessary to operate the ALU. For example, bits 0-5 are used to set up one of a number of branch conditions to be tested for branching. The bits 8-17 are used to provide the branch address, which can also be obtained from the OS register (return from a subroutine) or the X-bus (computed go to). The bits 18-41 are used to specify a 24-bit constant which can be gated onto the X or the Y buses respectively. A detailed description of all the fields is too long. The above description should be enough to give the reader a "feel" for the system.

4. MICRO LANGUAGE

A special purpose readable reference language, called MICRO, is available for writing microprograms for the processors of the BCC 500. The MICRO language has declaration statements and statements for execution. The declaration statements can be used to define macros, give symbolic names to registers, define parameter values, define branch conditions for repeated use in the program, etc. The set of statements for execution consists of the usual assignment (including multiple) statement, memory operations statement, branch instructions, microword-field assignment statements, etc. It is impossible to discuss the language in detail here. Instead, explanatory comments enclosed between "/*" and "*/" are imbedded in the sample microprograms provided in the later sections.

5. MICROSCHEDULER INTERFACE

A simplified diagram of the interfaces among the system resources and the microscheduler is given in figure 3. In this figure, the microscheduler and the user processors are hardware processors, whereas the swapper and the scheduler are software packages run on the system processors. All processors in the system make WAKEUP calls to the microscheduler to activate processes. If a process, which has received a wakeup call, is not in the main memory, the microscheduler inserts the identity of this process into the input stack of the scheduler. The scheduler, using its scheduling algorithm, assigns a priority to the process which cannot be changed by the other processors. It puts the process in its appropriate position in a queue and makes a SWAPIN call to the swapper. In some cases such as a page-fault condition, the microscheduler can make a direct SWAPIN call to the swapper. Due to lack of adequate memory space, the swapper may fail to

swap in a process. It then makes a GIVEUP call to the microscheduler, asking for the identity of a process that may be swapped out to make room. The microscheduler answers this call via a SWAPOUT call indicating the process that can be swapped out.

The microscheduler alters the work schedules of the processors by making SWITCH calls. A switch call contains the identity of the process to be worked on by the receiving processor. If a new process of preemptive priority preempts the current process on a processor, this information is sent back to the microscheduler via a RETURN call by the processor. If a running process blocks itself, the corresponding processor makes a BLOCK call to the microscheduler. The processor informs the microscheduler whether or not the blocked process should be swapped out (a policy decision made by the monitor).

All communications with the microscheduler are carried out via a stack in the main memory under suitable PROTECT and UNPROTECT mechanisms.

6. LIFE-CYCLE OF AN ACTIVE PROCESS

Figure 4 shows the life-cycle of an active process under control of the microscheduler. Consider an active process which receives a call from some other running process. The call is entered under protection into the top two words of the input stack of the microscheduler. The microscheduler periodically inspects the stack for calls from the outside. Upon finding such a call, the microscheduler checks the identity of the process for validity. If the identity is invalid, it ignores the call and deletes the entry. For a valid call, the microscheduler determines whether the call is for a wakeup or block.

WAKEUP CALL: The microscheduler merges the data word from the call into the program interrupt word of the process (PIW), stored in the process resident table. It checks to see if the process is either waiting in the microscheduler queue for a processor, or already running. In either case, nothing more needs to be done. For an interesting example of this situation see [2] pp. 271.

On the other hand, if the process is blocked, the microscheduler unblocks the process. It checks to see if the process is in the main memory. If the process is in the main memory, the microscheduler inserts it, according to it's priority, in a queue of processes waiting for processors. Note that if the inserted process has preemptive priority then it can preempt a running process. This means that the microscheduler may have to reallocate the processors. A preemptive priority structure is necessary because the system does not have a hard-wired interrupt mechanism. Preemptive priorities must be assigned to processes whose non-execution can lead to loss of information.

If the process is not in the main memory, it has to be swapped in. The microscheduler then puts the process in a stack of processes waiting for the scheduler. The scheduler determines the priorities of the processes independently, and inserts them in the input queue of the swapper. In some cases, the microscheduler may make a direct request for a

swapi to the swapper.

BLOCK CALL: Whenever a running process blocks, the monitor is activated. The monitor decides whether the blocked process should remain in main memory (page-fault) or be swapped out (input from terminal). This decision is passed onto the microscheduler via the block call. The microscheduler blocks the process and if so directed makes a swap-out call to the swapper.

If a process is caught in a timer-trap, the microscheduler does not block it but puts it on the input stack of the scheduler for future scheduling. The scheduler changes the priority of such a process based on its scheduling criterion and sends a wakeup.

RETURN CALL: Whenever a running process is preempted of its processor by a process with preemptive priority, the processor sends a return call to the microscheduler. The microscheduler removes the process from the run state and puts it in the microscheduler queue to wait for a processor.

7. PROCESSOR SCHEDULING

The microscheduler periodically checks the status of each processor and reallocates those processors which are either idle or can be preempted. The processors are directed to switch processes by means of the SWITCH call sent by the microscheduler. In principle, the SWITCH call provides the processor the identity of the new process to be run.

A processor has three possible states. It is either idle, or running a process, or running a process which has preempted another process. If the processor is in the last mentioned state, then the microscheduler does not send it a switch call until the process running on it blocks. A processor is switched only if it is idle or running a process which has not preempted another process.

Whenever the microscheduler enters a new process in its queue that has a preemptive priority, it sets up a schedule flag. This flag indicates that reallocation of the processors is necessary. When the microprocessor decides to reallocate the processors, it switches the highest priority process in the microscheduler queue with a process that has blocked.

8. MICROSCHEDULER INPUT STACK

Calls to the microscheduler are placed on a stack called USIB in the microprogram. Each call consists of two words. The leftmost six bits of the first word contains an opcode identifying the call, such as 1 for wakeup, 2 for block etc. The rightmost eighteen bits of the first word contains a pointer to the first word of a process's process resident table (in effect identifies the process). The second word contains the bits to be set by the microscheduler in the process interrupt word in the resident table, as a result of the call.

9. MICROPROGRAM FOR MANAGEMENT OF USIB

```
USIBIGIN: PROTECT (USIB);
/* Protects stack from use by other processors...*/
MAR ← USIBTOP, FETCH;
/* Get pointer to top of stack.....*/
Q ← SK7 ← M, MAR ← USIBASE, FETCH;
/* Get pointer to the bottom of stack.....*/
M EOR Q, GO TO EMPTY IF LB=0, Z←LUSIBE
/* Compare top and bottom pointers stored in M */
/* and Q by exclusive OR. If pointers same, out-*/
/* put of left boolbox LB=0. Branch to block */
/* labelled EMPTY. LUSIBE=No. of words in call..*/
MAR ← SK7, FETCH, Z ← Q-Z;
/* Get first word from stack.....*/
SK7 ← Q ← M, MAR ← MAR+1, FETCH;
/* Get second word from stack.....*/
R2 ← M;
/* Put second word in register R2.....*/
M ← Z, MAR ← USIBTOP, STORE;
/* Move pointer to top of stack down by LUSIBE */
/* words.....*/
UNPROTECT (USIB);
/* Unprotect stack. The first word fetched from */
/* stack is in SK7 and Q. The second word is in */
/* register R2.....*/
M ← Q LCY 4, Q ← 600 000 17B;
M ← M AND Q LCY 2, CALL UERROR IF LB=0;
/* Left cycling the contents of Q through M mask-*/
/* ed by 600 000 017 and the last AND operation */
/* leaves the opcode for the microscheduler in */
/* the rightmost bits of M. For a valid opcode */
/* this must be > 0. UERROR subroutine is called*/
/* otherwise.....*/
Q ← MAXOP;
/* Maximum value of opcode is loaded in Q.....*/
CALL UERROR ON Q-M < 0. Q ← OPTAB-1;
/* Call UERROR if the opcode exceeds its maximum */
/* allowable value.....*/
R5 ← M+Q, Q ← R2, DGO TO USIBIGIN;
/* R5 stores the pointer to the subroutine (wake-*/
/* up, block etc.) to be used by the microschedu-*/
/* ler as a result of this call. The subroutine */
/* is called in the next line. Q and R2 contains*/
/* the second word of the call. DGO TO is a de-*/
/* layed branch. The branch is executed after */
/* execution of the next instruction is complete.*/
MAR ← Z ← M ← SK7, CALL STKLK, .C ← 3, .TCX, .TCW;
/* STKLK causes a branch to the subroutine point-*/
/* ed at by R5. It also saves the return address*/
/* in a stack. Because of the delayed GO TO in */
/* the previous line, this return address is that*/
/* of USIBIGIN. Thus a return is made to USIBI- */
/* GIN after a subroutine such as block or wake- */
/* up has been executed. MAR contains the ad- */
/* dress of the PRT plus 3 (ie. the address of */
/* the PIW), where the 3 is merged from the con- */
/* stant field of the microword by TCX and TCW...*/
OPTAB: GO TO WAKEUP;
GO TO BLOCK;
GO TO BLOCKOUT;
GO TO GIVEUP;
/* End of microprogram for the management of USIB*/
```

10. FLOW-CHART OF THE MICROSCHEDULER

A complete description of all the microprograms is too long to be included in this paper. A flow-chart describing the operation of a simple micro-scheduler is given below.

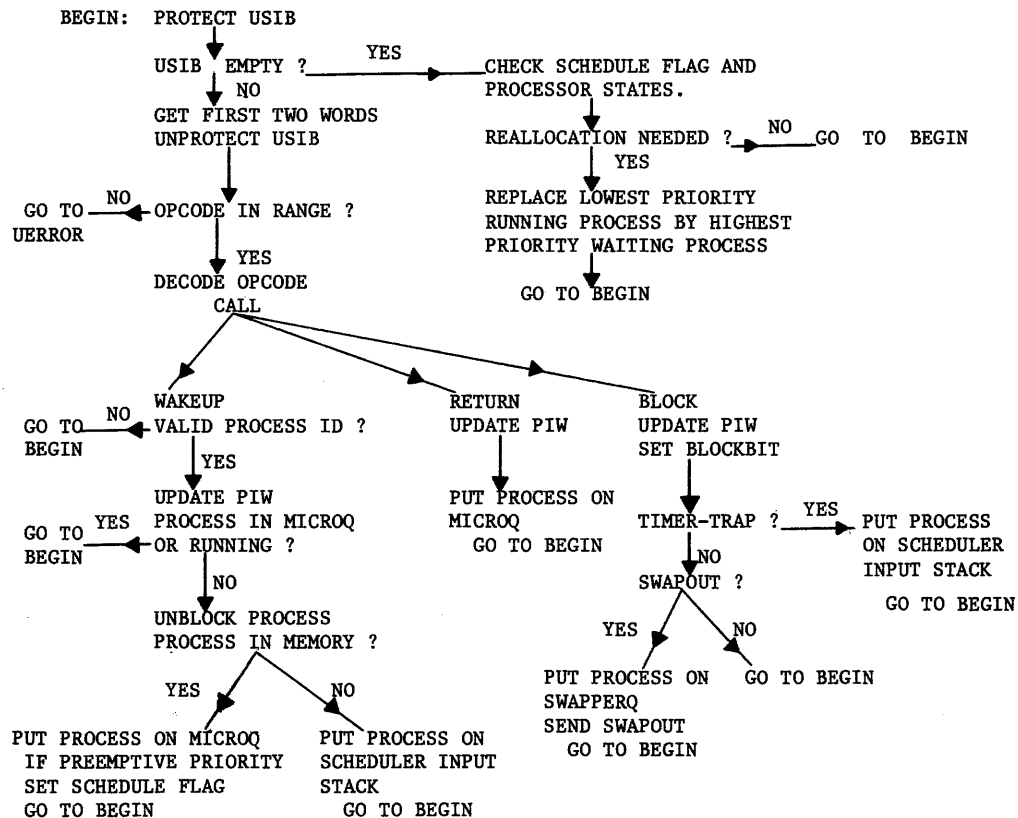
In the flow-chart, the usual housekeeping operations have been left out. Also the flow-chart does not include such operations as the management of real-time queues, which a microscheduler of a time-shared system must handle. A real-time queue is a queue of processes whose wakeup signals are specified by a real-time clock, and does not come from other processes. Basically, the microscheduler inspects its input stack periodically, and in response to calls left there by other processors it executes proper subroutines such as WAKEUP or BLOCK. It also checks the schedule flag and the states of the processors. Whenever necessary, it reallocates the processors and continues to loop around.

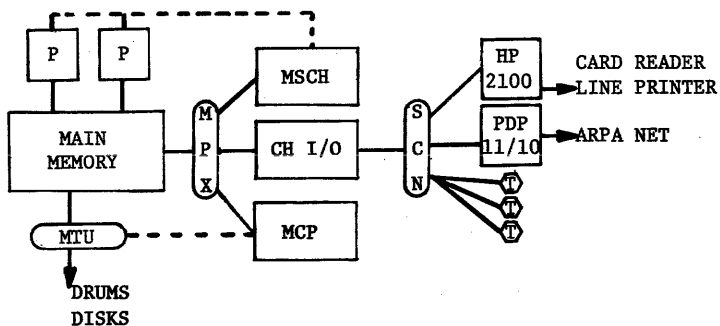
ACKNOWLEDGMENT

The author gratefully acknowledges encouragement and constructive criticism from Professor Wayne Lichtenberger of the department of Electrical Engineering, University of Hawaii.

REFERENCES

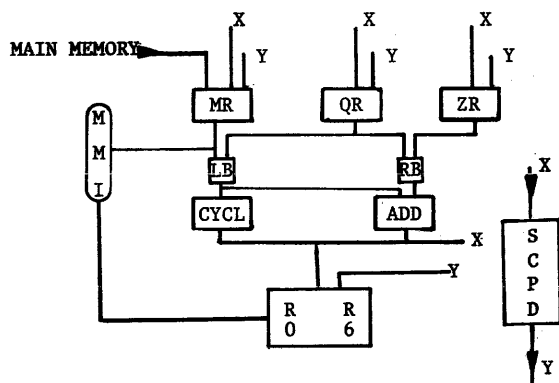
- [1] W. H. Burkhardt, R. C. Randel, Design of operating systems with micro-programmed implementation, NTIS Report, PB-224-484, September, 1973.
- [2] R. M. Graham, Principles of system programming, John Wiley, 1975.
- [3] B. W. Lampson, A scheduling philosophy for multi-processing systems, Communications of the ACM, vol. 11, No. 5, May, 1968.
- [4] B. H. Liskov, The design of the Venus operating system, Communications of the ACM vol. 15, No. 3, March, 1972.
- [5] J. V. Sell, Microprogramming in an integrated hardware/software system, Computer Design, vol. 14, No. 1, January, 1975.
- [6] W. G. Sitton, L. L. Wear, A virtual memory system for the Hewlett-Packard 2100A, preprints of the seventh annual Workshop on Microprogramming, ACM, September, 1974.
- [7] A. H. Werkheiser, Microprogrammed operating systems, preprints of the third annual Workshop on Microprogramming, ACM, October, 1970.





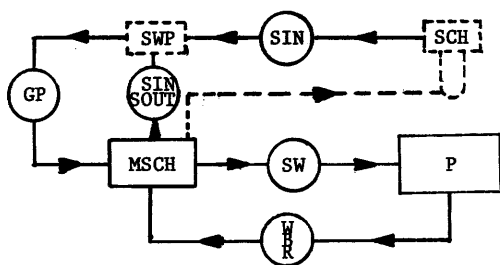
P - User Processor
 MSCH - Microscheduler
 CH I/O - Character I/O Processor
 MCP - Memory Control Processor
 MTU - Memory Transfer Unit
 MPX - Multiplexer
 SCN - Scanner
 T - Terminal

FIGURE 1



MR - M Register
 QR - Q Register
 ZR - Z Register
 MMI - Main Memory Interface
 SCPD - Scratch Pad

FIGURE 2



SWP - Swapper
 SCH - Scheduler
 SIN - Swap In
 SOUT - Swap Out
 GP - Giveup
 SW - Switch
 W - Wakeup
 B - Block
 R - Return

FIGURE 3

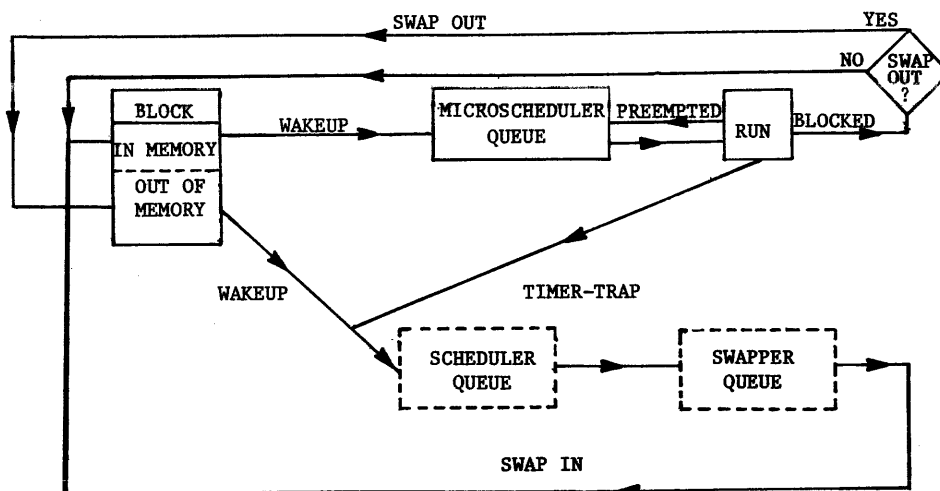


FIGURE 4

AN INSIGHT INTO PDP-11 EMULATION*

J. C. Demco
T. A. Marsland

Department of Computing Science
University of Alberta
Edmonton, Alberta
Canada T6G 2H1

In order to evaluate the Nanodata QM-1 as a universal host computer, an emulator for a contemporary computer, the PDP-11, was designed and constructed. It was required that the emulator be functionally equivalent to the target, without making excessive sacrifices in emulation speed. Some properties of emulation hardware necessary to achieve these goals are identified. In addition, the paper describes a monitor designed to support different emulators concurrently on a single host machine.

1. INTRODUCTION

Through the design and construction of an emulator for the DEC PDP-11/10 computer [1], the extent to which the Nanodata QM-1 [2, 3, 4] can serve as a universal host is being explored. The results summarized in this paper [5] identify some desirable properties of emulation hardware and show that complete emulation is possible without excessive sacrifices in emulation speed. One primary goal was that the emulator should execute all of the software for the target machine, rather than some specific package.

Many other computer emulations have either been for outdated machines with long memory access times, simple instruction formats and limited I/O capabilities [6] [7], or have not simulated I/O instructions exactly, but simply translated them into high-level requests to the host machine's operating system [8] [9]. In contrast, the study reported here considers the emulation of a contemporary computer, one with several instruction formats, addressing modes, and a main memory cycle time comparable to that of the host machine (fig. 1).

2. THE EMULATOR

With the QM-1, two distinct construction approaches are possible:

- Design a special microinstruction set, and implement it in nanocode. The emulator may now be built rapidly as a collection of microprograms.

- Implement the emulator's instruction set completely in nanocode; This is referred to as direct emulation, and should provide faster execution. Our emulator is essentially a direct one. Control store is used to hold tables for instruction decoding, a condition code bit map, and microroutines (written in the MULTI [10] instruction set) to handle I/O and control functions. These device drivers make the host peripherals serve the emulator as their target counterparts.

	Host QM-1	Target PDP-11
REGISTERS		
general purpose	32 18-bit	8 16-bit
special purpose	12 6-bit	device regs
	32 18-bit	(variable #)
	20 6-bit	
MAIN STORE		
width	18 bits	16 bits
maximum size	256K	28K
cycle time	960 nsec	980 nsec
CONTROL STORE		
width	18 bits	
maximum size	16K	
cycle time	160 nsec	
NANOSTORE		
width	360 bits	
maximum size	1K	
cycle time	160 nsec	

Fig. 1 Properties of User-Accessible Memory

*This study was supported by the National Research Council of Canada, Grant A7902.

Naturally the microroutines themselves are driven by nanocode, but speed in processing the I/O is of lesser consequence.

For each main store instruction, the high order nine bits of a PDP-11 word are used as an index into one of the control store tables. Each entry has two fields. The first is typically the nanoaddress of a setup routine, whose purpose is to prepare source and destination values. The second usually addresses an execute routine, which carries out the desired calculation. If an I/O device register is accessed during instruction execution, control is passed to a microroutine to initiate the I/O, before the execution of the next instruction.

2.1 Instruction Flow and Routine Descriptions

Fig. 2 is a block diagram showing the basic flow as the emulator executes PDP-11 instructions. The multi-way branch after the instruction FETCH routine reflects the execution of one of the setup routines mentioned in the previous section. The

boxes labeled SINGLE OP and DOUBLE OP indicate the use of one of the execute routines. The dotted lines represent conditional invocation of CALL, whose function is to pass control to a microroutine to handle the I/O, plus the HALT, WAIT, and RESET instructions. A summary of each nanoroutine follows:

- FETCH
Fetches the next instruction from main store and begins decoding it via a setup routine.
- MODE
A subroutine which calculates the effective address of a PDP-11 instruction operand, and returns its value. MODE is not shown in fig. 2 because it is called from many places. Error checking is performed, with control conditionally passed to SCHANGE.

- SCHANGE
The state-change routine is used to perform TRAP, EMT, BPT, and ICT instructions; it also handles I/O traps and error traps.

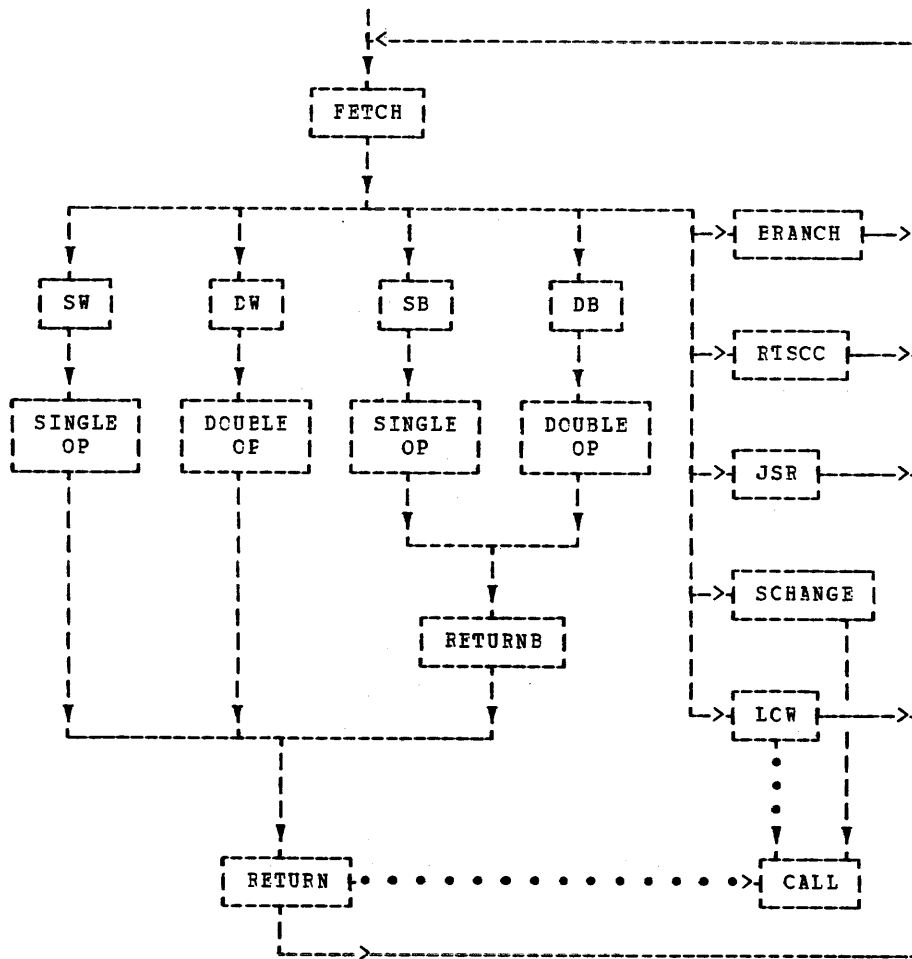


Fig. 2 Basic Flow of Control in the Emulator

• CALL

The invocation of micrsubroutines to handle I/O, HALT, RESET, WAIT, and logical interrupts is made by this routine.

