MU-5

INFORMATION PROCESSING 77, B. GILCHRIST, EDITOR
© IFIP, NORTH-HOLLAND PUBLISHING COMPANY (1977)

ASSESSING THE POWER OF AN ORDER CODE

SIMON H. LAVINGTON
The University, Manchester, GB and The University of Ife
Nigeria

ALAN E. KNOWLES
The University
Manchester, GB

When designing a new computer system or evaluating the architecture of an existing one, there is often a need to obtain comparative performance figures for several machines. It is then necessary to analyse these overall performance measurements into contributing architectural factors, making due allowances for differences in technology etc. One of the factors of prime interest is the power of a computer's order code, which is an indication of the efficiency with which a given task can be done. This paper develops two parameters for measuring the power of an order code, and gives examples to verify their usefulness for assessing high-performance processors in a scientific environment.

1. INTRODUCTION

For a given technology, the designer of a high-performance computer tries to maximise the rate of obeying instructions and also the "power" of those instructions. The power of an order code is a relative attribute, indicating the efficiency with which a given task can be performed. In this paper, attention is given to high-performance processors ("order code processors") and such measurable factors in their performance as indicate the relative power of their order codes when obeying scientific programs. Within these stated constraints, it is required to devise a set of simple parameters which will allow comparison of order codes when a particular algorithm, coded in a particular high-level language, is run on several different processors.

An initial study of this problem would suggest the following three run time parameters, whose values should be minimised by "powerful" order codes:

(a) size of object code, in bits
(b) number of instruction-fetches from main store
(c) number of operand-fetches from main store.

These parameters could be measured for one or more benchmark programs which are thought to typify the scientific task under consideration. The results would be affected by the quality of the compilers, but this is true of any assessment based on the use of meaningful programs as benchmarks. The more serious difficulties of employing the above three parameters to compare order codes are that the second parameter is sensitive to other architectural features of high-performance machines, such as loop-catching and look-ahead pre-fetch, whilst the third parameter is highly sensitive to features such as the size of fast operand buffers. Another problem is the practical difficulty of measuring the second and third parameters for computers not fitted with hardware performance monitoring facilities. A possible solution is to replace the last two parameters by some simpler factor depending on elapsed run time, suitably normalised to account for technology differences between the computers whose order codes are being compared.

2. THE EFFECT OF TECHNOLOGY DIFFERENCES

The choice of a basis for achieving normalisation of performance with respect to hardware technology is not simple. Once again, changes in architectural emphasis between machines tend to make the situation more complex. This is illustrated for three computers in table 1, which shows possible normalising factors based on the operand fetch time from main store, the fixed-point add time, and the floating-point add time. The computers concerned are the Ferranti Atlas, designed at Manchester University (1), the Control Data Corporation CDC 7600 and the present Manchester University research machine MU5.(2-4) They employ radically different CPU technologies, i.e: germanium junction transistors, cordwood type modules using silicon transistors, and small-scale integration ECL respectively. All three machines were designed to give, amongst other things, high performance for scientific computing.

Table 1
Possible normalising factors

	TOTAL OPERAND FETCH TIME FROM MAIN STORE (1)	FIXED-POINT ADD TIME (2)	FLOATING-POINT ADD TIME (3)
ATLAS	1 750nS	1 520 nS	2 610 nS
MU5	600nS	50 nS	250 nS
CDC7600	220nS	27.5nS	27.5nS

	NORMALISED OPERAND FETCH FACTOR	NORMALISED FIXED-POINT ADD FACTOR	NORMALISED FLOATING-POINT ADD FACTOR
ATLAS	1	1	1
MU5	2.9	30.4	10.4
CDC7600	8.1	55.3	94.9

Notes:
(1) Includes address translation time on Atlas and MU5; assumes no store clashes.
(2) Assumes no B-modification (Atlas); assumes operands available in central registers or Name Stores (7600, MU5); the time is for a stream of consecutive ADD instructions.
(3) Assumes double B-modification (Atlas); assumes operands available in central registers or Name Stores (7600, MU5); the time is for a stream of consecutive ADD instructions.

For ease of comparison the raw figures in table 1 have also been expressed as factors relative to Atlas. It is on the basis of such factors that any normalisation of overall performance (e.g., elapsed run time) would be attempted. As may be seen, the factors for e.g., the CDC 7600 compared to Atlas exhibit a wide variation, ranging from about 8:1 to 95:1 and, though an instructive insight into relative design emphasis, lead to uncertainties about which particular factor to select when carrying out order code comparisons.

It is therefore required to devise an order code parameter which needs no normalising, whilst being easy to measure. An approximate fulfilment of this requirement is now discussed.

3. THE DEVELOPMENT OF A SUITABLE PARAMETER

An acceptable solution to the measurement problem is to replace parameters (b) and (c) of section 1 by a new one:

Number of run time instructions obeyed.

This parameter is easily obtained for computers having hardware instruction counters, provided that a distinction can be made between instructions obeyed in a particular user program and those obeyed by the operating system or other processes. Such distinction can be made, for example, for Atlas and MU5.

To illustrate the use of this parameter, a comparison may be made between the relative power of the Atlas and MU5 order codes, when used for general scientific problems expressed in Algol. It is assumed for simplicity that two benchmark programs are sufficient to typify the application area under discussion. These Algol programs are:

GAMM: the five Gamm scientific loops,(5) as coded in Algol.

QSORT: the algorithm "Quickersort" (6) as applied to a 10 000-element randomised array.

Both programs have been used in several previous computer evaluations, e.g. (7) and (8). Table 2 shows various factors obtained from running on Atlas (using the Chilton Algol compiler) and MU5 (using the May 1976 Algol compiler). These two compilers are similar in the scope they offer and in the man hours taken to write them. Neither makes any attempt at source program optimisation. The MU5 compiler gives automatic array-bound checking as standard, due to the sympathetic hardware design of the array descriptor manipulation unit. Table 2 gives the comparison between the two machines. Columns one and two are the parameters which should be numerically smaller for the more powerful order code. Column one is self-explanatory and is usually considered to be of less significance in comparing machines. As an aid to comparison, the ratios of Atlas: MU5 are given, from which these two parameters indicate that the MU5 order code is somewhere between 2.4 and 6.5 times more powerful than the Atlas order code for these particular Algol jobs. When five more Algol programs, taken from the Oxford Benchmark,(9) were measured, the Atlas:MU5 ratio for the number of instructions obeyed was found to vary between 1.5 and 3.3, with an overall average for all seven jobs of 2.4.

In support of the use of column 2 in table 2 for assessing the relative power of order codes, it may be shown that the ratio of instructions obeyed for two computers is also equivalent to the ratio of normalised elapsed run times, provided that the normalising factor is carefully chosen to reflect the actual instruction speed observed for the particular benchmark program under consideration. For MU5 it has been possible to measure the elapsed times and average instruction times to about 1% accuracy by a special hardware monitor,(10) and the equivalent Atlas figures have been calculated to within 5% from the instruction-counter information and a knowledge of the type of orders executed for both programs. The raw elapsed times are shown in column 3 and the normalised ones in column 5, with the normalising factor being derived from the actual average instruction times for each program as given in column 4. As may be seen, the ratios of normalised elapsed times (2.6 and 2.4) are equal to the ratios of instructions obeyed.

The two proposed parameters of static code size and number of instructions obeyed still have certain practical disadvantages. The first parameter is usually the easier to obtain -except for compilers which intermix constants and monitoring information with object crde -but it is of less significance when assessing order code power for most applications. The second parameter has proved a valuable indicator, but is an impracticable measurement to obtain for computers without hardware instruction counters (e.g. the CDC 7600).

4. ASSESSING THE NUMBER OF INSTRUCTIONS OBEYED

An approximate technique has been devised for obtaining the number of user instructions obeyed during the running of scientific programs. It may be used for computers without hardware instruction counters but with facilities for measuring elapsed run time. The method consists first of estimating the average instruction time for a computer when executing scientific jobs in general, and then noting the elapsed run time for a particular program of interest. Dividing the elapsed time by the average instruction time then yields the approximate number of instructions obeyed.

It is assumed that the actual programs of interest are available in either Algol or Fortran. For each language a modular synthetic benchmark has been developed by Curnow and Wichmann,(11) which attempts to reproduce statistically the source statement profile typically exhibited by a general scientific program. Each of these Algol and Fortran benchmarks consists of eight modules, each of less than ten source statements. The modules may

Table 2
A comparison of the power of the Atlas and MU5 order codes for two Algol programs

	STATIC CODE SIZE (BITS)	NUMBER OF INSTRUCTIONS OBEYED (RUN TIME)	RAW ELAPSED RUN TIME (SECONDS)	AVERAGE INSTRUCTION TIME (NSEC.)	NORMALISED ELAPSED TIME
GAMM:					
ATLAS	8 784	302 080 000	1 027.44	3 400	1 027.44
MU5	1 344	114 645 000	30.35	265	388.48
ATLAS:MU5	6.5:1	2.6:1	33.9:1	12.8:1	2.6:1
QSORT:					
ATLAS	26 880	163 400 000	504.40	3 090	504.40
MU5	6 272	67 769 000	24.35	359	209.41
ATLAS:MU5	4.3:1	2.4:1	20.7:1	8.6:1	2.4:1

be run individually with conveniently large repetition counts, so that measurements of the elapsed time for each module may readily be obtained. Since the modules are of a manageable size, even allowing for calls on standard functions such as Sine and Cosine, it is quite possible, from an object code printout, to count manually the precise number of run time instructions obeyed. Thus the average instruction time for each module, and for the two complete benchmarks, may be obtained. Assuming the Algol and Fortran versions of the benchmarks are statistically representative, this then gives working estimates of the average instruction times to be expected in any Algol or Fortran scientific program, when run on the computer being investigated.
The technique may be illustrated using tables 3 and 4, produced by running Algol and Fortran versions of the Curnow/Wichmann benchmark on the CDC 7600 (12) and MU5 computers. The compilers used were the May 1976 versions for MU5 (no optimisation and full array-bound check), and Algol 4.1 (level 5F compiler, 5D run time system) and FTN4.5 (level 420 compiler, 406A run time system) with full optimisation (level 5 and level 2 respectively) and no array-bound checking, for the CDC 7600. (It may be noted that the MU5 compilers were produced on a very short timescale, and interface with a common Compiler Target Language.) The individual benchmark modules represent the following general constructs:

module 2: array elements
module 3: array as parameter
module 4: conditional jumps
module 6: integer arithmetic
module 7: trigonometric functions
module 8: procedure calls
module 9: array references
module 11: other standard functions

(Note that modules 1, 5 and 10 have been given zero weighting by Wichmann.)

As may be seen in tables 3 and 4, individual instruction times for MU5 and the CDC 7600 vary considerably, indicating considerable differences in hardware architecture and compiler strategy. The overall average instruction times for Algol and Fortran programs may be taken as 95 and 108 nanoseconds for the CDC 7600, and 348 and 374 nanoseconds for MU5. Using these, the actual number of instructions obeyed at run time may be estimated for any particular Algol or Fortran program, run on the CDC 7600 or MU5, for which only the elapsed run time is known. In particular the power of the CDC 7600 order code may be assessed, even though no hardware instruction counter exists for this machine. As an aside, it may be seen from tables 3 and 4 that the MU5 order code appears to be over four times as powerful for Algol whilst being slightly less powerful for Fortran, using the compilers previously specified. These figures have been confirmed for practical benchmark programs.

Table 3

Algol synthetic modules

	MODULE 2	MODULE 3	MODULE 4	MODULE 6	MODULE 7	MODULE 8	MODULE 9	MODULE 11	WHOLE PROGRAM
CDC 7600:									
MILLIONS OF INSTR. OBEYED	1.61	23.0	16.0	17.9	21.0	249.0	182.0	19.1	529.0
ELAPSED RUN TIME, SECONDS	0.25	2.32	1.85	1.93	2.13	22.9	16.3	1.85	50.1
AVERAGE ORDER TIME, NSECS.	155	101	115	108	102	92	89	96	95
MU5:									
MILLIONS OF INSTR. OBEYED	0.68	6.98	7.08	9.25	14.9	35.1	20.9	15.7	110.0
ELAPSED RUN TIME, SECONDS	0.13	1.17	1.91	1.80	6.81	12.9	6.68	7.08	38.5
AVERAGE ORDER TIME, NSECS.	187	167	269	194	458	366	319	450	348

Table 4

Fortran synthetic modules

	MODULE 2	MODULE 3	MODULE 4	MODULE 6	MODULE 7	MODULE 8	MODULE 9	MODULE 11	WHOLE PROGRAM
CDC 7600:									
MILLIONS OF INSTR. OBEYED	0.46	4.34	6.78	4.62	24.0	29.7	14.2	15.2	99.1
ELAPSED RUN TIME, SECONDS	0.06	0.40	0.49	0.27	2.18	3.76	1.82	1.45	10.7
AVERAGE ORDER TIME, NSECS.	132	92	73	58	91	127	128	95	108
MU5:									
MILLIONS OF INSTR. OBEYED	0.50	3.81	7.42	8.41	14.9	45.9	16.0	15.8	113.0
ELAPSED RUN TIME, SECONDS	0.09	0.98	1.86	1.58	6.86	14.1	9.60	7.13	42.2
AVERAGE ORDER TIME, NSECS.	185	256	251	188	461	307	599	451	374

5. VERIFICATION OF METHOD

The foregoing method depends very much on the reliability of the Curnow/Wichmann synthetic benchmarks. In order to assess this, 8 Algol and 14 Fortran programs were run on one computer (MU5) and the resulting 22 average order times were measured independently using a hardware monitor incorporating a crystal clock. The programs were taken from actual work. Table 5 shows that the order times vary considerably from program to program, with straight averages of 377 and 390 nanoseconds respectively for Algol and Fortran. This averaging assumes that each program is equally significant as a contribution to a general scientific benchmark -an assumption which is of necessity only an approximation. Thus the MU5 Algol and Fortran estimates of 348 and 374 nanoseconds given in tables 3 and 4 are quite reasonable, and may be taken as acceptable working values within the limits of conventional benchmark practice.

Table 5
MU5 average order times (observed)

ALGOL PROGRAM	AV. ORDER TIME, NSECS.	FORTRAN PROGRAM	AV. ORDER TIME, NSECS.
A1	395	F1	169
A2	497	F2	426
A3	233	F3	331
A4	486	F4	389
A5	245	F5	555
A6	538	F6	265
A7	359	F7	401
A8	265	F8	359
overall		F9	403
average:	377	F10	404
		F11	527
		F12	427
		F13	246
		F14	556
		overall	
		average:	390

See Appendix for the source of these programs.

6. CONCLUSIONS

For general scientific programs, two parameters have been developed to measure the relative power of a computer's order code. These are:

size of object code, in bits
number of run time instructions obeyed.

The second parameter, which can be shown to be equivalent to the normalised elapsed run time, is usually the more significant and the more difficult to measure. A method has been devised for estimating this second parameter for the case of computers having no hardware means for counting instructions. This method depends on the reliability of a modular synthetic benchmark, which has been tested against results for a number of actual scientific programs and found to give acceptable accuracy. The technique is currently being used in a design exercise aimed at developing a new architecture for high-performance computers.

ACKNOWLEDGMENTS

The authors wish to acknowledge the help of Professor T.Kilburn and the whole of the MU5 design team in obtaining the data for this work. Particular mention should be made of Dr.R.N. Ibbett, Dr.P.C. Capon, Dr.J.A. Linn and Mr.A.M. Addyman. The cooperation of Dr.B.A. Wichmann of the National Physical Laboratories is also acknowledged. The MU5 project is supported by the Science Research Council.

