MEMBERS—A Microprogrammed Experimental Machine with a Basic Executive for Real-time Systems*

J. K. Broadbent and G. F. Coulouris

Department of Computer Science and Statistics, Queen Mary College, Mile End Road, London E1 4NS

The aims and methods of a project investigating the direct interpretation of a high-level language by microprogram are outlined. Some features of a language for real-time applications specifically designed for interpretation by microprogram are described in the context of other related work. Factors involved in the use of microprogramming as a system development tool are discussed.

(Received November 1972)

1. Introduction

A conceivable goal for the designer of a computer system is to provide programmers of the system with a securely implemented programming interface at the highest level of expression compatible with the intended range of applications.

The secure implementation of high-level structures (i.e. data structures and functions that operate upon them) in a sufficiently general form to meet such a goal is costly in system performance when done by software and in development time when done by hardware. Nor is it easy for hardware and software development to co-exist on the same prototype. The increasing availability of microprogrammable logic computers that are competitive in cost with hard-wired machines of comparable performance offers the researcher or system designer a new strategy to demonstrate the feasibility and utility of his ideas.

Microprogrammable computers have facilities for two levels of program interpretation built in to their hardware. The higher level corresponds to the 'machine code' of conventional machines. However, an instruction in a machine code program is interpreted each time it is executed by another program written at lower, 'microprogram' level.

Microprograms normally reside in a special fast memory (called the 'control store') and exercise direct control over the units in the central processor of the computer. They can be designed so as to overlap many of their activities with delays that occur in transferring instructions and data to and from the main memory of the computer (see **Fig. 1**), and in more complex processors with delays in functional units.

Microprogramming was originally proposed (Wilkes, 1951) as a technique for the systematic implementation of machine code. It has more recently been demonstrated (Flynn, 1972; Weber, 1967) that microprograms can be used to interpret functions more complex than those found in conventional machine codes, resulting in faster execution of standard programming language functions and development of machine functions and data structures defined at a level more appropriate to user and system requirements.

A number of writers (Rosin, Freider, Eckhouse, 1972; Iliffe and May, 1972) have also noted that microprogramming offers a new technique for the investigation of untried computer system organisations. A computer with a writable control store offers the research worker and system designer the ability to evaluate a proposed system organisation at a cost that is low compared with the construction of specialised hardware.

While the crude processing power of an emulated computer system cannot be compared with a specialised hardware-based system at the higher end of the performance spectrum, for applications with relatively modest processing requirements, this limitation is often outweighed by other advantages. The primary benefit is the additional flexibility conferred by the ability to vary the definition for machine functions and data formats in the emulated machine. The availability of a writable control store may be seen as converting the tasks of logical checking and hardware debugging in hardwired machines into a fairly conventional program-debugging activity.

The secondary benefits include increased scope for the monitoring of behaviour in the emulated system through the addition of monitoring extensions to the emulation microprogram, and security of the microprograms against corruption by programs in the emulated machine.

Although there are clear benefits to the hardware manufacturer in the adoption of the simple, regular processor structure made possible by microprogrammed implementation, there is as yet little evidence that the benefits of microprogramming extend to improvements in the performance of a fully developed system. With few exceptions, the functions performed by microprograms could equally well be performed by hardwired logic. Most exceptions are based on the fact that functions of arbitrary complexity performing iterative operations such as vector sum and product, table sorting, etc. can be incorporated in the function set of an emulated machine with greater ease than the corresponding equivalent hardware extensions.

2. Hardware and supporting software

Our research is being carried out using an Interdata Model 4 computer, to which a 4K × 16 bit Writable Control Store has been attached. The configuration includes a 32K × 16 bit, 2·4 μsec memory, a drum backing store and a variety of peripherals. The microorder code of this computer is a simple 16-bit one allowing eight microinstruction types to be performed with operands held in 26 registers. Although not intended by its manufacturer for applications of this type, it has proved a useable, though not always convenient tool.