• RTSCC

Handles the group of instructions in the range 0002XX-0003XX.

• LOW

Instructions in the range 0000XX - 0001XX are the responsibility of LOW.

• RETURNB

Re-constructs a word after a byte operation.

• RETURN

Places the result in the location specified by the DST field. If a device word was either read or written during the instruction execution, control passes to CALL otherwise, control passes to FFTCB.

2.2 Memory Mapping and Utilization

The host memory requirements for each of its three address spaces are summarized in fig. 3. Since the target has a 16-bit word and the host main store word is 18 bits wide, two bits are available as tags. These tags are used to differentiate the target

machine's device registers from the remaining existent and non-existent memory.

- Bit 17 is 1 if the word does not exist: referencing this location will cause a trap via SCHANGE.
- Bit 16 is 1 if the word is a device register, in which case RETURN will pass control to CALL to perform the I/O function.

The Rotate-Mask-Index (RMI) unit is valuable here for determining the tag settings, but is not essential.

2.3 Implementation Shortcomings

Although the emulator successfully executes standard instruction diagnostics, memory and disk exercisers, and also Version 8.08 of the DCS-11 operating system, it does contain some deficiencies as the following details show:

- Odd PC values are ignored; in fact, no PC checking is done at all. The extra time and space required here is probably not justified.
- Stack overflow checking is incomplete for JMP and JSR instructions. Also, the change-of-state routine will not detect stack overflow. Correcting this inaccuracy is almost impossible, given the present structure of the emulator. Elimination of these shortcomings does not

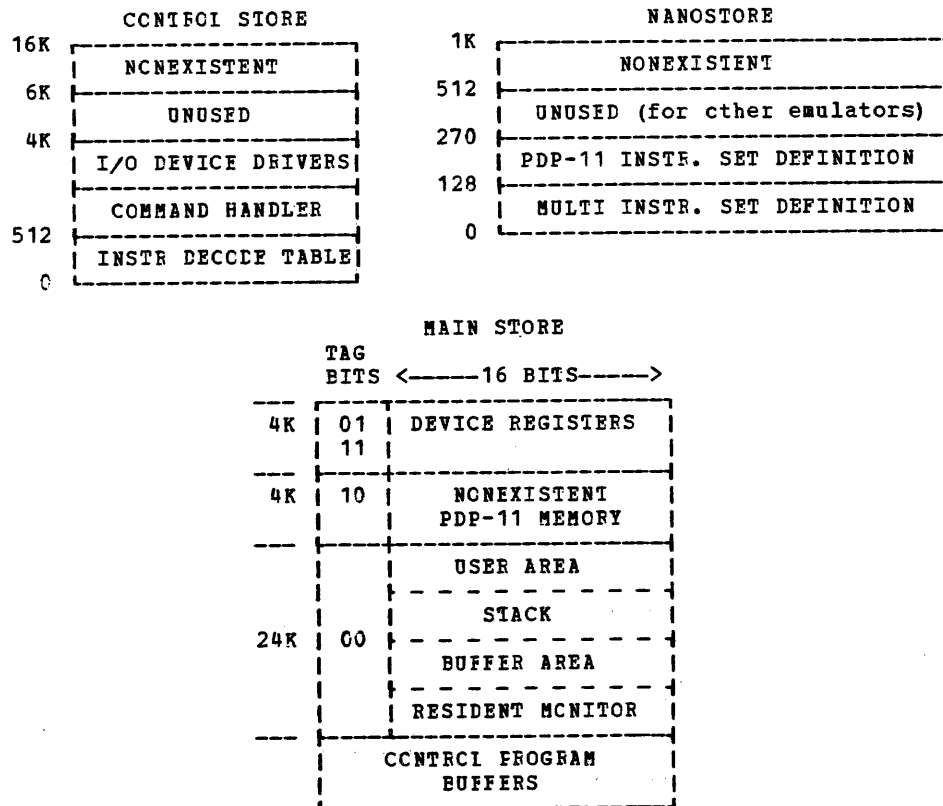


Fig. 3 Memory Allocation in the Host

warrant the excessive space overhead in the nanoprograms. In practice odd PC values should not arise, and stack overflow is detected in a real PDP-11 only after the user's buffers have been overwritten (fig. 3). In our case, the stack overflow problem stems from the fact that main store is not accessed through a common nanoprogram. The problem is a result of the limited subroutine nesting capabilities within nanostore and means that this emulator cannot be easily extended to handle the PDP-11/45 memory segmentation unit, because a common memory accessing mechanism is essential to deal with address translation.

Finally, the trace trap debugging feature was not implemented. It can be added to the microcode without modification to the nanoprogram portion of the emulator. No need was seen for the feature in our environment, and its implementation offered no new insight into emulation.

3. THE EMULATOR CONTROL PROGRAM

Several objectives guide the design of the Emulator Control Program (ECP). First, the control program must not impose architectural restrictions on an emulator. In fact, it should be possible to write an emulator without knowledge of ECP, and then to interface the two easily. Second, provision should be made for the concurrent support of different emulators [11]. Third, member emulators must be given low-level access to I/O devices through ECP and the emulator's device drivers. This is required in order that member emulators can be functionally equivalent to their counterparts, to the point of being able to transport existing software unchanged from the existing computer to the emulator. It was our hope that this approach to I/O handling would also spur thought in such areas as dynamic device ownership, fundamental differences between computer emulators and high-level language emulators, and even the question of whether or not a computer should know how to perform low-level I/O! Finally, basic control and debug facilities must be provided.

The design and implementation of ECP was heavily influenced by a similar system called CONTRCL [12], used by Nanodata Corporation to manage its Nova emulator.

3.1 Emulator - ECP Interaction

When the target accesses a device register, control is passed to the ECP. A table of address pairs is searched to match the device register address; the second address is the entry point into the device handler. Also passed to the I/O routine is a read/write flag. This process could occur twice in one main store instruction, if both source and destination are device registers. The device handler responds to

the request by accessing the I/O devices directly. At present, PDP-11 devices [13] which have been implemented include the simulation of an LA30 by a Tektronix 4023 CRT terminal, a PC11 paper tape reader (which is fed information via a Documentation-600 card reader!), an IP11 line printer, RK05 disks, and the KW11L line clock.

ECP's interrupt handling mechanism has two stages. Interrupts are caught first by the low level handler (with interrupts masked for only a short period), which then places the device's Unit Control Block (UCB) into a priority-ordered queue (fig. 4). A flag is set to signal a logical interrupt, and interrupt processing is completed. When the emulator is restarted, this flag is interrogated by the instruction fetch routine, and control is conditionally passed to the second stage. Once the logical interrupt routine gains control, a

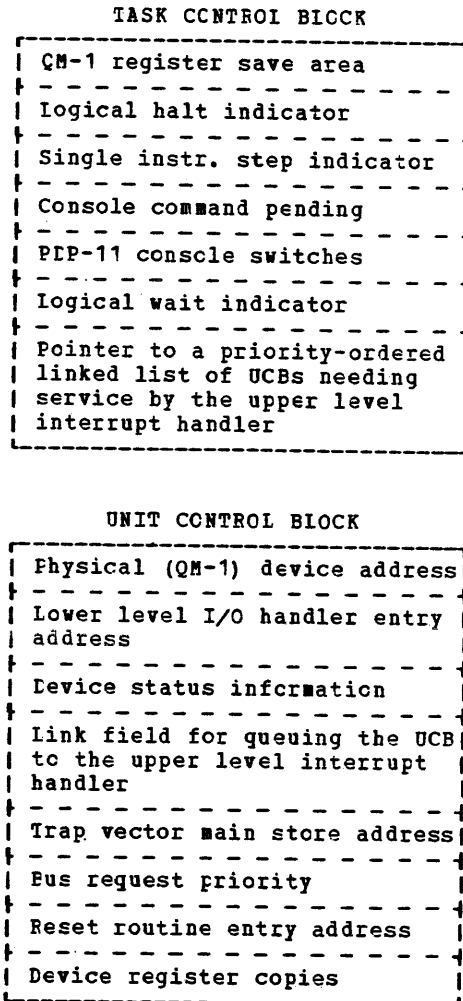


Fig. 4 ECP Control Block Structure (single emulator environment)

check is made first of the "command pending" word in the Task Control Block (TCB), and if necessary the command handler is invoked. The queue of UCB's is then inspected. If the CPU priority is lower than the bus request priority of the first UCB on the queue, a change of state in the emulator is forced via CHANGE. Otherwise, the emulator is restarted at the point of interruption.

This bi-level structure has two important features:

- No restrictions are placed on the way interrupts are handled by a member emulator's device drivers. This is especially important if virtual machines with different interrupt structures are to be supported concurrently.
- The handling of an interrupt by a device driver is completely independent of any emulator's software interrupt handler. In particular, loss of an interrupt from a device owned by an emulator which is not currently running does not occur

In a system with a small number of terminal devices, it may be desirable to have an emulator console double as the system console. This is easily done by providing a simple mechanism called console redirection to "point" keyboard interrupts at the appropriate handler, either the emulator's or the system's. When the primary console is "owned" by the emulator, receipt of a special control prefix passes the ownership to the ECP. Commands may then be executed, even while the emulator is active. A command is provided to return console ownership to the emulator. Similarly, more than one emulator may use the same console device.

One of the instructions which the control program has to handle is RESET, which invokes routines to re-initialize the devices that the emulator currently owns. Other obvious problems are for example emulating the HALT instruction, which should not stop the QM-1 (especially in a multi-emulator environment). Similarly the WAIT instruction cannot be dealt with by simply having an interruptible loop in nanocode, since the next interrupt need not necessarily come from a device which the emulator owns. Rather, WAIT sets the logical wait indicator in the TCB and decrements the PC so that the instruction is re-executed.

4. UNIVERSAL HOST EVALUATION

A number of architectural characteristics of the PDP-11 and of the QM-1 affect the emulation of the former by the latter. An investigation into those components of architecture which are appropriate for general purpose emulation has been made [14]; the statements following may also be extended to emulation in general.

Ideally a host machine should have significantly more registers than the

target. Of course, emulation of machines with a great many more registers, or registers wider than 18 bits, can be handled by maintaining them in control store. In our case the PDP-11 registers were easily accommodated in local store.

Fortunately, host main store is 2 bits wider than target memory, so data transfer is simplified; these extra bits are used as tags [15] to specify the existence or purpose of each word of the virtual machine memory (fig. 3).

The PDP-11 emulator is fairly fast, executing instructions at better than one-half the speed of a Model 11, approaching the physical limits imposed by the main store. The simpler instructions, and instructions using the simpler addressing modes, are relatively slower because of the rather long (2.5 microsecond) fetch and decode routine. On the other hand, use of the optional RMI unit to extract the 3-bit subfields in the operands reduces the instruction decode times by about 0.4 microseconds per operand.

4.1 Emulation Problems

The large number of buses and the presence of residual control in the host enhance its capabilities for parallelism. This is especially important in instruction fetch and decode, which is usually the most complicated part of instruction execution. Parallelism, however, is not sufficiently great for the PDP-11 emulation to check stack overflow concurrently with effective address calculation.

The QM-1 does not have an elementary N-way branch capability, and only one level of subroutine nesting is readily available at the nanoprogram level.

To emulate the PDP-11 efficiently, operations on various data widths are required. For example, the PDP-11 has byte operations which cannot be handled directly by the QM-1's shifter and ALU. There is no difficulty with arithmetic operations on bytes, since these are stored in the upper part of the word, but shifts and rotates require proper insertion of the carry bit - an operation which is awkward to perform. A desirable feature for a universal host machine is a truly variable-width arithmetic and shifting capability, including correct generation of conditions such as carry out and overflow. However, an extremely complicated (and almost unusable) structure might result. For example, the PDP-11 includes the carry bit in its shifts; the IBM 360 does not.

Condition codes generated by the host's hardware must undergo a non-trivial mapping to convert them to virtual machine condition codes. A powerful single-bit capability is required here, which the host does not have (table lookup is employed in the emulator). A similar problem exists whenever a signed conditional branch is processed, in order to take overflow into

account.

The difference in unit of memory addressability between host and target forces a good deal of time- and resource-consuming housekeeping on the emulator. A PDP-11 address must be shifted right by one bit to produce the corresponding QM-1 address. In the case of byte operations, the low-order bit of the PDP-11 address is used as a byte selector, and the byte which is not affected by the operation must be saved before the operation is carried out and restored after its completion. An efficient variable-width memory access capability would be an asset to a universal host machine.

4.2 Observations

The QM-1 was more than capable of hosting the complete emulation of a fairly complex machine, the PDP-11. Since the host could do many difficult things easily, we were perhaps overcritical whenever some features were awkward to implement. Nevertheless, the final emulation speed was within a factor of two of the target machine, comparing favorably with simulation, where a reduction factor of thirty is more realistic. In addition, the QM-1 supports a variety of other emulators: the Nova 1200, IBM 7094 and S/360, plus a number of lesser known machines.

In the area of multiple emulation, the common device drivers and the bi-level interrupt structure of the ECP provide uniform access to shared peripherals, without the need to inhibit interrupts for long periods of time. Work is continuing on that topic, and also on the design of universal file systems to simplify concurrent emulator support.

ACKNOWLEDGMENT

The repeated discussions with Steven Sutphen, regarding the hardware features of the PDP-11 and QM-1 computers, are much appreciated and helped reduce the inaccuracies in this study.

REFERENCES

- [1] Digital Equipment Corporation, PDP11 Processor Handbook, Maynard, Mass., 1975.
- [2] Nanodata Corporation, QM-1 Hardware Level User's Manual, 2nd ed., Williamsville, New York, 1974.
- [3] R. Rosin, G. Frieder, and R. Eckhouse, An Environment for Research in Microprogramming and Emulation, Communications of the ACM, vol. 15, no. 8, August, 1973, 748-760.
- [4] A.K. Agrawala and T.G. Rauscher, Foundations of Microprogramming, Academic Press, New York, 1976.
- [5] T. A. Marsland and J. C. Demco, A Contemporary Computer Emulation, Technical Report 76-1, Department of

Computing Science, University of Alberta, Edmonton, Alberta, February, 1976.

- [6] I. Schoen and M. Belsole, A Burroughs 220 Emulator for the IBM 360/25, IEEE Transactions on Computers, vol. C-20, no. 7, July, 1971, 795-798.
- [7] R. Benjamin, The Spectra 70/45 Emulator for the RCA 301, Communications of the ACM, vol. 8, no. 12, December, 1965, 748-752.
- [8] G. Allred, System/370 integrated emulation under OS and DCS, AFIPS Conference Proceedings, vol. 38, 1971, 163-168.
- [9] S. Tucker, Emulation of Large Systems, Communications of the ACM, vol. 8, no. 12, December, 1965, 753-761.
- [10] Nanodata Corporation, MULTI Multiprogramming Support System, Williamsville, New York, August, 1973.
- [11] J. C. Demco, Principles of Multiple Concurrent Computer Emulation, M.Sc. Thesis, Department of Computing Science, University of Alberta, Edmonton, Alberta, August, 1975.
- [12] Nanodata Corporation, CONTROL - Micro-System Control Program, Williamsville, New York, August, 1973.
- [13] Digital Equipment Corporation, PDP11 Peripherals Handbook, Maynard, Mass., 1975.
- [14] S. Fuller, V. Lesser, C. Bell, and C. Kawan, Microprogramming and its Relationship to Emulation and Technology, Seventh Workshop on Microprogramming, (Preprints), September, 1974, 151-158.
- [15] E. Feustel, On the Advantages of Tagged Architecture, IEEE Transactions on Computers, vol. C-22, no. 7, July, 1973, 644-656.

APPENDIX I

INSTRUCTION TIMING

The instruction execution times of the PDP-11 emulator and of the PDP-11/10 [1, Appendix E] are compared in the tables below. All timing information is in microseconds, unless otherwise noted.

SOURCE AND DESTINATION ADDRESS TIMES

Mode	PDP-11 SRC Time ¹	PDP-11 DST Time ²	Emulator SRC, DST Time
0	0.0	0.0	0.16
1	0.9	2.4	3.92
2	0.9	2.4	4.40 ³
3	2.4	3.4	4.96
4	0.9	2.4	4.72 ⁴
5	2.4	3.4	4.96
6	2.4	3.4	4.40
7	3.4	4.7	5.28

- ¹ - For SRC Time, add 1.3 usec for Odd Byte addressing.
- ² - For DST Time, and Odd Byte addressing:
 1. add 1.3 usec for a ncn-modifying instruction (CMPB, BITB, TSTB).
 2. add 2.4 usec for a modifying instruction.
- ³ - If register is 6 or 7, subtract 0.08 usec. If increment is 1, add 0.08 usec.
- ⁴ - If register is 6 or 7, subtract 0.08 usec. If decrement is 1, subtract 0.08 usec.

BASIC TIME

Single Operand

Instr Time = Basic + DST

Instr.	PDP-11 Basic Time	Emulator Basic Time ¹
CLR	3.4	4.88
COM	3.4	4.88
INC	3.4	4.96
DEC	3.4	4.96
NEG	3.4	5.12
ASR	3.4	5.92 ²
ASL	3.4	6.00 ²
ROR	3.4	6.00 ²
ROL	3.4	6.00 ²
ADC	3.4	5.12
SBC	3.4	5.28
TST	2.2	4.88
SWAB	4.3	6.16

- ¹ - If Byte instruction, add 0.80 usec for odd address, 0.72 usec for even address.
- ² - If Byte instruction, add 0.80 usec.

Double Operand

Instr Time = Basic + SRC + DST

Instr.	PDP-11 Basic Time	Emulator Basic Time ¹
ADD	3.7	5.12
SUB	3.7	5.60
BIC	3.7	5.12
BIS	3.7	5.12
CMP	2.5	5.36
BIT	2.5	5.20
MOV	3.7 ²	5.12 ³

- ¹ - If Byte instruction, add 1.44 usec.
- ² - 3.1 usec if Word instruction and Mode 0.
- ³ - If Byte instruction and DST Mode is 0, add 0.24 usec.

Branch Instructions

Instr.	PDP-11 ¹ branch	Emulator branch	Emulator no branch
BGE	2.5	3.20 ²	2.72 ²
BLT	2.5	3.20 ²	2.72 ²
BGT	2.5	3.68 ²	3.20 ²
BLE	2.5	3.60 ²	3.12 ²
BR	2.5	2.64	---
others	2.5	3.12	2.64

- ¹ - Subtract 0.6 usec if no branch.
- ² - Depending on N and V settings, add 0.0 to 0.48 usec.

Jump Instructions

Instr Time = Basic + DST

Instr.	PDP-11 Basic Time	Emulator Basic Time
JMP	1.0	3.52
JSR	3.8	3.76

Control, Trap, and Miscellaneous Instructions

Instr.	PDP-11 Instr Time	Emulator Instr Time
RTS	3.8	4.96
RTI	4.4	6.56 ¹
CLR CC	2.5	4.56
SET CC	2.5	4.56
HALT	1.8	3.92 ¹
WAIT	1.8	4.32 ¹
RESET	100 msec	4.88 ¹
ENT	8.2	8.32 ¹
TRAP	8.2	8.32 ¹
EPT	8.2	9.36 ¹
IOT	8.2	9.36 ¹

- ¹ - Then invoke micro routine.

DESIGN PROBLEMS IN EMULATING THE MIX COMPUTER ON THE MICRODATA 1600*

T. Don Dennis and O. G. Johnson
 Department of Computer Science, University of Houston
 Houston, Texas 77004

This paper presents an overview of an emulator for the MIX computer written in Microdata 1600 microcode. The MIX computer thus emulated is a variant of the original MIX computer as described in Volume 1 of The Art of Computer Programming by Donald Knuth. Basic changes involve the utilization of 8 bit bytes along with the ASCII character code.

1. INTRODUCTION

The MIX computer [5] is widely simulated for instructional purposes by departments of computer science. Hence, it is felt that there should be broad interest in the existence of MIX firmware. In this paper, we report the availability of a MIX emulator for a Microdata 1600 with 32K and AROM

The limited micro capabilities of the 1600 along with the architectural differences of the two machines created considerable micro program design problems. The identification of these problems, the solution techniques, and the subsequent creation of a new generation MIX computer constitute the content of this paper.

Only major design problems are considered here. Detailed descriptions of the design along with flowcharts, listings and user's manual are available in [8]. The second section discusses memory allocation. The third considers the layout and management of file registers and the last section discusses the overall firmware design.

2. MEMORY ALLOCATION

The first design decision was how to emulate the memory architecture of the MIX computer. The memory resources available on the Microdata 1600/30 and memory requirements of the MIX machine are reflected in figure 1. With 32K of core memory the Microdata 1600/30 is larger than the MIX 1009. Hence, one might think that emulation of the entire 4K words of MIX memory would cause no real problems. There were problems, however. The following three questions indicate the difficulties considered.

1. How many Microdata bytes should be used to emulate each MIX word?
2. How many bits should be used in each byte?
3. How should any extra Microdata memory be used?

	Microdata 1600/30	MIX 1009
Words of Memory	undefined byte addressable	4,000
Bits/Byte	8	1/sign 6/data
Total No. Of Bytes	32,728	4,000 sign 20,000 data
Total No. of Bits	261,824	124,000
Character Code	ASCII or EBCDIC	Knuth's Code
Numeric Code	binary 2's compliment	binary sign plus magnitude

Fig. 1 - Microdata-MIX Hardware Comparison

In examining the first of these problems, either five bytes per MIX word or six bytes per MIX word might seem to be the best solution. The five bytes per MIX word solution would require packing the sign and the first MIX data byte together. This packing would result in more available MIX memory but would inhibit uniform handling of all five data bytes. Uniform handling and the ability to generalize the firmware for partial word operations should be a primary consideration, however. Therefore, the six-bytes-per-MIX-word solution was selected in which the sign byte and 5 data bytes would each be assigned to a separate Microdata byte.

