# The MU5 name store

R. N. Ibbett and M. A. Husband

*Department of Computer Science, The University, Manchester M13 9PL*

**One of the most important concepts in MU5 is the use of a Name Store to hold the small number of frequently used named variables found in high level language programs.**

**This paper describes the implementation of this store as part of the MU5 processor and presents some results of measurements of its performance.**

(Received May 1976)

## 1. Introduction

The MU5 processor is designed around an order code which reflects the structure of high level languages (Kilburn, Morris, Rohl and Sumner, 1968). In these languages, operands can be divided into two major classes (*a*) named operands, which can be reals, integers, or descriptors of arrays, and (*b*) array elements. Statistics taken on the Manchester University Atlas computer have indicated that over a large range of programs 80% of operand accesses are for named operands and 20% for array elements. Furthermore, the number of different names used in a program is generally less than a hundred, and the number of names declared in a routine within a program is almost always less than 64. Thus, it is possible to keep these named operands in a small high speed store incorporated into the central processor and thereby avoid the long access time to the main store for a high percentage of accesses.

If this store had to be addressed explicitly by the programmer, it is unlikely that compilers would make full use of it and the full speed of the processor would not be realised for programs written in high level languages. If the store is addressed associatively, however, and forms part of a 'one level' store with the main store of the system, then it is invisible to the programmer and can operate equally efficiently for programs written in high level languages or coded directly.

To assess the size of store which would be required, six programs using large numbers of names were analysed and the complete patterns of name accesses for those programs recorded (Odeyemi, 1970). The operand access patterns were used to investigate the performance of simulated buffers of different sizes, and it was found that a 32 line buffer (equivalent to 256 bytes) almost completely trapped the named operands for these programs. This was only possible because of the definition within the order code of named quantities. With a more conventional order code such as that of the IBM 360, for example, where all operand accesses must be treated alike, and where, in fact, all store accesses for operands and instructions are treated alike, the same sort of efficiency is only obtained when a buffer of 16K bytes is used, as in the 'cache' memory of the Model 85 (Liptay, 1968). In practice the MU5 name store has been expanded to 52 lines (416 bytes), for reasons explained in the next section.

## 2. Name store hardware

The various activities involved in executing an order in the MU5 central processor are carried out by separate functional units (Ibbett, 1972). Instructions are accessed by the instruction buffer unit (IBU in **Fig. 1**) via the store access control unit (SAC) which is linked to the local store and to other stores via the exchange. The primary operand unit (PROP) accesses the operand specified directly by the instruction and sends it with the function to the *B*-Arithmetic (modifier) unit, to PROP itself for use as a variable, or to the secondary operand unit (SEOP) for use either as a descriptor specifying a secondary operand (e.g. an array element) or as a variable to be used within the SEOP or as a variable destined for the accumulator unit.

Both PROP and SEOP contain name stores. The PROP name store contains 32 lines, four of which are reserved for the operating system, leaving 28 available for user programs. This name store deals mainly with B/D names (i.e. named quantities used



**Fig. 1 The MU5 CPU**



Downloaded from https://academic.oup.com/comjnl/article/20/3/227/751714 by guest on 19 April 2024