APPENDIX

The origin of the Algol and Fortran programs appearing in table 5 is as follows:

A1: program A4 of Oxford University benchmark(9)
A2: program A12 " " "
A3: program A24 " " "
A4: program A25 " " "
A5: program A26 " " "
A6: the chess program produced at the Science Research Council's Atlas Lab. (13)
A7: the National Physical Labs. Quickersort program (see section 3).
A8: the National Physical Labs. Gamm program (see section 3).
F1: the Science Research Council's Atlas Lab. benchmark ATMOL, devised by P.E. Bryant.
F2: program EXI8A of London Univ. benchmark (14)
F3: program GAMMA " " "
F4: program BESSEL " " "
F5: program LSQ " " "
F6: program HEIGHTH " " "
F7: program ODE " " "
F8: program SC " " "
F9: program ERFREQ " " "
F10: program MULLER " " "
F11: program TSU I/O " " "
F12: program EX15 " " "
F13: the GAMM program of A8, transposed to Fortran
F14: the simulation of the SM104 minicomputer .(15)

REFERENCES

(1) T.Kilburn, D.B.G.Edwards, M.J.Lanigan and F.H.Sumner, One-level storage system, I.R.E. Transactions, vol. EC-11, April 1969, 223-235.
(2) T.Kilburn, D.Morris, J.S.Rohl and F.H.Sumner, A system design proposal, Proc. IFIP Congress 1968, D76 - D80.
(3) D.J.Kinniment and D.B.G.Edwards, Circuit technology in a large computer system. Radio and Electronic Engineer, vol. 43, no. 7, July 1973, 435 - 441.
(4) R.N.Ibbett, The MU5 instruction pipeline, Computer Journal, vol. 15, no. 1, Feb. 1972, 42 - 50.
(5) J.Heinhold and F.L.Bauer eds., Fachbegriffe der Programmierungstechnik, Ausgearbeitet vom Fachausschutz Programmieren der Gesellschaft für Angewandte Mathematik und Mechanik (GAMM), Oldenburg, München, 1972, (German).
(6) R.S.Scowen, Quickersort, algorithm 271, Communications of the ACM, vol. 8, no. 11, 1965, 669.
(7) B.A.Wichmann, Algol 60 compilation and assessment , Academic Press, 1973.
(8) P.Verstege and B.A.Wichmann, Experimental data-base for computer performance evaluation. NPL report NAC62, May 1975.
(9) C.H.Cheetham, Oxford University benchmark test, report, Oxford University, Dec. 1969.
(10) M.A.Husband, R.N.Ibbett and R.Phillips, The MU5 computer monitoring system, Proceedings of the European Computing Conference on Computer Performance Evaluation, Sept. 1976, 17 - 28.
(11) H.J.Curnow and B.A.Wichmann, A synthetic benchmark, Computer Journal, vol. 19, no.1, 1976, 43 - 49.
(12) A.M.Addyman, Run time instruction statistics for the Wichmann benchmark on the CDC 7600, Internal document, Manchester University,1976.
(13) A.G.Bell, P.J.Hallowell and D.H.Long, A Universal benchmark?, Software Practice and Experience, vol.3, 1973, 355 - 357.
(14) P.H.Hughes, University computer benchmark report: Atlas/6600/1108, University of London Atlas Computing Service Report, August 1967.
(15) S.H.Lavington, Processor Architecture, NCC Books, 1976.

OPERAND ACCESSING IN THE MU5 COMPUTER

F.H. Sumner
Department of Computer Science
University of Manchester

In the design of modern high performance computers one of the major problems is that of obtaining instructions and operands from the store at a rate sufficient to match the extremely high speeds of the logic in the central processor and the arithmetic units. The development of this problem can be very clearly seen if we consider the operations needed to be obeyed to access the ith element of a vector A and to add it to a central accumulator. The Atlas Computer, designed in the late 1950's, has a one address code with two modifier fields. The operations are achieved by the two instructions shown in Fig. 1, which also shows the times for the individual stages, the first column being the times for the Atlas system. The starred items are the four accesses to the store and these require 7.2 μs of the total elapsed time of 11 μs. By overlapping the different operations the Atlas can in fact obey a string of orders of this type at a rate of approximately 4 μs for the pair, that is an overlap factor of approximately 3. When we came to the design of our present computer the MU5 in the late 1960's the speeds of the logic and the stores had increased by a factor of 8 over those of Atlas. The access time to the store is made up of many parts, as listed in Fig. 2, and the overall increase in access time is only a factor of 4. The actual store access is only 145 ns out of a total access of 520 ns, of which 150 ns is accounted for by the time taken for information to travel between the central machine and the store. The times taken for the operation required for the accessing and addition of a(i) to the accumulator are given in the second column of Fig. 1. The total is now 2760 ns, of which 2080 ns are required for the four store accesses. It was one of the design aims of the MU5 to obey repeated operations of this type at a rate of about 150 ns for the two operations. This would require an overlap factor of approximately 18, which is probably impossible.

In computers such as the CDC 6600 and CDC 7600 the technique for matching the high speed available in the logic is to have a small number of explicitly addressed central registers which contain the operands and which therefore reduce the number of accesses to the main store. The attainment of the maximum possible speeds in such register systems is very dependent upon putting the right operands in the registers at the right time. This can be accomplished by careful hand coding but is far more difficult for a high level language compiler. The compiler writer would prefer his operands to be in a single store of effectively infinite length. This in turn implies a large core store which is expensive and not very fast. All the operands would be brought from this store as required, and we have seen above the effect of this type of organisation. This situation of conflicting designs - on the one hand the high speed engineer's machine which is difficult to program, and on the other hand the compiler writer's machine which is difficult to make fast - has been getting more acute as computers have become more complex and more use has been made of high level languages. In the design of the MU5 we have attempted to bring the two approaches together. We began this process by studying the behaviour of the Atlas system when obeying programs written in high level languages. In these languages there are basically two classes of operand - named operands, which can be reals, integers or descriptors of arrays, in the one class, and array elements in the other class. One of the major languages in use on the Atlas is Atlas Autocode. This is an Algol-like language written by members of the Department, and we modified the compiler for this language so that it would produce statistics on the run-time usage of the different types of operand. By monitoring the accesses to the operands in some hundred different programs we found that 80 per cent of the accesses were for named operands and 20 per cent of the accesses were for array elements. If this is rather difficult to believe, consider the very simple expression A = B(i), that is taking the value of an element of a one dimensional array and putting it into a named real quantity. To obey this expression we would require three

accesses to names for A, i and the array name of B and only one array access for the actual element of the array. Having established the two classes of operand which are to be brought from the store, and having determined the relative number of accesses for the different types, the next problem is to determine how many different names are in use in a program. Again by a study of the programs submitted to the Atlas Computing Service the number of names used and the number of names declared in a program was, generally speaking, under a hundred, and the number of names declared in any routine was nearly always less than 64. This is the number of names declared statically at the beginning of the routine and the number of names in dynamic use in any area of the routine will be less than this. There is therefore the possibility of keeping these named operands in a small store close to the central processor and therefore of obtaining very fast access to such operands.

Some of the basic design features of the MU5 are summarised in Fig. 3. It is expected that there will be more than 16 processors in the system at any time, but that not more than 16 of these will be active. The organisation of the segmented store permits a process to have private, shared or common segments, and these have access control for read, write and obey. The three classes of object which need to be brought from the addressable memory are instructions, named quantities and array elements. These can therefore be placed in distinct segments and the mode of access and the management of the different segments can reflect the type of object that they contain. For example, the instructions are in pure procedures and are held in obey only segments. The method of accessing instructions has been described in a previous lecture to IRIA (reference
For the purposes of this lecture it will be assumed that a continuous sequence of the correct instructions arrives at the central processor. The named operands can generally be kept in one segment (maximum size 64K) and the organisation of this segment is shown in Fig. 4. There are three central registers associated with the accessing of operands in this segment: these are the stack front register (SF) which contains the address of the stack

front, the name base register (NB) which contains the address of the first named variable of the current routine and the extra named base register (XNB) which can be set to point to the first named variable of any routine. The directory is accessed by using names relative to a dummy name base of zero.

The address development of named and stacked quantities is shown in Fig. 5. In general terms there are two groups of named operands - integers and array descriptors on the one hand, which are required to evaluate the addresses of array elements, and reals which go directly to the main arithmetic unit (floating point, fixed point or decimal). A small 32-line associative store is provided to trap the frequently used integer and array names. This is shown in Fig. 6. The performance of the final system will depend critically on the hit rate achieved in this store. In order to gain some idea as to the correctness of the design four Atlas Autocode and two Fortran programs were taken from the University computing service, programs using a large number of names were chosen, and the complete pattern of name accesses for these programs was determined and stored on a set of magnetic tapes. These operand access patterns were then used to investigate the performance of buffers of different size and design. The variation of hit rate with number of lines for one program is shown in Fig. 7, which also lists the hit rates for a 32-line buffer for the six programs considered. All the Atlas Autocode programs had hit rates in access of 99 per cent for a store as small as 24 lines. Experiments on replacement algorithms suggested that a simple cyclic system was sufficient and that any money or space spent on more elaborate algorithms would have been better used in increasing the length of the store. These operands, which the system almost completely traps in the store, are defined quite naturally by the compiler in the production of the object code. This is due to the close similarity between the structure of the high level language program and the structure of the hardware. For the Atlas Autocode programs the integer and array names represent some 50 per cent of the total names, that is 40 per cent of all operand accesses. The figures for the Fortran programs were 30 per cent and 25 per cent respectively. The

importance of a high hit rate for this buffer is clearly shown by the rate at which this part of the central processor can execute instructions. If all operands required are in the store, the rate is one instruction every 40 ns, but if an operand has to be brought from the main store, a gap of some 800 ns occurs in the instruction pipeline.

So far only the accessing of named operands has been considered. The development of an address of an array quantity is illustrated in Fig. 8. Consider the accessing of an element of a one dimensional array a(i + j + 3), 3 B orders are required using the named integers i and j and the literal 3 to compute the value of i + j + 3 in the B register. Then an order is obeyed in which the N field defines the 64 bit descriptor of the array a and the k field indicates the action to be taken, the F field defines the function to be performed on the array element when it has been accessed. The 64 bit descriptor contains the origin of the array, the type of element and the bound of the array. The descriptor arithmetic unit checks that the displacement i + j + 3 in register B is within range then forms the address by adding B to the origin and accesses the store for the operand. On obtaining the value of the array element this is sent together with the function to the appropriate arithmetic unit.

Some array elements are indices or array descriptors, but the vast majority are floating point numbers, decimal numbers or characters and as such are processed in the main arithmetic unit. The following description refers to that group of array elements and the other operands which go to the main arithmetic unit.

The BD name store and the BD arithmetic units are collectively referred to the primary operand unit or PROP. The PROP obeys some orders completely but for others the function and either the operand or the operand address are passed to the secondary operand unit, the SE OP. This is shown in Fig. 9. The function passes via an eight stage queue or pipeline to the arithmetic unit (referred to in the Figure as ACC). This permits up to 8 functions to be queued up waiting for operation in the arithmetic unit, and whilst the function is moving down the queue there will probably be

sufficient time to obtain the required operand from the store. There is thus far less need to try to trap these operands in fast buffers. However, eight lines of buffer are required for each type of operand, array, name or literal, as there could be 8 functions in the queue each with the same type of operand, and there has to be somewhere for each of these to be stored as the arithmetic unit could be busy for sufficient time for all 8 operands to be brought from the store. Any operands which can be trapped in buffers in the SE OP will be an advantage as this will reduce accesses to the store and will also decrease the chances of delays in operands being supplied to the arithmetic unit.

The final choice of buffers is shown in Fig. 9. Any literal operand goes to one of the 8 lines of literal buffer and the tag generator forms a pointer which joins the function in the queue and is later used to select the operand for transmission to the arithmetic unit.

The virtual address of an array element is presented to an 8 line associative store and if equivalence occurs the tag goes to the queue. If not equivalence occurs one of the 8 lines of the 128 bit wide buffer is selected. The tag is sent to the queue and the tag and virtual address are sent to the store. The required operand together with the other half of the 128 bit word is brought from the store and placed in the buffer. This buffer is two words wide as the main store is also two words wide and it is economical to bring two words if there is a reasonable chance of the second being used. This is quite likely for array quantities which tend to be scanned sequentially. Traces of the array operand accesses for the six programs referred to above indicate that this small buffer has a hit rate of approximately 60 per cent.

For named real quantities the action is the same except that only single words are brought from the store, this buffer was made as big as possible, regard being paid to both economic and space considerations. 24 lines were the most that it was possible to include and with this configuration the hit rates for the named real operands were 99.07%, 87.06%, 99.71% and 99.77% for the four AA programs, and 73.28% and 72.48% for the two Fortran programs.

These hit rates are very satisfactory for such a small buffer and in this situation misses do not necessarily incur a penalty as the function is quite likely to be delayed in the queue whilst the arithmetic unit is busy.

For the Atlas Autocode programs the operands trapped in the buffers are over 99% of the names and 60% of the array elements, that is 92% of all operand accesses. This is quite a good hit rate as the total buffer size is only 576 bytes. Furthermore, those operands which must be accessed quickly to maintain a good overlap in the pipeline are virtually all trapped in the buffers. For the Fortran programs the overall figure is 80%, but again nearly all of the essential operands are trapped in the BD buffer.

The restriction of array elements to a particular buffer should permit the using of a buffer which makes considerable use of the sequential mode of addressing of most arrays. A range of such buffers was studied with the traces of the six programs, and whilst it was not possible to include any of these designs in the current version of the MU5 it is of interest to note that hit rates in excess of 90% were obtained, on a 16 line two word wide buffer in which a system of predictive loading was employed. This is a further proof of the value of permitting the structure of the hardware to match the structure of the data of the programs being obeyed.

May 1971

101, 1, 0. ADDRESS OF i

320, 1, 0, ADDRESS OF ORIGIN OF a

ALTER CONTROL	0·3	80
*ACCESS FIRST INSTRUCTION	1·8	520
DECODE & FORM OPERAND ADDRESS	0·5	120
*ACCESS FIRST OPERAND	1·8	520
LOAD INTO B1	0·4	80
ALTER CONTROL	0·3	80
*ACCESS SECOND INSTRUCTION	1·8	520
DECODE & FORM OPERAND ADDRESS	1·1	200
*ACCESS SECOND OPERAND	1·8	520
ADD OPERAND TO ACCUMULATOR	1·2	120
	11·0 μsec	2,760 ns
	*7·2 μsec	2,080 ns

Fig 1 ADDITION OF $a(i)$ TO ACCUMULATOR

CABLE	40
PRIORITY LOGIC	30
ASSOCIATIVE	50
READ	50
CONCATENATE	20
CABLE	28
BUFFER ADDRESS	15
SEND ADDRESS TO STACK	35
ACCESS	145
CABLE	30
LOAD BUFFER	10
CABLE TO CPU	20
STORE ACCESS CONTROL	15
CABLE TO OPERAND UNIT	32
	520

Fig 2 ACCESS TIME

1. CHARACTER LENGTH 8 BITS.
   BASIC INSTRUCTION 16 BITS.
   FLOATING POINT NUMBER 64 BITS.

2. MINIMUM NUNBER OF CENTRAL REGISTERS.

3. INSTRUCTION FORMAT

F	k	n
7	3	6
FUNCTION	TYPE OF OPERAND	OPERAND ADDRESS RELATIVE TO A BASE ADDRESS

4. LARGE VIRTUAL ADDRESS SPACE ARRANGED IN A SEGMENTED MANNER.

   ADDRESS FORMAT

P	S	P	L	B
PROCESS	SEGMENT	PAGE	LINE	BYTE
4	14	16		2

THE STORE IS PAGED OVER 3 LEVELS, A 250 ns PLANTED WIRE STORE, A 2·5 μsec CORE AND A DRUM. THE PAGE SIZE IS VARIABLE IN POWERS OF 2 FROM 16 TO 64K LINES