*This work was supported by National Science Foundation Grant DCR-74-1782

There are three disadvantages of this approach:

1. Not all 4000 words of MIX memory can be emulated on a 16K Microdata.
2. Address translation from MIX address to Microdata address and vice versa would be time consuming.
3. Detection of MIX word boundaries would be difficult given only the corresponding Microdata address.

The use of 32K Microdata memory solves problem 1. In fact, 8,728 bytes of Microdata memory would be still available at six bytes per MIX word. Problem 2 can be solved with a small amount of microcode; however, address translation is still time consuming. Problem 3, however, is the most difficult. Recall that MIX Input/Output instructions handle the sign byte of each MIX word differently from the data bytes. On Input the sign bytes are set positive and on Output the sign bytes are ignored. Figure 2 illustrates the Input operation.

S	1 st	2 nd	3 rd	4 th	5 th	
O	C	O	N	T	E	MIX word 0000
1	N	T	S		P	MIX word 0001
1	R	I	O	R		MIX word 0002
0	T	O		I	N	MIX word 0003
1	P	U	T			MIX word 0004

Before Read

	S	1 st	2 nd	3 rd	4 th	5 th	
MIX word 0000	0	A	B	C	D	E	
MIX word 0001	0	F	G	H	I	J	
MIX word 0002	0	K	L	M	N	O	
MIX word 0003	0	P	Q	R	S	T	
MIX word 0004	0	U	V	W	X	Y	

After Read

Read 80 characters into MIX memory starting at MIX word 000.

In 0,(10)

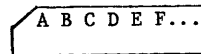


Fig. 2 - MIX Input Operation

Input/Output on the host machine, however, occurs one byte at a time, and the microprogram must use Microdata addresses to store each data byte; thus the firmware must be able to sense MIX word boundaries. One way to do this is to divide each Microdata address by six. If the remainder is zero then this byte corresponds to a MIX sign byte and could be handled accordingly. Another possible solution is to assign a counter to each I/O device that is set to zero when an I/O operation is initiated on that device; then each time a data byte transmission occurs this counter is tested to see if it is equal to zero. If not, the data transmission would take place and the counter would be decremented. But if the counter

is zero the Microdata address corresponds to a MIX sign byte and this byte should be either zeroed or ignored, depending on whether the operation involved is Input or Output. The counter would then be set to 5 and normal handling of data could resume. Either of these two methods, dividing by six or running a special counter would solve the problem of MIX word boundary detection but neither is easily accomplished.

Having tentatively adopted the six-byte-per-MIX-word solution, the next decision concerned how many bits should be used in each MIX byte. Knuth specifies each MIX byte should hold at least 64 values, but at most 100 values. This range allows MIX to be implemented as either a binary or a decimal machine. This implies that any binary implementation would have to use 6 bit data bytes. However, the Microdata accepts 8 bit operands and produces an 8 bit result plus a high order carry to be used as a link bit in multiple byte operations. If MIX were to be emulated with a 6 bit byte then the following format of figure 3 would result.

S	N	N	N	N	N	N	N
N	N	1 ₁	1 ₂	1 ₃	1 ₄	1 ₅	1 ₆
N	N	2 ₁	2 ₂	2 ₃	2 ₄	2 ₅	2 ₆
N	N	3 ₁	3 ₂	3 ₃	3 ₄	3 ₅	3 ₆
N	N	4 ₁	4 ₂	4 ₃	4 ₄	4 ₅	4 ₆
N	N	5 ₁	5 ₂	5 ₃	5 ₄	5 ₅	5 ₆

S - Sign Bit
N - Not used
K_i - ith bit of Kth byte

Fig. 3 - 6 Bit MIX Format

This format complicates all arithmetic instructions in MIX. If this format is allowed, the existing Microdata hardware for doing arithmetic operations cannot be used as intended. Incrementing a two byte counter, a very common and usually very simple operation is now fairly involved. The hardware Link bit provided in the Microdata ALU cannot be used to indicate a carry, so firmware logic must be developed to handle the high order carry. Figure 4 shows one way of incrementing two 6 bit bytes on the Microdata ALU.

- * Assume P1 contains the 6 high order bits of the counter.
- * And P2 contains the 6 low order bits of the counter.
- * Also assume P1 and P2 are carried in the following form.

* 00111111	High order 2 bits - zero
* 00222222	
INC 2	Increment P2
TN 2,X'40'	Test for high order carry
JP RTN	No high order carry so continue
- * High order carry has occurred

LT X'3F'	Load mask into T register
AND 2,T	Clear carry from P2
INC 1	Increment P1
- * Now overflow is possible

TZ 1,X'40'	Test for overflow
JP OVERFL	Overflow has occurred
- * Return

Fig. 4 - Microcode for 2Byte/6Bit Incrementation

Using the Microdata's ALU as intended a two byte (8 bit) counter can be incremented as shown in figure 5.

```

* Assume P1 and P2 again contain the counter
  INC 2      Increment low order 8 bits
  ADD 1,L,C  Add Link bit to high order 8
            bits, set condition flags
  TZ 0,X'01' Test for overflow
  JP  OVERFL Overflow has occurred
*
  Return
  
```

Fig. 5 - 2 Byte/8 Bit Incrementation

If the 6 bit format doubles the number of instructions required to increment a two byte counter, it should be clear that involved instructions such as Multiply, Divide, Add, Subtract, Shift, Char, and Num would be considerably longer as well. Recall also that MIX, being a sign plus magnitude machine, already conflicts with the Microdata's ALU since it is a two's complement unit.

In light of these complications an 8 bit data byte was adopted for the emulation of MIX. The format is shown in figure 6.

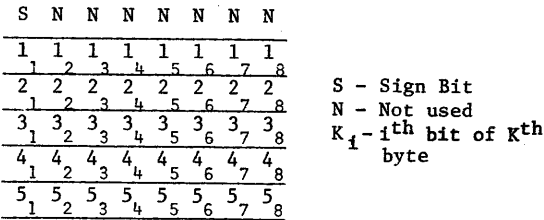


Fig. 6 - MIX 8 Bit Format

This eight bit data byte called for the adoption of another character code. Knuth's code, a 6 bit code, could have been used; however, it was felt that since the teletype, line printer, and disk worked in ASCII, it would be more advantageous to use ASCII than forcing Knuth's code onto these devices via firmware.

The remaining memory allocation problem concerned what to do with the 8,728 bytes of surplus Microdata memory. The two choices are obvious:

1. Extend MIX memory by 1,454 words.
2. Provide some sort of system support (temporary storage).

Solution 1 enhances the MIX machine from the user's point of view. However, MIX is supposed to have only 4,000 words of memory. In fact, if programs are to be written according to Knuth's rule "that no more than sixty-four values are ever assumed for a byte" [1], the largest memory location addressable is location 4,095 (2⁶-1). It would also seem that 4,000 words of memory is more than enough for educational purposes. Thus, adding more memory to the MIX machine offers no real advantage.

Solution 2 offers some advantages that are not obvious at first. All Input/Output in MIX takes place in concurrent mode. That is, an I/O operation is started via an IN or OUT instruction but a major portion of the I/O operation takes place

while the user executes other instructions. Certain information must be available to the microprogram in order to carry out these block data transfers. However, this information, memory address, and counters, must be stored somewhere besides the MIX registers available to the user. This surplus memory provides an ideal, and in fact the only place to temporarily store this type of information. A decision was also made, for reasons that will be discussed later, to keep MIX's index registers in main memory instead of the secondary files. Thus, with this type of privileged data in main memory, it is necessary to have an area of core that the MIX user cannot use. If certain MIX memory locations were dedicated to the above functions then the MIX user could alter concurrent I/O operations as well as the contents of the Index registers. It was thought that this was both dangerous and unnecessary for beginning programmers.

Having adopted solution 2, one more question arose; where should MIX memory begin and where should the surplus memory reside? Three possible answers were considered; the two which follow are obvious:

1. The surplus resides at Microdata address 0000-8727 and MIX memory resides at 8728-35,728.
2. MIX memory resides at 0000-24,000 and the surplus at 24,001-35,728.

The third possibility, and the one which was chosen, was to begin MIX memory at Microdata 0000 and then alternate one MIX word with two surplus bytes throughout Microdata memory. This did not affect the availability of the surplus memory but it did solve two problems previously discussed in this paper. Figure 7 illustrates the above solution.

Microdata Address	MIX Address	
00000	00	sign byte
00001		1 st data byte
00010		2 nd data byte
00011		3 rd data byte
00100		4 th data byte
00101		5 th data byte
00110		surplus
00111		surplus
01000	01	sign byte
01001		1 st data byte
01010		2 nd data byte
01011		3 rd data byte
01100		4 th data byte
01101		5 th data byte
01110		surplus
01111		surplus
10000	02	sign byte
10001		1 st data byte
10010		2 nd data byte

Fig. 7 - MIX Memory Assignment

Using this memory layout, the addressing problems of the 6 byte per MIX word solution were solved. Each MIX word is six Microdata bytes long but now each MIX word begins on a Microdata address which is a multiple of eight. Thus conversion from MIX address to Microdata address can be accomplished by shifting the MIX address three places left. Conversion from Microdata address involves shifting the Microdata address 3 places right. MIX word boundaries are also easy to sense. Any

Microdata address ending in 000 is a word boundary. Surplus bytes are also easy to detect since their addresses all end in either 110 or 111.

It should be noted that the surplus data bytes are invisible to the MIX user. This interweaving of MIX words and surplus data bytes also allows a minor extension of MIX memory from 4000 words to 4096 words. Some of these extra 96 words were dedicated for purposes not included in Knuth's specifications. For example, one word in high core is trapped by the microprogram if an illegal address, opcode or I/O device is encountered. Two bytes are dedicated to each I/O device, one to store the device status byte upon completion of an I/O operation and one as a trap address in case of an I/O error on that device.

3. FILE REGISTER ALLOCATION

The second design decision concerned the mapping of MIX's registers into the hardware available on the Microdata 1600/30. The registers which were to be emulated along with their lengths' are reflected in figure 8.

A Register	Sign plus 5 bytes
X Register	Sign plus 5 bytes
Instruction Register	Sign plus 5 bytes
Instruction Counter	2 bytes
Jump Save Register	2 bytes
Index Register 1	Sign plus 2 bytes
Index Register 2	Sign plus 2 bytes
Index Register 3	Sign plus 2 bytes
Index Register 4	Sign plus 2 bytes
Index Register 5	Sign plus 2 bytes
Index Register 6	Sign plus 2 bytes
Overflow and Comparison	1 byte

Fig. 8 - MIX Register Requirements

The Microdata provides 30 eight bit registers for the microprogrammer to use in emulating the registers of target machines. The idea of storing some of MIX's registers in main memory was also considered. It was decided that if simpler firmware logic would result from certain registers being stored in main memory then the speed gained through this simpler and therefore faster logic might easily make up for the time spent paging registers in and out of main memory. One should also note that the sum of the lengths of the registers listed in figure 8 is 41 bytes. Thus there were more MIX register bytes to emulate than there were file registers.

Initially it was decided to place the X register and the Index registers in main memory and to page them as required into the secondary file. Nine file registers (1-9) were reserved in the secondary file to hold the registers that were currently paged-in. A page map was to designate which registers were in memory and which were in the secondary files, as well as, which MIX register was in which set of Microdata files. Using this set-up, either the X register and one Index register, or up to 3 Index registers could be in the secondary files at any given time. The X register could fit into two possible slots, either register 1-6 or registers 3-9, and Index registers could fit into either registers 1-3, 4-6, or 7-9. This paging algorithm

plus the other five MIX registers consumed a total of 29 registers, leaving one register free.

Although this method did allow the emulation of all of MIX's registers and free work space could be created at almost any time by paging the registers in the secondary file out to memory, the overhead involved was considered very high. Instructions concerning the X registers were complicated since it could be in two possible positions. Index register routines were complicated since they could be in any one of three places in the secondary file. The accounting involved in keeping track of the current location of each MIX register file required three file registers and a considerable amount of AROM. Some sort of scheduling algorithm was also required to determine which register should be paged out in order to make room for the incoming register. It was thus decided that this strategy was too costly both in terms of firmware logic and file registers. The paging concept was then amended to allow only one Index register in the file registers at any given time. The X register would reside permanently in the secondary files. This simplified the X register instructions considerably. The Index register instructions were also simplified since now there were only six different Index register's instructions instead of thirty-six. For example, there are six Load Index instructions, one for each register; but all six are effectively represented by one load Index routine since all the Index registers are loaded into the same place in the secondary file. This routine calls the paging routine to page in the required Index register and then loads this register with the proper contents. The same is true for the Load Index negative, Store Index, Jump On Index, Enter Index and Compare Index instructions. The accounting problem associated with the paging system was also simplified. The page map was now 3 bits long--these bits contained the number of the Index register currently rolled in from memory, or the value zero if all the registers were currently rolled out. This ability to page all the Index registers out to memory provided an easy way to create free work files when the need arose. By allowing only one Index register in the secondary files at any one time, the need for a scheduling algorithm was eliminated. If Index register 1 is in the secondary files and Index register 2 is required, either for indexing or by an Index instruction, Index register 1 must be paged out and then Index register 2 paged in.

The detailed flow chart of the paging mechanism, called the Index Register Supervisor, can be seen in chapter V and microcode for the routine is found in chapter VI of [8].

Having solved the problem of too-many-MIX-registers-and-not-enough-files, the allocation of MIX registers to Microdata files was begun. The A register, the Instruction register and the Instruction Counter were assigned to the Primary file while the X register, Overflow and Comparison Indicators, Index registers, Index Map and Jump register were assigned to the Secondary file.

The Instruction Counter and Instruction register were placed together to facilitate the fetch routine. The A register was placed in the same file with the Instruction register, since the A register is the register most likely to be involved in the

next instruction fetched. Although only one free work file remained in the Primary file, the three files in the Instruction Counter containing the Operation code, the Index register and the sign of the Memory Address normally become available following the execution of the Instruction Decode Routine. These four free work areas in the Primary file are available in most cases.

The remaining MIX registers, the X register, the home position for the Index registers, the Index Map, the Overflow and Comparison indicators and the Jump Save register were grouped together out of necessity since the secondary file was the only place left to put them.

Figure 9 illustrates the file mapping that was finally selected. The A register was allocated Primary files P1, P2, P3, P4, P5, and P6. The X register was allocated ten matching registers in the Secondary file. This alignment simplified the different A and X instructions in much the same way that paging simplified the Index instructions. The only difference between a Load A and a Load X instruction is the periodic selection of the Secondary files instead of the Primary files.

		E T I I Z N O				P S O			
A Register	sign	P1	sign	S1	X Register				
	1st byte	P2	1st byte	S2					
	2nd byte	P3	2nd byte	S3					
	3rd byte	P4	3rd byte	S4					
	4th byte	P5	4th byte	S5					
Free work Register	5th byte	P6	5th byte	S6					
		P7	OLEG x iii	S7	Overflow, Comp., & Index Map				
Instruction Register	sign	P8	sign	S8					
	A1 address-ho	P9	1st byte	S9	Index Register i				
	A2 address-lo	P10	2nd byte	S10					
	I index spec.	P11		S11	Free work Registers				
	F field spec.	P12		S12					
C op code	P13		S13						
Instruction Counter	1st byte-ho	P14	1st byte-ho	S14	Jump Register				
	2nd byte-lo	P15	2nd byte-lo	S15					

Fig. 9 - MIX File Allocation

The Instruction register was placed in P8-P13 with P7 being the one free work file. This placed P7 next to the sign byte of the instruction address, which is one of the first files in the instruction register to become free. The Instruction counter was then assigned to P14 and P15, the remaining Primary files.

The Overflow and Comparison Indicators, a 4 bit register, and the Index Map, a 3 bit register, were combined into Secondary file S7. The home position for the Index registers was assigned to S8, S9, and S10. The Jump Save register was aligned with the Instruction Counter in S14 and S15. This left S11, S12, and S13 as free work registers.

It should be noted that eleven file registers can be freed after instruction decode if they are needed. P7, S11, S12, and S13 are always free. P8, P11, and P13 are free after instruction decode. S8, S9, and S10 can be freed by paging the current Index register out to memory after the effective

operand address has been computed. Finally, the seven bits of S7 can be packed into S1 with the sign of the X register if necessary, freeing S7.

The mapping allowed simplified coding of Index instructions of A and X instructions as well as ample work space and resulted in the successful emulation of the MIX 1009 computer.

4. FIRMWARE LOGIC DESIGN

Having defined the MIX Machine in terms of the Microdata's hardware, firmware design could begin. First a general overview of the system was composed. The following is an explanation of the overview as presented in figure 10.

Start Routine - performed following cold start and prior to the execution of any MIX instructions; enables external interrupts; enables the real time clock; initializes the teletype; loads the Instruction Counter from a dedicated high core address.

Fetch Routine - fetches the next instruction into the instruction register from the address contained in the Instruction Counter.

Addressing Routine - computes the effective operand address.

Decode Routine - examines the Instruction Operation Code and transfers control to the corresponding firmware instruction module.

Instruction Modules - firmware routines that execute the individual MIX instructions.

Interrupt Handler - executed after the execution of the last instruction and prior to fetching the next instruction; acknowledges and handles external interrupts from I/O devices; acknowledges and handles internal interrupts from the real time clock and console panel.

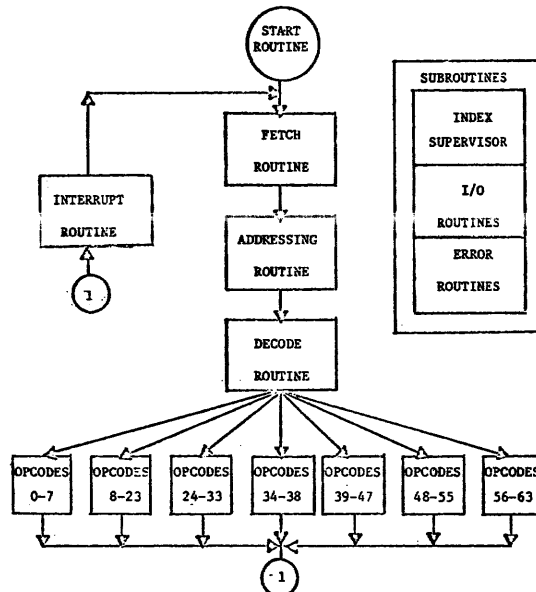


Fig. 10 - System Overview

Subroutine Package - Index Register Supervisor:

handles the paging of MIX Index registers; I/O routines: used by the Input/Output instructions as well as the interrupt handler; Error Routines: handles user errors such as illegal addresses, illegal opcodes, illegal I/O device numbers, and I/O errors.

An attempt was made to keep the MIX emulator as modular as possible. This facilitated program development and debugging as well as simplifying the decode routine. The decode routine divides the MIX instruction set into seven groups, each group representing a type of MIX instruction.

The seven Groups are:

1. Opcodes 0-7 Arithmetic-Logic instructions
2. Opcodes 8-23 Load instructions
3. Opcodes 24-33 Store instructions
4. Opcodes 34-38 Input/Output instructions
5. Opcodes 39-47 Jump instructions
6. Opcodes 48-55 Enter and Increment instructions
7. Opcodes 56-63 Compare Instructions

Detailed descriptions of each of these groups, along with flowcharts and microcode are given in [8]. The same reference includes a user's guide as well as a complete program listing. Parties interested in using the emulator should contact the authors. The code is public domain and the authors will distribute the program at cost.

REFERENCES

- [1] MIX, Publication number AL 1/73 03808, Reading Mass., Addison-Wesley Series in Computer Science and Information Processing, 1970.
- [2] Microdata, Computer Reference Manual, Microdata 1600/30, publication number RM 20001630-1, Irvine, Cal., Microdata Corporation, 1973.
- [3] Microdata, Micro 1600 Computer Reference Manual, publication number 71-1-1600-001, Irvine, Cal., Microdata Corporation, 1971.
- [4] Microprogramming Handbook, Second Edition, Irvine, Cal., Microdata Corporation, 1972.
- [5] D. E. Knuth, Fundamental Algorithms Vol. 1, The Art of Computer Programming, (R. S. Varga and M. A. Harrison, eds.), pp 120-153, Reading, Mass., Addison Wesley, 1969.
- [6] R. K. Richards, Digital Design, (R. K. Richard, ed.), pp. 341-368, New York, N.Y., 1971.
- [7] W. King and T. D. Dennis, A Paging System for the Control Memory in a Minicomputer System, COMPCON 75, Tenth IEEE Computer Society International Conference, San Francisco, Cal., Feb. 25-27, 1975.
- [8] T. D. Dennis, A Microprogrammed MIX 1009 Emulator for the Microdata 1600/30 Computer, Master's Thesis, Computer Science Department, University of Houston, Houston, Texas, 1975.

EXTENSIBILITY - A NEW APPROACH FOR DESIGNING
MACHINE INDEPENDENT MICROPROGRAMMING LANGUAGES

David J. DeWitt
Computer Sciences Department
University of Wisconsin-Madison

This paper describes a new technique for designing high level machine independent microprogramming languages. In Section 1.0 we will discuss some design considerations for microprogramming languages and will review the previous efforts in the area of microprogramming languages. In Section 2.0, we will discuss extensible languages - what they are and why they are useful. Then in Section 3.0, we will show why an extensible microprogramming language resolves most of the difficulties inherent in designing a language for microprogramming. This section will also include a description of the language EMPL - our extensible microprogramming language. We will define its syntax, give some examples of how its extensible features can be used and finally will demonstrate the feasibility of constructing such a compiler.

1.0 Design Considerations for Microprogramming Languages

In designing a microprogramming language there are four primary design goals to achieve:

- G1: The language must facilitate writing programs. It should make provisions for writing well structured programs and thus should include the appropriate control constructs (such as "IF THEN...ELSE..." statements and "DO WHILE" loops).
- G2: The language should be "readable" so as to facilitate the task of redesigning a program if that proves necessary at a later date.
- G3: The language should be machine independent so that programs written in it are portable.
- G4: It should be possible to compile the language into very efficient microcode* for a variety of microprogrammable computers.

These goals are not very different from the design objectives for any programming language except that the requirements for very efficient code production (objective G4) are generally more severe for microprogramming. This requirement results from the fact that control memories are usually implemented with "state of the art"

This work was done at the University of Michigan, Department of Computer, Information, and Control Engineering

*Efficient microcode is microcode which executes in a minimum amount of time.

technology so that they are as fast as possible. Thus, they are expensive and therefore are generally small (compared to main memories). Thus, "tight" microcode is needed so that a minimum number of words in the control memory are used. The second reason why efficiency is so important is that microcode is generally run very frequently. An order of magnitude difference between the execution frequency of an assembly language routine in an operating system and a microcoded routine in an emulator is not unrealistic. Therefore, saving just a couple of microinstructions here and there can result in a dramatic performance improvement at the assembly language level. A good example of where efficiency is obviously very important is in the sequence of microinstructions which interpret an assembly language statement on the IBM 360.

In order to motivate the purpose of our Extensible Microprogramming (EMPL), we will next examine some different levels of microprogramming support. We will also analyze how successfully each achieve the four design objectives we have presented.