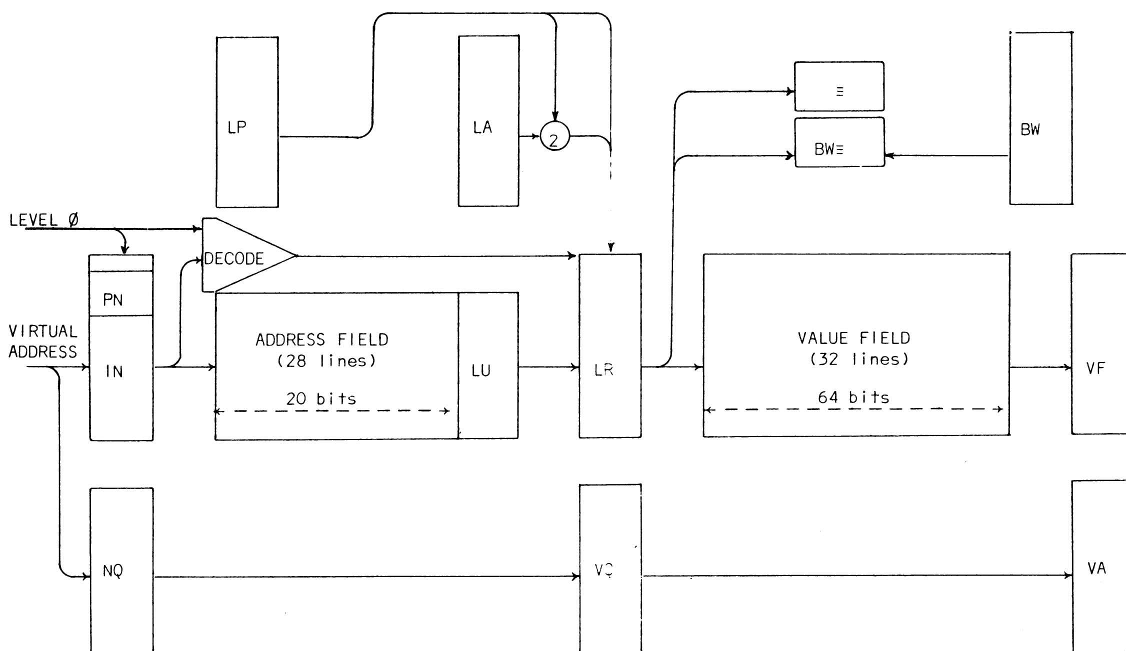

**Fig. 2 The basic PROP name store hardware**

as variables by units other than the accumulator unit, or used as descriptors), while the 24 line name store in SEOP deals mainly with *A* names (i.e. named quantities used as variables in the accumulator unit). Each line in these stores may contain either two adjacent single word operands (32 bits) or one double word operand (64 bits). Two separate name stores were provided in an attempt to overcome the 'Acc write back' problem as described below.

MU5 is a highly overlapped machine with up to 12 instructions at some stage of execution between the PROP name store and the accumulator unit. Thus, during the execution of the following sequence of instructions, which form the scalar product (SCALPROD) of two vector arrays *X*1 and *Y*1 (each of length LIM)

*L*1: ACC = *X*1[*B*]
ACC * *Y*1[*B*]
ACC + SCALPROD
ACC ⇒ SCALPROD
*B* CINC LIM :: compare *B* with LIM and increment *B*
IF < 0, → *L*1 :: if *B* < LIM, jump to *L*;

there are four instructions separating the order writing to the name SCALPROD and the order wishing to read it again. If SCALPROD were in the PROP name store, a long gap would be created since the ADD ('+') order would have to wait in PROP for its operand to be returned by the STORE ('⇒') order. By having a name store in the SEOP, the ADD order can proceed along the pipeline until it is within two stages of the accumulator unit before picking up its operand.

Each line of both name stores consists of two main parts, a 64-bit value register and an associatively addressed register containing the virtual address of the operand. In the PROP name store (**Fig. 2**) the address field is 20 bits wide—four bits for the process number, 15 for the address within a segment and one bit to distinguish the four operating system lines. The segment address is not included since all names are in the same segment (the name segment) within a process. The name store in the SEOP forms part of a larger operand buffer system (OBS) which is also used for array elements, and the 14 segment digits are therefore included in the address field of the OBS name store (Woods and Sumner, 1974). In addition to the address and value registers in each line, three special digits are used, one, line-in-use (LU), indicating whether the line contains valid information, the second indicating whether the contents of the value field have been overwritten by the action of a 'write to store' order, line altered (LA), and the third indicating which line of the name store is to be used when entering a new name into it, the line pointer (LP).