Fig 3 SOME FEATURES OF MU5

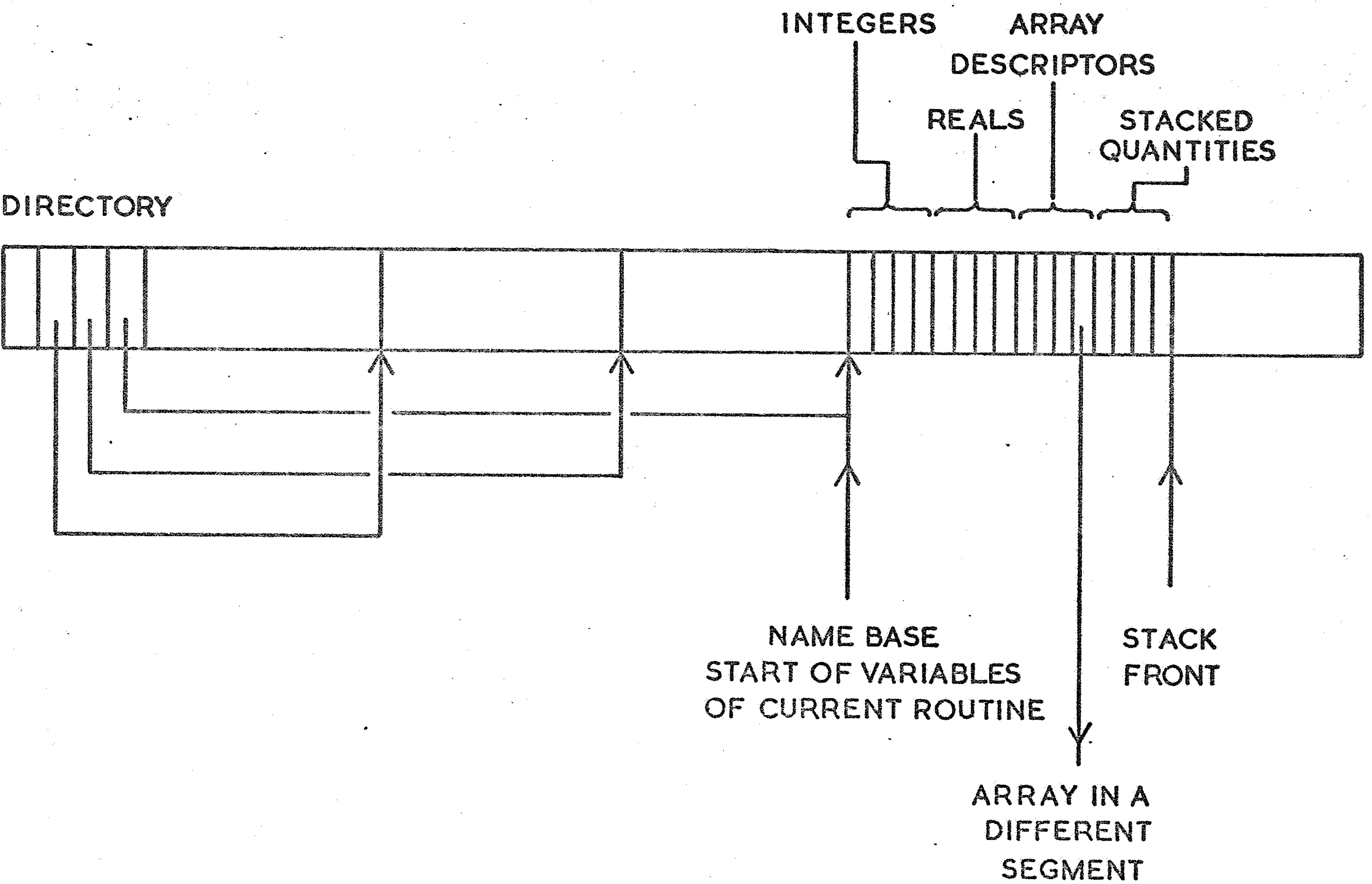


Fig 4 NAME SEGMENT ORGANIZATION

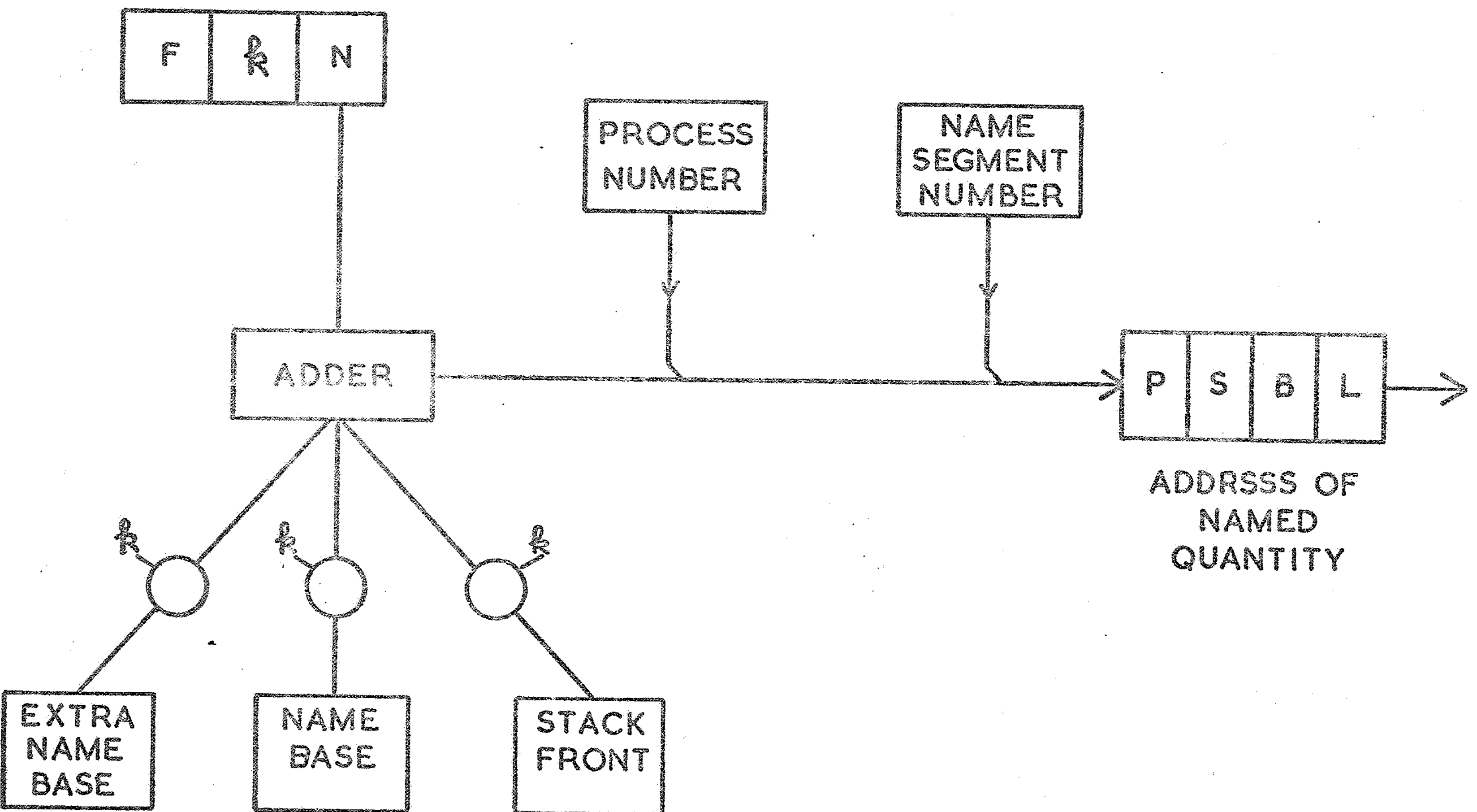


Fig 5 ADDRESS DEVELOPMENT (NAMES)

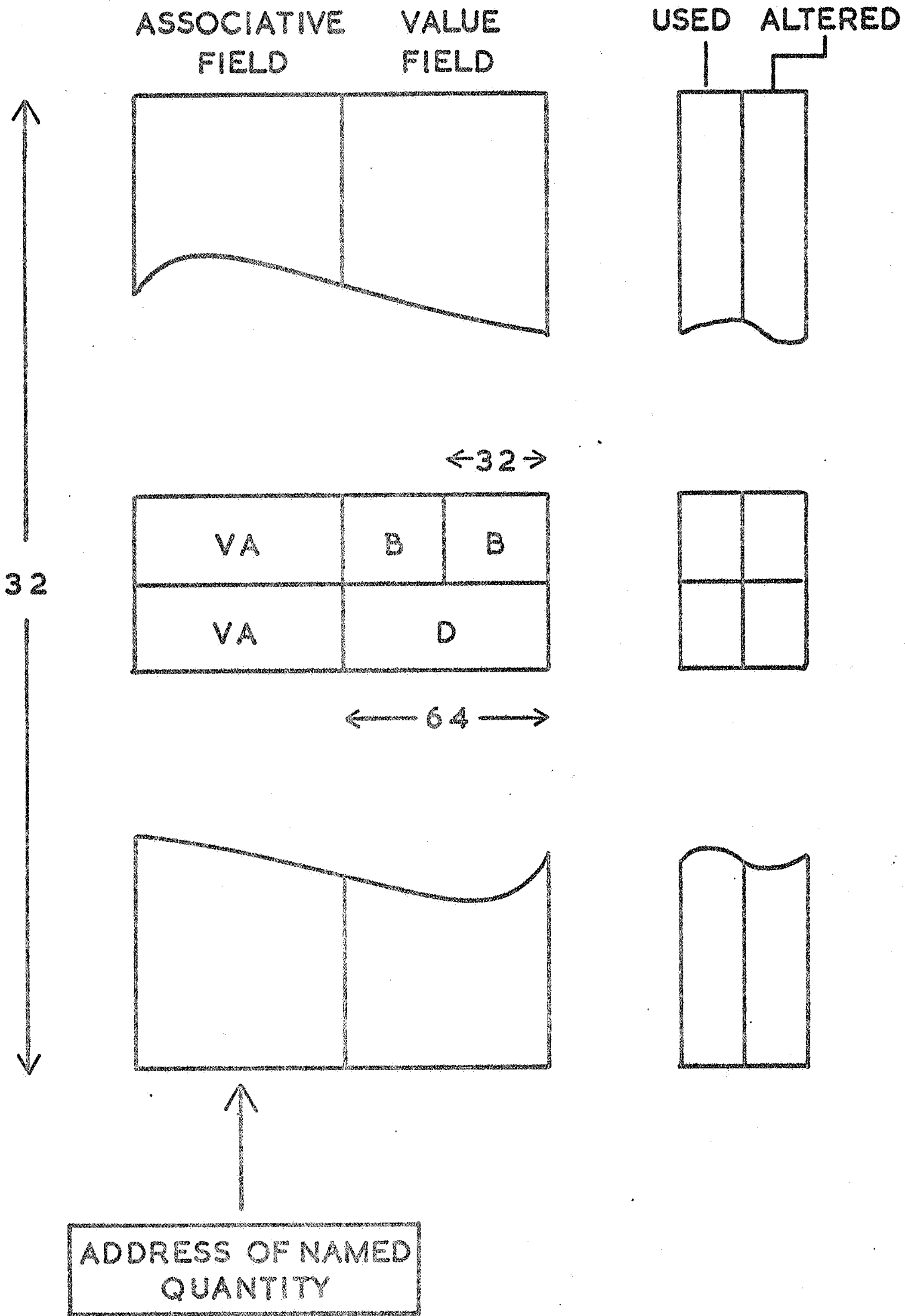


Fig 6 NAME STORE FOR B/D NAMES

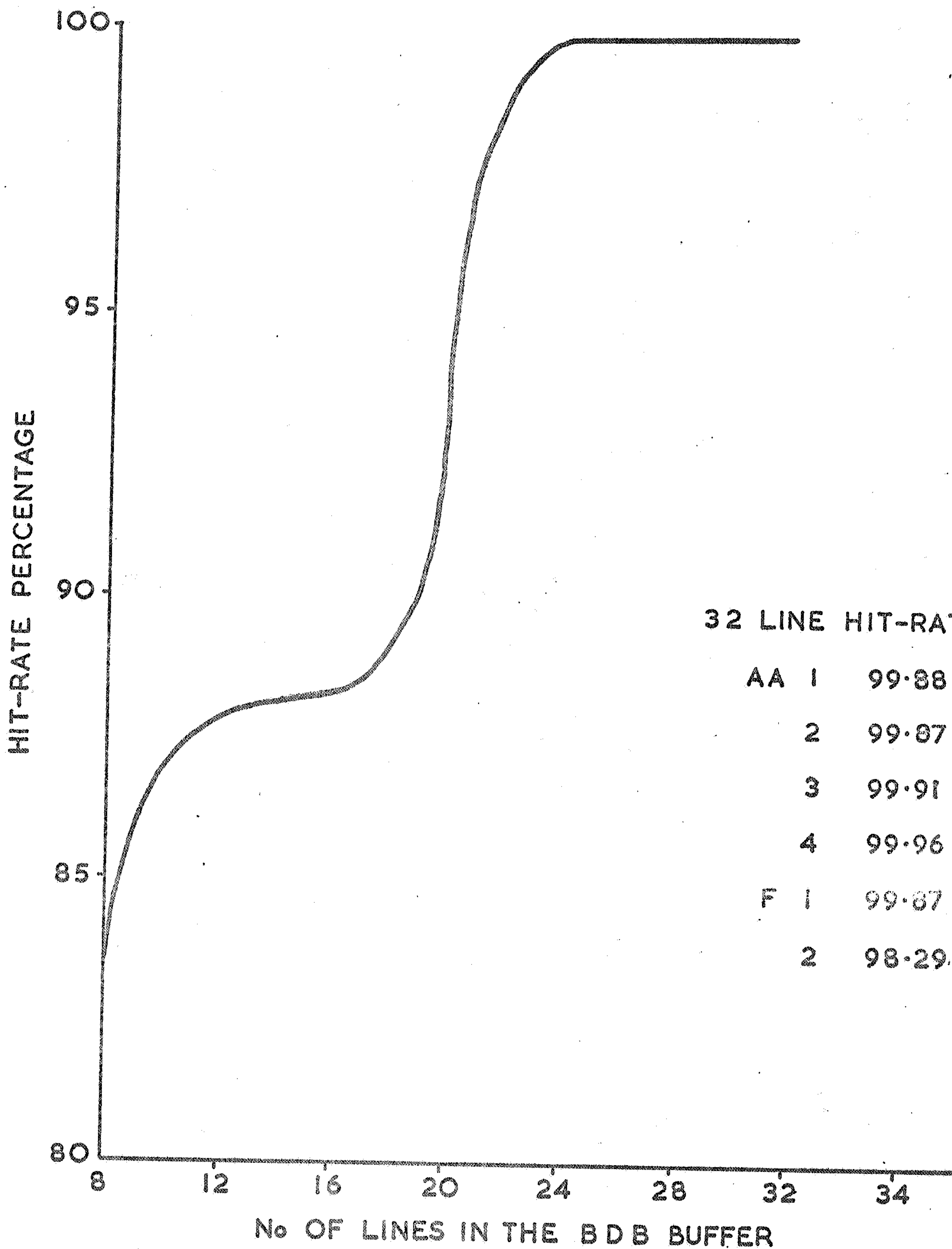


Fig 7 NAME STORE HIT-RATES

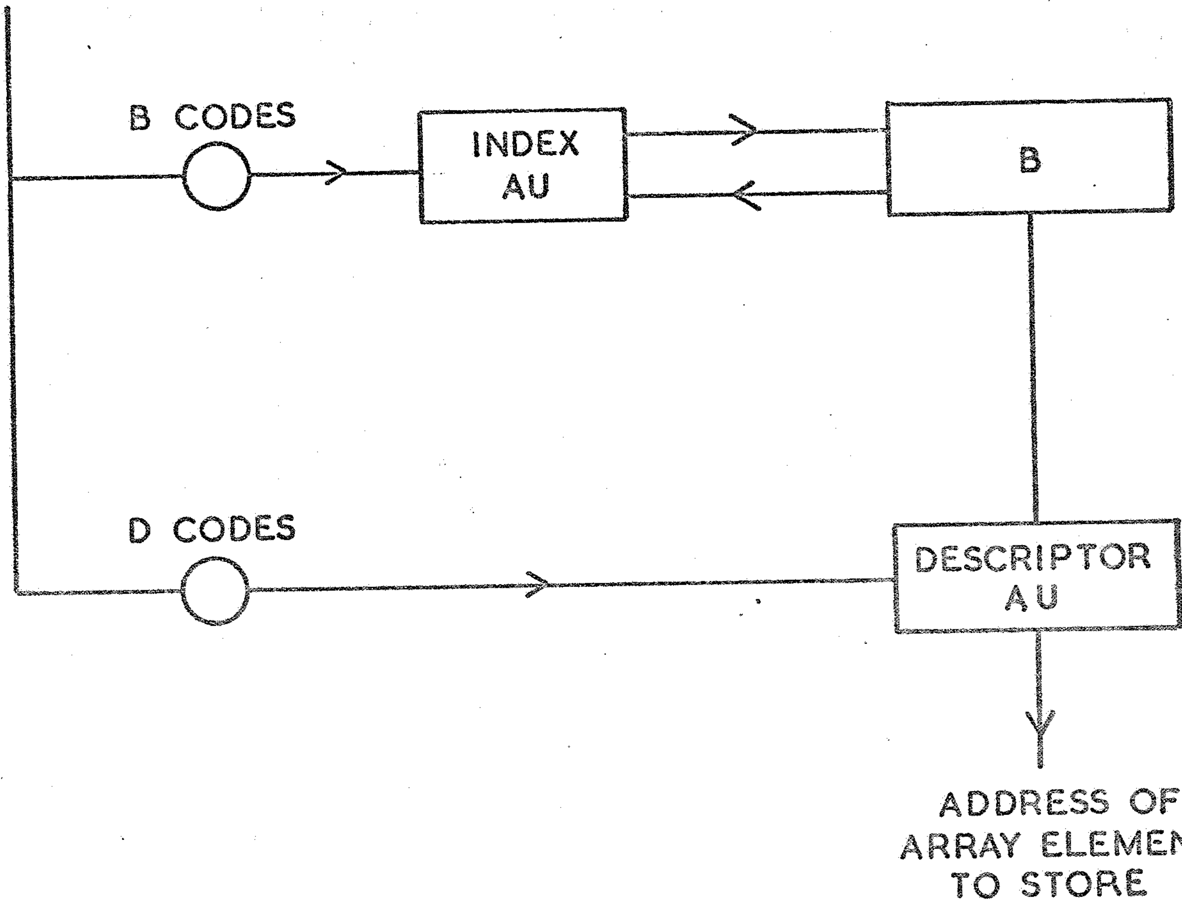


Fig 8 ADDRESS DEVELOPMENT (ARRAY ELEMENTS)

Instruction Fetching in High Speed Computers

F. H. Sumner

Department of Computer Science

University of Manchester

Text of Lecture given at IRIA (Paris), September 1969

I would like, first of all, to give a little background information on the research computer which is now being designed and built in the Department of Computer Science at Manchester University (this has been described in a paper published in the Proceedings of the 1968 IFIPS Conference).