1.1 Micro-Assemblers

Assemblers* translate symbolic instructions on a one to one basis into executable microinstructions. For microprogrammable computers with vertical microinstruction formats, the symbolic instructions look very similar to typical assembly language instructions. Examples of such assembly languages can be found for the Microdata 1600 in [MIC71], for the Burroughs B1726 in [DEW73], and for the Interdata 85 in [INT73]. For computers with horizontal microinstructions each symbolic instruction contains a specification of each microoperation that should be executed during the microinstruction. No concurrency analysis is done. Mirager [CLA72], an assembly language for the Argonne Micro-Processor*, is an example of such an assembly language.

Such assembly languages achieve only one of the desired goals G1 through G4. They fail to ease (to any significant degree) the burden of writing programs. Programs written in such assembly lan-

*Except for assemblers which provide MACRO capabilities.

*The term "microprocessor" as used here refers to a microprogrammable processor, not an LSI component.

guages certainly are not transportable and are generally not very readable since they are full of 'GO TOs'. The assembler can, however, produce efficient microcode but will do so only if the programmer writes an efficient program.

1.2 Tailored Languages

Tailored languages are microprogramming languages similar in design to PL/360 [WIR68]. The syntax of these languages is similar to the syntax of high level languages such as ALGOL or PL/1. Control constructs such as "IF THEN...ELSE...", "DO CASE", and "DO WHILE" are permitted. However, the programmer must have a working knowledge of the architecture of the target machine in order to write code in these languages. Generally symbolic variables are not permitted, so the programmer must keep track of which operands are in which registers and must make an explicit assignment between variables and physical memory locations. SIMPL [RAM73] (a single assignment microprogramming language), the Datsaab FCPU Microprogramming Language [BLO73], and MPL (a microprogramming language for the Microdata 32/S computer [MIC73]), are examples of tailored microprogramming languages.

Tailored languages partially succeed in facilitating the task of writing microprograms. While the program may be fairly readable, it is not portable at all. The code produced should be about as efficient as the code produced by an assembler for the same computer.

1.3 Machine Independent Languages

While the set of machine independent microprogramming languages could include a language like ALGOL, the compilation of such languages into microcode is a very complex process. Furthermore, languages like ALGOL require a complex runtime environment which would be very difficult to maintain at runtime by a microprogram. Therefore, a high level language for microprogramming, while supporting ALGOL like control constructs, simple data structures such as vectors, and arithmetic expressions, should not be expected to handle complex operations such as character string manipulations. Such a high level language, if the code produced is sufficiently efficient, would achieve all the goals G1 through G4.

There have not been many efforts to produce such a language. Ramamoorthy et. al., [RAM73] claim that SIMPL is such a language. Careful analysis of SIMPL reveals that it is a language tailored for the CDC 6600. MPL, a microprogramming language developed by Richard Eckhouse [ECK71] at SUNY/Buffalo is the best attempt to define a machine independent language. MPL is a procedure oriented language which has six basic data types:

1. machine registers - both real and virtual
2. memory - main and control memory
3. local and auxiliary storage
4. testable events - condition flags
5. constants
6. variables - which only take on constant values

MPL has two basic statement types:

1. Simple assignment statements, with operands in the above classes, which permit typical operations (e.g., arithmetic and logical)
2. "If" statements which permit branching on the value of some testable event.

Although Eckhouse claims that his language is machine independent, it is so in only a very limited sense. It is capable of describing and compiling microprograms for a wide variety of computers but since real machine registers show up as predefined variables, programs written in MPL are not transportable from machine to machine. Furthermore, MPL is not designed for taking advantage of the concurrency available on those target machines which can support parallel operations. The compiler for MPL does not attempt to detect parallel microoperations and so will fail to produce very efficient microcode for any machine which has horizontal microinstructions.

In conclusion, "high level" machine-independent microprogramming languages are still very much in the research and development phase. Existing languages such as PL/1 or ALGOL are too complex to be used for this purpose. MPL is not sufficient for the reasons which were described previously. Finally, any high level language constructed in a conventional manner, may be its very definition, be incapable of producing efficient microcode for a wide range of machines. Some machines support, in hardware, multiply operations. Some even have floating point hardware. Some have elaborate decode and branch microoperations. Any microprogramming language must be capable of taking advantage of each unusual facility on each machine if users are going to use this language as a tool to write their microcode. This diversity of available features makes a high level microprogramming language difficult to design because if the language is too complex (because it attempts to incorporate every imaginable feature) then it becomes too difficult to compile. If it is not able to handle floating point operations when the target machine has microoperations which manipulate floating point operands then the compiler is not going to be able to produce efficient microcode. These observations have motivated the development of an extensible microprogramming language. In the next section we will present an overview of extensible languages. Then, in Section 3.0, we will describe EMPL - an extensible microprogramming language.

2.0 Extensible Languages

The purpose of this section is to introduce the basic concepts of extensible languages. We will not describe the syntax of a particular extensible language such as ALGOL-68, but will instead describe the features that most extensible languages share. Furthermore, we will not concern ourselves with implementation details at this time. Rather, after we describe our extensible language EMPL, we will discuss how the extensible features of EMPL can be implemented.

In [CHE68], Cheatham presents two extreme design approaches for programming languages. They are referred to as the shell approach and the core

approach. The shell approach calls for implementing a universal programming language that includes every conceivable feature a user might ever want. PL/1 is an example of a shell language. A core language would be designed to include only very basic features but would permit the user to extend the language so that it includes those features that he desires. Thus, the user can customize the language so that it becomes appropriate for the type of problems he wishes to solve.

The basis of an extensible language is a concept which is known as a type (types are called classes in SIMULA67 [DAH70] and modes in ALGOL68 [WIJ74]). A type permits a user to define a new data structure and a class of operators which can operate on instances of this new data structure. Types, as envisioned by Campbell and Wyeth [CAM74], also can be used to provide a form of protection since the only operators allowed defined within the type declaration. Once a type has been defined, the user can, within the scope of the type definition, declare instances of that type and can include the new operations (as described in the type definition) in statements. Whereas the set of legal operators for common types such as integer and real are defined implicitly by the syntax of the core language, the set of legal operators for a particular type must be specified explicitly in the type definition. The definition of a Campbell and Wyeth type includes:

1. A template which specifies the internal structure of each instance of a type. This template consists of a set of objects, a collection of object operations which specify the ways in which the objects can be manipulated, and a set of private procedures. Each private procedure, which also manipulates objects, is executed for every instance of a type which is declared.
2. A set of type-objects, type-operations, and type-procedures. The type-objects are objects outside the template but inside the type definition. For each type definition there is only one instance of each type-object regardless of how many instances of the type there are. Type-operations are operations which manipulate type-objects. Type-procedures are procedures which are executed when an instance of a type is declared. Initializing the value of a type-object to zero might be the effect of executing a type-procedure.

The object operations and type-operations are the only elements of the type definition which can be accessed outside the scope of the definition.

3.0 An Extensible Microprogramming Language

3.1 Extensibility and the Design Objectives

While it should be apparent that it is not a difficult task to design a microprogramming language that is easy to use and is self-documenting, it is hard, however, to design a language that is both portable and capable of being translated into efficient microcode for a variety of target machines. We feel that the core approach to language design can be used to design a microprogramming language which will achieve all four of the design objectives stated in Section 1.0. We will state without argument that any well designed

high level language achieves the first two objectives - ease of programming and readability. What we must demonstrate is that such a language can be both portable and capable of being translated into efficient microcode.

Recall that the motivation for the core approach to language design is to implement a programming language which permits the user to extend the set of data structures and operations included in the core so as to customize the language for the class of problems he wishes to solve. A microprogramming language which is designed according to the core methodology will permit the programmer to customize the microprogramming language to fit the target machine. That is, if the target machine supports microoperations which manipulate floating point operands, the language can be extended so that the user can declare and manipulate floating point operands. Since the language has been customized to take advantage of the floating point capabilities of the target machine, the microcode produced should be more efficient than if the floating point operations were implemented using fixed point operations. Analogously, if the target machine has a stack, the programmer can extend the language by declaring a new type named STACK. Example 1.0 contains a type declaration for a stack. The objects comprising the stack include a vector of integers for the stack itself named STK and a pointer to the current top of the stack named STKPTR. The two object operations are PUSH and POP. PUSH adds a new element to the top of the stack. Thus, by extending the language to include a stack and the

```

type STACK
  template
    integer STK (16), STKPTR,LIMIT
    initially: STKPTR=0;
    operation PUSH (Accepts integer VALUE);
      IF STKPTR=16 then ERROR;
      ELSE DO;
        STKPTR = STKPTR+1;
        STK(STKPTR)=VALUE;
      END;
    operation POP (Returns integer VALUE);
      IF STKPTR=0 then ERROR;
      ELSE DO;
        VALUE=STK(STKPTR);
        STKPTR=STKPTR-1;
      END;
  ENDtemplate;
endtype;

```

Example 1.0

operations PUSH and POP, the user can express his algorithm using this new type and the microcode produced will take advantage of the stack which is available on the target machine.

It may be difficult to understand how portability can be maintained if the systems programmer is permitted to extend the language so that it takes advantage of the hardware features of a particular target machine. Assume that we have written a program for a target microprogrammable machine A which supports a hardware stack. Thus, at the beginning of the program we would include a type declaration for a stack similar to the one found in Example 1.0. When the program is compiled to run on Machine A, each occurrence of the object operation

PUSH will be compiled into one microoperation which will perform the PUSH microoperation. If at a later date we wanted to run the program on machine B, which did not support a hardware stack, we would set a toggle on a control card. This toggle would indicate that the stack should be maintained as any vector would be on the target machine and that each occurrence of PUSH is not compiled into a single microoperation but is instead compiled into the sequence of microoperations which corresponds to the compilation of the statements included in the declaration of the object-operation PUSH.

Thus, by setting a toggle for each type declared in a program, the program can be transported from one machine to any other machine as long as a compiler exists to compile the core language for the target machine. Therefore, the extensible approach to designing a microprogramming language can achieve all of the design objectives which were specified in Section 1.0.

One should, however, realize an important fact about such an extensible language - a separate compiler is needed for each target machine. Actually, as we will demonstrate in Section 3.3, the compilation process can be subdivided into a machine independent subprocess and a machine dependent subprocess. The lexical analysis and syntactical analysis processes are machine independent and do not need to be rewritten for each compiler. It is only the code generation process that is machine dependent and thus must be written separately for each target computer. The fact that the code generation process is machine dependent implies that the intermediate language statements are machine dependent. Thus, the output of the compiler is a sequence of intermediate language statements each of which corresponds to an executable microoperation (after register allocation is done) on the target machine.

3.2 EMPL - The First Extensible Microprogramming Language

We are now ready to define the syntax of our extensible microprogramming language. The syntax of EMPL is:

3.2.1 The Syntactic Description of EMPL

The BNF description of EMPL is as follows:

```

BNF of EMPL
<PROGRAM> ::= <PROGRAM HEAD> <STATEMENT LIST> EOF
<PROGRAM HEAD> ::= <BASIC STATEMENT>
                <PROGRAM HEAD> <BASIC STATE-
                MENT>
<BASIC STATEMENT> ::= <EXTENSION STATEMENT> ;
                    | <DECLARATION STATEMENT> ;
                    | <EXTENSION OPERATOR> ;
<STATEMENT LIST> ::= <STATEMENT
                    | <STATEMENT LIST> <STATEMENT>
<STATEMENT> ::= <STANDARD STATEMENT>
               | <EXT STATEMENT>
<EXT STATEMENT> ::= <IF STATEMENT>
                  | <RESTRICTED STATEMENT>

```

```

<STANDARD STATEMENT> ::= <CALL STATEMENT> ;
                    | <PROCEDURE DEFINITION> ;
                    | <RETURN STATEMENT>
                    | <LABEL> <STANDARD STATE-
                    MENT>
                    | <EXECUTE STATEMENT> ;
<RESTRICTED STATEMENT> ::= <ASSIGNMENT> ;
                    | <GROUP> ;
                    | <GO TO STATEMENT> ;
                    | <LABEL> <RESTRICTED
                    STATEMENT>
                    | ;
<IF STATEMENT> ::= <IF CLAUSE> <STANDARD STATE-
                    MENT>
                | <IF CLAUSE> <TRUE PART> <STAN-
                    DARD STATEMENT>
                | <LABEL> <IF STATEMENT>
<IF CLAUSE> ::= IF <LOGICAL EXPR> THEN
<TRUE PART> ::= <STANDARD STATEMENT> ELSE
<GROUP> ::= <GROUP HEAD> END
<GROUP HEAD> ::= DO ;
                | DO <WHILE CLAUSE> ;
                | <GROUP HEAD> <STATEMENT>
<WHILE CLAUSE> ::= WHILE <LOGICAL EXPR>
<PROCEDURE DEFINITION> ::= <PROCEDURE HEAD>
                        <STATEMENT LIST> END
<PROCEDURE HEAD> ::= <LABEL> PROCEDURE ;
<CALL STATEMENT> ::= CALL <IDENTIFIER>
<RETURN STATEMENT> ::= RETURN
<GO TO STATEMENT> ::= GO TO <IDENTIFIER>
<LABEL> ::= <IDENTIFIER> ;
<EXECUTE STATEMENT> ::= EXQ <OPERATION>
<OPERATION> ::= <VARIABLE>
                | <IDENTIFIER> <ARGUMENT LIST>
<ARGUMENT LIST> ::= <ARGUMENT HEAD> <IDENTIFIER> )
<ARGUMENT HEAD> ::= ( <IDENTIFIER> ,
                    | <ARGUMENT HEAD> <IDENTIFIER> ,
<ASSIGNMENT> ::= <VARIABLE> <REPLACE> <EXPRESSION>
<REPLACE> ::= =
<VARIABLE> ::= <IDENTIFIER>
                <SUBSCRIPT HEAD> <PRIMARY> )
<SUBSCRIPT HEAD ::= <IDENTIFIER> (
<PRIMARY> ::= <CONSTANT>
                <IDENTIFIER>
<EXPRESSION> ::= <TERM>
                | <TERM> <OP> <TERM>
                | <OPL> <TERM>
<OP> ::= + (two's complement)
        | - (two's complement)
        | *
        | /
        | &
        | |
        | @
        | ] &
        | ] |
        | ] @
        | RSHL
        | LSHL
        | ROTL
        | ROTR
        | <IDENTIFIER>
<OP> ::= [
        | -
        | <IDENTIFIER>
<TERM> ::= <VARIABLE>
        | <CONSTANT>
<LOGICAL EXPR> ::= <VARIABLE> <RELATION> <TERM>
<RELATION> ::= <|> | <|=| <|<|> | <|> | <|=|
<DECLARATION STATEMENT> ::= DECLARE <DECLARATION
                                ELEMENT>

```

```

<DECLARATION ELEMENT> ::= <IDENTIFIER> <TYPE>
                        | <BOUND HEAD> <CON-
                          STANT> ) <TYPE>

<TYPE> ::= FIXED
        | <IDENTIFIER>
<BOUND HEAD> ::= <IDENTIFIER> (
<EXTENSION STATEMENT> ::= <EXTENSION HEAD>
                          <OPERATOR LIST> END-
                          TYPE
<EXTENSION HEAD> ::= TYPE <IDENTIFIER> <TEM-
                     PLATE>
<TEMPLATE> ::= <DECLARATION LIST> <INITIAL BLOCK>
<DECLARATION LIST> ::= <DECLARATION STATEMENT> ;
                    | <DECLARATION LIST>
                      <DECLARATION STATE-
                        MENT> ;
<INITIAL BLOCK> ::= <INITIAL HEAD> END;
<INITIAL HEAD> ::= <INITIALLY DO;
                  | <INITIAL HEAD> <ASSIGNMENT> ;
<OPERATOR LIST> ::= <EXTENSION OPERATOR>
                  | <OPERATOR LIST> , <EXTENSION
                    OPERATOR>
<EXTENSION OPERATOR> ::= <OPHEAD> <OPBODY> END
<OPHEAD> ::= <OPERATOR NAME> <MICROOPSPECIFICA-
             TION> ;
            | <OPERATOR NAME> <PARAMETER LIST>
              <MICROOPSPECIFICATION> ;
<OPERATOR NAME> ::= <LABEL> <OPERATION>
<PARAMETER LIST> ::= <ACCEPTS LIST>
                    | <RETURNS LIST>
                    | <ACCEPTS LIST> <RETURNS
                      LIST>
<ACCEPTS LIST> ::= <ACCEPTS HEAD> <IDENTIFIER> )
<ACCEPTS HEAD> ::= <ACCEPTS> (
                  | <ACCEPTS HEAD> <IDENTIFIER> ,
<RETURNS LIST> ::= <RETURNS HEAD> <IDENTIFIER> )
<RETURNS HEAD> ::= RETURNS (
                 | <RETURNS HEAD> <IDENTIFIER> ,
<MICROOPSPECIFICATION> ::= <MICROOP HEAD>
                          <BLOCK INDEX>
                          <PROCESSOR INDEX>
<MICROOP HEAD> ::= MICROOP: <OPCODE>
<OPCODE> ::= <CHARACTER STRING>
<BLOCK INDEX> ::= <CONSTANT>
<PROCESSOR INDEX> ::= <CONSTANT>
<OPBODY> ::= <DECLARATION LIST>
           | <EXT STATEMENT>
           | <OPBODY> <EXT STATEMENT>

```

3.2.2 EMPL Data Types and Variables

The only data type defined in the EMPL core is integer. A variable may be declared as either a single integer or as a vector of integers. In order to eliminate the need for a complex run time environment, the scope of all variables in EMPL is global. That is, all declaration and extension statements must precede the first executable statement of the program. Furthermore, variables are not passed to procedures.

3.2.3 EMPL Extension Statements

EMPL provides two methods by which the programmer can extend the constructs defined in the core of EMPL. These methods are called extension statements and extension operators. Extension statements, like types, permit the user to define new data types and new object operations to manipulate the data type. Example 2.0 contains an EMPL description of a stack and the associated object operations PUSH and POP. Other than the obvious

syntax differences between Example 1.0 and Example 2.0, there are two important differences between types as defined by Campbell and Wyeth [CAM74] and types as we have defined them. The first difference is that EMPL types do not permit the definition of type-objects, type-operations, and private procedures to manipulate type-objects. While type objects can serve a useful function (e.g., keeping track of the number of different items in an inventory), it is not apparent that such a tool is valuable in a microprogramming language. Furthermore, since the purpose of the type feature in EMPL is to permit the user to customize the language to take advantage of the hardware features on the target machine and thus there is a corresponding microoperation for each object operation, there is not likely either a storage element which corresponds to the type-object or a microoperation which corresponds to the type-operation.

The second important difference between Campbell and Wyeth's types and EMPL types is that the definition of each object operation in an EMPL type declaration also specifies the microoperation which corresponds to the object operation. The microoperation specification includes three parts: the opcode, the block index, and the processor index. The function of the opcode is obvious. The block index, a component of the Control Word Model [DEW75], is used during the optimization of the intermediate code. The processor index specifies the ALU which executes the microoperation. A value of zero for the processor index (as in Example 2.0) specifies that no general purpose ALU is used. The specification of the operands which the microoperation will use is done implicitly by the compiler by examining the operands specified in the program statement which utilizes the object operation.

In order to complete this discussion of extension statements we need to discuss how instances of new types are declared and how object operations are specified in executable statements. We will postpone describing the specification of object operations until Section 3.2.4 where executable statements are discussed in detail.

Instances of new types are declared in EMPL just like fixed variables by a DECLARE statement. For example, in order to declare an instance of a data type named STACK (as defined in Example 2.0) the following declaration statement would be used.

```

TYPE STACK
  DECLARE STK(16) FIXED;
  DECLARE STKPTR FIXED;
  DECLARE VALUE FIXED;
  INITIALLY DO;
      STKPTR=0;
  END;
  PUSH: OPERATION ACCEPTS(VALUE)
        MICROOP: PUSH 3 0;
        IF STKPTR=16 THEN ERROR; /*OVERFLOW*/
        ELSE DO; /* PUSH VALUE */
            STKPTR=STKPTR+1;
            STK(STKPTR)=VALUE;
        END;
  END,
  POP: OPERATION RETURNS(VALUE)
       MICROOP: POP 3 0;
       IF STKPTR=0 THEN ERROR; /*underflow*/
       ELSE DO; /* POP VALUE */

```

```

        VALUE=STK(STKPTR);
        STKPTR=STKPTR-1;
    END;
END,
ENDTYPE:

```

EXAMPLE 2.0: EMPL Description of a Stack

```
DECLARE ADDRESS_STK STACK;
```

Thus for each new type defined, the programmer may declare instances of that type by specifying the type name instead of "FIXED" in a declaration statement.

The second method by which a programmer can extend the constructs provided in the EMPL language is with an extension operator. Extension operators are defined exactly as object-operations in type declarations (except that they are terminated with a semicolon instead of a comma) and are used to define new operators in the language that can be applied to variables of type FIXED. Extension operators are similar to high level language macros except for the fact that the definition of each extension operator includes the definition of a microoperation to perform the specified operation. Example 3.0 contains an EMPL description of an extension operator which is called EXTRACT. The purpose of the EXTRACT operation is to extract a sequence of bits from an operand starting at bit position SBIT and finishing at bit position SBIT+CNT-1. Both the Burroughs B-1700 and the Palyn Corporation's EMMY computer have a micro-operation which performs an extract operation. It is a very useful operation for decoding instructions.

The obvious syntactical differences between extension statements and extension operators is that extension statements provide a new data type; instances of which can be declared by using a DECLARE statement. There is,

```

/* EXTRACT OPERATION */
/* AN EXTENSION OPERATION WHICH EXTRACTS */
/* A SEQUENCE OF BITS FROM A VARIABLE */
/* STRING IS THE VARIABLE, SBIT THE */
/* STARTING BIT POSITION AND CNT IS THE */
/* NUMBER OF BITS TO BE EXTRACTED */

EXTRACT: OPERATION ACCEPTS(STRING,SBIT,CNT)
        RETURNS(EXSTR)
        MICROOP: EXTRACT 5 2;
DECLARE STRING FIXED; /* INPUT VARIABLE */
DECLARE SBIT FIXED: /* STARTING BIT POSITION */
DECLARE CNT FIXED; /* NUMBER OF BITS */
DECLARE EXSTR FIXED; /* RESULT */
DECLARE RSHAMNT FIXED; /* NUMBER OF PLACES TO SHIFT RIGHT */

EXSTR = STRING LSHL SBIT; /* SHIFT OFF THE LEFT
                          HANDBITS WHICH ARE
                          NOT WANTED.
                          SHIFT IN
                          ZEROS */

```

```

RSHAMNT = WORDLENGTH-CNT; /* CALCULATE THE NUMBER
OF POSITIONS TO BE SHIFTED
RIGHT. WORD-LENGTH IS A GLOBAL
RESERVED VARIABLE WHICH
INDICATES THE WORD LENGTH OF
THE TARGET MACHINE */

```

```
EXSTR = EXSTR RSHL RSHAMNT;
```

END;

EXAMPLE 3.0

however, an important difference in the realization of these two constructs. The definition of a new type implies that there exists a piece of hardware on the target machine which supports instances (at least one) of the new type. For example, the STACK defined in Example 2.0 could be realized in hardware on the target machine.