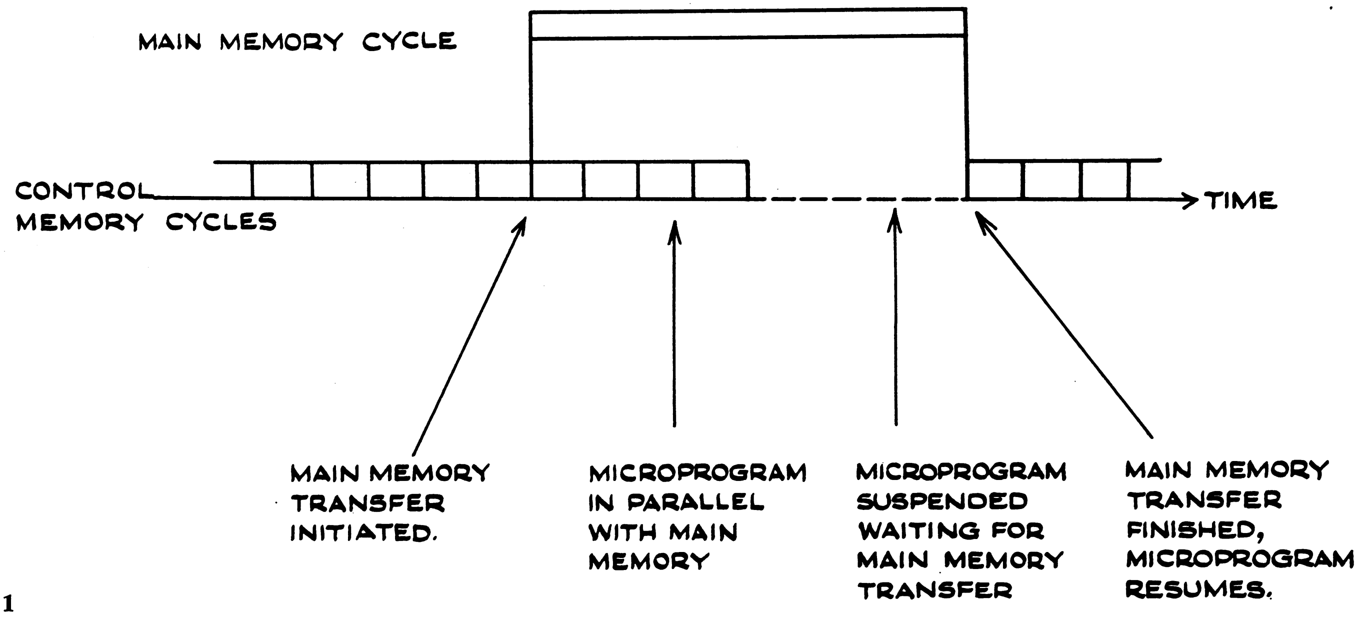
When later in this paper we give performance figures it should be remembered that the microexecution time is 400 nsecs for logical and register load operations and 800 nsecs for arithmetic operations. The machine is slow by current standards and the microinstruction set is oriented towards implementation of the standard Interdata machine code. Furthermore the 16 bit microinstruction, despite a high degree of encoding, is unable to allow general masking or control transfers at the microcode level in one microorder.

The development of microprograms is assisted by a number of software systems, including a microassembler, a microprocessor simulator, a debugging package and a specially-developed operating system with filing facilities and interactive command language (Cole, 1972). All of this supporting software is programmed in the standard Interdata assembler language, and can be executed using the standard Read Only Control Store in the Interdata 4 computer.

3. Real-time systems

We have chosen to investigate the design and implementation of programming systems for small and medium-scale real-time

*Originally presented at Datafair, Nottingham, April 1973



MAIN MEMORY CYCLE
CONTROL MEMORY CYCLES
TIME
MAIN MEMORY TRANSFER INITIATED.
MICROPROGRAM IN PARALLEL WITH MAIN MEMORY
MICROPROGRAM SUSPENDED WAITING FOR MAIN MEMORY TRANSFER
MAIN MEMORY TRANSFER FINISHED, MICROPROGRAM RESUMES.