The basic philosophy of this computer is that, as present high level languages demand from the programmer information about the types of operand he intends to use, then this information should be available to the computer to increase the efficiency of its own store organisation.

To be more specific, Algol and similar languages require the user to define sets of named operands; e.g. reals, integers and arrays within each routine. The basic instruction of our computer is of 16 bits single address type (Fig. 1).

F	N
9	7

where F defines the function and name type, and N is the name of the operand in the current routine. The addition of the contents of a name base register to N gives the virtual address of the name in the store.

Array elements can be accessed by obeying orders of the above type to compute the displacement down the array then obeying an order where the name points to a descriptor of the array, e.g. its origin and the type and size of its elements.

Compilers have, in general, made poor use of central registers. We believe that this type of code with no explicitly named fast registers will permit the writing of efficient compilers which produce efficient object code.

The proposed hardware for the system contains a store of cycle time 250nsec. and a pipeline approach to the obeying of instructions which permits a theoretical maximum rate of processing of one instruction (16 bits) every 40nsec. This could never be maintained if all instructions and operands were requested from the main store in a direct manner.

The system design derives enormous benefit from the concept of naming. An analysis of a large number of programs run on the University Atlas showed that 80% of operand accesses were for operands that in the new system would be named quantities, and a further study showed that the number of names currently being used is quite small and that an associative store of only 32 lines would trap over 99% of these operand accesses. (Fig.2).

Thus, 4 out of 5 operand accesses go to a very fast associative store which can be accessed and read in 2 overlappable beats each of 40nsec. This is instead of going to the main core store.

The proposed computer has an elaborate system of virtual addressing and paging, and the management of this adds approximately 100nsec. to each access to store. This, together with priority circuits and cable times, results in a final access time of about 500nsec. even though the access time of the store is 130nsec.

The value of trapping 80% of the operand accesses in the name store is now very clear.

The remaining operand accesses are to array quantities and advantage can be taken of their predominantly sequential nature.

The major problem therefore remaining is how to deal with accesses to the store to read instructions.

Before going on to consider solutions to this instruction problem, I would like to go back several years to the design of the Atlas. At the time of its design this was one of the fastest computers in the world and there was a considerable amount of overlap of the various stages of instruction execution in order to achieve the speed. However, it was decided that all test codes and other orders which may or may not transfer control would be taken to completion, this takes about 7 sec., before the request for the next instruction was made to the store. This simplified

the design and at the time it was not felt that the effect would be significant. The computer has an instruction counter which counts one for most orders, two for floating point multiply, and four for floating point divide. This permits easy timing of the system.

After the machine was completed various tests of its speed were made using long sequences of instructions and it was concluded that the average time per count was 2µsec.

However, observarions of the number of orders obeyed in a day gave an average time of 3µsec. The discrepancy was originally put down to store clashes but when a later version of the system with a much larger store gave the same results, it was concluded that the overlap in practice was not as good as had been expected, but no particular reasons could be put forward.

At the time when we were beginning to think about our next computer project, modifications were made to our Atlas to permit the collection of statistics on the number of orders of the different types which were obeyed. These showed that over a very large sample of programs, the proportion of orders which may transfer control was about 20%. This clearly explains the 2, 3µsec. discrepancy

$$0.8 \times 2 + 0.2 \times 7 = 3\mu\text{sec.}$$

The experiment also yielded the distribution shown in Fig.3 for the number of orders between transfer type orders for quite a large sample (several minutes of computing).

The general reaction to this graph was one of disbelief but it is interesting to analyse the orders executed in obeying a simple piece of Atlas Autocode. (Fig. 4).

Other statistics provided by the Atlas monitoring are summarised in Fig. 5.

For the new computer system the maximum rate at which the central processor can execute instructions is one 16 bit instruction every 40nsec. It is expected that a transfer will occur about every 15 instructions. The store of the new computer can supply instructions at the required rate but in order to cope with the long access time, instructions will have to be

requested very far in advance of being obeyed.

A simple early call approach is all right for long sequences of instructions but the incidence of a transfer completely wrecks the system. Assuming one jump in 15 orders, an access time of 500nsec. and 6 orders in the pipeline, gives an average instruction time of

$$(14 \times 40 + 1 \times 740) = 87nsec$$

i.e. less than half the maximum speed.

The distances jumped make it clear that a simple instruction buffer of reasonable size will not be much use.

The first system that we considered is summarised in Fig.6. All instructions proceed as if there are no transfers and when a transfer occurs and the new address has been computed this is presented to a small associative store to get the first few orders at the jumped to address, this permits an early call to the main store to get subsequent orders without any additional delay. The loss here is a pipeline of instructions, 240nsec. plus the access time to the small store, 80nsec., total 320nsec. (The 240nsec could be slightly reduced for unconditional transfers).

If the jumped to address was always in the store this would give a maximum rate of:

$$(1/15)\ (14 \times 40 + 320) = 55nsec.$$

In order to test the efficiency of this type of approach detailed traces of several programs were made and a typical result for various numbers of lines of associative store is:

n	% in buffer	limit (nsec).
4	45	74
8	80	64
16	81	64

However, each line has to be very wide to give the early call chance to catch up after a transfer, probably about 200 bits, and it was felt that this was rather expensive.

The system that we propose to implement is summarised in Fig. 7.

In order to make an early call system work it is necessary that the requests to the store should be for the correct instructions even if these instructions are not in sequential addresses. The system starts with the associative store empty and a stream of instructions from sequential addresses is brought from the store and presented to the central processor. When a control transfer occurs, it is easy for the central processor to detect that the next order in the pipeline, which should of course be out of sequence is, in fact, in sequence, because each instruction is accompanied by a single bit indicating that it is an " in sequence" instruction. A request is therefore made to the store for the instruction at the jumped to address and subsequent requests to the store are for the instructions following this address. At the same time an entry is made in the associative store which consists of the jumped from address on the associative side and the jumped to address on the value side. As can be seen from the diagram, all instruction addresses are presented to the associative store before being allowed to go to the main store. If, therefore, at some later time, a request is about to be made to the store for the instruction in the jumped from address, it is easy for the system to detect this and substitute a request for the instruction in the jumped to address, and to continue the stream of instructions from this address. When the instruction from the jumped to address is presented to the central processor, the extra bit is set to say "not in sequence". Therefore, if the transfer again occurs the next instruction in the pipeline will be the correct one and there will be no break in the overlapped operation of the pipeline. This means that whenever a jump occurs which has not been anticipated, there will be a gap of 740nsec. (500 + 6 x 40) but for all anticipated transfers there will be no gap.

This model has been tested with the same program traces as the previous system with the following results:

n	% correct	limit (nsec)
4	43	66
8	74	52
16	76	51

The reason for the fall in the percentage of correct accesses relative to the same number of lines of the previous system is that the new system will occasionally predict a transfer which does not occur.

The lines of the associative store in the new system are only 48 bits wide and the limit on the computing speed for the same number of lines is considerably less than in the previous system. It is proposed to implement a system with 8 lines of associative store.

The results for the different systems are summarised in Fig. 8. Figures 9 and 10 show the relationship between the percentage of correct associations and the limiting speed for different occurrences of transfer instructions from 1 in 5 to 1 in 20 for the two systems considered.

FEATURES OF THE INSTRUCTION SET

1. SINGLE ADDRESS CODE

2. OPERANDS ADDRESSED BY NAME

3. OPERANDS MAY BE OF ANY OF THE COMMONLY OCCURING TYPES

4. CENTRAL REGISTERS ARE PROVIDED ONLY FOR FUNCTIONAL REASONS

5. FAST STORAGE FOR FREQUENTY USED NAMED OPERANDS AND STACKED PARTIAL RESULTS PROVIDED BY ASSOCIATIVE STORAGE

FIG. 1

STORE ACCESSING BY NAME

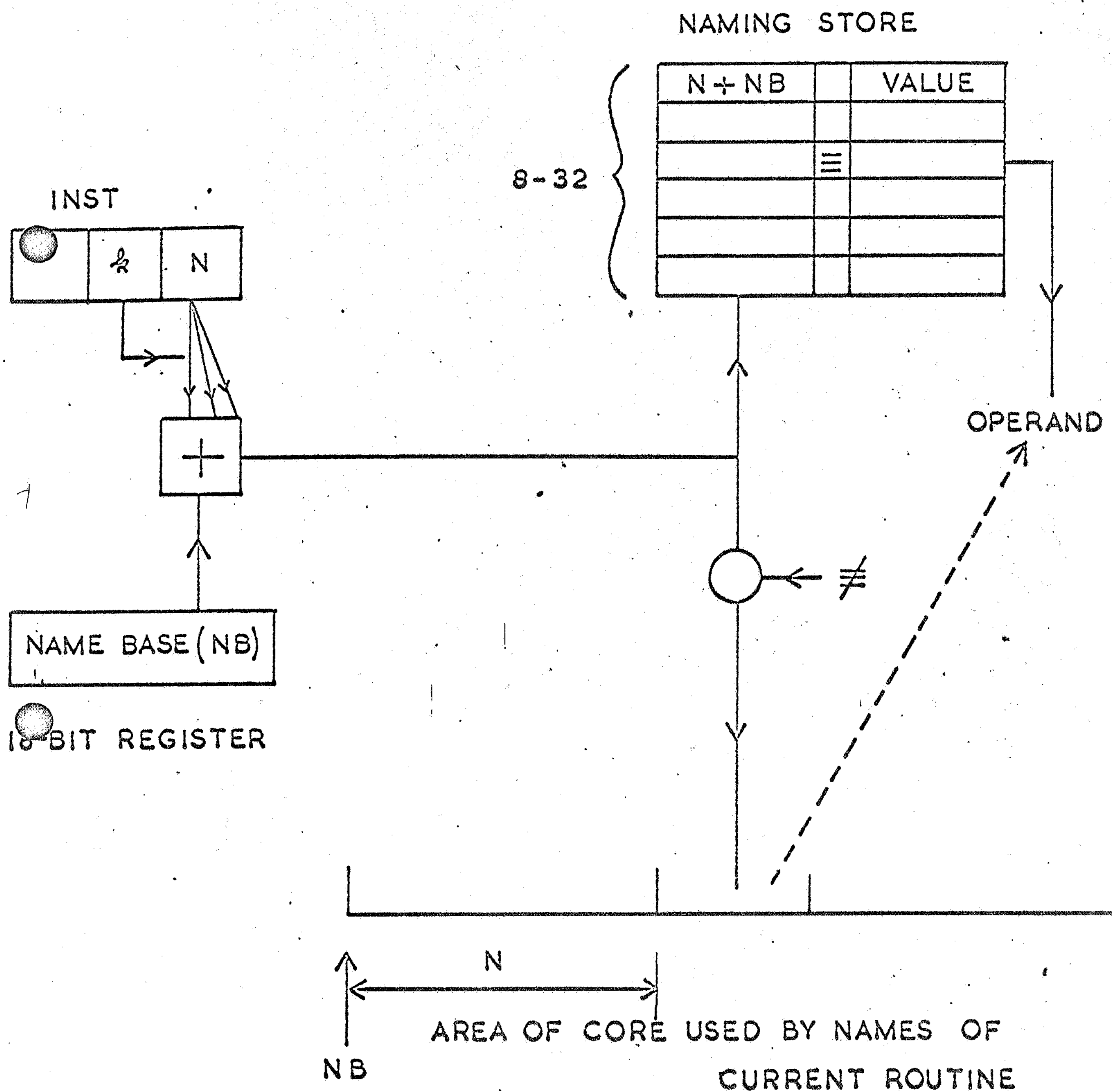


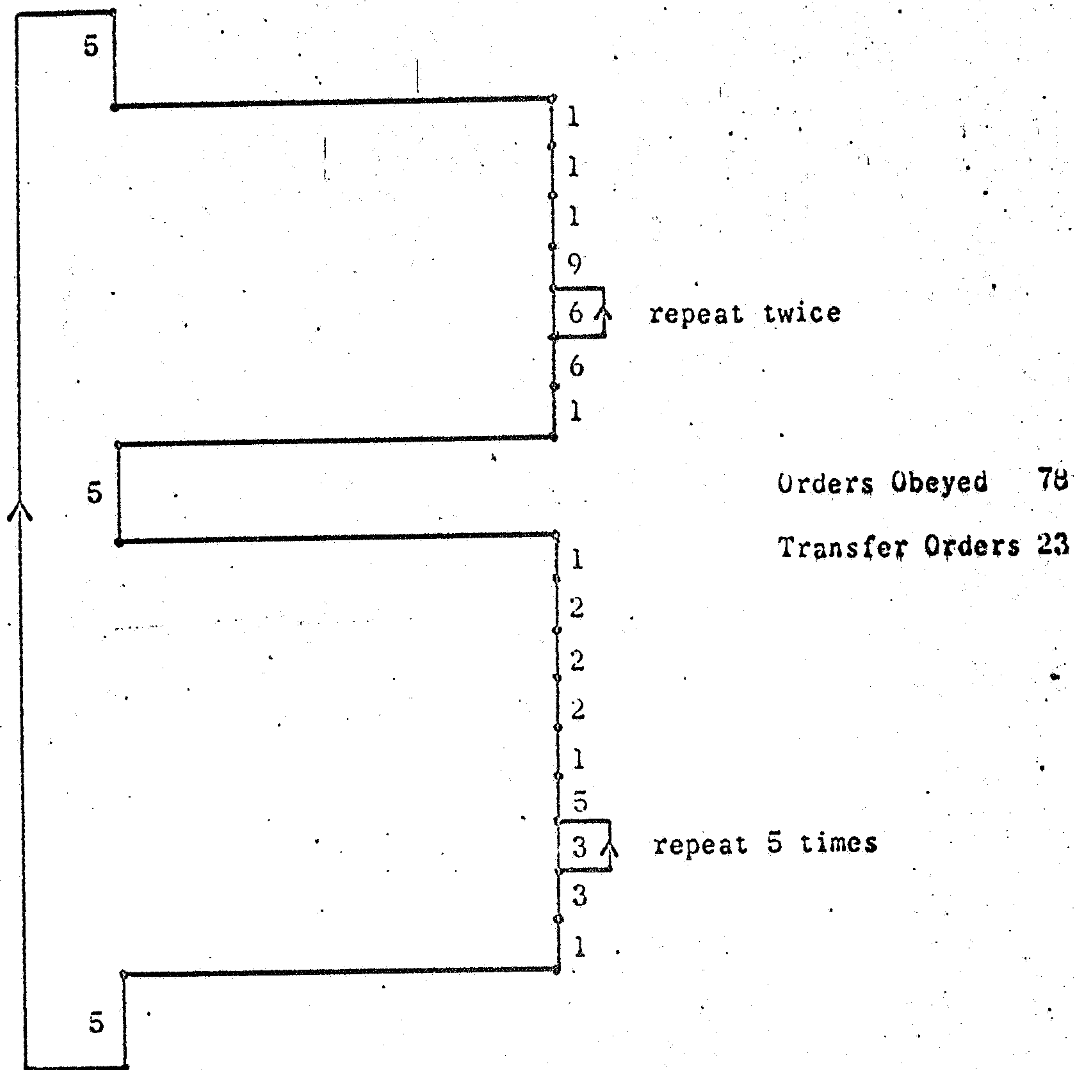
FIG. 2

Distribution of number of orders obeyed
between transfer orders

2 3 4 5 6 7 8 9 10 11 12 13 14 15

FIG. 3

```
cycle i = 1, 1, n
a(i) = sqrt(b(i)) + cos (c(i))
repeat
```



Analysis of a simple piece of code

FIG. 4

Results from monitoring the Manchester University Atlas

Percentage of obeyed orders which may transfer control	20%
Percentage of these which actually cause a transfer	70%