The normal action in either name store is for the virtual address of the required operand to be copied into an interrogate register (IN) and to be presented to the associative field. If the address is identical to an address in one of the associative registers, and the line is in use, an equivalence occurs and a corresponding digit is set in the line register (LR). The digit in the line register then accesses the appropriate register in the value field and the value of the operand is read out and copied into the value field register (VF). In parallel with this operation a check is made to determine whether an equivalence actually occurred in the associative field (this is done by OR'ing together the digits of the line register and testing for a '1'). The action taken if no equivalence is found depends upon the type of order requiring the operand and the various possibilities are described in Section 3.

In addition to the normal accessing and updating operations, the name stores have provision for certain special actions required to implement the paging system used to translate the process generated virtual addresses into real store addresses. Most of these actions are controlled by means of 'privileged operands' in the instruction set, i.e. a set of locations within the processor (known as V-store) accessible only to executive and supervisory software. The special actions occur when an order writes to a V-line. They include resetting the name store (i.e. setting all LU and LA bits to '0'), purging the name store (i.e. copying all altered lines back to local store and then setting all LU and LA bits to '0'), and searching the address field for a given page address. The PROP name store also incorporates a mechanism for allowing accesses made via descriptors to the name segment to check its contents. Descriptor addresses are generated at a later stage on the pipeline than the PROP name store but earlier than the OBS name store. Thus the latter can be checked without special action, whereas the former must be re-examined by an order which has already passed beyond it.



Downloaded from https://academic.oup.com/comjnl/article/20/3/227/547 by guest on 19 April 2024