Fig. 1

applications. We take the term 'real-time applications' to include as one of its most important elements the problem of designing and implementing operating systems controlling a number of asynchronous processes and/or devices. 'Real-time' also denotes those applications where the computing system is required to guarantee to a high degree of certainty that the resources to perform a given task in a given time can be made available. As a preliminary exercise, a small-scale survey of users of systems in these categories was conducted. This has been reported upon elsewhere (Coulouris and Broadbent, 1971).

4. High-level language implementation

There appear to be at least three distinct approaches emerging to the implementation of high-level languages on a microcoded processor. To some extent these reflect different approaches to the design of microcoded processors and are influenced by control store technology.

With increasing use being made of encoded microinstruction sets the differences between microcode and machinecode tend to be blurred especially when the microcode is interpreted by a further lower level of stored program steps (Rosin, Freider, Eckhouse, 1972). Flynn (1972) has proposed the use of a single micro machine code which then becomes the target code for compilers with the main memory as an extension to the control store. Such a solution can hardly find much favour with those compiler writers who complain that in general machines do not represent the structures required by high-level programming languages.

A second approach is to have separate target codes and microcoded interpreters for each programming language supported. Using a writable control store it would be possible to load an interpreter, at one extreme for an entire work shift or at the other extreme for each time slice on a system running several languages simultaneously.

The third approach is to define a system architecture at a high level incorporating the store allocation and stacking mechanisms required by most high-level languages and then to extend that interface for particular languages by introducing new data types, control structures and machine functions.

In MEMBERS we have adopted an emulated machine architecture that is designed primarily to support a particular programming interface within an environment that approximates to the second of the above approaches. It is intended however that by providing for a high degree of generality in the binding of program references, some scope for the use of the third approach will also be retained. The programming interface mentioned has been formulated as a language for programming the MEMBERS machine. The language is entitled FLUID. FLUID programs are expressed in a concise and relatively high-level syntactic form (e.g. at a similar level to POP-2 programs). The functions available in FLUID include all of the machine and system functions available in the MEMBERS system, rendering the use of lower-level programming languages unnecessary.

5. MEMBERS implementation

We shall now describe how microcode is being used in specific areas for implementation of MEMBERS and how its use allows us to take particular approaches to system design.

5.1. *Order code interpretation*

One of the earliest 'soft' uses of microprogramming was Weber's (1967) EULER translator and run-time system. Since then several APL interpreters have been microcoded (e.g. Hassitt, Lageschulte and Lyon, 1973). In such systems the essential advantages of amendment at statement level to a running program are retained. The MEMBERS interpreter is a compromise between a full interpreter in the software sense and the limited meaning attached to microcoding of a traditional order code. The result is that much structural information is retained in a useful and directly intelligible form for monitoring, diagnostics, etc. while allowing programs to be amended at run time by, for example, recompiling individual functions. The binding of a procedure or function's external references and the creation of its variables can be repeated on each application (i.e. procedure entry) so that any changes in name space are automatically incorporated. This binding convention is similar to that of POP-2 (Burstall and Popplestone, 1968), i.e. binding to the run time rather than the compile time environment.

Of course such methods impose an overhead but that overhead is reduced by use of microcode. The minimum time for function call is 30 microseconds with 20 extra microseconds for binding of each external reference when necessary.