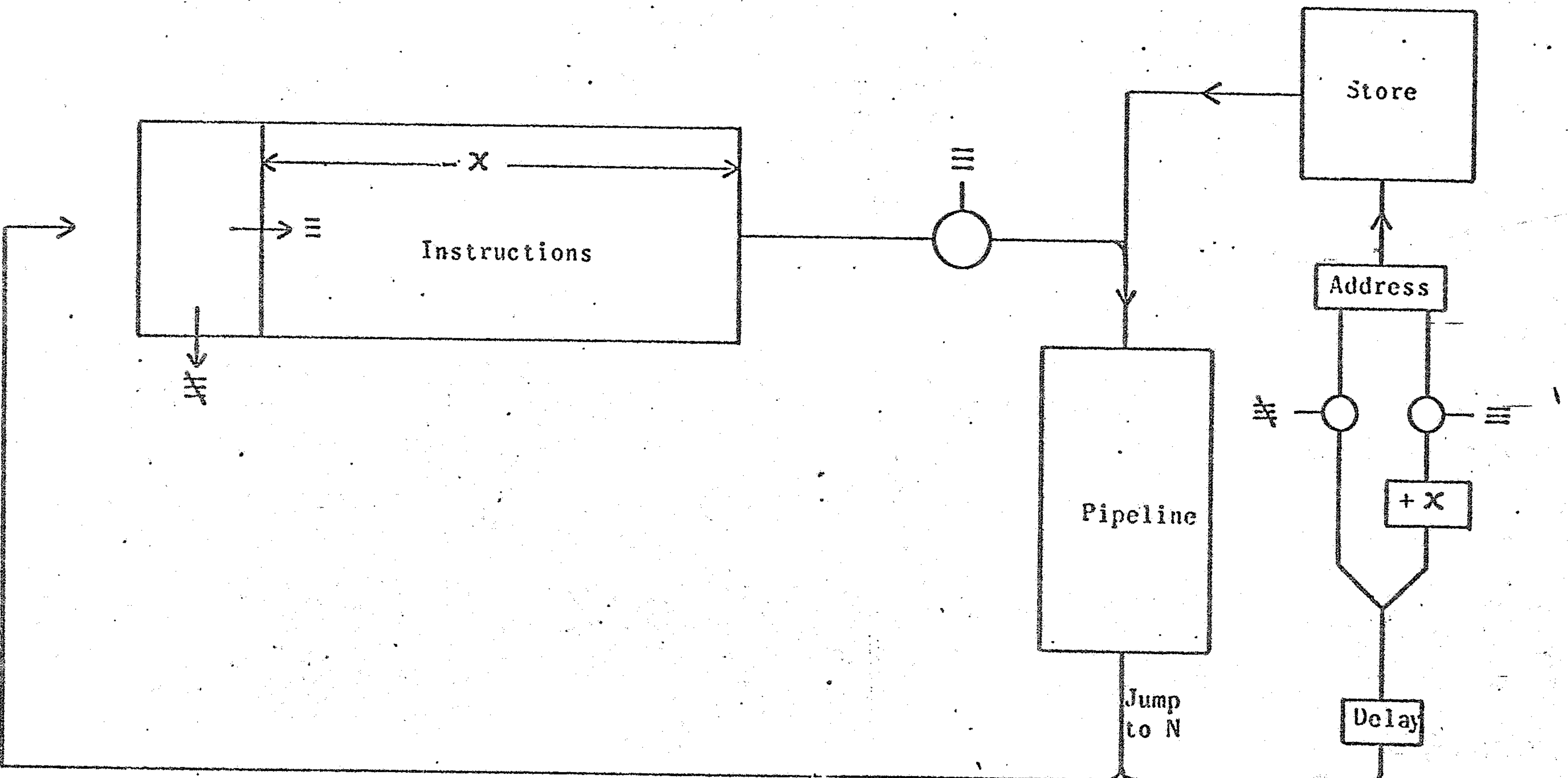
14% or 1 in 7 orders transfer control

Distribution of transfer orders

	Obeyed	Jump
Unconditional transfers	5.5%	5.5%
Test and count	5.0%	3.9%
Arithmetic tests	9.5%	4.6%
	20.0%	14.0%

Distance of jumps		
	34%	< 4
	6%	4-16
	60%	> 16

FIG. 5



System 1 - Associatively Addressed Buffer

FIG. 6

Proposed early call system with control tracing

FIG. 7

Limiting Speeds Imposed By Instruction Fetching

Assume
1. Maximum rate of processing = 40 ns/16 bits
2. Length of pipeline = 6 stages = 240 ns
3. Access to store for instruction = 500 ns
4. Access to buffer for instruction = 80 ns
5. One jump every 240 bits

Simple early call system

Limit = 87 ns

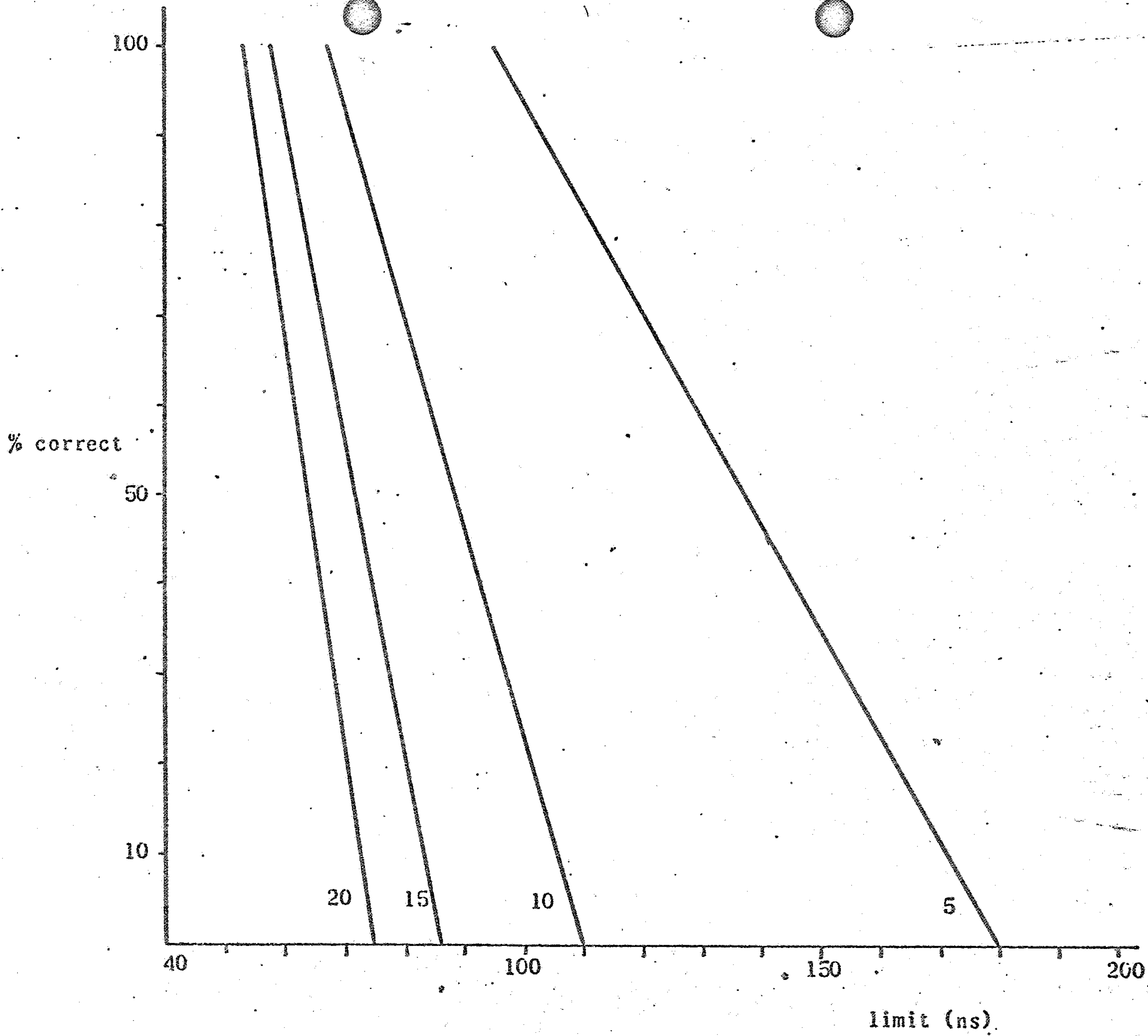
System 1. n lines of associatively addressed buffer each containing 24 bits of address + 160 bits of instruction

n	% in buffer	limit (ns)
4	45	74
8	80	64
16	81	64
-	100	53

System 2. n lines of associatively addressed buffer each containing 2 x 24 bits of address

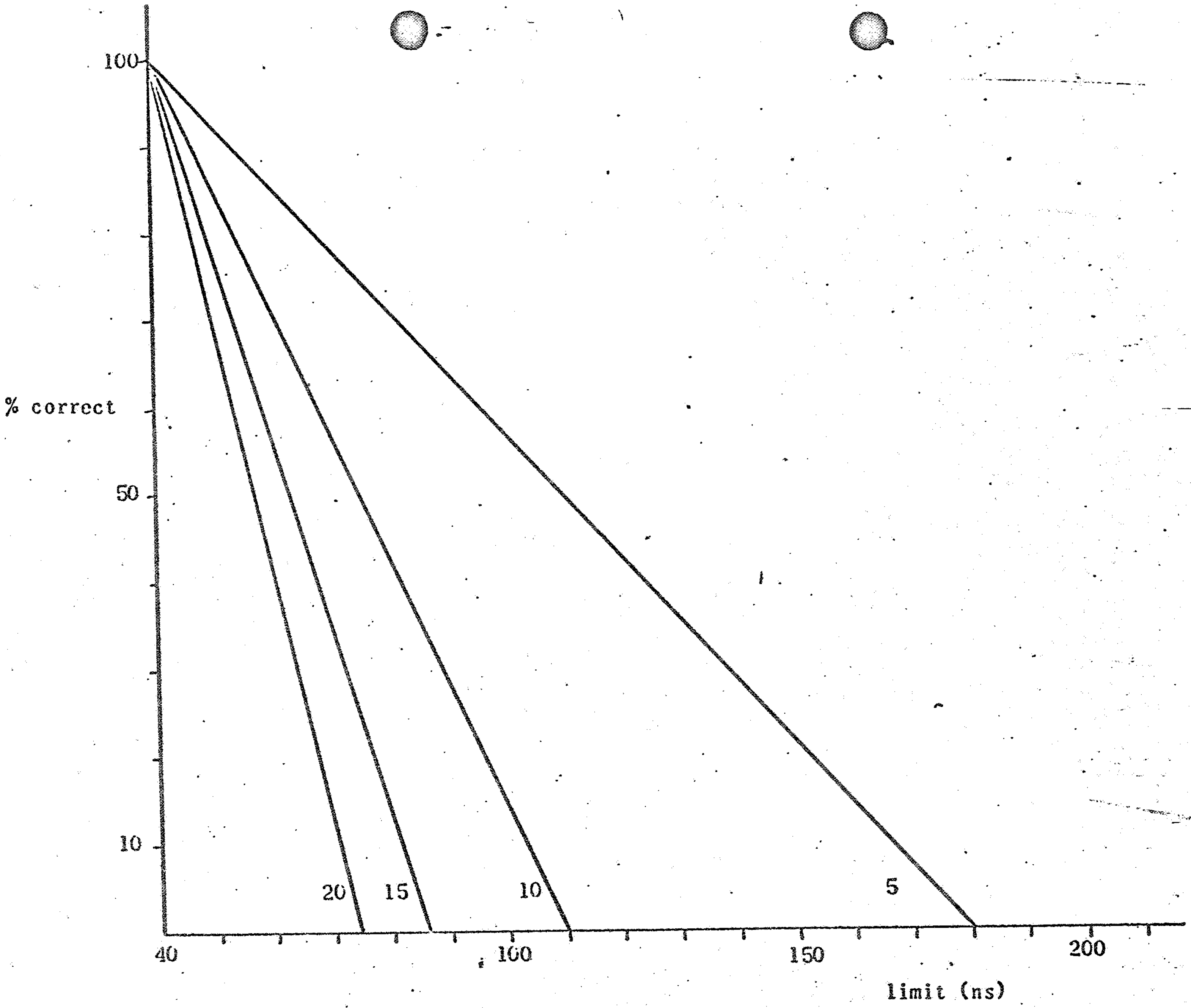
	n	% correct	limit (ns)
	4	43	66
PROPOSED SYSTEM	8	74	52
	16	76	51

FIG. 8



System 1 - Associatively Addressed Buffer

FIG. 9



System 2 - Control Tracing

FIG. 10

INFORMATION PROCESSING 74 – NORTH-HOLLAND PUBLISHING COMPANY (1974)

MU5 – AN ASSESSMENT OF THE DESIGN

F. H. SUMNER

The University of Manchester, Manchester, England

(INVITED PAPER)

The first proposals for the MU5 computer were presented at the IFIP Congress in 1968. The system is now built and this paper summarises the design. Attention is drawn to two of the most significant features, firstly the addressing of operands with the division into two classes, names and array elements, and secondly the fetching of instructions so that the gaps in processing caused by branch orders are minimised. The correctness of the structure is examined and possible extensions are suggested.

1. INTRODUCTION

At the IFIP Congress in 1968 a paper was presented [1] describing the initial ideas for the MU5 computer system. In the six years since 1968 this design has been completed and the hardware and much of the software for the system has now been implemented. The system is expected to be made available to users early in the summer of 1974 and the following period will be devoted to a detailed study of the behaviour and performance of the complete system.

Since freezing the design in 1971 subsequent work on implementing the hardware and software has increased our belief in the correctness of many of the design features and has also revealed areas where changes would lead to improved performance. It is the purpose of this paper to assess the design of the MU5 in the light of the experience gained in bringing it to a working system.

2. SUMMARY OF SYSTEM DESIGN

There have been many papers describing the system in detail (e.g. [2 and 3]). In this paper a brief description will be given which covers the significant features of the design.

The complete system is a complex of several processors and stores linked by an exchange as shown in fig.1. It is expected that stores and processors will be added and removed as the project develops. Each main processor has direct access to its own store as well as the common accesses through the exchange; this is essential if they are to operate at high speed.

The aim in the design of the MU5 central processor was to produce a machine whose structure is well suited to the needs of modern high level languages. It was also hoped that the system would have a power of some twenty times Atlas.

Following our experience on ATLAS in which relatively inefficient use was made of the fast registers it was decided that in MU5 there would be no explicitly addressed fast registers, all operands would be addressed as though in the virtual address space of the current process. The address space for each process is quite large and is divided as follows.

14 bits to define the segment
16 bits for page/line
2 bits for the byte address

The page/line division is not fixed, the page size is variable in powers of 2 from 16 to 64K. The management of the variable page size has not proved

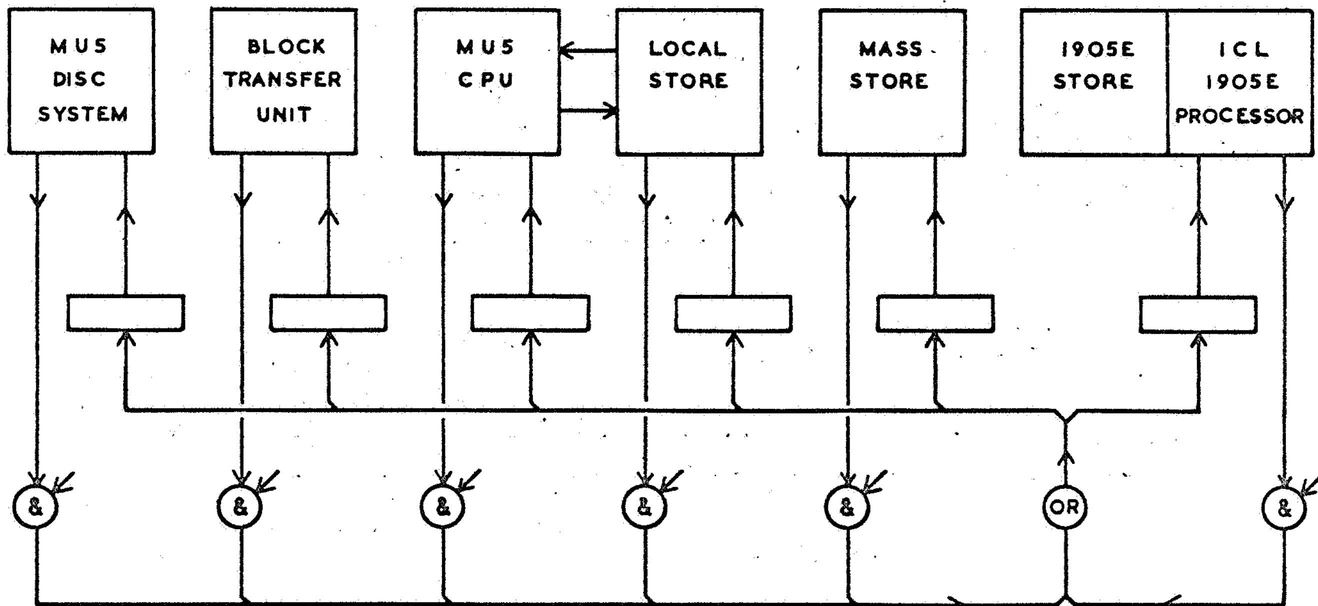


Fig. 1. The MU5 Multicomputer System.

too difficult and the additional overheads beyond normal fixed length paging should not be significant. The page size may be different for each segment and one of the major programs of research over the next two years will be the investigation of the advantages and disadvantages of this flexibility.

High level languages generally demand from the programmer information about the types of operand he intends to use. In the MU5 design, this information is not thrown away but is made available to the computer hardware so that it can increase the efficiency of the store organisation.