On the other hand, the declaration of a new extension operator does not necessarily imply the existence of some additional hardware to hold data items. Rather, it implies the existence of either an additional ALU to execute the new operator or it implies that an existing ALU can perform more than the standard arithmetic, logical, and shift operations. For example, on the B-1700 the extract operator is performed by a special ALU while on the EMMY machine it is performed by the same unit which executes the rest of the shift and rotate operations. Thus, by permitting the user to define different types of extensions to EMPL, he can describe two entirely different situations in the EMPL language.

3.2.4 EMPL Arithmetic, Logical and Shift Operations

EMPL supports only very simple assignment statements. Only one variable can be placed on the left hand side of the equal sign and only one operator and two operands can be placed on the right hand side. The arithmetic operators and logical operators are:

+, -, *, /, &, |, ⊕ (exclusive or)
~~⊗~~, ⊘, ⊙

When the operator specified is a shift or rotate operator, the left hand operand is the operand to be manipulated and the right hand operand specifies the shift/rotate amount. For example, in Example 3.0, the assignment statement
 EXSTR = STRING LSHL SBIT;
 does a logical left shift on the variable STRING and places the result in the variable EXSTR. The number of positions shifted are specified by the variable SBIT. The other shift and rotate operators are RSHL, ROTL (rotate left), and ROTR (rotate right).

It is also possible to use an operator defined in an extension operator statement as the operator in

an assignment statement if it uses only two source operands and produces only one result. Those extension operators which use more than two source operands or which produce more than one destination operand, must be specified in an EXECUTE statement. For example, to use the extract operator the programmer would specify

```
EXQ EXTRACT(IR,FBIT,BITCNT,OPCODE);
```

The compiler will associate the global variable IR with the variable STRING in the definition of the EXTRACT operation. FBIT will be associated with SBIT, BITCNT with CNT, and OPCODE with EXSTR.

In order to perform an object operation on an instance of a type which was declared in an extension statement, the programmer must use an EXECUTE statement. When an EXECUTE statement is used for this purpose, the first element of the argument list of the object operation identifies the instance of the type which is to be manipulated by the object operation. For example, assume that an instance of stack was declared in the program as:

```
DECLARE ADDRSTK STACK;
```

Then, in order to push a value of the operand RETURNADDR onto the stack ADDRSTK, the programmer would specify:

```
EXQ PUSH(ADDRSTK,RETURNADDR);
```

ADDRSTK specifies which instance of the type STACK is to be used and RETURNADDR is associated with the variable VALUE (see Example 2.0). The result of executing this statement would be to push the current value of RETURNADDR onto the stack ADDRSTK.

3.25 EMPL Control Constructs

EMPL's control constructs are also very simple. EMPL supports "IF THEN...ELSE..." statements for conditional operations and a "DO WHILE" construct for specifying loop operations. While a "GO TO" statement is included, its use is not encouraged.

The two remaining control constructs which are legal in EMPL are CALL and RETURN statements to transfer control to and from procedures. Since all variables (except those within extension statements and operators) have global scope, no declaration statements are permitted within procedures. Thus, there is no need to pass parameters between the calling statement and a procedure. Therefore, the syntax of the CALL statement does not permit the specification of an argument list.

In conclusion, the syntax of EMPL is very simple and straightforward. It was designed with the objective of maximizing the usefulness of the language, while minimizing the complexity of the compiler necessary to translate the source program into the intermediate language form. In the next section we will discuss some of the implementation details. However, since EMPL was designed to minimize the complexity of the run time structures necessary to support the compiled code, it is not difficult to construct a compiler for EMPL.

3.3 Construction of an EMPL Compiler

In this section we will discuss some of the construction details of a compiler to translate programs written in EMPL into machine dependent intermediate language statements. Since such a compiler has not yet been constructed we must pro-

vide enough details so that the reader is convinced of the feasibility of such an undertaking. Each intermediate language (I L) corresponds to an executable microoperation on the target machine.

Rather than discuss the structure of the complete compiler, we will focus our attention on the two aspects of the compilation process which are not generally found in typical compilers. These aspects are the generation of machine dependent IL statements from machine independent EMPL statements and the implementation of the extension statement and extension operator constructs. We are not proposing that one can construct one EMPL compiler that will translate EMPL statements into IL statements for all microprogrammable computers, but rather that it is possible to construct a machine independent skeleton for an EMPL compiler. For example, the XPL translator writing system provides a skeleton which can be used for writing compilers. Since XPL is meant to be a general purpose compiler compiler, the user needs to:

1. Define the BNF of the target language so that tables can be generated for the parser.
2. Write a scanner to perform the lexical analysis phase of the compilation process.
3. Construct a set of procedures for semantic analysis and code generation.

What we are proposing to do is to extend the idea of XPL by providing an EMPL skeleton which includes a scanner to do the lexical analysis phase of the compilation process - translating EMPL statements into a string of tokens. The EMPL skeleton will also include the tables for parsing EMPL. It will employ a mixed strategy parsing algorithm as in some versions of XPL [MCK70]. The third task, writing the procedures to do the semantic analysis and code generation processes, can also be partially included in the EMPL skeleton. By careful analysis, the productions can be divided into machine independent and machine dependent subsets. Since the machine independent procedures only need to be written once, they can be included in the EMPL skeleton. The machine dependent procedures, including all those for code generation and others which collect information for later code generation in the parsing process, need to be written for each target machine.

In the remainder of this section we will discuss the code generation procedures and the procedures which are necessary to implement the extensible features included in EMPL.

The complexity of the code generating procedures depends on the nature of the target machine. For example, take the EMPL multiplication operator (*). If the target machine does not support a hardware multiply, then the code generation procedure for the multiply operation must emit a sequence of intermediate language statements which implement, for example, an algorithm to perform the multiply in terms of shifts and adds. If the repertoire of microoperations on the target machine includes a multiply instruction then the procedure to generate code for an EMPL multiplication will be very simple; it may just take one intermediate language statement to implement the multiply.

The motivation for keeping the syntax of EMPL simple should now be apparent. By restricting the set of legal constructs we have greatly simplified the process of writing the machine dependent code generation procedures.

As mentioned earlier, there is a control toggle associated with each extension statement and each extension operator. During the process of code generation the compiler will have to process object operations and extensions operations. By examining the control toggle for the operation the compiler can decide what it should do. If the control toggle is set off, then the compiler knows that it should emit the microoperation associated with the definition of the object operation (in the type definition) or the extension operator. As an example let us return to the extract operation defined in Example 3.4. Assume that the extract operation is specified as:

```
EXQ EXTRACT (IR,FBIT,BITCNT,OPER);
```

Then if the control toggle associated with EXTRACT is off, the compiler would emit (by examining the definition of EXTRACT maintained in the symbol table):

```
EXTRACT 5 2 1 3 0 OPER IR FBIT BITCNT
```

This statement is a typical intermediate language statement, the general format of which is defined in [DEW76]. The statement can be interpreted as follows:

- i. EXTRACT is the opcode
- ii. 5 is the Block index
- iii. 2 is the processor index
- iv. there is 1 destination operand
- v. there are 3 source operands
- vi. there are no literal operands
- vii. OPER is the destination operand
- viii. IF,FBIT, and BITCOUNT are the source operands

The reader should note that no register allocation has been done. This fact also simplifies the code generation process since the operands of the intermediate language statements are the same as the operands in the EMPL statements.

On the other hand, if the control toggle associated with the extract operation had been on, then the compiler would not have emitted the above IL statement. Instead, the compiler would have used the definition of the extract operator and do an "macro" expansion. Thus, the EMPL statement:

```
EXQ EXTRACT(IR,FIBT,BITCNT,OPER);
```

would have been replaced with the following sequence of EMPL statements.

```
OPER = IR LSHL FBIT;
RSHAMNT = WORDLENGTH-BITCNT;
OPER = OPER RSHL RSHAMNT;
```

This sequence of code would then be compiled into a sequence of IL statements which could be executed on a target machine which did not have an extract operator.

Having discussed the implementation of extension operators we should next discuss how extension statements are implemented. As mentioned previously EMPL types do not permit the definition of type-objects, type-operations, or private procedures to manipulate type-objects. EMPL extension statements only allow the user to define data types. The implementation of object operations is handled exactly as extension operators are. Implementa-

tion of objects is different. If the control toggle associated with the extension statement is on, then the compiler does not have to do anything with each object specified in the type definition. However, if the control toggle is off, then the compiler for each instance declared must reserve a memory location for each object in the type definition. Then when an object operation is performed on an instance of a type, the compiler will substitute the address of the proper memory location as the operand.

4.0 Conclusions

In conclusion while such an EMPL compiler has not yet been constructed, we feel that there are no insurmountable obstacles to prevent the successful implementation of an EMPL skeleton. We have clearly demonstrated that an extensible microprogramming language can achieve each of the design objectives stated in Section 1.0. The intermediate phase of an EMPL compiler is a sequence of machine dependent intermediate language statements. Each intermediate language statement corresponds to an executable microoperation except that no register allocation has been done. This sequence of microoperations is then transformed into a sequence of microinstructions which are executable. This final phase of the compilation process involves the tasks of register and processor allocation. It is performed by a branch and bound algorithm which uses a Control Word Model [DEW75] description of the target machine to take advantage of whatever concurrency is permitted by the target machine. The sequence of microinstructions produced is optimized so that it will run in minimum time. This second phase has been completed and is described in [DEW76].

Acknowledgements

The author would like to express his appreciation to Professors W.E. Riddle and D.E. Atkins for their constructive and critical suggestions.

5.0 References

- [MIC71] Microdata Corporation, Microprogramming Handbook, second edition, 1971.
- [INT73] Interdata Corporation, Model 80 Microinstruction Reference Manual, March 1973.
- [DEW73] DeWitt, D. J. et. al., "A Microprogramming Language for the Burroughs B1726," MICRO6, September 1973.
- [CLA72] Clark, Robert, "Mirager, the 'Best-Yet' Approach for Horizontal Microprogramming," ACM National Conference, August 1972, Boston, Massachusetts.
- [WIR68] Wirth, N., "PL/360, A Programming Language for the 360 Computers," JACM VOL15-1, pp. 37-74, 1968.
- [BLO73] Blomberg, L., and Lawson, H., "The Data-saab FCPU Microprogramming Language," SIGMICRO/SIGPLAN Interface meeting, May 1973.
- [RAM73] Ramamoorthy, C. V., et. al., "A Higher Level Language for Microprogramming," MICRO6, September 1973.
- [MIC73] Microdata Corporation, Computer Programming Language MPL, 1973.
- [ECK71] Eckhouse, "A High Level Microprogram-

- ming Language (MPL)," AFIPS, Vol. 38, 1971.
- [CHE68] Cheatem, J. E., Fisher, A., and Jorrand, P., "On the Basis for ELF - An Extensible Language Facility," SJCC 1968.
- [DAH70] Dahl, O. B., and Nygaard, K., "The Simula 67 Common Base Language," Norwegian Computing Centre, 1970.
- [WIJ74] Wijngaarden, A., et. al., Revised Report on the Algorithmic Language Algol 68, Supplement to Algol Bulletin No. 36, 1970.
- [CAM74] Campbell, R. H., and Wyeth, D., "A Computer Architecture for a Language Permitting Programmer Defined Types," The University of Newcastle-upon-Tyne, Computer Laboratory Report, July 1974.
- [MCK70] McKeeman, W. M., et. al., A Compiler Generator, Prentice Hall, New Jersey, 1970.
- [DEW76] DeWitt, D. J., "A Machine Independent Approach to the Production of Horizontal Microcode," Ph.D Thesis, University of Michigan, June 1976.

REALIZING A VIRTUAL MACHINE

Brian Forbes, Tom Weidner, Ron Yoder, Tony Pitchford
 Computer Systems Group, Burroughs Corp.
 Mission Viejo, California 92675

Historically, microprogramming has been viewed as an emulation tool or a means of extending hardware instruction sets. This usually results in a piecemeal application of microprogramming techniques. The concept of a virtual machine, in which firmware realizes the complete functional structure of an idealized computer, is often debated but infrequently attempted. This paper examines microcoded interpretation of high-level languages and microcoded support of complex algorithms in a front-end peripheral processor as implemented for a commercial computer subsystem. The virtual machine and its supporting structure are outlined and specific advantages of microprogramming are discussed. Microprogramming is presented as a generalized systems resource.

1.0 INTRODUCTION

This paper is intended as an illustration of the capabilities of microprogramming rather than merely another description of a particular application. Its scope includes direct microcoded interpretation of high-level languages, real-time operating systems for front-end processors, and microcoded support of complex processing requirements. We view microprogramming as a flexible systems resource rather than the contemporary concept of a hardware curiosity [1], and describe its use in an existing commercial product.

2.0 THE APPLICATION

The application centers around magnetic-ink encoded (MICR) documents such as checks or credit card receipts. Typically, one day's processing involves reading several million documents and physically separating them into several hundred groups based upon the encoded data. The peripherals which accomplish this, called reader/sorters, can read up to 1625 documents per minute, directing each document to one of 20 (typically) output hoppers or 'pockets'. Between ten and fourteen reader/sorters transmit document images to a central computer where they are applied as transactions against a large data base.

After a document passes the read head, the controlling computer must accept the data, decide which pocket the document belongs in, and send this information to the reader/sorter before the document reaches the first pocket; the practical maximum response time is on the order of 37 milliseconds. The pocket-selection algorithm involves validation of data fields by means of field delimiters and check-digit techniques, a table look-up to determine the destination pocket, and the accumulation of various totals. Historically this algorithm is installation dependent, sometimes changing on a weekly basis; the length of time it requires per document directly affects the number of reader/sorters a computer can handle on a real-time response basis.

Added to this are the tasks of database management, on-line inquiry, and possibly batch processing. As the complexity of processing requirements and the volume of documents increase, the need to off-load the real-time aspects of reader/sorter control becomes critical.

The solution chosen was to interpose a small number of front-end processors between the reader/sorters and the central computer. Each reader/sorter processor (RSP) drives up to four sorters simultaneously, communicating with the host system via a standard peripheral interface. Having each RSP oversee multiple reader/sorters increases the cost-effectiveness of the subsystem, affording considerable resource concentration. This makes more effective use of both local storage and data paths, since a reader/sorter represents a low data transfer rate, on the order of 3 kilobytes per second. See fig. 1.

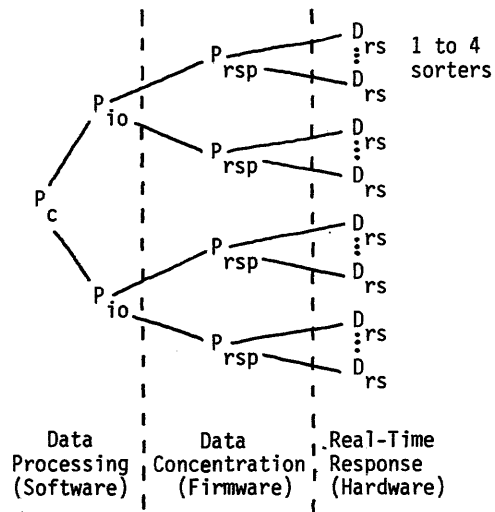


fig. 1 - A Reader/Sorter System Configuration

2.1 THE READER/SORTER PROCESSOR (RSP)

The reader/sorter processor is responsible for the real-time handling of up to four sorters, multiprogramming user-written pocket-select routines, passing document images to the host system, and communicating with the host system when exception conditions arise. With 4 sorters running at 1600 documents per minute, the RSP has about 9 milliseconds in which to handle each document. As each document is read, its image is placed directly into host storage and written to a disk audit file.

Each RSP is vertically microprogrammed with a 16-bit microinstruction length. It possesses 8KB of read/write control store with a cycle time of 167 nanoseconds. Main storage is addressable to the bit level; control store address space may extend to main store so that large microprograms can overflow into main storage with some ensuing speed penalty. Features of the RSP architecture such as bit addressability, variable-length ALU, internal stack, and the instruction set will be discussed following the description of the virtual machine.

3.0 SYSTEM DESIGN PHILOSOPHY

The reader/sorter subsystem presents three different interfaces to its users. To the data processing program, the reader/sorter is simply a unit record device. As long as no exception conditions arise, image capture is the primary concern. Unless the program overrides them, the system should provide default action to handle most exceptions. To the pocket-select routine, the sorter is a virtual peripheral capable of performing certain physical functions, while the data processing program in the host is mainly a consumer of document images. The applications programmer wants to express the pocket-select algorithm in a natural, flexible manner, using data types and control structures suited for the application and insulated from the machine-language level. To the host operating system, the RSP and its attached sorters should resemble standard peripherals, requiring as little active supervision as possible.

To this end, we have implemented two major pieces of firmware:

- A virtual machine for controlling a reader/sorter which is supported by a microcoded interpreter for a high-level language.
- An operating system which multiprograms four pocket-select routines, off-loads real-time supervision of the reader/sorters, and implements an intelligent interface with the central computer.

In this paper we will concentrate on the virtual machine and its implementation.

4.0 THE VIRTUAL MACHINE

A few remarks on what is meant by a virtual machine may be in order before we discuss the implementation. We consider direct interpretation

of high-level languages by microcode to be the vehicle which leads to our virtual machine development. An intermediate language (S-language) is generated by the compiler which in turn will be processed by the microcoded interpreter residing in control store. However, it is not our contention that this, in itself, constitutes a virtual machine. On the contrary, a virtual machine must also reflect some underlying structure. Store, registers, stacks, and environment all encompass the more general notion of virtual machine. The following paragraphs describe an S-machine, not just the S-language interpreted by the machine.

The routines which control the flow of the information digested by the reader/sorters are written by the applications programmer in a COBOL-like language called 'SCL'. It has been modified to suit the application, but the choice of a model language was due solely to the nature of the application: COBOL is the primary language used in commercial areas. A typical control routine may be around 1000 source statements in length.

The intermediate language is generated by applying only the analysis phase of translation to the SCL statements. The execution is performed on this high level language through direct interpretation by a microcoded interpreter. One of the advantages immediately apparent is the ease of translation due to the absence of the normal synthesis phase of compilation. The S-operators generated form a one-to-one relation with source statements, giving the S-language an attribute generally associated with assembly languages: there is a 'MOVE' operator, a 'PERFORM' operator, a 'GO TO DEPENDING' operator, etc. An example of an SCL language extension is the 'MATCHES' operator, which allows the comparison of two strings for similarity instead of strict equality; this is useful for handling mis-read characters from the reader/sorters.

As mentioned earlier, a virtual machine implements a logical structure as well as a set of operations. Once COBOL was chosen as the base language, the form of the COBOL machine, its data structures and underlying architecture, was still flexible. Rather than emulate historic 'flat' storage structures, we chose a descriptor-based machine where each data item is described by a 24-bit word; reference to a data item is by descriptor number. This scheme results in further code compaction and makes it easy to 'tag' certain items for special handling by the interpreter. See fig. 2.

An example of this is the handling of subscripted variables. The general symbolic form of an SCL 'MOVE' statement is:

```
MOVE ABLE TO BAKER
```

which is compiled as:

```
MOVE, i, j
```

where 'i' and 'j' are the descriptor numbers for ABLE and BAKER. If ABLE is a one-dimensional array, however, the statement

```
MOVE ABLE(CHARLIE) TO BAKER
```

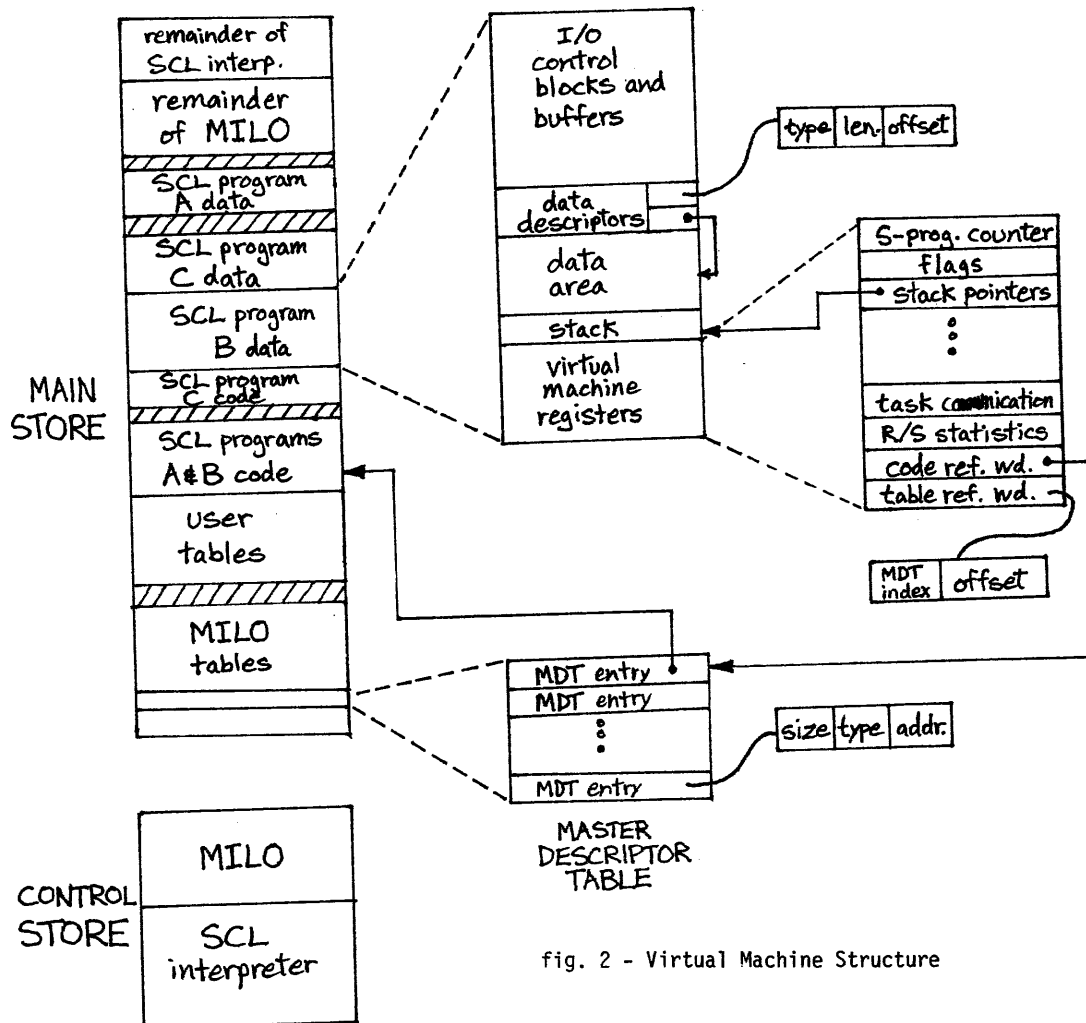


fig. 2 - Virtual Machine Structure

compiles as

```
MOVE , i , k , j
```

where 'k' is the descriptor number to CHARLIE. The interpreter, upon fetching ABLE's descriptor, discovers it requires one subscript, fetches 'k' and indexes ABLE by CHARLIE before fetching 'j'. Bounds checking is done by the micro-code based on descriptor information. The compilation and interpretation of such subscripted instructions are simple recursive processes. The evaluation of arithmetic expressions as subscripts, however, such as

```
MOVE ABLE(CHARLIE + DOG) TO BAKER
```

results in multiple S- ops and more compiler ingenuity; this is a traditional COBOL problem which could be alleviated by the introduction of a push- down computation stack.