The machine order code adopted is largely reverse Polish. Each instruction is only one byte long but can have certain operands such as constants, strings and switch vectors in following bytes. It may appear paradoxical to choose such a short machine code when frequently one of the advantages

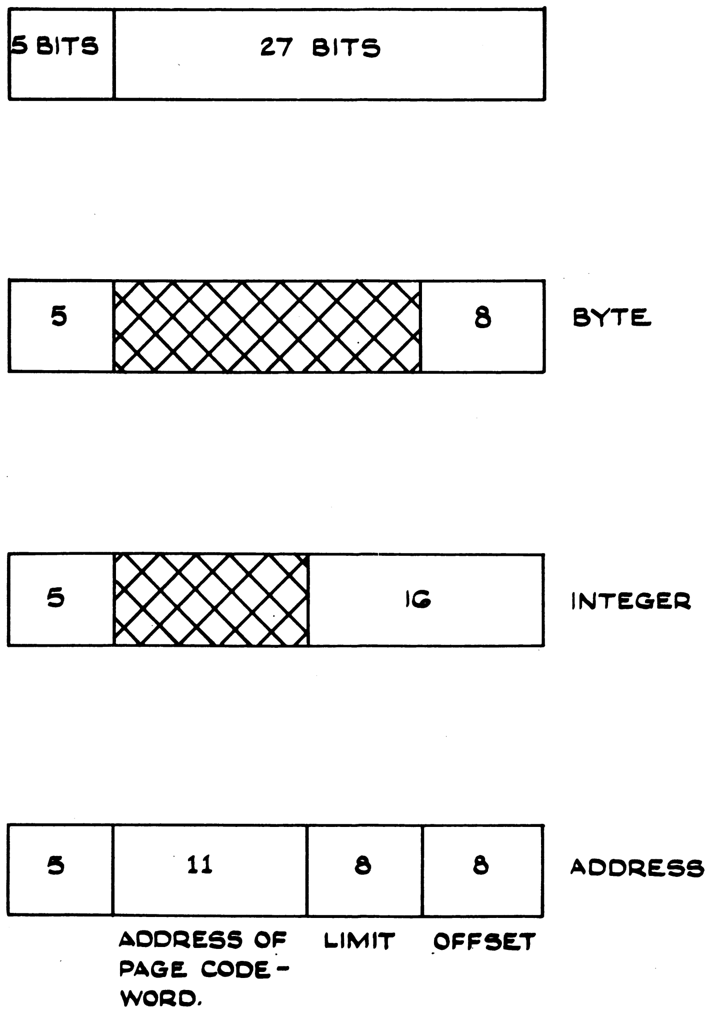


Fig. 2

claimed for microprogramming is the ability to implement complex machine instructions. However, the complexity in our case derives not from a diversity of operations but from the interpretation of relatively simple addressing and arithmetic operations on a variety of data types or structures, the checking of the validity of arguments for operations and the taking in line of processing costs associated with dynamic store allocation. In such a situation the decoding of a machine instruction is only a small part of the total operation, accounting for 2 microseconds out of a 20 microsecond operation. The positive advantage is the order code's compactness and when designing for a small computer system, compactness is an important design aim. Compactness is also particularly important in a paged system where a dense order code will obviously reduce the number of page faults in executing a program.

As examples we can give the timing and length of compiled code for certain language statements. A, B, C are locally declared named variables and assuming B, C to contain 16 bit integers we have

STATEMENT	LENGTH (BYTES)	TIME (μ SECONDS)
$A \leftarrow A + 1$	3	40
$A \leftarrow B + C$	4	80

The above figures are approximations based on counting microorders and exclude interrupt handling. Actual execution speeds can vary with the state of the stack. The orders generated for referencing externally declared variables are different and although the same amount of space is used they take slightly longer in execution.

5.2. *System monitoring*

The monitoring facilities being implemented in MEMBERS are for monitoring execution of functions and access to data structures in the target system.

Monitoring is initiated by calling a microcoded system function which marks the descriptor of the function or data structure to be monitored and links it to a monitoring routine. The monitoring routine can be either software or microcode, the user being completely free to define his own software monitoring routines if so desired.