The types of operand commonly in use in high level languages may be listed as follows:-

(a) Literals consisting of fixed-point and floating point constants.
(b) Primary operands consisting of:-
  (i) integer and real variables
  (ii) data descriptors
  (iii) names of functions and routines
(c) Secondary operands consisting of:-
  (i) elements of arrays and vectors
  (ii) strings of elements of structured data sets.

Only primary operands, which we have called named quantities, can be addressed directly. The instruction format of MU5 has two fields the function and an address; this address is of a primary operand and defines the operand within the current procedure the full virtual address being obtained by adding a name base to the "name" N. All named quantities are kept in segment zero and the remaining segments are used for instructions (in pure procedure segments) and for array elements etc. These latter are accessed indirectly by means of a named descriptor, which defines the origin, and a precalculated indexed quantity which gives the dispalcement from the origin. The system of operand accessing is shown schematically in fig.2. It is expected that the majority of instructions will only be 16-bits long though of course longer ones of 32, 48 and 80-bits are provided for long literals or for long addresses.

The component parts of the MU5 processor are shown in fig. 3. It was stated above that a design aim was a performance of 20 times ATLAS and to this end the instruction unit is designed to prefetch instructions and to supply these to the primary operand pipeline (PROP) at a maximum rate of one every 40 nsec. This rate is impossible within PROP if all operands have to be brought from the store as the access time including address translation and cables is about 500 nsec. To overcome this there exists within PROP a 32 line associative memory with an access time of 40 nsecs. which traps up to 32 named quantities. If a name is in the fast memory a rate of one instruction per 40 nsec. can be maintained by PROP but the penalty for a miss will be a gap of at least 500 nsecs. The performance of the entire system is therefore critically dependent on the performance of this associative memory. A hit rate in the region of 99% is expected.

Instructions concerned with index arithmetic are completed within PROP but instructions which result in accessing an array element, or which require use of the main arithmetic units, proceed to the secondary operand pipeline (SEOP).

3. OPERATION OF THE SYSTEM - OPERAND ACCESSING

The operation of the system is best illustrated by a simple example, consider the evaluation of the following simple sequence.

1) a := a + b(i+3)
   i = i + 1
   go to 1) if i<n
(a and the elements of b are floating point numbers)

The first instruction loads the value of the named real quantity into the floating point arithmetic unit;as "a" is a named operand it should be read out of the name store in PROP and then sent via SEOP to the arithmetic unit. Then come two orders to form i + 3; i is read from the PROP name store;3 is a literal;and the arithmetic is done in the index arithmetic unit.

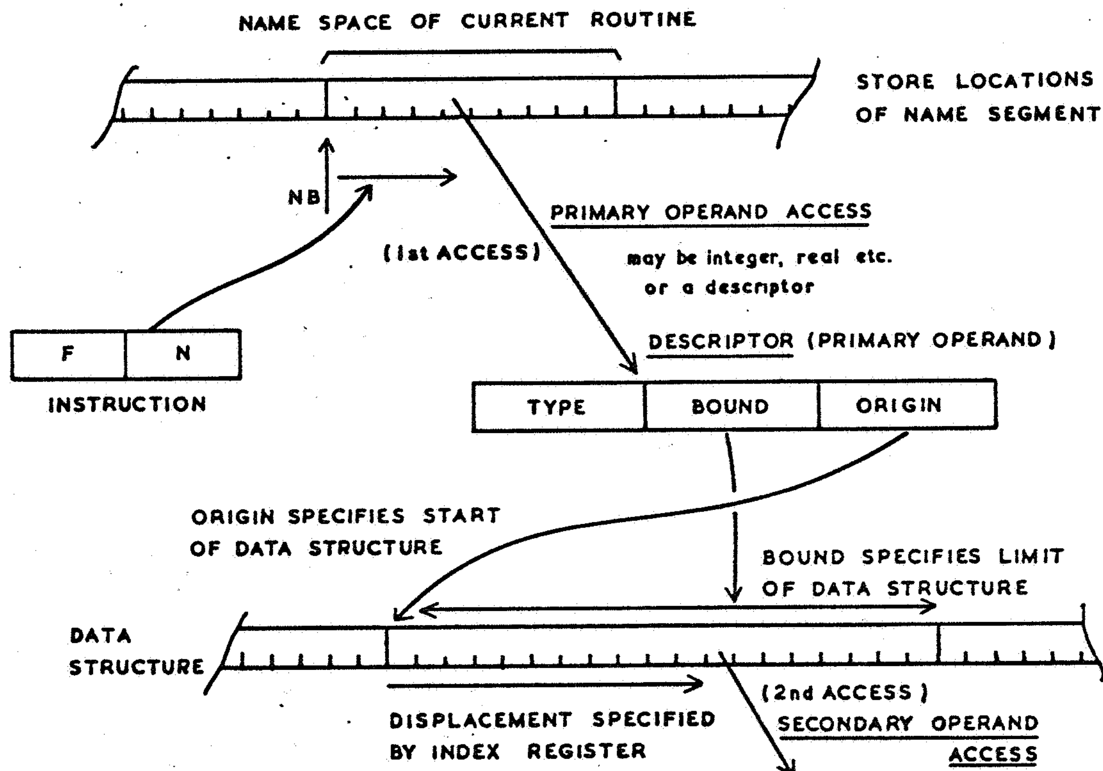


Fig. 2. Operand Address Development

index arithmetic
B-UNIT
instruction accessing
initial instruction processing
INSTRUCTION BUFFER UNIT
PRIMARY OPERAND PIPELINE
DESCRIPTOR UNIT
SECONDARY OPERAND PIPELINE
MAIN ARITHMETIC UNIT
address modification
operand manipulation
request for instructions
processor's dedicated store
instructions
STORE ACCESS CONTROL
LOCAL STORE
interface with store system
ADDRESS ROUTES
DATA ROUTES
TO EXCHANGE

Fig. 3. The MU5 Central Processor.

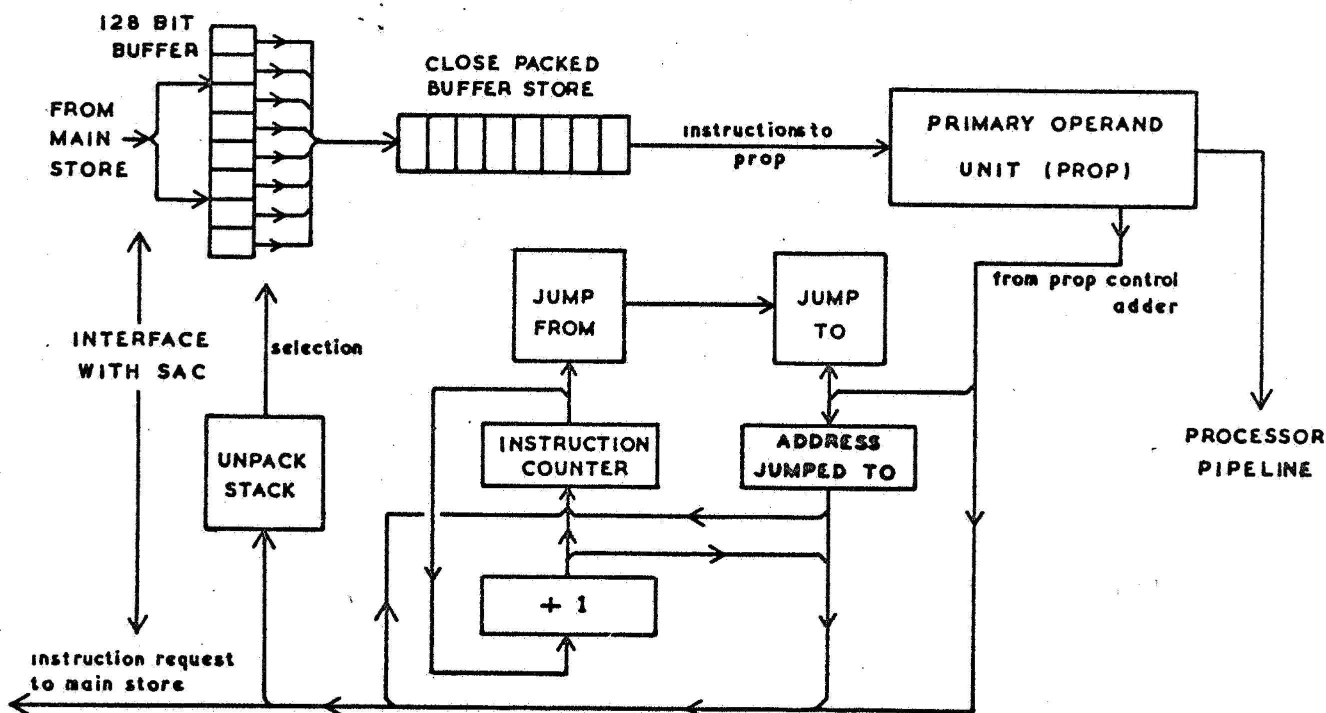


Fig. 4. Instruction Fetching.

These orders will be completed long before A gets through SEOP to the floating point arithmetic unit. The next order reads the descriptor for the vector B from the name store, forms the address of B(i+3), reads the value of this element from the store and sends it via SEOP to the floating point arithmetic unit for addition to a. The final order of this group writes the contents of the floating point arithmetic unit back into the value of a in the name store. This writing obviously presents many problems; there will be a long delay before the information is available and any subsequent reference to the name store for a will have to be detected and held up. These and other arguments led to the decision to split the name store into two parts, one remaining in PROP to trap integers and array descriptors and the other part for reals being placed at the end of the SEOP near to the floating point arithmetic unit. All the names still remain in the same segment of the virtual space and the hardware is therefore required to perform detailed checks on where any particular virtual address is currently living. A better solution may be to extend the concept of matching the address structure with the language structure and to define a distinct segment of virtual space for reals. The associative memory to trap elements of this new segment could then be placed near the arithmetic unit which manipulated reals and the organisation and management of this associative memory could be arranged to match the pattern of usage for reals. A further major difficulty with the present design is illustrated if B(i+3) in the above problem is replaced by B(i,j). A commonly used way of accessing an element of a two-dimensional array is to use the first descriptor and the value of i to read a descriptor for the ith row from a vector of descriptors; this is then used together with j to access the jth element of this row. The access for the second descriptor will go via SEOP as it itself is an array element. Whilst a separate path is provided so that the access is not held up by operands destined for the floating point arithmetic unit, the access for the element B(i,j) is still a very lengthy process. A further extension of the above argument is to place both single descriptors and vectors of descriptors in the same segment and to have a third associative memory placed near to the address calculation unit to trap elements of this segment.

4. OPERATION OF THE SYSTEM - INSTRUCTION FETCHING

Returning to the program given above the next orders increment i and jump if i is less than n. In any pipeline system discontinuities in the instruction stream caused by branch orders give severe degradation in the performance and an average pipeline efficiency of 20 to 25% is generally considered to be acceptable. When a conditional branch is obeyed the worse delay is A + B where A is the access time for the jumped to instruction and B is the length of the pipeline. For small loops all the instructions may be trapped in a fast buffer reducing the loss to something approaching B. However, B is approximately the same magnitude as A and the losses are therefore still large. Furthermore, for large or complex loops, trapping of the loop instructions within a small buffer is not possible.

The MU5 instruction fetch unit(IFU)is shown in fig. 4. A small associative memory is used to remember previously obeyed pairs of jumped from and jumped to addresses so that on a subsequent pass through the same part of the program the early call system will see the jumped from address and cause the orders from the jumped to address to be prefetched. Thus if the jump occurs again there will be no pipeline gap at all. This simple system will have gaps on the first obeying of a jump and also when an order which has previously jumped decides to go straight on. However, it is expected to correctly predict about 60% of all jumps and this approach appears to be far more cost effective than a large buffer of more conventional design. Only branch orders with literal operands can be predicted in this way so many orders such as return links still cause gaps of A + B. The system could easily be expanded to include branch orders with operands from store locations but other more ambitious improvements are also possible. If the associative memory in the IFU could be written to by the object program it would be possible for the compiler to plant instructions which cause the appropriate address pairs (jumped from/jumped to) to be set in the buffer before the first occurrence of the jump and also to be deleted after the last occurrence. These extensions would prevent many of the remaining pipeline gaps and the overall efficiency could be greatly increased. This does not appear to be an unreasonable demand on the compiler-writer particularly for code which is frequently obeyed such as that within the compiler library.

5. POSSIBLE EXTENSIONS TO THE SYSTEM

The problem above of accessing vectors of descriptors is a particular example of the more general problem of list processing, where, for the present purposes this means any situation in which the next operand cannot be accessed until the present one has been brought back to the CPU and possibly processed in some way. A pipeline processor is unsuited to this type of problem as little or no overlap of instruction execution is possible. The evidence we have to date is that the object code of FORTRAN and ALGOL 60 will be executed efficiently but the compilers themselves will be less efficient and furthermore the code executed in the operating system will be even less suited to the structure of the hardware. The optimistic way of looking at this is that the operating system orders will be obeyed as efficiently as in conventional computers and the compilers will be better and object code will be very good. As shown in fig. 1, MU5 is a multi-computer system and it was always intended that all the input/output would be done by the 1905E rather than the MU5 processor. Perhaps the correct extension of the design is to have other computers in the system which are designed to be good at obeying compiler code or operating system code. For an overall system ratio of say 10% operating system, 15% compiling, 75% execution it would be easy to design processors for these particular jobs. The drawback of this approach is that if the balance of activities varies one or other of the special purpose processors will be overloaded, and even if this does not happen over long periods there are bound to be short bursts of activity for individual processors with the others remaining idle. These extra processors will not however be expensive and it seems reasonable to investigate the possibilities of extending the basic MU5 concept of the structure matching the problem to the structure of the whole system matching the structure of the complete work load.

ACKNOWLEDGEMENT

MU5 is the work of a large research group at the University of Manchester under the general direction of Professor T. Kilburn.

REFERENCES

[1] T. Kilburn, D. Morris, J.S. Rohl and F.H.Sumner, A system design proposal, Information Processing 68, North-Holland, Amsterdam, 1969.

[2] D. Morris, G. Detlefsen, G.R. Frank and T. Sweeney, The structure of the MU5 operating system, The computer journal, May 1972.

[3] R. N. Ibbett, The MU5 instruction pipeline, The computer journal, February 1972.

Presentation to be given at 1978 AFIPS National Computer Conference,
Anaheim, California, June 5 - 8, 1978.

THE EVOLUTION OF MANCHESTER COMPUTER DESIGN.

by S. H. Lavington
Department of Computer Science
University of Manchester
Manchester M13 9PL U.K.

Introduction.

Four diagrams are given which show the development of computer architecture at Manchester University over a 30-year period. The diagrams show the overall structure of the Mark I, Atlas and MU5 computers, illustrating the evolution of particular features such as address-generation registers and the one-level store. For clarity only the principal data paths and visible registers are shown; more details can be found in References 1 - 3 and the bibliographies given therein.

Of the four computers under discussion, the two Mark I's were serial synchronous machines, whereas Atlas and MU5 were parallel, asynchronous and pipelined. None of the computers employed micro-programming. The time for a simple fixed-point addition on the four machines was respectively: 1.2 msec., 1.8 msec., 1.52 microsec. and 50 nanosec.

The terminology used here has been unified and brought up to date compared with the original usage. Some of the less-common abbreviations appearing in the diagrams are explained as follows:-

AM, AL - the most-significant and least-significant halves of a double-length accumulator.

B-Store - contains B-lines, equivalent to general-purpose (index) registers

Descrip. - data-descriptors (64 bits long), used to hold the main properties of structural data such as arrays, records, etc.

Distrib. - distributor, a unit within the Atlas CPU which distributed data to/from the various stores according to address decoding.

Exch. - Exchange, a time multiplexed cross-point switch with a data throughput of 89 M bytes/sec., allowing communication between up to 10 units in the MU5 multi-computer configuration.

LCS - large core store.

Name Stores - associatively-accessed operand buffers, used on MU5 for scalars, descriptors, vectors, and the most-recently accessed locations on the stack. The Name Stores are transparent to the user.

SAC - store access controller, a unit arbitrating between asynchronously-arriving access requests to main store from the CPU and various autonomous channels. SAC also contains the address translation hardware (virtual-to-real), as page address registers (Atlas) or current page registers (MU5).

STU - store-to-store transfer unit, used principally in MU5 for transferring pages autonomously between main store and LCS.

V-store - the collective term for all control flip-flops and registers including status registers, peripheral buffers, channel control and interrupt registers. These are assigned 'V-store addresses' and are made accessible (with necessary protection) via normal computational instructions on Atlas and MU5.

(1) <u>The prototype Mark I (1948) - see Figure 1.</u> Williams Tube

This was primarily intended as a realistic test for the random-access storage system (electrostatic) and so the computational facilities were sparse. It first ran a program on the 21st June 1948.