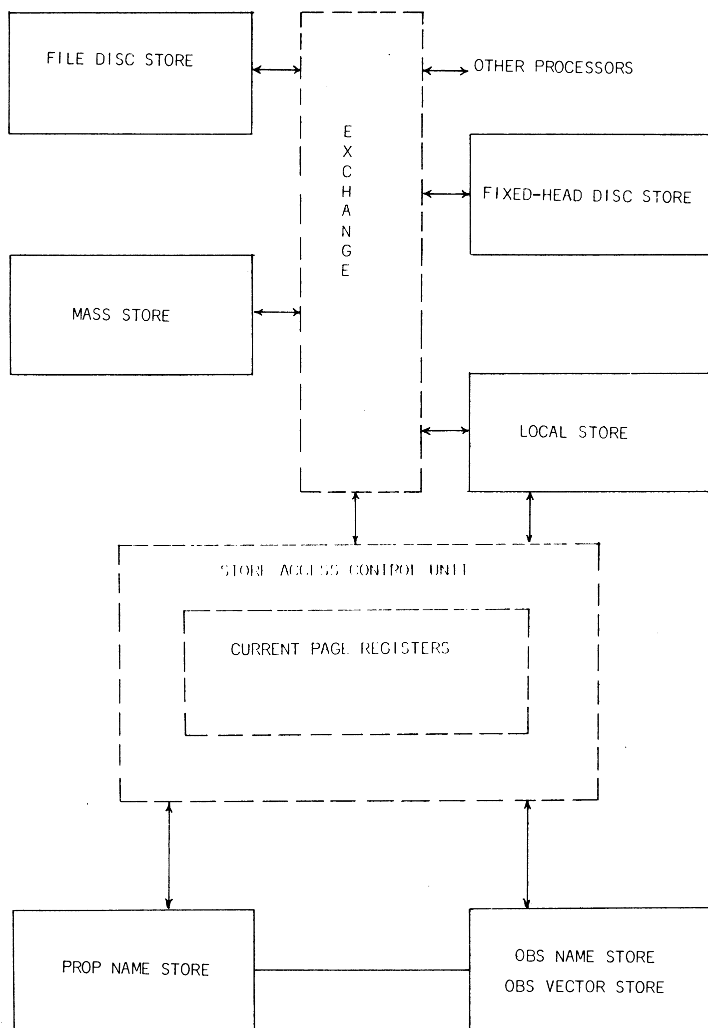


**Fig. 3 The MU5 storage hierarchy**

### 3. One level storage

The 'one level storage' concept developed for the Atlas Computer (Kilburn, Edwards, Lanigan and Sumner, 1962) has been extended in MU5 to cover four levels of real storage—the integrated circuit name stores, the 260 nsec. cycle time plated wire local store, a 2·5 μsec. ferrite core mass store, and a fixed head disc store (**Fig. 3**). Operand accesses for names are initially made to the name stores; if the required operand is not found, the name store must be updated by bringing a new word into it and discarding an old one. An access is therefore made, via a set of current page registers (CPRs) contained in SAC, to the local store. The CPRs translate the virtual address of the operand into the local store real address. If the CPRs indicate that the required operand is either not in the local store or is not currently accessible via a CPR (CPR $\neq$), then an interrupt is generated and the appropriate software is entered to update the CPRs and to organise the transfer of a page of information via the exchange between the mass or disc stores and the local store. Clearly, it would be inefficient to enter software to organise a one word transfer between the local store and a name store, so these transfers are organised by hardware alone.

In organising such a transfer, the hardware must decide which line to replace and must take into account the effects of store orders. To maintain the speed advantage of the name stores, store orders update the value of an operand in a name store but not that in the local store (*cf* 'store-through' in the IBM cache stores where both the buffer and the main store are updated together). Thus the new word required for one name store may already be in the other name store (and have a different value from the copy in the local store), and the old word may have to be written back to the local store before it is discarded. The decision concerning which line to replace requires the use of a replacement algorithm, and the effects of various replacement algorithms have been studied by simulation (Section 4). The algorithm actually incorporated is a simple cyclic one, implemented because it uses a minimum of additional hardware.

The detailed actions which take place when a non-equivalence (NEQ) is detected depend upon whether the order is destined for the accumulator unit or not and whether the required operand is already in the 'wrong' name store. If names were to be exchanged between name stores on each possible occasion, this would lead to inefficiencies. (Although a particular variable may not need to move between stores, a 32-bit variable used mainly by the *B*-arithmetic unit may be contained within the same 64-bit word as a variable used mainly by the accumulator unit, and the full word could be exchanged many times.) The number of occasions on which the exchanges are made is reduced by only allowing them to occur for orders which write to the operand concerned.

Thus for an accumulator order the normal situation is for a non-equivalence to occur in the PROP name store and an equivalence to occur in the OBS name store. If equivalence occurs in the PROP name store, however, then for a 'load' (i.e. non-store) order the operand is carried through from the PROP name store as if it were a literal, and no access is made to the OBS name store. If equivalence occurs for a store order the normal action of the PROP pipeline is inhibited and a hardware routine is entered which deletes the entry for the operand in the PROP name store, writing it back to the local store if necessary, i.e. if it has been altered by a previous store order whilst in the PROP name store. The pipeline is then restarted and when the order reaches OBS an access to its name store is made. This produces a non-equivalence and the operand is therefore accessed from the local store and written into the OBS name store.

When a PROP name store non-equivalence occurs for a non-accumulator order, the OBS name store must be checked before the operand is accessed from the local store. The non-equivalence is detected when the order has reached the value field register and the operand address is in the virtual address register (VA in Fig. 2). The normal action of the PROP pipeline is inhibited and a special order is created ahead of the order finding the non-equivalence. This order leaves PROP for SEOP and causes the OBS to access its name store. If the operand is not in this store then OBS makes an access to the local store via SAC on behalf of PROP, so that the operand, when available, will be returned directly to PROP.

If the operand is in the OBS name store it is returned to PROP via the normal internal highway used for operands returning to PROP from SEOP. For a 'load' order the operand is simply aligned with the order in PROP and the pipeline is restarted. For a 'store' order OBS deletes the entry in its name store and PROP updates its name store as for an order which found non-equivalence in the OBS name store.