5.3. *FLUID*

FLUID is both a command language in the sense that the user can type individual commands or paragraphs for immediate execution and a programming language in the usual sense that he can compile procedures for execution as and when required. In fact FLUID is designed to be the software programmer's lowest interface with the computer. Therefore all his communications to the system have to be processed by a FLUID translator. This in turn means that the FLUID translator is a critical item of software which should be as small and efficient as possible consistent with maintaining clarity in the programming language. Fortunately, it has been possible to design the MEMBERS machine code to facilitate this. The approach that has been adopted also includes minimising the diversity of syntactic forms in FLUID to such an extent that

$$1 \leftarrow 2;$$

is a syntactically correct assignment statement as would be

$$10(A, B, C);$$

a procedure call. Provided the left and right hand side of the assignment or the procedure name and parameters are syntactically correct as expressions then the respective FLUID commands will be syntactically correct. But because of the interpretive nature of the language these faults will be detected and fully diagnosed at run time. We would also argue that neither of these above 'errors' is a type likely to be made by a programmer with sufficient frequency to warrant compile time checks.

The result of evaluating an expression such as A + 1 or SIN (X/2) is a 32 bit element of which the first 5 bits is a tag and the remaining 27 bits an argument interpreted according to the tag. For example, in numeric elements only eight of the 27 bits are used for a byte and 16 for an integer but with addresses of store the 26 bits is broken down to contain address of page codeword (or descriptor), current offset from start of page and limit (**Fig. 2**).

Store is allocated in vectors and the data types allowed for a vector include bytes, integers, addresses and mixed. In the first two cases the vector is a sequence of 8 or 16 bit elements but when the value of one of these elements is the result of an expression it is converted to a 32 bit element with the correct tag. Conversely when a 32 bit tagged numeric element is assigned to a byte or integer vector element various coercion rules are followed and the result stored in the compact form without a tag. Vectors of addresses or mixed types are both sequences of 32 bit tagged elements.

The use of run-time checking gives several immediate benefits (as has been noted by Iliffe in the Basic Language experiment, (1972)).

1. Reduction in object program size for certain classes of application;
2. Localising of programming errors, particularly through dynamic array bound checking;
3. Improved diagnostic capabilities.

However it is becoming apparent that the retention of program structural information in a form which is distinguishable to the software and the microcoded interpreter could have many other advantages. We shall consider two here.

5.4. *Data types and parameter checking*

We have used a simple encoding scheme to represent data types, e.g.

[BY]	bytes
[VCIN]	vector of integers

These are actually expressions which when evaluated have a 32 bit tagged element as a result. A function is provided in the language whose result is the address of a new vector of store of length and type specified by parameter. For example the statement

```
BUFFER ← NEWVEC (100, [VCBY]);
```

would result in the address of a vector of 100 bytes being assigned to BUFFER. Depending on the size of vector required NEWVEC takes upwards from a best time of 20 microseconds to execute. Some time later we may wish to pass the value of BUFFER as a parameter. Now parameter type checking is something which all good programmers surely agree is necessary but few actually carry out consistently because it can be time consuming, and anyway what do you do if you get an error? The advantages of parameter type checking by the interpreter are that when errors do occur they are often monitored immediately as errors in the procedure call and become subject to either the user specified recovery procedures or the standard default diagnostics.

Ideally the parameter checking should include testing of upper and lower bounds and vector dimensions at procedure entry. What we are implementing is more limited and makes use of the type identifiers described previously. So in an example the function declaration might begin

```
FUNCTION LIST;
PARAM B[VCBY];
```

When the function is called the interpreter checks that the actual parameter is the address of a vector of bytes. Bound checking will of course occur automatically when the vector is actually referenced.

5.5. *Memory management*

A garbage collector is necessary in a dynamic store allocation system which neither imposes on nor requires of the programmer a discipline with respect to the allocation and release of memory segments. To perform garbage collection it is well known that the system must be able to distinguish between addresses and numbers in a program address space and obviously that is one reason for the tagging described earlier. But given that these distinctions have been introduced there are other ways in which store management can make use of the information available.