Williams in 1949 in reply to Queen's a day question

"Well, we had an idea for storage & then pressed on regardless and built a computer round that store without stopping to think too much"

Stores later used on 701 & 702

The Manchester Mark I (1949) - see Figure 2.

This enhanced Mark I first performed useful work (on Mersenne Primes) in April 1949, and the nature of such problems suggested the inclusion of multi-length arithmetic facilities and a fast multiplier. Multiplication took about 2.16 milliseconds. The B-line, first introduced in October 1948 as a relocation register to facilitate the overlaying of 'pages' from drum, was extended in 1949 to an 8-line B-store of eight general-purpose index registers.

Atlas - see Figure 3.

The B-store proved so useful on the Mark I that on Atlas it was enlarged to 128 lines, amongst which were three Program Counters (for 'main', 'extracode' and 'interrupt' control to speed context-switching). For ease of programming, store overlays were automated. Thus the main core and drums formed a 'one-level store' organised as a paged virtual memory of size 8M characters (excluding the V-store). The handling of many slow I/O devices was simplified by the V-store concept and interrupt vectoring. The fast ROM ('fixed store') contained 'extracode' extensions to the normal instruction set such as the common mathematical functions, as well as operating system procedures concerned with store management, I/O handling, test programs, etc. The read-only nature of this store permitted the use of a ROM technology about five times faster than contemporary main store RAM technology. It may be seen that Atlas contained many hardware innovations which eased the operating systems' tasks. The on-line spooling and multiprogramming features of the Atlas operating system gave an impressive work throughput and fast job turn-round time. The first Atlas came into operation in Manchester in December 1962.

MU5 - see Figure 4.

The many Atlas general-purpose registers proved not to be fully utilised by high-level language programs and were thus replaced in MU5 by explicit address-offset registers (e.g. Name Segment, Name Base, Stack Front), explicit structure-accessing hardware (the descriptor mechanism), and fast associative operand buffers (the Name Store). The MU5 instruction set, whilst still basically

one-address format, took advantage of these explicit registers to allow production of fundamentally more efficient object-code - measured to have an improvement of about 2.5 times when compared with Atlas compiled code. Since by the late 1960's there was little speed difference between available RAM and ROM technologies, there was no reason to have a separate ROM on MU5. However the Atlas concept of 'common software' persists and on MU5 all systems software is placed in Common Segments in the virtual memory to expedite sharing of libraries, compilers, etc. The Atlas virtual memory has been extended on MU5 to include explicit segmentation, variable page sizes, and enhanced facilities for sharing and protection. MU5 also marks for Manchester the move away from a single centralised mainframe. In MU5 all input/output and some of the one-level store management activity and file manipulation has been removed from the main CPU, thus leaving it freer for user-code execution. The Exchange and the MUSS portable operating system makes possible the connection of MU5 into a flexible multicomputer network of distributed processing capability. MU5 came into operation in the Autumn of 1974 and is still believed to be the fastest Algol processor under the sun, though the same cannot be said for Fortran.

Post-script.

The foregoing has selected but a few topics to illustrate the evolution of Manchester computer architecture. All the machines described have had their industrial derivatives, (the Ferranti Mark I, the Ferranti Atlas and, to a large extent, the ICL 2980). There has been a parallel development of Manchester software, starting in 1952 with what is thought to have been the first compiler (Ref.4). It should be stressed that much has been omitted from the present paper, and serious enquirers should work from the bibliography given in Reference 1.

Acknowledgement.

The author has pleasure in acknowledging the many helpful discussions with Professor T. Kilburn and all of the Manchester design group.

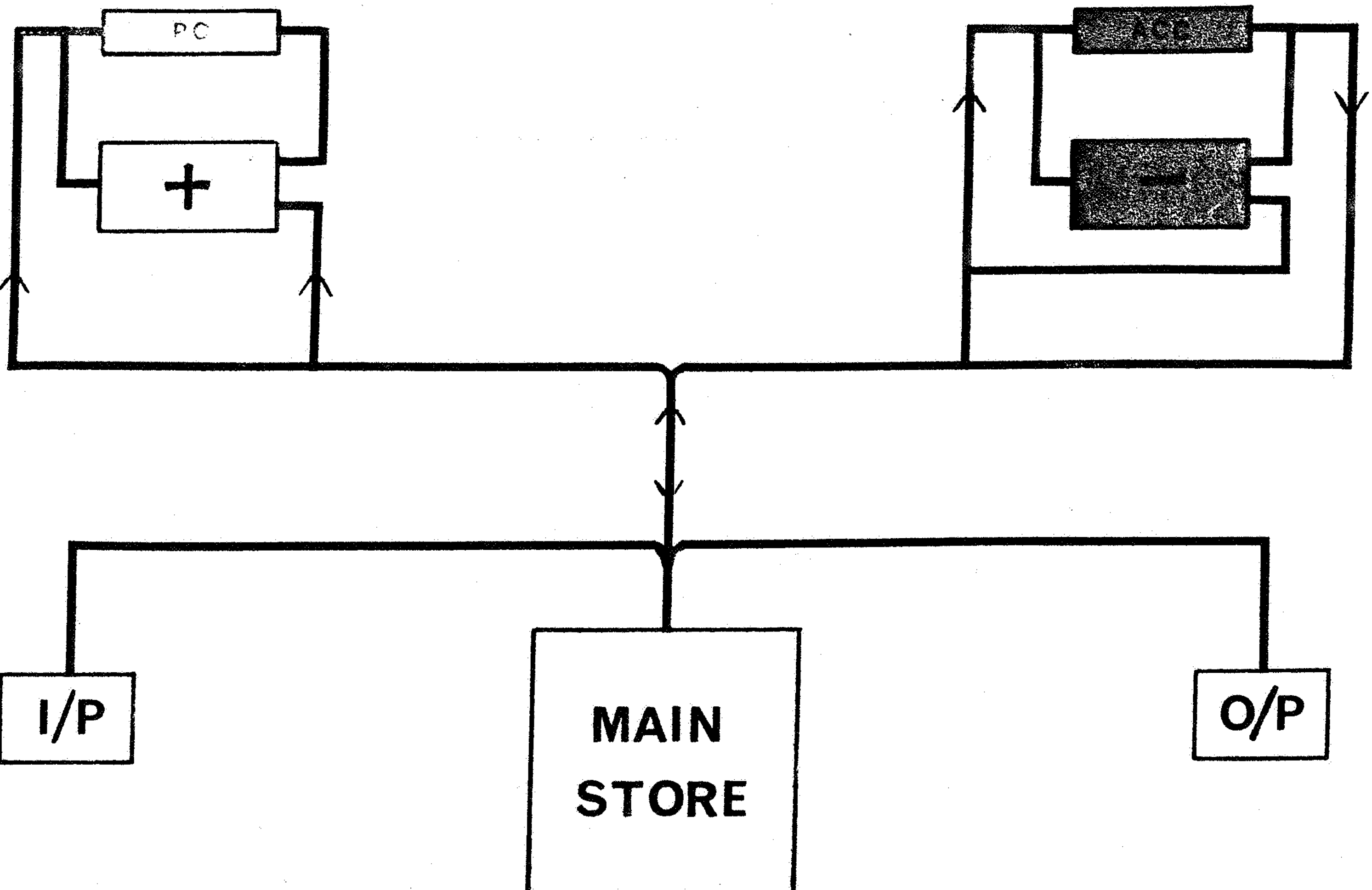
References

1. Lavington, S.H. A history of Manchester computers. National Computing Centre Publications, Manchester, England, 1975

2. Lavington, S.H. The Manchester Mark I and Atlas: a historical perspective. CACM Vol. 21 No. 1 January 1978, pages 4-12

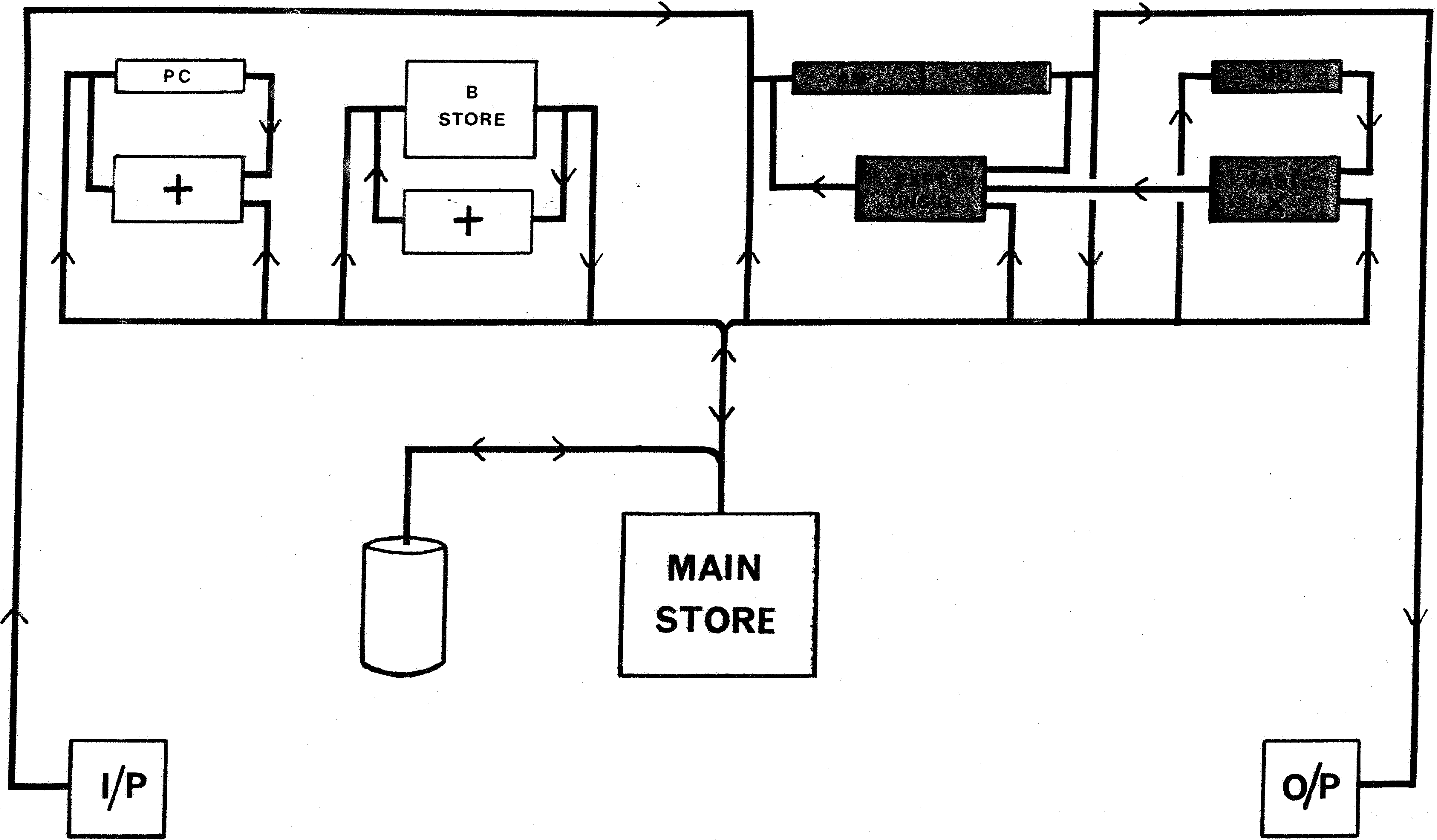
3. Ibbett, R.N. and Capon, P.C. The development of the MU5 computer system. CACM Vol. 21 No. 1 January 1978, pages 14-25

4. Knuth, D.E. and Pardo, L.T. The early development of programming languages. Report STAN-CS-76-562 from Computer Science Department, Stanford University, August 1976