The physical interpreter resides in 6 K bytes of store. Typically, 5KB reside in control store. To increase efficiency, most frequently used S- operators are guaranteed placement in control

storage. In an attempt to maintain structured design a very narrow interface exists between interpreter and operating system.

When discussing compilation we mentioned a lack of synthesis. For the most part that is true. However, due to certain criteria pertaining to the application, the interpreter contains other interpreters and as a result some code (tables) are generated. One such inner interpreter is a data-driven microcoded document scanner used in assimilation of data read by the reader/sorters. Another is a microcoded table search. Based on a description given by the customer using SCL statements, a table is built associating groups of documents to be read by the sorters as particular subsets of items. As each document is processed this table will be searched to determine membership. The interpreter, upon encountering a 'SEARCH' statement, causes an invocation of the table search interpreter which acts on the table (code) resulting in the required membership information. This information then becomes the result of the original 'SEARCH' command. The table contains bit patterns which represent S- operators

to the inner interpreter. These operators are either 2 or 4 bits in length, depending upon the expected frequency of use, and provide considerable space savings.

The virtual machine is thus a group of nested machines, within each of which the machine structure can be optimized for a particular task according to application criteria. In the case of table search, it was possible to achieve both size and execution time reductions over many search algorithms by providing a machine geared to certain kinds of searching. For example, a binary search on 5300 eight-digit keys would require 32 K bytes, whereas the specialized search algorithm requires only 14 K bytes and takes approximately the same amount of time.

5.0 THE OPERATING SYSTEM (MILO)

The collection of routines which assists the virtual machine has been given the name 'MILO'. Each system and virtual machine support function, such as storage management and logical I/O, is microcoded. MILO occupies 16 K bytes, of which 3 KB, representing the functional core, reside in control store. Besides providing support for the interpreter, MILO maintains the interface with the host operating system.

5.1 INTER- PROCESSOR COMMUNICATION

Inter- processor communication takes place via message queues maintained in the host's main store. All RSPs feed a common input queue to the central operating system; access is interlocked using the central store's 'swap' capability, whereby it is possible to write a word and retrieve its previous contents. The host system maintains an output queue and an available message queue for each active RSP. Interprocessor interrupts are used mainly to indicate that a previously- empty message queue now contains a message.

The bulk of the messages sent by MILO are 'buffer full' messages, indicating to the host that a block of document images in the host's store is complete, ready to be made available to the data processing program. Other messages to the host include exception condition notification, logging of reader/sorter statistics, and a 'handshake' to certify the RSP's integrity during idle periods.

Messages sent to MILO consist mostly of sorter start/stop and pocket- select routine load/unload commands. Should a pocket- select routine commit a run- time error, such as an array index which exceeds the array bounds, the host can request a dump of the virtual machine. When the output queue is idle, MILO expects periodic handshakes as a measure of the host's well- being.

5.2 VIRTUAL MACHINE SUPPORT

MILO provides many services to the SCL interpreter. These functions are performed by a driver which is the only MILO routine having detailed knowledge of the virtual machine, and which isolates the interpreter from the details of the operating environment. Support services include buffer handling, physical and inter- processor I/O, and

breakout/restart capabilities. The driver thus translates between the external world and the virtual machine.

5.3 SYSTEM RESPONSIBILITIES

MILO performs simple storage management, allocating and de- allocating space in local storage as required, and implements reentrant S- code. All reference to storage areas is via descriptors, so that it is possible to move areas while they are in use without affecting their users. The task structure which supports multiprogramming is capable of handling both microcoded and SCL routines, which results naturally from the soft nature of the interpreter. Therefore it is possible to multiprogram parts of MILO with itself, simplifying the implementation of asynchronous functions. Loading, unloading, and dumping of pocket- select routines in the background are accomplished in this way.

System integrity is of particular concern because the RSP has access to the host's main storage and system disk. The programmability of the RSP allows more sophisticated checking than is available in the hardware by enforcing narrow interfaces to external resources. Further, MILO performs detailed reasonability checks on all messages and tables accessed in the central computer's store, to detect host hardware or software failure. In such an event, MILO will shut down all reader/sorters and terminate operations in an orderly fashion.

6.0 ATTRIBUTES OF MICROPROGRAMMED IMPLEMENTATION

During the two years of design and implementation, and especially during the final stages, the most apparent attribute was flexibility. Modifications were easy to make to both operating system and interpreter as particular facets of the application became more clear.

An excellent example of flexibility occurred during the final testing stage of the product. An applications group had coded a particular routine using SCL which performed the task of removing hyphens from strings while moving them from one data area to another. Due to the weakness of COBOL in editing strings, such SCL-coded procedures would be inherently slow. Considering that other SCL users would need to perform the same task, the interpreter was altered: a variant was added to the MOVE statement within the interpreter allowing the more general solution of replacing any character by another during the MOVE. The particular subset of this solution required by the application group was a MOVE replacing '-' by 'null'. The change to both interpreter and SCL compiler took but a short time and resulted in both a decrease in object code and a tremendous increase in efficiency.

Our system, being vertically microprogrammed, lends itself very nicely to mandatory features such as fault diagnosis, statistical bookkeeping, and error recovery. The operating system is free to perform diagnostics and logging in the background. A program dumping facility, which runs concurrently with pocket- select routines, is provided for dumping just those areas pertaining to the routine which committed an error. Continuous logging of system and application information is done

automatically by the RSP/host including statistical and timing analysis.

7.0 BASE MICROPROCESSOR

Virtual machine implementation differs from emulation of existing machines in the demands placed upon the base microprocessor. It is usually the case that the base and emulated machine are quite similar with respect to hardware structure. Often, a base microprocessor will be tailored towards a specific emulation. The virtual machine designer, on the other hand, postpones engineering decisions as long as possible, and must then map his conceptualizations onto a suitable base machine. The RSP offers many features, both hard and soft, which make it highly suitable for the implementation of generalized virtual machine architectures.

7.1 BIT ADDRESSABILITY

The ability to store and retrieve a bit string of any length from any bit position in memory allows efficient storage utilization and obviates the necessity for the virtual machine's word length to be a multiple of the base machine's. In the RSP, the width is restricted by the capacity of the ALU (24 bits); micro-coded loops make it possible to handle longer strings.

7.2 VARIABLE LENGTH ALU

The effective width of the arithmetic/logic unit is controlled by the contents of an auxiliary register, as is the arithmetic mode (BCD or binary). This affects carry out, normalization, and other functions of the most-significant bit. It is as easy to implement 19-bit machines as it is to implement 16-bit machines, or arbitrarily wide words by means of loops. The use of loops to effect operations wider than the actual ALU is simplified by a microinstruction which sets the ALU width to the minimum of 24 and the contents of two registers which represent the remaining bit lengths of the operand strings being processed. If one operand is shorter than the other, this takes care of the odd bits remaining after taking the smaller bit length modulo 24. The final ALU width also controls the fetch from memory, so that only those bits which are valid are used.

7.3 REGISTER MANIPULATION

A central feature in the RSP is the bit rotator and masking facility. Besides providing a one-clock shift or rotate of a 24-bit register by any number of bits, it allows the extraction of any subfield of a register to another register. The RSP processor can test a total of 128 flags, representing the entirety of several 24-bit registers, subfields of others, and condition flags; the latter include static comparisons between registers, whether certain registers are odd or zero, and so on, which obviates passing the register contents through the ALU to test them. The combination of the rotator and bit test operators simplify the decoding of S-operators, inspection of packed information such as operand descriptors, and the maintenance of virtual machine status.

7.4 SCRATCHPADS AND STACK

There are 32 24-bit scratchpads available to the microprogrammer for use as base registers, virtual machine registers, and fast-access temporary storage. A 32-word stack is provided for subroutine management and for temporary operand storage.

7.5 BASE MACHINE SOFTWARE

The micro-implementation language of the RSP belongs to the class of higher level machine-dependent microprogramming languages [2]. Although each statement generally represents one microinstruction, provision is made for block structure, declaration of symbolic data areas, and macros. Statements may be grouped for conditional execution and IF-THEN-ELSE is available. English verbs are used as keys rather than mathematical notation for assignment and arithmetic operations. The principles of structured programming are more easily applied to this class of languages, wherein parametric subroutines provide a facility for logically subdividing tasks. Thus, both software and hardware cooperate in making the realization of arbitrary virtual machines effective using the RSP as a base machine.

8.0 CONCLUSION

At the risk of beating a quotation to death, we repeat Rosin's definition of microprogramming:

'Microprogramming is the implementation of hopefully reasonable systems through interpretation on unreasonable machines.'

Undoubtedly many unreasonable machines exist, however, we would disassociate the RSP with such a group. Perhaps the underlying reason for the success of this project has been the quality of the RSP architecture.

We hope this discussion has demonstrated the advantages of viewing microprogramming as a generalized programming tool in all areas of system development. In particular, vertical architectures extend the applicability of good programming practices to yet another level. Unfortunately, microprogramming is usually seen by the computing community as no more than an emulation tool or a means of extending hardware instruction sets. It is time to remove the undue mysticism surrounding microprogramming.

REFERENCES

- [1] Ashok K. Agrawala, Tomlinson G. Rauscher, Foundations of Microprogramming, Academic Press, Inc., 1976.
- [2] Infotech Information, Infotech State of the Art Report 23: Microprogramming and Systems Architecture, Maidenhead, Berkshire, Eng., 1975.

THE FORTRAN PROJECT - A MULTIFACETED APPROACH
TO SOFTWARE-FIRMWARE HIGH LEVEL LANGUAGE SUPPORT

Gideon Frieder
Department of Computer Science, State University of
New York at Buffalo, 4226 Ridge Lea Road
Amherst, New York 14226

ABSTRACT

A description of a novel approach for the support of high level languages is presented. This approach consists of different machines for different task structures, even if those tasks are written in the same high level language. At the same time, a high degree of similarity of the machines is maintained.

Preliminary results and the outline of future work are briefly summarized.

INTRODUCTION

The implementation of machines which support high level languages, via a mix of software-firmware techniques is an established technology. The widely quoted example of such an approach is the B1700, although there are others which use such techniques.

As applauded as this approach is, it is not without its problems or limitations. The main two problems which seem to be inherent in the present approach are:

- (a) All machines are designed to be virtual memory machines, using basically a common mechanism for virtual memory management.
- (b) All machines are assumed to be always in a multiprogrammed environment.

While these two problems seem to be desirable features rather than problems, they do cause sometimes unnecessary overhead and, what is more important, they preclude language and task dependent enhancements. That fact causes the price/performance of certain types of tasks to be less than one would expect. For example, it hardly seems useful to do simple student type jobs on the same machine that does extensive production in highly memory consuming tasks. The types of optimizations necessary are completely different.

In what follows, we shall illustrate our solution to the aforementioned problems on a medium scale machine, in the framework of the FORTRAN language. We propose to support a language by a series of machines which are close enough to guarantee equivalence of

results, but which are distinct enough to enable task oriented optimization and full exploitation of the machine capability.

The FORTRAN project is described in the next section, while Section 3 compares our approach to others and presents some preliminary results and expectations. In that section we shall also comment on our future steps.

2. THE FORTRAN PROJECT

The FORTRAN project consists of two main parts, the first of which is further subdivided into six parts:

- (a) Design of a general framework (i.e., instruction set) for support of FORTRAN-like languages.
- (b) Implementation of (a) in a fixed memory, multiprogramming environment under Burroughs MCP.
- (c) Implementation of (a) in a Burroughs MCP virtual memory, multiprogramming environment.
- (d) Implementation of (a) in a stand alone, task oriented virtual memory.
- (e) Development of a common compiler for the support of all Machines.
- (f) Development of the system components (i.e. loader) necessary for the support of the project.

The second independent part has not yet started. It will use heavily the machinery developed in the first part and some notions developed by Stockenberg[1] in order to answer the rather old question of compilation vs. interpretation that apparently was settled in the 50's but reappeared with

the introduction of user interpretive machines like the QM-1 and B1700. We reserve further comment on this question until we shall have actual results.

At this point in time, (a) and (b) are done. There is one person working on (c), one on (d), one on (e) and none on (f). Part (c) is 75% done, (d) 50% and (e) 10%.

The first part of the project, with its six subdivisions, intends to prove the desirability of multifaceted support of high level languages, as opposed to the current practice that provides one language for all machines or one language for each HLL. The various machines that we propose are geared toward four different modes of operation:

- (a) Small programs, for trying out ideas, simple one time computations, education, training and module debugging.
- (b) Production or development in a multiprogrammed timeshared environment.
- (c) Production in a dedicated environment, optimized for the specific language but not taking into account the specific task structure of the program.
- (d) Highly optimized, program dependent production.

The fourth mode is to be handled in part II of the project.

For each of the first three modes we propose a slightly different machine. The main differences lay in the actual microcoded implementation and addressing structure. As far as the user is concerned, the machines are identical in function, though not in speed. In particular, a program running on the different machines will produce identical results in all of them. The compiler for the three machines will be identical except for the code production phase. The fourth mode will be entirely different and, as mentioned before, we reserve any comment at this time.

For all modes, we now have ready the basic instruction emulation. For the first operational mode we have the machine designed and operational. We shall report in length about this machine in another place[2]. For reasons of brevity and prevention of duplication, we do not produce here any elaboration nor references on different FORTRAN machines, different FORTRAN language analyses and related topics. In particular, we do not elaborate of our goals and motivations to which caused us to produce yet another FORTRAN machine. The reader is again referred to [2]. For this presentation suffice

it to say that the machine is multiple addressing registers, dual accumulator machine with one, two and three address instructions. Preliminary measurements show it to be twice the speed of the Burroughs machine. It was designed using the statistics accumulated by Knuth[4] and using data structures which take into account the architecture of the B1700. It is easily adaptable to other machines and, as a matter of fact, we intend to adapt it to the QM-1.

The recent publication of the new proposed standard for FORTRAN[5] is a cause of major concern to us, as it imposes artificial, harmful and unnecessary restrictions on possible machine designs. When the final standard will be published, we shall reevaluate our machine and adapt it to the standard, even if we shall be forced to pay by efficiency in storage utilization and speed.

The machine as designed incurs no overhead for virtual memory (the overhead of Burroughs Fortran Machine for virtual memory was not taken into account in our comparisons). It does operate in the MCP multiprogrammed environments.

The mode 2 machine is identical to the first one in all aspects except in the address length which reflects the larger addressing space which is typical to a virtual memory machine. The virtual memory uses segment organization, with maximum allowable segment size. In this sense, it is not different from the Burroughs implemented FORTRAN machine. We do not expect any significant difference between our former measurements and those contemplated for the mode 2 machine. Thus we expect a factor of approximately two in CP bound jobs but, obviously, no significant improvement of I/O and virtual memory bound jobs.

Mode 3 machine is the most interesting in the first part of the FORTRAN project, and is an absolute prerequisite for a successful attempt of the second part. Its instruction set is identical in structure and function to that of the first two machines. It is a stand alone machine in the sense that there is no other program running with it, and therefore it has full and absolute knowledge of both the status and availability of all system resources. It therefore minimizes the interaction with the operating system, as the only unexcepted possible event is the console interrupt. This interrupt is on the human time-scale and therefore requires responses of the order of magnitude of 100 msec, which is a very long time in the logic time-scale. One can therefore eliminate large parts of the MCP

interaction. In medium scale machines, the interrupt handling is checked in each instruction by a microcode sequence which, in some cases, is very long compared to the actual instruction execution. Some rather heavily used instructions have short microcode sequences which can be shortened even further by judiciously eliminating the interrupt checking. This places some burden on the compiler, but as this mode is a production mode, one should be ready to accept a slightly longer compilation phase.

In the machine organization that we propose, i.e., in an addressing register machine, instructions like load address, compare registers and branch, increment and decrement and fetch indexed operand are very frequent. We expect a considerable enhancement in their speed based on the mere fact that a single user (stand alone) machine is completely cognizant of its environment. We believe that this approach was not proposed before.

Second desirable feature of such a machine is in its memory management. Here again one should be able to utilize the fact that a single program written in a static and simple language is running on the machine. Thus prepagging of code and data segments should be possible by utilizing the growing body of compile time flow analysis algorithms and procedures[6], and by implementing run time data gathering mechanisms in microcode. The data gathering instructions will be compiled at those places that will be dictated by the static flow analysis. Again, we have done nothing concrete in this area, as the machine is just being brought to its first operational stage. We do expect, however, to be able to run large scale programs on rather small memory, slow disk configurations. We shall report on the usefulness of dynamic measurements for paging consideration once the actual measurements will be available. The measurements will then be used as a basis for the development of the mode 4 machine.

Our ability to compare and measure these machines depends on the availability of a common compiler. Such compiler is now under development. During the initial stages of the development we encountered a couple of non-trivial problems, which stem from our belief that the length of real, integer and logical variables should be different. We report on these problems elsewhere[7].

3. COMPARISONS TO OTHER APPROACHES, RESULTS AND EXPECTATIONS

There are essentially four possible approaches to the support of high-level

languages on a machine, and these are:

- (a) one machine language for all high level languages
- (b) one machine language for each high level language
- (c) multiple machine languages for each high level language
- (d) direct execution of the high level language.

Approach (a) is obviously inferior to (b), as one cannot assume that a general machine language will be better than a specialized one. We therefore limit ourselves to b, c, and d. We claim that c is superior to both (b) and (d).

In the case of (b) one assumes that all tasks written in a high level language are of the same nature, at least as far as the support that they require in the machine is concerned. We claim that with the advent of virtual memory in medium and small machines and growing knowledge of static program analysis, specific data gathering and memory control instructions can control the flow of the microcode so as to optimize performance far above the fixed language support case.

As for the second extreme, i.e., direct execution of high level languages, the previous arguments still apply. One should not eliminate the benefits of compile time analysis and optimization achieved by them. If one uses such analysis in order to produce better, "optimized" high level language, then one should go through with the small additional step necessary to produce code for a properly designed support machine.

The results that we already have at hand show that our machine is superior in speed to the Burroughs designed FORTRAN machine and superior in code density to IBM 370, Burroughs B1700 FORTRAN machine and CDC CYBER. This obviously is not enough and further comparisons, especially to specific FORTRAN machines mentioned in [2] are called for.

We expect to be able to produce a common compiler which will operate in the four different modes. We expect that our machine design will prove the possibility to run virtual memory tasks on a small scale machine with the same efficiency that is accomplished today by static memory machines like the VARIAN 74. We also expect to incorporate dynamic data gathering for paging and memory management optimization and turn them into standard techniques.

4. ACKNOWLEDGEMENTS

I am indebted to Rina Shachar for her cheerful cooperation, to Peter Lutz for

his insights into compiling and to Bob Pekarske for his effort in the preliminary stages of the mode 3 machine.

REFERENCES

- [1] John Stockenberg: Private Communication, See forthcoming Ph.D. Thesis from Brown University. Part of the techniques due to John Stockenberg were presented in (3).
- [2] Gideon Frieder and Rina Shachaf: Machine Design for high speed FORTRAN support on a B1700. To be published.
- [3] G. Frieder: Microprogramming and Operating Systems, Infotech State of the Art report, Vol. 23, 1975, p. 393.
- [4] Donald E. Knuth: An Empirical Study of FORTRAN programs, Department of Computer Science, Stanford University, 1970.
- [5] X3J3 Committee Draft of Proposed ANS FORTRAN, SIGPLAN Notices, Vol. 11, No. 3, 1976.
- [6] See for example: Susan L. Graham, Mark Weghan: A fast and usually linear algorithm for global flow analysis, JACM 23, 1 (1976) p. 172.
- [7] Peter Lutz: A dedicated FORTRAN machine: Some problems with equivalence. To be published.

AUTOMATED PROOFS OF MICROPROGRAM CORRECTNESS

W. H. JOYNER, Jr.
W. C. CARTER
G. B. LEEMAN, Jr.

Computer Sciences Department, IBM T. J. Watson Research Center
Yorktown Heights, NY 10598 USA

This paper presents a method for verifying microprograms with computer aid, and examples of its application to actual systems. The specifications for an architecture and those for the computer on which it is to be implemented are both described formally, with the microcode supplied as data to the low level description. A correspondence between the two descriptions is then formalized, and a system of programs is used to prove mathematically that the correspondence holds. This interactive, goal-directed system not only provides a proof that microcode performs as specified, but more often aids in detecting and correcting microprogram errors. Several errors in actual implementations, some of which were difficult to detect using test cases, have been discovered in this way.

1. INTRODUCTION

The increased use of microcode, residing in a control store or in main memory, for implementation of machine architectures makes a demonstration of the correctness of the code a necessary part of the design verification of a computer. Ramamoorthy and Shankar [11] have examined loopfree vertical microprograms, while Patterson [10] has proposed a method for proving that microprograms written in a high level language are correct. Below we describe a partially automated system which has been used to detect errors in microcode and to certify microprograms as correct.

In proving the correctness of microprogrammed implementations, all of the facets of machine operation must be explicitly described. However, particularly for microcode implementations of computer architectures, assertions for correctness such as those due to Floyd [4] are not easily formulated. Our approach is to give the specifications for correct implementation as an abstract machine schema [1,8], having a well specified tree control structure which operates upon a state vector of machine components and is determined by a library of macro routines. The state vector components are treated as APL-like variables; the macros are written in a Vienna Definition Language/APL format [1]. The attributes of the computer on which the specified architecture is to be implemented are also embodied in such an abstract machine. Next it is proved that the abstract computer schema, controlled by the microprogram, simulates the architectural machine schema. This is a generalization of Milner's algebraic simulation between programs [9].

Previous work [1, 7, 8] has been done using abstract simulation to prove the correctness of microcode. However, these proofs, though formal, were carried out by hand. It became apparent from them that the individual parts of such proofs were not of great complexity, but that the main impediment to human proofs was the generation and organization of these many separate parts. Since this number of parts increases with the size of the implementation being verified, it became clear that some automated aid would be necessary. In the following we shall describe abstract machines and simulation in more detail, and briefly describe an interactive system, MCS, [2]

designed to aid in proving simulation between programs. Finally, examples will be given from experiments in using MCS.

2. ABSTRACT MACHINES AND SIMULATION

The abstract machines which describe the architectural level and the register transfer level of a computer each consist of two parts: the abstract syntax and the macro library. Implicit in these descriptions is a control algorithm for abstract machine schemas. The abstract syntax, for our purposes, is a list of the components of the machine: registers, storage, switches, lines, etc. We associate with each of these a shape, or dimension, in the style of APL. Registers have associated with them their width in bits; main storage has the dimensions of a matrix. Fig. 1 shows the abstract syntax for the architectural description of the S-machine, described in Section 4. The macro library for each machine is a list of macro definitions for

```
{(MEM 16777216 32)
 (STK 32)
 (SX 32)
 (SCC 32)
 (SSW 1)}
```

Example of abstract syntax showing
dimension of each machine component

Figure 1

a machine with a tree control structure, each consisting of its name, formal parameter list, and either a tree into which the macro expands, or several such trees whose selection depends on predicates over elements of the abstract syntax. Fig. 2 is an example of a macro definition in the VDL/APL notation. If the macro exec pgm appears as the leaf of a control tree, it is replaced by the tree shown if the switch SSW is on, and is removed and not replaced (indicated by Ω) otherwise. The macro body shows assignments to local variables and calls to other macros (underlined) which may or may not return values.