MEMBERS store management is not just concerned with garbage collection but also with integrating the core and drum into a conceptual one-level store by means of paging. Each process in the machine has a single stack for allocation of space for parameters and variables to functions in current execution and for expression evaluation. Therefore by scanning the stack looking for addresses a first approximation can be made to a list of pages being used by that process. More specifically by examining the top part of the stack allocated to the most recently called function a list of pages in current use can be obtained, i.e. by looking at the stack we can obtain a good idea of the process's working set. The scheduler can use this approach to run only those processes whose working set it believes to be in main store and perhaps more importantly to pre-fetch pages for processes which it wishes to run in due course.

6. Conclusions

The recent availability of relatively convenient facilities for the development of microprogrammed interpretive systems has stimulated widespread interest in the subject. We have reported here on progress in the design of a high-level programming system specifically intended for microprogrammed interpretation.

In our own environment, the presence of a suitable microprogrammable computer has led to a number of other related research activities, ranging from the simple emulation of existing machines to the interpretation of several high-level languages. All of these studies have indicated the importance of software aids and systematic techniques for microprogramming. Much work remains to be done in these areas.

The performance and usefulness of FLUID and other micro-interpreted systems remain to be evaluated. Results available at the time of writing suggest that even for the relatively complex machine architecture implied by FLUID, microprogrammed emulation can provide an implementation that is acceptable for many purposes. Further judgements must await a future report.

7. Acknowledgements

We should like to acknowledge the contributions made to the MEMBERS project by several post-graduate students and research staff, in particular, M. S. Cole, P. Macres, B. Neil and W. M. Newman have made substantial direct or indirect contributions to the work reported here.

The work reported here has been carried out with the support of a research grant from the Science Research Council (Ref. B/RG/500). J. K. Broadbent also wishes to express his gratitude for the support he has received from ICL Research Division and the SRC under the SRC Industrial Studentship scheme.

References

WILKES, M. V. (1951). *The best way to design an automatic calculating machine*. Manchester University Computer Inaugural Conference.

FLYNN, M. J. (1972). *International Advanced Summer Institute on Microprogramming*. Eds. Boulaye & Mermet, Publ. Herman, Paris.

WEBER, H. (1967). A Microprogrammed Implementation of EULER on IBM System/360 Model 30, *CACM*, September, Vol. 10, No. 9, pp. 549-558.

ROSIN, R. F., FREIDER, G., and ECKHOUSE, R. H., Jnr. (1972). An Environment for Research in Microprogramming and Emulation, *CACM*, August, Vol. 15, No. 8, pp. 748-760.

ILLIFFE, J. K., and MAY, J. (1972). Design of an Emulator for Computer Systems Research, *International Advanced Summer Institute on Microprogramming*, Eds. Boula e & Mermet, Publ. Herman, Paris.

COLE, M. S. (1972). An Operating System for Microprogram Development, *presented at the InterUniversity Computer Science Colloquium*, Edinburgh, September 1972. (See also MEMBERS Report 11, Department of Computer Science & Statistics, Queen Mary College).

COULOURIS, G. F., and BROADBENT, J. K. (1971). Some User Reactions to Operating Systems: a selective survey, *International Symposium on Operating System Techniques*, Queen's University, Belfast.

HASSITT, A., LAGESCHULTE, J. W., and LYON, L. E. (1973). Implementation of a High Level Language Machine, *CACM*, April, Vol. 16, No. 4, pp. 199-212.

BURSTALL, R. M., and POPPLESTONE, R. J. (1968). POP-2 Reference Manual, *Machine Intelligence 2*, Edinburgh, Oliver and Boyd.

ILIFFE, J. K. (1972). *Basic Machine Principles*, 2nd Edition, MacDonald.