Before a new value may be written into either name store, a line must be selected for the new operand. The actions taken in PROP and OBS are slightly different in this respect due to the different ways in which the two units access operands from local store: OBS must always have one free line in its name store while PROP is able to free a line during the time it is waiting for an operand to be returned from either OBS or SAC. The actions taken to free a line in the PROP name store and to bring in a new operand are described briefly below.

When a PROP non-equivalence is detected, the address of the operand causing the non-equivalence is in VA (Fig. 2), whilst the two succeeding orders have their addresses in VQ and NQ. The content of VA is therefore sent off to OBS and the content of the line pointer (LP) is copied into LR in order to select a

Downloaded from https://academic.oup.com/comjnl/article/20/3/227/751714 by guest on 19 April 2024

line for deletion. If this line is the target line for an outstanding 'B ⇒' order (LR ≡ BW), the content of LR is copied, with a one digit shift, into LP and back into LR to select the next line, whilst if the line selected has already been altered (as indicated by LA) the address and value are read out and used to update the local store. When the new operand is received by PROP, its address and value are written into their respective fields, the corresponding LU bit is set to '1' and the corresponding LA bit is set to '0' or '1' as appropriate. The content of LR is then copied back into LP, and, after the pipeline has been restored to its former state by recycling the contents of VQ and NQ through IN, the normal instruction execution sequence is restarted.

If all instructions find their operands in the correct name stores they may proceed, in the absence of other hold-ups, at the maximum rate of one instruction every 50 nsec. When a PROP name store non-equivalence occurs, however, a delay of about 1,200 nsec is introduced. Thus it is very important that a high hit rate be obtained in the PROP name store.

## 4. Replacement algorithms

In addition to software investigations carried out to assess the size of name stores required, software simulation of the affects of different replacement algorithms was also carried out (Khaja, 1972). A number of algorithms were tried and assessed using the following criteria:

(*a*) the replacement algorithm for PROP must be able to select a line for replacement within the time required to return the operand from OBS or SAC

(*b*) the replacement algorithm should use a minimum of hardware for its implementation

(*c*) the replacement algorithm should provide a greater increase in performance than that which would be gained by using the same amount of hardware to increase the size of the store.

Some of the algorithms studied were cyclic, random, current use, least recently used, and past unwanted. The cyclic algorithm is simplest to implement, and since it requires a minimum of additional hardware it serves as a standard by which to compare more complex algorithms. A random replacement algorithm tends to be slightly better than cyclic but not enough to warrant the extra hardware needed for its implementation. The current use algorithm makes use of one or more bits per line to indicate which lines are in current use. In the limit, one five bit counter is used for each line, producing the least recently used algorithm. Every time a line is used all counters are incremented and that for the currently used line is cleared. Consequently the line with the highest count is the least recently used, or conversely, not a member of the currently used set, and is therefore a candidate for replacement. Although this algorithm is significantly better than cyclic, it would have been so costly to implement in MU5 technology that a greater gain could have been achieved for the same cost by doubling the size of the PROP name store. In the past unwanted algorithm a currently unwanted register is used to record which lines were unwanted during an interval '*t*'. At the end of this interval, this register is copied into the past unwanted register and all lines not wanted in either interval are candidates for replacement.

This algorithm is not too costly to implement and, when '*t*' is chosen carefully, possibly dynamically, it yields significant improvement over a cyclic algorithm. Due to space limitations, however, a simple cyclic algorithm was implemented, but some appreciation of gains and costs of other algorithms was obtained.