The architectural or specification level machine is completely defined by formalizing its principles of operation as an abstract machine. The second machine specification results in a general purpose computer controlled by the instructions in the control store.

This computer is made into a special purpose machine by associating the actual code to be verified as a value of the component which contains it (e.g. the actual microcode in the control store).

```

execpgm =
  SSW = 1 → exec-pgm
           execinstr (op, adl)
           advctr
           adl: instrprep (id, ix, op, ad)
                id: a[0]
                ix: a[1]
                op: a[2+16]
                ad: a[8+124]
                a: fetchword (SCC)
  SSW = 0 → Ω

```

A tree control macro

Figure 2

In order to simplify the proofs of simulation by breaking them into parts (see [8]), we choose as points of control at which we are interested in establishing a correspondence (the stopping points) a small subset of all possible values of the VDL control tree. This choice (see [8]) decomposes the simulation relation R into components R_1, \dots, R_n , one for each pair of control points at which a relation is to be established. Each component contains both control information, specifying for each machine the point of control at which a correspondence must hold, and simulation conditions, detailing the desired correspondence. The control information usually consists of a certain form of control tree, but may also include predicates (stopping conditions) over the state vector variables which further constrain the points of simulation. The simulation conditions are predicates (usually equalities) relating the state vector variables of the two machines. A sample simulation component appears in fig. 3.

<u>exec-pgm</u>	control tree for specification level (single macro on tree)
NIL	stopping conditions (none)
<u>exec-mpgm</u>	control tree for processor transfer level
l = 2:CSAR	stopping condition
MSMEM = SMEM	simulation conditions
MXS = SX	
MSCC = SCC	
MSSTK = SSTK	
MSSW = SSW	

Elements of a Simulation Component

Figure 3

Given initial values for the state vector quantities and an initial control tree, the control algorithm for the abstract machine is applied. Briefly, if the leaf L of the control tree is a macro, then L is replaced by the expansion of the macro if the latter contains no predicates. If the macro has one or more nested paths of predicates, the first path with all its predicates true is chosen; the tree structure at its end replaces L . If L is an assignment statement, then either local or global variables are appropriately modified (see fig. 2).

The decomposition of the simulation relation suggests a decomposition of the problem of proving algebraic simulation. We must, for each pair of stopping points corresponding to a simulation component R_i : (1) assert that the simulation conditions corresponding to component R_i hold; (2) run each abstract machine until another stopping point is reached (i.e., apply Milner's functions F, F' between domains); (3) verify that the pair of

points reached corresponds to a component R_i of the simulation relation; and (4) prove that the simulation conditions of this component hold. Proving condition (4) is equivalent to showing that if the pair (d, d') is a member of R_i , then $(F(d), F'(d'))$ is a member of R_i . Below we describe an automated system which aids in this proof.

3. THE MCS SYSTEM

MCS (for "Microprogram Certification System") is written in LISP and provides interactive aid for proving that a stated relation between two machine descriptions is a simulation. A critical problem in the hand proofs of microprogram correctness [1, 7, 8] was to assure that all pairs of simultaneous paths (from one component of the simulation relation to the other) were taken and all theorems generated and proved. In MCS this bookkeeping function is embodied in an interactive, goal-directed, and problem independent supervisor [2]. From the user's point of view, this supervisor provides a uniform interface through which he manipulates a tree of goals. Using a set of standard commands, he controls the direction of the proof of simulation and observes its progress by invoking the components of MCS which do the actual work of proving simulation - the path tracer, simplifier, verification condition generator, theorem prover, etc.

The data structure upon which the MCS supervisor operates is an AND goal tree, and the user manipulates this tree in a problem-reduction fashion [2]. The initial goal entered by the user in proving simulation between programs indicates that the problem is to prove that a given simulation relation holds between two given machine descriptions. This goal generates one subgoal for each of the components of the simulation relation. Each of these subgoals, called a goal of class TRACEIT, consists of the state vector, control tree, and macro library for each machine, a current predicate list of assumptions, and the complete simulation relation. The control tree of each machine is taken from the component of the simulation relation to which the subgoal corresponds. The state vector is formed by assigning a unique symbolic value to each component of the abstract syntax, in preparation for the symbolic execution of macros. Presently this value is formed by prefixing "\$" to the name of the machine component. The predicate list is created from the predicates given in the stopping conditions and simulation conditions of the simulation component, although some of these predicates may not appear explicitly in the predicate list but may be reflected in the initial values of the state vector quantities. Fig. 4 shows the elements of a goal of class TRACEIT, generated from the simulation component of fig. 3.

To achieve each of these goals, the user must show that starting from each goal and running both abstract machines to their set of pairs of next stopping points, the correct simulation conditions hold at each pair of points reached. The abstract, or symbolic, interpreter carries out the symbolic execution [2, 6] of each machine by employing a simplified version of the control algorithm in [8]. Normally, the abstract interpreter is applied first to the specification description until a stopping point is reached, and then to the processor level. In each case, the interpreter checks at each step to

see if one of the stopping points defined by the components R_1, \dots, R_n of T has been reached; if so, a subgoal of class TRACEIT is generated. Note that all loops or potentially infinite recursive expansions in the machine description must have associated stopping points in some component R_k .

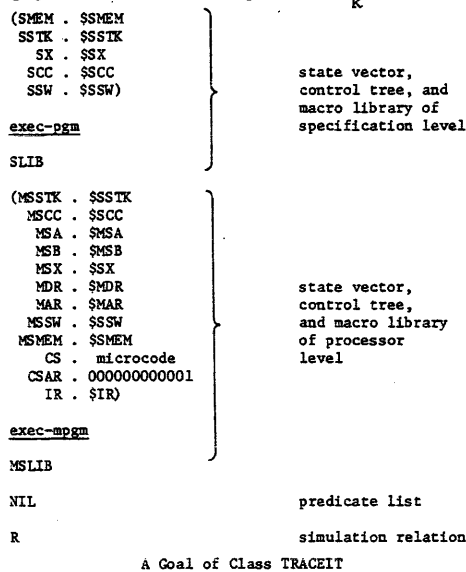


Figure 4

The symbolic execution chiefly affects two aspects of the interpretation. First, the expressions in assignment statements are symbolic, and cannot usually be evaluated to numeric or boolean values. Second, when predicates are encountered in expanding macros, they cannot always be evaluated to "true" or "false". Both of these problems are partially solved by a simplifier, which performs the symbolic computation done in MCS. Whenever the interpreter encounters an assignment statement, a predicate in a conditional, or the passing of arguments to a macro, the simplifier is invoked to use some of its 400 reduction rules to "evaluate" APL and logical expressions by returning a simpler form if possible. When a predicate which cannot be resolved in this way is encountered, multiple subgoals, one for each possible value of the predicate, are generated, and the user selects the subgoal which he wishes to pursue first.

When a goal is generated for which both descriptions have reached stopping points, the verification condition generator is invoked. This first verifies that another point of correspondence R_i in the simulation relation has been reached, and then instantiates values from the two state vectors into the list of simulation conditions given in R_i .

A goal, consisting of a theorem and the predicate list of the current goal, is generated for each of these instantiated conditions. An example of a generated theorem appears in fig. 5.

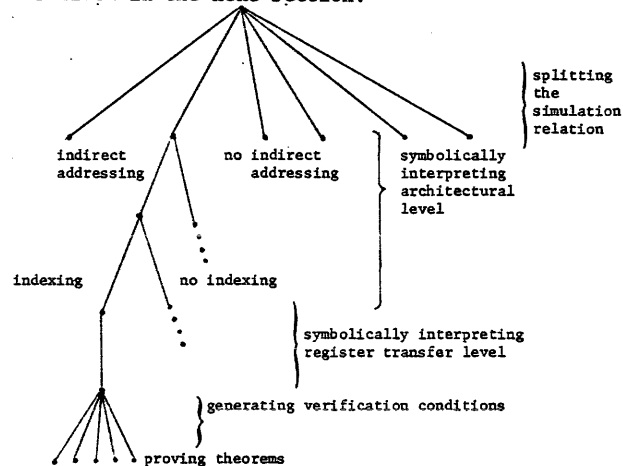
```
((8p0),
((32p2) τ (21$SX)
+21$SMEM [21$SCC[8+124]; 8+124])
[8+124]) [18] = 8p0
```

GENVC Generated Theorem

Figure 5

The theorem prover is invoked on goals of this form. If all the theorems on all branches are proved true, the supervisor will mark the top level goal as achieved, and simulation will have been shown. Errors in the microcode (or in the formal descriptions) are detected by being unable to prove theorems at the leaves of the goal tree, by invalid branches or by failure to reach a valid stopping point. By tracing back toward the root, information about the particular instruction or place in the description at which the error occurred can be obtained.

Fig. 6 shows a portion of the goal tree (corresponding to operand-fetch) for the S-machine experiment, described in the next section.



Portion of Goal Tree for S-Machine Experiment

Figure 6

4. THE S MACHINE EXPERIMENT

The S machine is a simple hypothetical computer described by Haralson and Polivka [5]. The stack is in main memory, which is an array SMEM of 2^{24} 32-bit words. Additional components in the architecture of S include a one-bit switch SSW, an index register SX, an instruction counter SCC, and a stack pointer SSK, each of size 32 bits. There are 27 machine instructions. The one-address instructions need an address operand; the zero-address instructions manipulate the stack and require no operand.

The micromachine MS which emulates S has entities MSMEM, MSSW, MSX, MSCC, and MSSTK corresponding to the components of S. It possesses in addition general purpose registers MSA and MSB and a memory data register MDR, each of size 32 bits; a 24-bit memory address register MAR; a five-bit instruction register IR; a twelve-bit control store address register CSAR; a sixteen-bit control store data register CSDR; and a control store CS containing a maximum of 300 sixteen-bit microinstructions. The microcode which we have proved correct with MCS has 172 microinstructions. This code has a major loop beginning when a machine instruction is to be executed. This loop is divided into two sections. In the first, IFETCH, the operands for the current instruction are determined. For zero address instructions the given operand is used; for one address instructions indexing and/or indirect addressing is performed. In the second section, IEXECUTE, the machine instruction operation code is used to calculate a branch to a segment of microcode which performs that instruction. Each of these segments

is straight line code, except for the handling of the two shift instructions, each of which contains a simple loop.

Each of the two machines was described formally by means of the VDL/APL language mentioned earlier. As pointed out above, natural places to choose for stopping points depend upon the architectural description (in this case the S machine), and upon the structure of the microcode. In the original S description [1], each machine instruction was described separately; in the microprogram, portions of the code were shared by several instructions. To reduce the number of paths and shorten their length, new macros were added to the S description to consolidate the work previously done in each instruction. These changes produced a second architectural description which was easily shown by hand to simulate the initial one. The proof was completed by showing a simulation relation between the new description and the register-transfer description MS and using the transitivity of simulation [7].

The original hand proof of simulation had two pairs of domains or stopping points: begin instruction (R_0) (isomorphic to end instruction) and machine stopped (R_2). The change in description allowed the addition of the new domains R_1 (zero address instruction calculation), R_2 (one address calculation), R_3 (left shift verification condition), and R_4 (right shift verification condition).

All of the pairs of paths emanating from the six components of the simulation relation were traced. In the complete attempt at verification of the S-machine microcode using MCS, 45 pairs of corresponding paths were traced, and 236 verification conditions were generated. Of these, 224 were proved (two with user interaction by splitting into cases); inability to prove the remaining twelve indicated microprogram errors, which were corrected. Three errors had been found by hand, and MCS found an additional error [3]. The changed code was then validated.

5. THE HTC EXPERIMENT

MCS is now being applied to a real computer architecture. The HTC (for Hybrid Technology Computer) is a product of the IBM Federal Systems Division at Huntsville for space flight applications [12]. Its architectural specifications require it to support the System/360 standard instructions (no decimal or floating point operations) with sixteen 32-bit general purpose registers, 16 to 64 K bytes of main storage, and a single I/O channel for three types of I/O. This architecture is implemented in a machine having a 1K memory of 64-bit microinstructions, a 64x10 bit read-only memory for instruction decoding, a 64x16 bit scratch pad memory, an ALU and two input multiplexers, three working registers, and a 16-bit wide data path.

Part of the increased difficulty with the HTC lies in formalizing the description of the architecture and register transfer levels. Architecturally, the HTC differs from the System/360 in several ways. In addition to emulating the specified instruction set, the HTC microcode also implements interrupt and I/O handling. Thus the formal architectural description must be pieced together from the System/360 documentation and other HTC information. Also, in the main machine instruction definitions, parts of the specifications (such as condition code settings) were left

intentionally ambiguous to allow implementation on various S/360 models; the architectural description must allow for this vagueness. The APL ? operator is used in these cases, and specific simplification rules are included for handling expressions with this operator. The single 16-bit wide I/O channel of HTC has been modeled by an array of 16 columns and an infinite number of rows. Input from the channel is done by taking the first row of the array as the input word; output is done by concatenating rows onto the end of the array. In the HTC, the register-transfer level has a 16-bit data path with 16-bit register and storage locations; the architectural level is 8-bit byte-oriented. For example, each 32-bit general purpose System/360 register is stored in two non-contiguous locations in the HTC scratch pad memory, and the 16Kx8 bit main store of the architectural description is implemented in an 8Kx16 bit memory in the hardware. Thus the APL expressions stating the equalities between the machine components are more complex than in the S-machine, where the architectural components are a subset of those at the register transfer level.

The HTC register transfer level description describes how the microcommands in a microinstruction act to modify the values in the registers and storage in a real computer. Since the validity of the simulation proof depends upon the accuracy of this description, only hardware entities are made part of the micro machine state vector. However, macros should be written to be direct, and perform calculations so that simulation speed is rapid. These requirements conflict frequently. Presently, for efficient microcode validation a good knowledge of both the microcode and the data flow is required.

The HTC is a real machine, so the methods by which it physically starts (stops) and the consequences of its timing conditions and asynchronous actions must be modeled. Starting and stopping concerns both the architectural and register-transfer levels while the timing conditions primarily affect the register transfers. Since the HTC is an airborne computer in the Space Ultra-reliable Modular Computer (SUMC) family, its connections with the outside world are controlled by microprogrammed Test Support Equipment (TSE) which simulates an HTC channel and also has lines directly connected to the HTC. The control lines (activated by buttons on the TSE console) and their functions are: Power on/off; Reset (with microprogram assist) and soft-stop (wait for external interrupt).

The basic architectural functions which may be initiated are IPL and Read Paper Tape (for program entry). The support functions are the usual maintenance functions - register display, stop and display registers when a particular address (in main storage or in control store) is reached, clear and test main store, and display main storage or control storage locations. All of these functions are initiated by the TSE and performed under microprogram control using the HTC channel. Since these functions are implemented in microcode, an architectural level description for them must be given and a simulation between the two levels specified and proved.

The basic HTC machine cycle is 550 ns and the main storage cycle is 700 ns. One microinstruction is performed per machine cycle. The action of the registers is asynchronous, but the microcommand actions are performed roughly sequentially except for interactions between the main store and the

computer. The action of the registers is determined by sequential leaves on the HTC control tree, as was shown in the S-machine example. The timing interaction between the computer and main storage is considered only to ensure that the contents of affected registers, the Storage Data Register and the Instruction Register, are valid. All timing is thus relative to microinstruction execution, so a pseudo counter, CTR, was introduced. Whenever a microinstruction is read, CTR is incremented. Each of the two registers affected by timing is replaced by two pseudo registers. The first contains the usual register contents and is set as if there are no timing restrictions. The second is set to the contents of CTR at the time the first is set. When one of these registers is accessed, the value of the second register is compared with the contents of the CTR, and if the difference is too small a timing error is signaled.

Finally the size of HTC compared with the S-machine makes necessary a more complex simulation relation, more paths to be interpreted, and more difficult theorems to be proved. The HTC implements an instruction set three times larger than that of the S-machine and has eight times as many microinstructions, each of which is four times larger.

At present the HTC architectural and register transfer levels have been formally described, the simulation relation has been partially formulated, portions of the microcode have been proved correct, and some errors have been detected. Several of these have been subtle errors which are difficult to detect using test cases. For example, one of them would occur only when fetching a 16-bit instruction from the last halfword of addressable memory. Another more serious flaw was found in the implementation of the BALR or branch-and-link instruction: in cases where the link information was to be stored in the same register containing the branch address, the branch address was erroneously lost. All of these errors were not found by testing or simulation: they were detected because of unprovable theorems or unexpected branches being generated in the certification process. The goal structure of MCS enabled the user to go directly to a small segment of microcode and correct the error.

6. CONCLUSIONS

Our aim in the MCS system has been not only to obtain proofs of correctness, but also to detect and correct errors. The problem reduction approach provided by the MCS supervisor has several advantages. Most important of these is that no path to be traced can be overlooked. Indications of error, such as being unable to prove a theorem, failing to reach an expected stopping domain, and unexpected branches, occur when the user invokes a rule in a particular subgoal. Examination of the branches of the goal tree leading to the error indication provides some information as to the cause of the error. Also, the goal tree record of the way in which the goals are manipulated permits partial proofs; certain sections of program or microcode can be verified even before correctness criteria, in the form of simulation relations, have been developed for other parts.

The most useful MCS routines are simplification of symbolic APL expressions, and the symbolic interpreter. The immediate simplification of expressions and processing of predicates encountered contributed

greatly to the simplicity and intelligibility of the final theorems generated.

Of course, the human contribution to the automated validation of implementations consists of more than interaction with the MCS system. Describing the abstract machines which embody the system specifications is by no means trivial, especially when the English "principles of operation" are vague. Also, specification of the simulation relation requires some understanding of how the program being verified works (location of loops, etc.), though use of MCS to interpret symbolically a single program may provide aid in developing simulation conditions and stopping points. The judicious choice of stopping points can greatly reduce the number of the paths which must be followed and the theorems which must be proved.

The successful detection and correction of errors in a small microprogram using the interactive aid of MCS, and our progress toward the verification of an actual microcoded implementation, have confirmed our beliefs that computer aid in validating the design of computer systems is needed and valuable, and that the notion of simulation between programs facilitates this automation.

REFERENCES

- [1] A. Birman, On proving correctness of microprograms, IBM J. R&D. 18, 4 (May 1974), 250-266.
- [2] A. Birman and W. H. Joyner, A problem reduction approach to proving simulation between programs, IEEE Transactions on Software Engineering SE-2, 2 (June 1976), 87-96.
- [3] W. C. Carter, W. H. Joyner, and G. B. Leeman, Automated experiments in validating microprograms, Fault Tolerant Computing Symp. 5, Paris, (July 1975), 247.
- [4] R. W. Floyd, Assigning meaning to programs, Proc. Symposia in Applied Mathematics 19, 1967, 19-32.
- [5] K. Haralson and R. Polivka, Microprogram training - an APL application. Proc. 4th Int. APL User's Conf., 1972.
- [6] J. C. King, A new approach to program testing. Proc. Inter. Conf. on Reliable Software, Los Angeles, (April 1975), 473-481.
- [7] G. B. Leeman, W. C. Carter and A. Birman, Some techniques for microprogram validation. Proc. IFIP, Stockholm, (August 1974), 76-80.
- [8] G. B. Leeman, Some problems in certifying microprograms, IEEE Transactions on Computers C-24, 5 (May 1975), 545-553.
- [9] R. Milner, An algebraic definition of simulation between programs. Proc. Second Inter. Conf. on Artificial Intelligence, London (September 1971), 481-489.
- [10] D. Patterson, The design of a system for the synthesis of correct microprograms, MICRO-8 Proceedings, Chicago (Sept. 1975), 13-17.
- [11] C. V. Ramamoorthy and K. S. Shankar, Automatic testing for the correctness and equivalence of loopfree microprograms IEEE TC, C-23, 8 (Aug. 1974), 768-781.
- [12] Technical Description of the SUMC Computer Model 2B, IBM Doc. #4CW 00013, Dec. 31, 1975.

A MICROPROGRAMMED MACHINE ARCHITECTURE FOR EFFICIENT MATRIX MULTIPLICATION

Robert W. Nowlin
Northern Arizona University
Flagstaff, Arizona

Donald Gustafson
Texas Tech University
Lubbock, Texas

ABSTRACT

In the modern world of control and estimation theory, matrix multiplications are ubiquitous. Minicomputers designed as general purpose machines do not have instruction sets designed to efficiently implement these multiplications. A microprogrammable machine may be capable of efficient matrix multiplications if it has the proper architecture. A Hewlett-Packard HP-2100 minicomputer was used to investigate architectural and efficiency problems. Algorithms were developed to calculate execution times for any n , m , and T where n and m refer to the matrix dimensions and T is the basic machine instruction time. The execution times for various size matrix operations indicated savings of up to 80% for microcoded operations.

1. INTRODUCTION

Many equations of modern control, digital filtering, Kalman filtering and related problems involve matrix-vector multiplications which must be calculated to obtain their solution. These matrix-vector products have the general form shown in eq. (1)

$$\underline{Ax} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} \quad (1)$$

where \underline{A} is a $n \times m$ matrix and \underline{x} is a $m \times 1$ column vector. In computing a solution to problems where the product \underline{Ax} is needed, considerable time is involved for any reasonable size n and m . Hence if a real-time, on-line solution to the problems of modern control and signal processing is to be computed, a method for calculating \underline{Ax} efficiently must be developed. The method developed in this paper is to design an appropriate architecture and firmware which will yield an efficient implementation of matrix-vector product algorithms. The procedure used to derive this appropriate architecture and firmware is outlined below.

First a machine language implementation of the algorithm to compute \underline{Ax} is produced so that comparisons of execution times with subsequently developed microcoded versions can be made. An algorithm is developed to calculate execution times for any n , m , and T where n and m refer to the matrix size and T is the basic machine language instruction execution time. Next, various size matrix-vector products are microcoded. A $n \times 2$ matrix times a 2×1 vector microprogram is

implemented and run. A Hewlett Packard, HP-2100, minicomputer was used to determine both machine language and microcode execution times. Algorithms are developed to calculate execution times; actual execution times for various size matrix-vector products are computed for comparison. Finally half-word length (8-bit) versions of a $n \times 2$ matrix times a 2×1 vector and a 4×4 matrix times a 4×1 vector are microcoded and run. Algorithms for execution time calculation are developed; execution times are computed for various size products; and comparisons are made with the aforementioned machine language and full-word length versions. This development of microroutines reveals the architectural characteristics necessary for an effective implementation of matrix-vector products on microprogram controlled computers.