**Table 1 Average values for name store hit rates and interactions**

| | | |
|---|---|---|
| | In either name store | 96·1% |
| | In correct name store | 86% |
| B/D | Name accesses | |
| | NEQs | 8·0% |
| | SAC. access | 2·9% |
| | OBS. read | 3·3% |
| | OBS. delete | 1·8% |
| A | Name accesses | |
| | NEQs | 5·6% |
| | SAC. access | 1·0% |
| | PROP. read | 2·8% |
| | PROP. delete | 1·8% |

## 5. Performance evaluation

A system performance monitor (M. A. Husband *et al.*, 1976) has been used to measure the efficiencies of the two name stores and the interactions between them for a set of 95 programs. This set of programs contains both FORTRAN and ALGOL jobs ranging in complexity from simple student jobs to large scientific programs. For most programs it is found that 80% (±5%) of operand accesses are for named variables, that fewer than 120 names are used in each of the programs, and that in every program 95% of name accesses are to fewer than 35% of the names used. These figures confirm the earlier Atlas results and suggest that high name store hit rates should be obtained. In fact, it is found that over 96% of name accesses find their operands in one or other name store. **Table 1** shows the average hit rates obtained as well as indications of the degree of interactions between the two name stores. It can be seen from this table, however, that although 96·1% of name accesses find their operands in one or other name store, only 86% of these accesses find their operands in the correct name store. Of the remainder, 3·9% require an access via SAC to the local store (2·9% + 1%), whilst 6·1% of accesses (3·3% + 2·8%) require the operand to be read from the wrong name store, and 3·6% of accesses (1·8% + 1·8%) require their operands to be deleted from one name store and transferred to the other.

Thus the performance of the processor as a whole is not as high as anticipated due to the comparatively high and largely unforeseen proportions of interactions between the two name stores. The main reason for this discrepancy is the way in which procedure calls are implemented in MU5. Parameters for procedures are stacked into the PROP name store and may subsequently be used as OBS names. Conversely, it is also possible for one procedure to use a particular word in the address space as an OBS name and for a subsequent procedure to use the same addressed location as a PROP name.

It has been demonstrated, therefore, that although the use of two name stores does not produce individual hit rates within each name store as high as anticipated, the name store concept itself is viable. Furthermore, simulations of single name stores, of varying widths and numbers of lines, have been performed using address traces produced on MU5 (Y. L. Husband, 1976), and these have demonstrated that hit rates in excess of 99% are obtainable using a single name store of only 512 bytes. Thus in the short term software techniques and simple hardware modifications which will reduce the amount of interaction between the two name stores in MU5 are being investigated, whilst in the long term it would seem that a single name store should be used in future machine designs, instead of the two name stores implemented in MU5, and that the 'ACC write-back' problem therefore be solved by some other means. A number of solutions are possible, each with its own advantages and disadvantages, but further investigation is required before any definite conclusions can be drawn.

**Acknowledgements**

The MU5 project has been supported by the SRC and ICL. The authors would like to thank all members of the MU5 team who have contributed to the work described here.

Downloaded from https://academic.oup.com/comjnl/article/20/3/227/751744 by guest on 19 April 2024

**References**

ODEYEMI, I. A. (1970). Experiments on Operand Buffer Stores, Ph.D. Thesis, University of Manchester.
LIPTAY, S. J. (1968). Structural Aspects of the System/360 Model 85 Part II: The Cache, *IBM Systems Journal*, Vol. 7, No. 1.
IBBETT, R. N. (1972). The MU5 Instruction Pipeline, *The Computer Journal*, Vol. 15, No. 1. Reproduced in *The Auerbach Annual—1973 Best Computer Papers*.
WOODS, J. V. and SUMNER, F. H. (1974). Operand Accessing in a Pipelined Computer System, *IEE Conference on Computers—Systems and Technology*, November 1974.
KILBURN, T., EDWARDS, D. B. G., LANIGAN, M. J., and SUMNER, F. H. (1962). One Level Storage System, *IEE Transactions on Computers*, Vol. EC-11, No. 2.
KILBURN, T., MORRIS, D., ROHL, J. S., and SUMNER, F. H. (1968). A System Design Proposal, IFIP Congress, Edinburgh, August 1968.
KHAJA, W. (1972). The Implementation of the Name Store and the Associated Replacement Algorithms in the MU5 Computer, Ph.D. Thesis, University of Manchester.
HUSBAND, M. A., IBBETT, R. N., and PHILLIPS, R. (1976). The MU5 Computer Monitoring System, The European Computing Congress, September 1976.
HUSBAND, Y. L. (1976). Operand Buffering in High Speed Computers, Ph.D. Thesis, University of Manchester.