2. ASSEMBLY LANGUAGE ROUTINE

The assembly language routine for an $n \times m$ matrix times an $m \times 1$ vector is shown flowcharted in fig. 1. The routine was programmed to handle only integers that had been scaled three octal places, thus actually allowing three octal place accuracy. No checking for overflow or underflow was made. The multiplication of integers is justifiable since the main concern is not to just demonstrate feasibility or viability of this approach but to develop routines amenable to real-time, on-line systems applications. In such applications, information into the computer derives from an analog to digital converter which produces integerized digital values of the analog signal. Scaling is normally employed between the computer and the system both on input and output. Checking for an overflow condition would also normally be carried out, but the action taken upon an overflow detection is system dependent, e.g., an abort might be necessary, further scaling might be sufficient, etc. Therefore, the routine developed is sufficiently general and adequate.

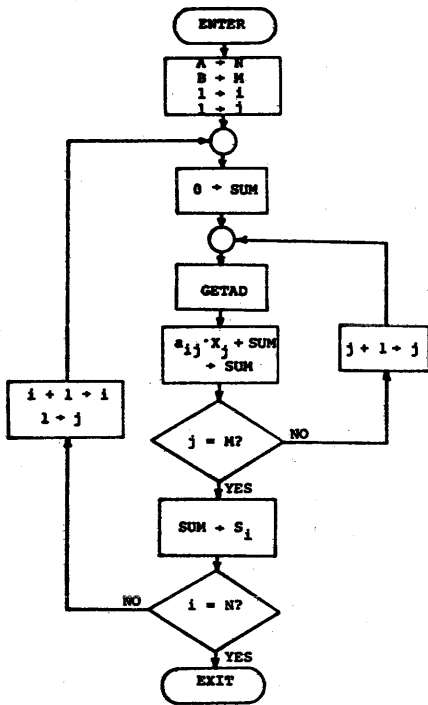


Fig. 1 - Flowchart for assembly-language matrix-vector multiplication.

The elements of A , the a_{IJ} 's, were stored sequentially by rows, i.e., first, a_{11} , then a_{12} through a_{1m} , then a_{21} , etc. The algorithm used for accessing each array element is given in eq. (2).

$$\text{Address}(a_{IJ}) = (I-1)N + J + \text{Address}(a_{11}) - 1 \quad (2)$$

The time of execution for this routine can be calculated from the following equation,

$$t_e = 29.5mT + 10.5nT + T \quad (3)$$

where n and m refer to the matrix size and T is the basic machine instruction execution time (1.96 μsec for the HP-2100). Equation 3, along with all other equations for t_e in this paper, was determined by actually summing the times to execute each instruction and noting the dependency on m and n in processing the algorithm. The times for a 2×2 matrix times a 2×1 vector, a 4×4 matrix times a 4×1 vector, and an 8×8 matrix times an 8×1 vector which were calculated for latter comparisons are, respectively,

$$\begin{aligned} t_{22} &= 274.40 \mu\text{s} \\ t_{44} &= 1009.40 \mu\text{s} \\ t_{88} &= 3867.08 \mu\text{s} \end{aligned} \quad (4)$$

3. MICROCODED FULL-WORD MATRIX-VECTOR MULTIPLICATION ROUTINES

The flowchart for a microcoded multiplication of an $n \times 2$ matrix times a 2×1 vector is shown in fig. 2. To save memory references and unnecessary

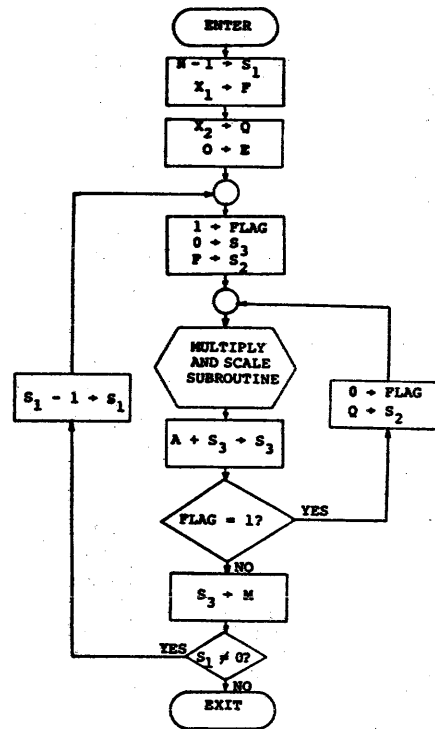


Fig. 2 - Flowchart of an $n \times 2$ matrix times a 2×1 vector.

programming, x_1 and x_2 are first read into WCS and stored in registers F and Q , respectively. All scratch-pad registers are used: S_2 and S_4 in the multiply subroutine to temporarily hold the multiplicand and multiplier as are A and B to hold the results; S_3 contains the running sum, and; S_1 to count n . Each a_{IJ} is retrieved from memory as needed; and the result $a_{11}x_1 + a_{12}x_2$ is then stored in memory. The Flag flip-flop is used to determine when this result is completed.

The algorithm for calculation of time of execution was determined to be

$$t_e = 13T + 110nT \quad (5)$$

where n is the number of rows in the matrix and T is the microinstruction execution time (196 ns for the HP-2100). The time to execute this microprogram for a 2×2 matrix times a 2×1 vector was calculated from eq. (5) to be 45.668 μs . Comparing this with the time to perform the same operation in assembly language, a savings factor of six (6) or about 230 μs is accomplished.

As was demonstrated, all available registers plus the Flag flip-flop were used to implement this microprogram. Since only microinstructions and no data (other than eight bit constants stored in the least significant eight bits of certain microinstructions) can be stored in WCS, a severe limitation, this is the maximum size matrix-vector product that can be implemented on the HP-2100. However, microprograms of a 4×4 matrix times a 4×1 vector, an 8×8 matrix times an 8×1 vector, and an $n \times m$ matrix times an $m \times 1$ vector were flowcharted and actually microcoded, but not implemented. These flowcharts are shown in figs. 3, 4, and 5, respectively. These routines

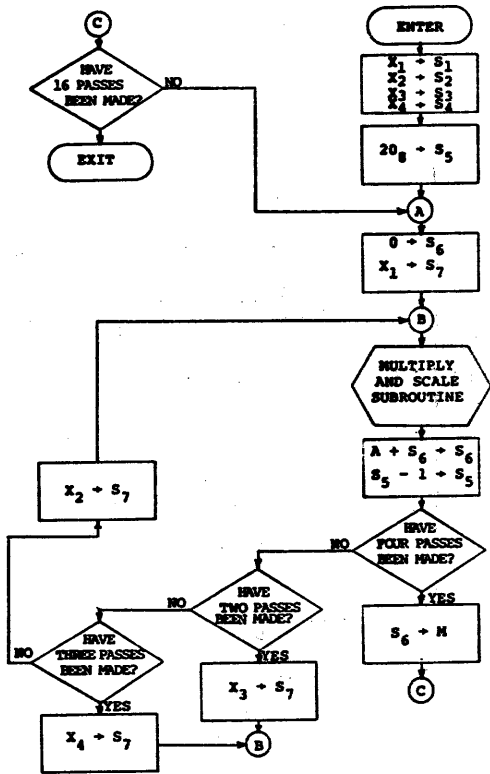


Fig. 3 - Microprogram flowchart for a 4 x 4 matrix times a 4 x 1 vector.

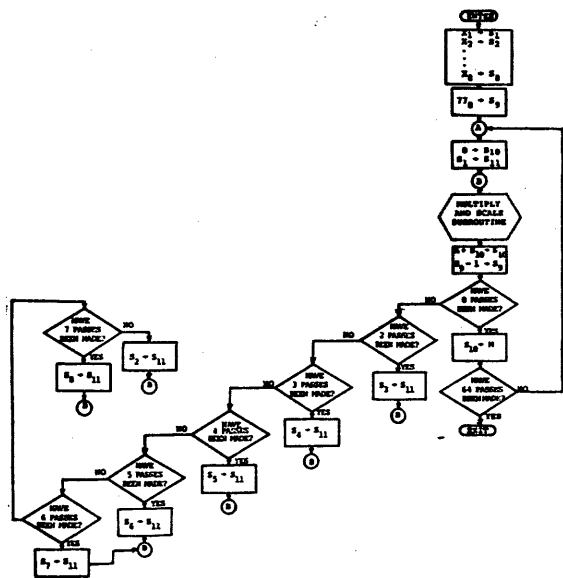


Fig. 4 - Microprogram flowchart for an 8 x 8 matrix times an 8 x 1 vector.

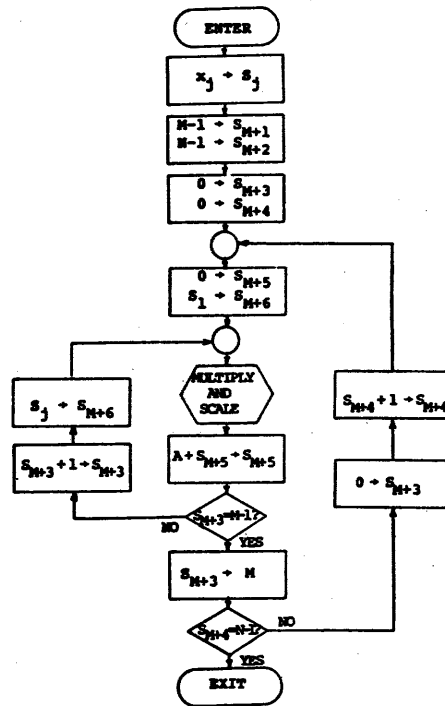


Fig. 5 - nxm matrix times mx1 vector.

require one storage register for each element of the vector (4, 8, and m), two scratch registers plus the A- and B-registers for the multiply subroutine, one scratch register for the counter for the square matrices and four counters for the nxm matrix, and one register for the running sum; or, 8, 12, and m+7 scratch registers, respectively. As seen, these are more than are available on the 2100, a limitation that could be alleviated if data could be stored and retrieved directly from WCS other than as eight bit constants.

The time of execution algorithms for the first two of these microprograms is given as

$$t_e = 5T + 7nT + 59n^2T + nT \sum_{i=1}^{n-2} (n-i) + 2nT \sum_{i=2}^{n-1} (n-i), \quad (6)$$

where n refers to the (square) matrix size, T is the microinstruction execution time, and i is a counter which depends on the number of decisions to be made (n-dependent). The times of execution for these programs are

$$\begin{aligned} (4 \times 4) \quad t_e &= 200.116 \mu s \\ (8 \times 8) \quad t_e &= 860.244 \mu s \end{aligned} \quad (7)$$

Comparing these times with those corresponding ones found using the assembly language routine (i.e., 200.116 μs vs. 1009.40 μs and 860.244 μs vs. 3867.08 μs), savings ratio of about 4.8 and 4.5 are effected. The algorithm for calculating execution time for the nxm matrix times the mx1

vector microprogram is

$$t_e = 2T + 5mT + 58nmT + 4nT + nT \sum_{i=1}^{m-2} (m-i) + 2nT \sum_{i=2}^{m-1} (m-i), \quad (8)$$

where again n and m refer to the matrix dimensions (n rows and m columns), T is the basic micro-instruction execution time and i is an m -dependent count. Time of execution for a 4×4 matrix times a 4×1 vector is approximately the same as the above corresponding 4×4 time.

4. MICROCODED HALF-WORD MATRIX-VECTOR MULTIPLICATION ROUTINES

As was noted above, the largest full word length, matrix-vector product microroutine within the capabilities of the 2100 was an $n \times 2$ matrix times a 2×1 vector. This was due to a shortage of scratch-pad registers for storage of vector elements. If the word lengths are halved to 8-bits (7 bits plus sign), then two vector elements can be stored in the same register and the product of a 4×4 matrix times a 4×1 vector come within the capabilities of the machine. Also observe that approximately two-thirds of the execution time for the above routines was consumed in the multiply subroutine. Since in a half-word routine 16-bit multiplies are not needed, but only 8-bit multiplies, an initialization of the counter to eight will effectively reduce multiply times by half. This results in a considerable savings of time. The flowcharts for an $n \times 2$ matrix times a 2×1 vector and a 4×4 matrix times a 4×1 vector are illustrated in figs. 6 and 7.

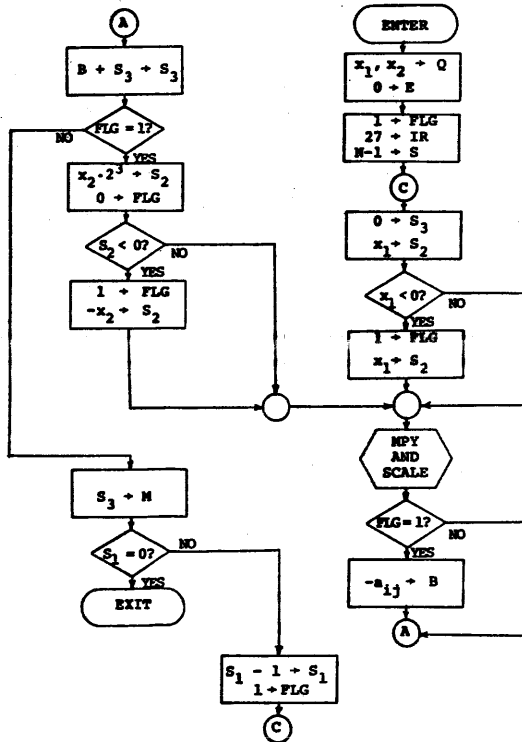


Fig. 6 - $n \times 2$ matrix times 2×1 vector, half-word.

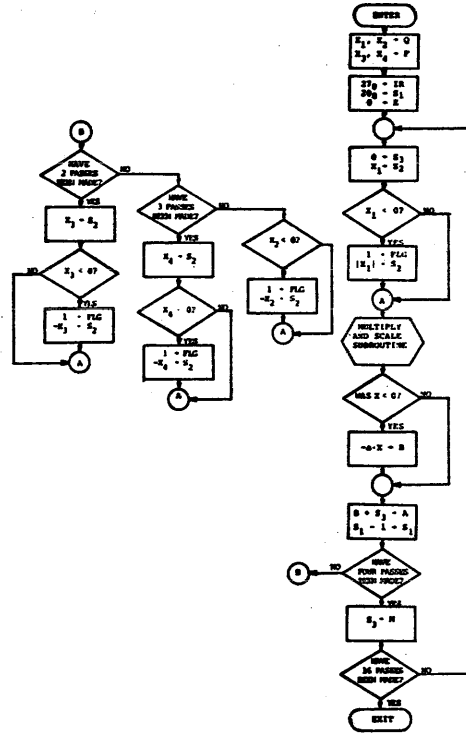


Fig. 7 - 4×4 matrix times 4×1 vector, half-word.

The algorithms for calculation of execution times for both worst case (all vector elements negative) and best case (no vector element negative) are given by eqs. (9) and (10), respectively.

Worse Case: $t_e = 7T + 107nT$
 Best Case: $t_e = 7T + 101nT$ (9)

Worse Case: $t_e = 5T + 4.5nT + 53.5n^2T + nT \sum_{i=1}^{n-2} (n-i) + 2nT \sum_{i=2}^{n-1} (n-i)$

Best Case: $t_e = 5T + 4nT + 49.5n^2T + nT \sum_{i=1}^{n-2} (n-i) + 2nT \sum_{i=2}^{n-1} (n-i)$, (10)

where n , T and i have the same meanings as in the previous equations. The respective times of execution for a 2×2 matrix times a 2×1 vector and a 4×4 matrix times a 4×1 vector are:

Worse Case: $t_e = 43.316 \mu s$
 Best Case: $t_e = 40.964 \mu s$ (11)

Worse Case: $t_e = 180.908 \mu s$
 Best Case: $t_e = 167.972 \mu s$ (12)

Equation 10 can also be used to calculate the time of execution for an 8×8 matrix times an 8×1 vector. These times are:

Worse Case: $t_e = 787.332 \mu s$
 Best Case: $t_e = 736.372 \mu s$ (13)

Even though considerable time is saved in these half-word, matrix-vector product microroutines, the savings is not as great as anticipated. This is due to the additional programming necessary to extract each vector element from the scratch-pad registers, to detect and account for negative vectors, and to scale each vector to accomplish the 8-bit multiply routine. But as the value of n increases (see eq. (9) and (10)), greater savings are realized. This becomes even more apparent by referring to fig. 8 where it is seen that as n increases, the lines depicting execution times versus matrix size for the assembly language and microcoded full- and half-word routines all diverge. As also shown, that in microcoding 8-bit word length routines, larger matrix-vector products can actually be implemented in the computer. This aspect is discussed more fully in the succeeding section on the architecture proposed to effectively implement these matrix-vector products.

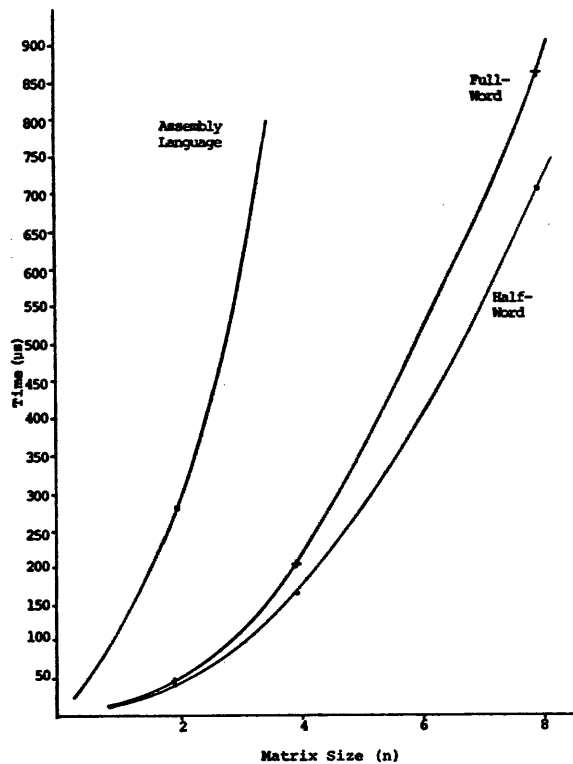


Fig. 8 - Time of execution vs. matrix size.

5. PROPOSED ARCHITECTURE

From the preceding discussion concerning implementation of microroutines for matrix-vector products, the limitations of the HP-2100 for efficiently microprogramming such problems are evident. A proposed architecture to alleviate these limitations and allow an efficient implementation of an $n \times m$ matrix times an $m \times 1$ vector is shown in fig. 9 and described

in the following paragraphs.

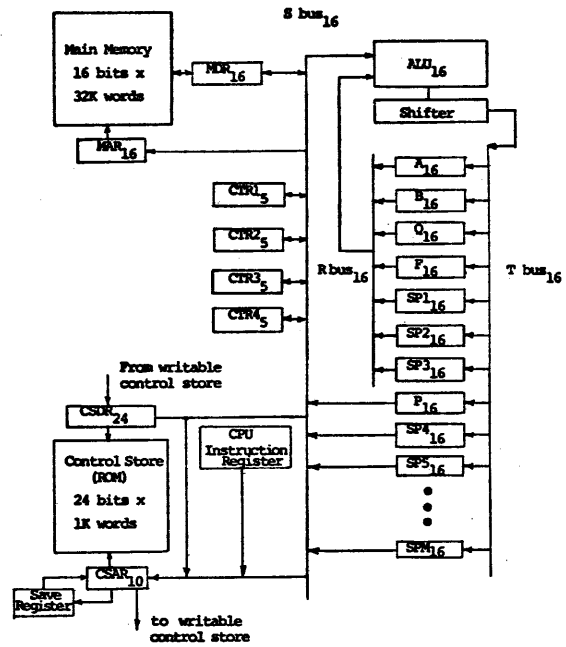


Fig. 9 - Proposed architecture for a microprogrammed machine.

Assuming that the bus structure and micro-instruction format remain fixed, the most severe limitation of the HP-2100 is a shortage of scratch-pad registers. By providing more scratch-pad registers, larger size matrix-vector products can be implemented, greater computational versatility results, and the execution speed of many programs can be increased considerably. To implement the $n \times m$ matrix times $m \times 1$ vector product, $m+7$ registers are needed. These $m+7$ registers are shown in fig. 9, where m scratch-pad registers are shown as "S-bus" registers and the other seven are shown as "R-bus" registers. Four of the "R-bus" registers are labeled as in the HP-2100, i.e., A, B, Q, F. Providing additional registers on the R-bus results in greater versatility, a reduction in number of micro-instructions to implement a given function, and hence, a reduction in execution time. All of these registers are also assumed to be general purpose registers and not latches as are the scratch-pad registers in the HP-2100. The scratch-pad registers in the HP-2100 are likely to develop a "race" condition if loaded while being interrogated. This limitation prevents the microprogrammer from specifying the same scratch-pad register in both the S- and T-bus (in the same microinstruction) which leads to more micro-instructions than necessary if these registers are made general purpose.

If it is desired to implement the half-word length (8-bit) version of this microroutine, the m scratch-pad registers could be made 8-bit registers. To efficiently incorporate these eight bit registers, the capability to detect the most significant bit (sign bit) as on or off must also be implemented. With these features added,

the half-word length microroutines can be much more efficiently executed since each eight bit vector would not have to be extracted from sixteen bit registers and a sign detection made.

Figure 9 also shows several additional five bit counters on the S-bus. These counters are not necessary to implement this matrix-vector product since the $m+7$ registers include the registers necessary for counting. The counters are shown to indicate that several of the $m+7$ registers may be replaced by shorter length (5 bit) registers to be used as counters.

Another limitation of the HP-2100 is that only microinstructions can be stored in and executed from WCS, that is no data can be directly accessed in WCS. The only data available in WCS is stored as eight bit constants in the least significant eight bits of microinstructions containing a "CR" or "CL" micro-order in the S-bus field. The "CR" and "CL" micro-orders direct the computer to read the eight bit constants stored in bits 0-7 of the microinstruction onto the least (CR) or most (CL) significant bits of the register specified in the T-bus field. The feature to read and store data directly into WCS locations by microprograms resident in WCS would greatly enhance the capabilities of the machine. However, the need for this feature is mitigated by the addition of sufficient scratch-pad registers; but if m is large, the cost of such registers might become prohibitive. So there is a trade-off here, either incorporating as many registers as necessary to implement a problem or incorporating several additional registers and adding the capability to read and store data into WCS. These features would significantly increase speed of execution of many microprograms and greatly enhance the computer's overall capabilities--not only for matrix-vector products but for a wide class of problems.

REFERENCES

- Agrawala, A. K., and Rauscher, T. G. "Microprogramming: Perspective and Status," IEEE Trans. Comp., Vol. C-23, No. 8, August 1974, pp. 817-837.
- Hewlett-Packard. 2100 Computer Microprogramming Guide, Hewlett-Packard Company, Cupertino, California, February 1972.
- Hewlett-Packard. 2100 Computer Microprogramming Software, Hewlett-Packard Company, Cupertino, California, September 1973.
- Hewlett-Packard. 2100A Computer Reference Manual, Hewlett-Packard Company, Cupertino, California, December 1971.
- Jones, Louise H. "A Survey of Current Work in Microprogramming," Computer, Vol. 8, No. 8, August 1975, pp. 33-38.
- Stewart, G. W. Introduction To Matrix Computations, Academic Press, New York, New York, 1973.