# Book reviews

*Content Addressable Parallel Processors*, by Caxton C. Foster, 1976; 233 pages. (*Van Nostrand Reinhold*, £8·40)

This title arouses interest since it implies a high performance processor structure which is not conventional. Comments on the loose cover further encourage this view, but in reality the book deals with the basic structure, mode of operation and application of small content addressable memories (CAM). The first two areas are reasonably well presented, Chapters 2, 4 and 7 relating mainly to structure and Chapter 5 to algorithms for performing a variety of search, arithmetic and array operations. The chapter on applications tries to cover too wide a field and more detail on a smaller number of significantly different applications would have provided a more convincing story.

The two chapters discussing systems which have been designed and built are also disappointing in that no factual comparison in respect of performance or programming with a conventional machine has been provided. Some description of their structure is necessary but most of one chapter is cluttered with irrelevant detail such as the bit allocation for fields in the instruction word. Finally, the author's design featured in the last chapter is mainly concerned with the basic structure of a CAM and there is only a minimal attempt to relate it to a computer system, thus no useful system design criteria emerge.

Despite this criticism the book would be useful to computer scientists, computer engineers and users wishing to acquire some knowledge of CAM's.

B. G. EDWARDS (Manchester)

*Modern Factor Analysis*, 3rd Edition, by H. H. Harman, 1976; 487 pages. (*University of Chicago Press*, £12·60)

Harman's well known book on factor analysis now appears in a third, revised, edition. Compared with the second edition there have been considerable changes. In his preface the author states that the basic material remains essentially unchanged. To accommodate the new material two new chapters have been introduced and there has been considerable revision of the structure and content of the other chapters. In all, eight of the sixteen chapters contain new material. The extensive bibliography has been updated from the second edition and is very comprehensive. In particular, this reviewer applauds the change in the style of citation of the references in the main text from numerical to author date. The overall length has been kept to that of the second edition by some useful pruning; for example most of the hand calculations have been eliminated.

These changes have made a significant improvement on the second edition and this new edition should maintain the high reputation of Harman's book. It is perhaps inevitable that the text is not quite as up to date as the references and one might feel that the approach to numerical techniques is a little dated although appropriate references are made.

The mathematically inclined reader might prefer a more concise treatment. These are minor shortcomings and this book will be invaluable to anyone interested in factor analysis. It is pleasant to report that even the price does not seem too unreasonable for a book of such wide coverage and usefulness.

G. J. JANACEK (Norwich)

*Automata*, by David Hopkin and Barbara Moss, 1976; 170 pages. (*Macmillan Press*, £8·95 hard cover, £3·95 paper)

This slim volume covers the field of automata from finite state machines to Turing machines. It is liberally supplied with worked examples and exercises (but no solutions). I did not note any obvious typographical errors except for a glaring transposition (page 74) of nearly half a paragraph from a subsequent page.

The topics covered are nearly the same as those discussed in Marvin Minsky's *Computation—Finite and Infinite Machines*, but the general approach to the subject and style of presentation is very different. This book is an honours course text, compact and formal in style, where Minsky's book is much more suited to the solo reader.

The appendix on logic will enhance the value of the book as a work of reference for students who have followed the course, but mainly as an *aide-mémoire*, since 20 pages from truth tables to existential quantifiers is fairly compressed.

A second appendix with potted biographies of leading names mentioned in the text, adds a nice human touch. For example even though an alumnus of QCC, I never knew until now that George Boole died in Cork in 1864 as a result of lecturing in wet clothes.

H. R. A. TOWNSEND (Edinburgh)

Downloaded from https://academic.oup.com/comjnl/article/20/3/227/751714 by guest on 19 April 2024