PROTOTYPE MARK 1 — 1948

FIG. 1



MANCHESTER MARK 1 — 1949

FIG. 2

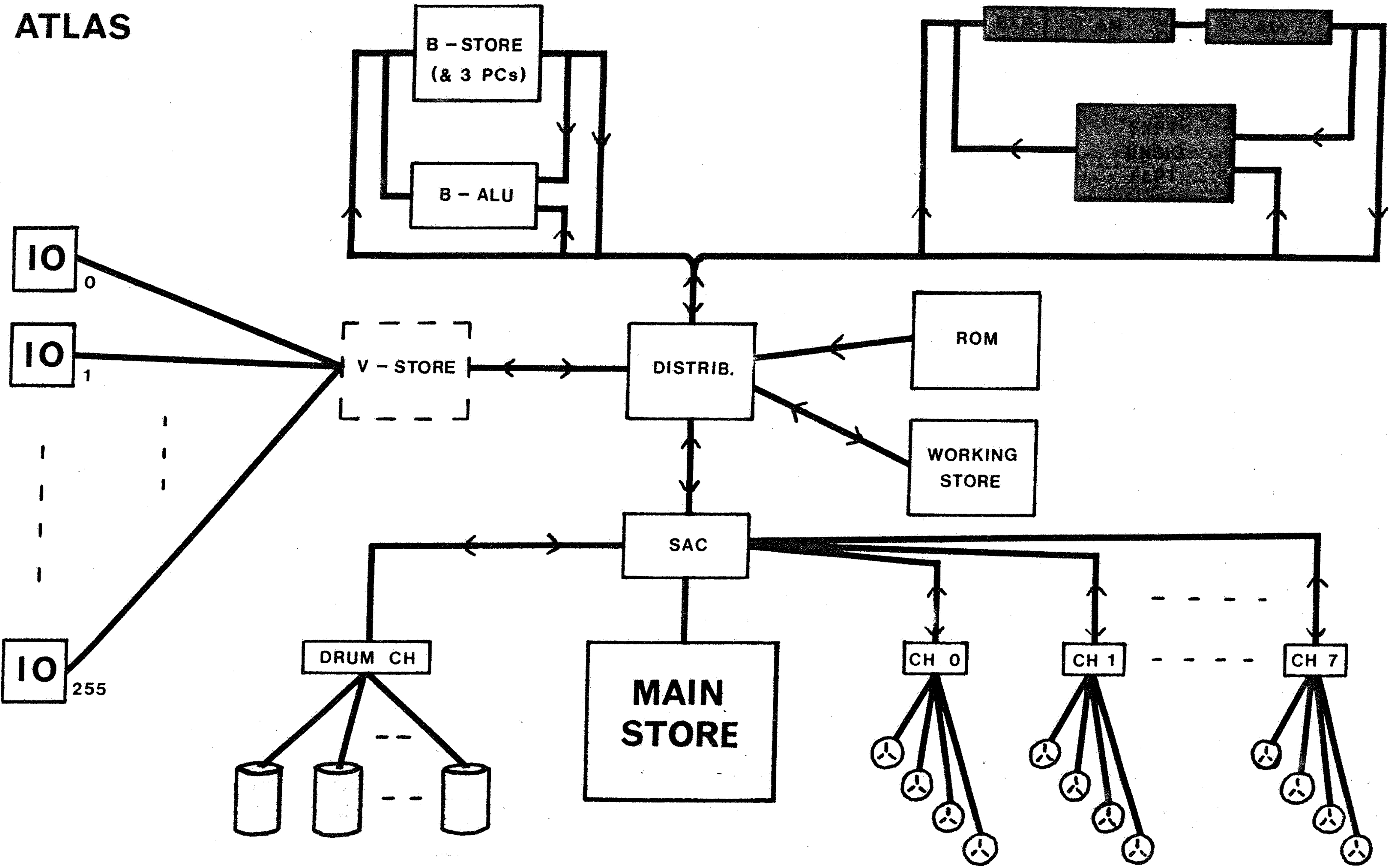


FIG. 3

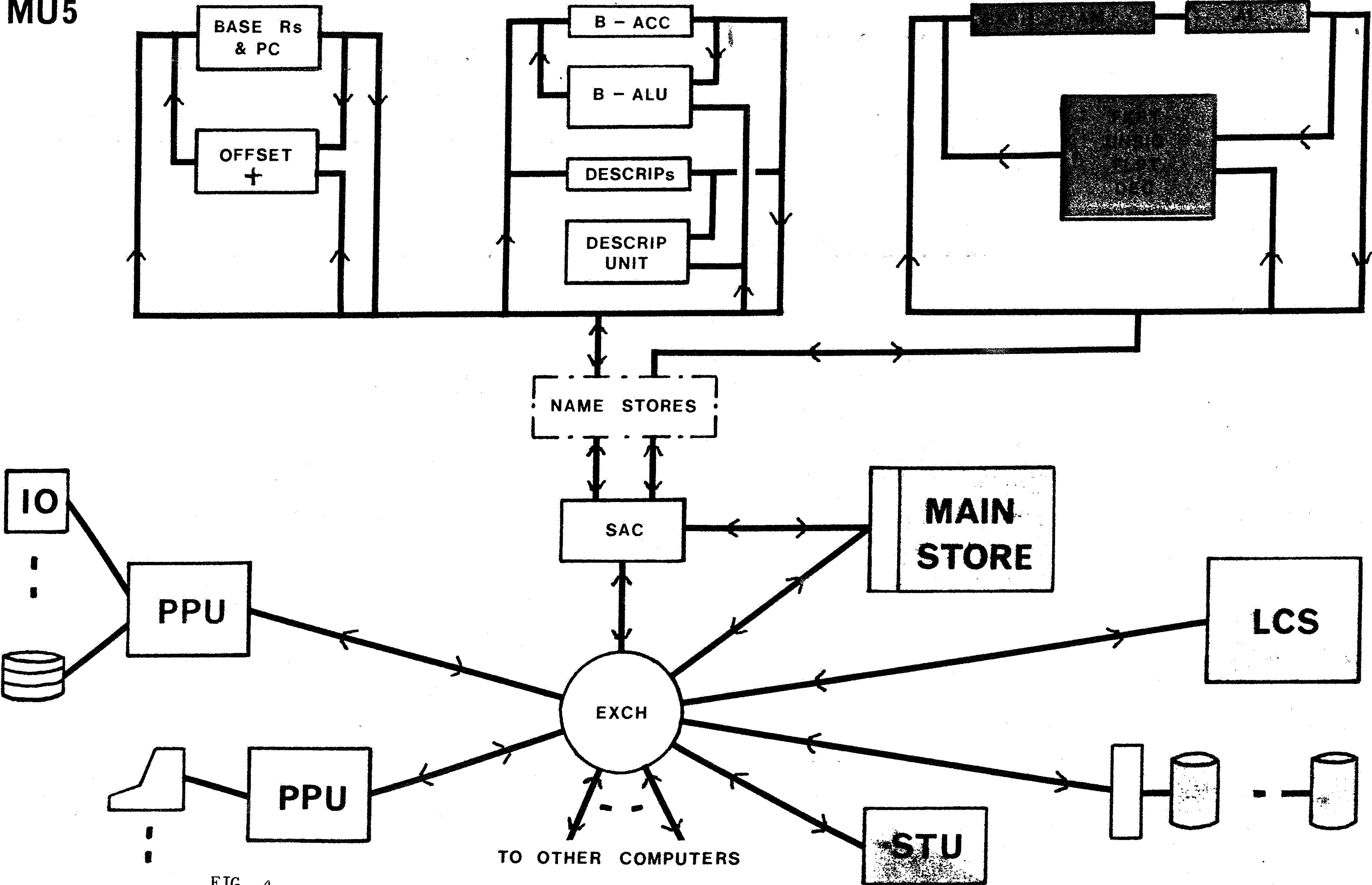


FIG. 4

THE ARCHITECTURE OF MU5

by

Derrick Morris, Ph.D., Professor of Computer Programming
Department of Computer Science, The University, Manchester.

ABSTRACT

The MU5 project at the University of Manchester, Department of Computer Science has been concerned with the design, construction and programming of a system intended for use as a general purpose computer utility. In this paper a description is given of the organisation of the instruction set and the way it shapes the architecture of the CPU.

This page in chapter 1

The design of MU5 was approached through the order code of the main CPU and the organisation of the stores (Ref.1). An instruction set was chosen to satisfy the following conditions:

(1) a) the generation of efficient code by the high level language compilers must be easy
(4) b) Programs must be compact
(3) c) The instruction set must allow a pipeline organisation of the CPU leading to a fast execution rate
(2) d) Information on the nature of operands (e.g. scalar or array element) should be available to the operand buffering scheme.

Economic arguments decided that the store should have several levels in which size would decrease as unit cost increased. The system design was aimed at making at least two levels of "core", a fixed head disc and a moving head disc appear to the software as a one level store. Provision was also made for the slower stores to be shared by several computers, possibly with different order codes and peripheral handling capabilities. A single operating system would be distributed across the machines. The job pattern assumed for this system is a timesharing one in which the sum of the job sizes would be approaching the size of the fixed head disc but the store requirement of individual jobs would usually be less than the size of the fastest "core". Minimising the store requirement of jobs has been an important consideration (hence condition (b) above). replace by brief O.S. requirement

There will only be time in this talk to consider the design of the CPU, but since the development of the system has now reached the stage where a full assessment can begin it is an appropriate time to revisit this design.

Choice of Instruction Format

The first step was to decide how many operands an instruction should have. Also whether the operand addresses were to be full store addresses, register addresses or store addresses contained in registers. It was decided from the start that, in order to comply with condition ~~(a)~~ d?, there should be an address form equivalent to each of the operand forms permitted by the high level languages. That not more than one such address should appear in each instruction followed from (b) and (c).

A second decision taken as a result of condition (a) was that no general registers would be provided for storing frequently used operands. Instead the hardware would incorporate associative memory for this purpose.

Thus the choice of format was between the zero address (stacking machine) type and some form of one address code. The attraction of the stacking machine from a compiler point of view is well known. In particular the simplicity of the algorithm for translating from ALGOL to Reverse Polish makes it the best choice for condition (a). However a zero address format was not chosen for the following two reasons.

First, measurements taken on Atlas indicated that the CPU spent more than half its time executing the (hand coded) basic library routines. Thus from a performance point of view this small amount of hand coded software was just as important as the user code. Most of these hand coded sequences worked out worse for the stacking machine than for the single address machine. This was usually because the main calculation, the address calculations and the control counting interfered with each other on the stack. A machine with several stacks would have worked rather better, for example

a control stack
an index stack
an address stack
and the main stack

This sort of arrangement would also fit the pipeline requirement better since the stacks could be distributed along the pipeline.

The second argument against the stacking machine would apply equally to a multi-stack organisation. Consider the example A:= B + C. For the two types of instruction format under consideration it would be coded as follows:

```
STACK B         ACC = B
STACK C         ACC + C
ADD
STORE A         ACC => A
```

If the operands normally come from store the execution times of the above sequences would be about the same being controlled by the access time for A, B and C. However if an operand buffering scheme is utilized giving a high hit rate (say > 90%) for operands such as A,B and C the access time to the stack becomes important. On MU5 the stack and the operand buffers would be the same speed, and the above example would have caused 6 stack accesses. Some, but not all, of the accesses could have been overlapped.

The instruction format eventually chosen for MU5 represented a merger of single address and stacking machine concepts. All the arithmetic and logical functions take one operand from an accumulator and the other operand is specified in the instruction address. Thus a sequence such as

```
ACC = B
ACC + C
ACC => A
```

typifies the style of simple calculations. However, there is a stack, and variant of the load order (*=) causes the accumulator to be stacked before being reloaded. Also a special address form exists (STACK) which unstacks the last stacked quantity. Thus the above example could be written in a form approximating to Reverse Polish, as follows

```
ACC = B
ACC *= C
ACC + STACK
ACC => A
```

A more realistic use of the stack is in conjunction with parenthesised subexpressions. For example, the expression (A+B)*((C+D)/(E+F)) would compile into

ACC = A
ACC + B
ACC *= C
ACC + D
ACC *= E
ACC + F
ACC ⊘ STACK
ACC * STACK

It is interesting to observe that if the first operand of a pair is stacked it subsequently appears as the second operand. Therefore for the non-commutative operations - and / the reverse operations ⊖ and ⊘ have to be provided.

ACC/OPERAND	means	ACC = ACC/OPERAND
ACC⊘OPERAND	means	ACC = OPERAND/ACC

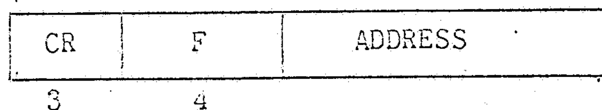
Only one stack is provided but there are five "accumulators" or Computational registers. Each may stack its content hence the effect is the same as having 5 stacks providing the order of unstacking corresponds to the way the stacked quantities are interleaved. This is usually the case. If it were not the conventional stacking machine would not be acceptable. Each of the five computational registers serves a dedicated function and they are distributed along the pipeline in close proximity to the arithmetic unit associated with that function. The arithmetic units are:

The B unit - used for index arithmetic and control counting
The D unit - used for address modification, bound checking etc
The A unit - the main arithmetic unit providing fixed point, floating point and decimal facilities

The registers are:

BM	a 32-bit modification register
DR	a 64-bit register for vector "descriptors"
XDR	similar to DR and used with DR by the string move orders
X	a 32-bit fixed point register in the A unit
A	a 64-bit register in the A unit.

The instruction format provided for operating on these registers is



One combination of the CR bits distinguishes a second format on which the main functions are for control branching and the manipulation of "addressing" registers. The remaining 7 combinations qualify the function (F) as follows:

1 fixed point operations on BM
2 fixed point operations on X
3 floating point operation on A
4 decimal operation on A
5 unsigned fixed point operations on A
(used for multi length working)
6 manipulation of DR and XDR
7 string processing functions (mainly for COBOL)

There is close similarity in the functions provided in groups 1 - 5, the following being typical:

=	load (32 bit operand)
='	load (64 bit operand)
*=	stack and load
⇒	store
+	add
-	subtract
*	multiply
/	divide
≢	Logical non-equivalence (exclusive or)
V	Logical or (inclusive or)
<-	Shift
&	Logical and
⊖	reverse subtract
COMP	compare
CINC	compare and increment
⊘	reverse divide

Until the mechanics of address generation have been described the example below will not be completely understood. It is given at this point to emphasise the close correspondence between the high level language form and the machine code. Each line except the JUMP order would be a 16 bit instruction if the example was taken from an average ALGOL program.

Example: W:= Z[I-1] * F + C(P,Q)/Y[J*3+K]

becomes

```
     BM = I
     BM - 1
     A = Z[BM]
     A * F
     STACK A
     STACK LINK L1
     STACK P
     STACK Q
     JUMP C
L1:  BM = J
     BM * 3
     BM + K
     A / Y[BM]
     A + STACK
     A => W
```

Address Generation

The aims of having an address form for each kind of high level language operand, and having compact instructions conflict. It was therefore decided to allow different sizes of address and to choose an encoding which represented the most common operand forms in the shortest addresses. It was also decided to have dedicated addressing registers whose functions related to the constructs of high level languages, rather than general purpose modifiers. This helped to satisfy conditions (c) and (d) as well as keeping the address size down.

An examination of the operands in high level languages led to the conclusion that provision should be made for

SCALARS
ELEMENTS FROM ARRAYS OR OTHER STRUCTURES
LITERALS
FUNCTIONS (i.e. PROCEDURE CALLS)

In the first two cases the procedure organisation of languages allows for operands which are:

LOCAL, (to the current procedure)
NONLOCAL (OR COMMON)
GLOBAL
STACKED

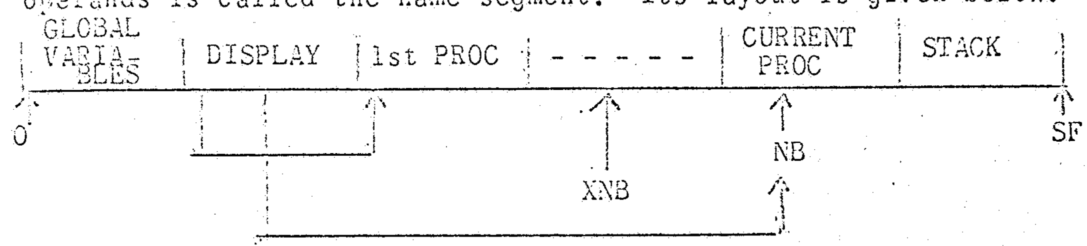
It has already been mentioned that the design incorporates a special functional unit (the D unit) for providing access to arrays and other structures. The route into these structures on MU5 is via "descriptors" which are accessed like scalars. Thus the problem with variables was viewed as one of making a primary access for an operand which would be a SCALAR or ARRAY DESCRIPTOR, then in the latter case passing the operand to the D unit for it to make a secondary access. For convenience, it was decided to allow literals of up to 64 bits to be coded explicitly into the instruction. To obtain the generality required for the ALGOLlike languages the mechanism for procedure calling had to be integrated into the stack concept.

Before continuing to describe the address generation in detail it is necessary to consider the expected store layout and the function of the dedicated registers. These are:

NB a pointer to the scalars and descriptors of the current procedure
XNB a pointer used to access any non-local or common scalars and descriptors
SF a pointer to the stack
0 this is a pseudo register always zero giving access to global scalars and descriptors.

For reasons discussed later the overall storage organisation provided each program with a segmented virtual store. One segment is used for the named operands (i.e. the scalars and descriptors) and the rest are used for the secondary operands (i.e. elements of arrays and other structures) and the code. The segment holding the named

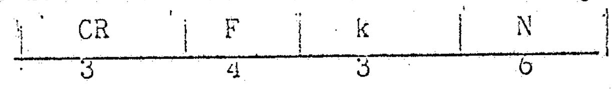
operands is called the name segment. Its layout is given below.



On entry to a new procedure the link stored contains NB, NB is reset to SF and SF is moved on by the number of names in the new procedure. Also an entry is made in the display.

Address Encoding

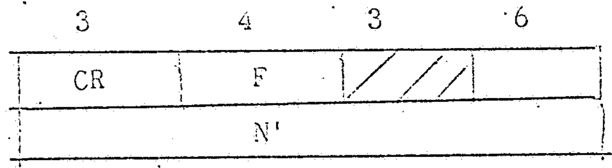
In the 16 bit instruction format the encoding chosen is



The N field corresponds to the operand name; e.g. the first declared name has N = 0 the second has N = 1 etc. Of the 8 combinations of k, which notionally specifies the kind of name, one is reserved to distinguish the extended addressing forms and the rest are:

- k = 0 - use N as a 6-bit signed literal
- k = 1 - use N as a register name (e.g. BM, X, etc)
- k = 2 - use N as the name of a 32-bit local scalar i.e. operand is 32-bit store line (NB+N)
- k = 3 - use N as the name of a 64-bit local scalar i.e. operand is 64-bit store line (NB+2N)
- k = 4 - use N as above but then pass operand to the D unit for a modified secondary access (name[BM])
- k = 5 - spare
- k = 6 - as k = 4 but secondary access is unmodified i.e. name[0]

There are three main requirements not met by the above which are provided in the extended address forms. First, there are the non local the global and stacked operands. Second, there are the procedures in which more names are declared than can be encoded in the 6-bit N. Third, there are the literals bigger than 6-bits. Thus both the k field and the N field have to be extended and the general form of extended instruction has the 32-bit format



From the detailed encoding of K given below it will be seen that in some cases the N' is omitted and the instruction again reduces to 16 bits. Also in the case of literals more 16 bit pieces may be added

up to the maximum instruction size of 80 bits. The K bits contain two 3 bit fields k' and ℓ which function as follows

K =	k'	ℓ
	3	3

k' = 0 means that the operand is a literal - in this case ℓ gives the length of the operand (16, 32 or 64 bits), and indicates whether it must be sign extended or zero filled.

k' = 1 is spare

k' = 2, 3, ..6 When ℓ = 2 the action is identical to the corresponding k values except the 16-bit N' replaces the 6 bit N. For ℓ = 0, 1 and 3 the action is similar except the base added into the N' is not NB but SF, 0 and XNB respectively. The action when ℓ = 4, 5, 6, 7 is explained below.

k' = 7 In this case the primary address is computed as above but instead of it then being applied to the name segment it is applied to the "V-store". Only the O/S is allowed to access V-store it contains, for example, the page address registers.

For l = 4, 5, 6, 7 the N' field is omitted hence the instruction is again only 16 bits.

ℓ = 4 is the operand from STACK mentioned earlier. Thus as well as taking the operand from the top of the stack it decrements SF. By combining ℓ = 4 with the appropriate values of k' the stacked quantity may be used as the operand or a descriptor used to access a secondary operand (e.g. STACK, STACK[B], STACK[0]).

ℓ = 5 can only be used with k' = 4 and 6. No primary operand is fetched and a secondary access is made using the current value of DR (e.g. DR[B], DR[0]).

ℓ = 6, 7 come accidently out of the decoding to mean the same as l = 2, 3 but with 0 replacing N'.

Secondary Operands

Descriptors are passed to the D unit together with an indication of whether or not modification is required. The unmodified descriptor is retained in the DR register and can be used again. If modification is specified the modifier is taken from BM.

Two main types of descriptor are provided. They are:

String Descriptors	Ts	LENGTH	ORIGIN
	8	24	32
Vector Descriptors	Tv	BOUND	ORIGIN
	8	24	32

String descriptors describe strings of bytes. If the string is short enough it can be accessed as a normal operand. Short strings are always right justified and filled out to the register size with zeros. A more usual use of the string descriptor is in conjunction with the string processing functions.

In the vector descriptors the type bits Tv control the modification and give the size of element in the array. This may be 1, 2, 4, 8, 16, 32, 64 or 128 bits, but the present MU5 hardware does not implement the sizes 2 and 128. As with strings short operands are right justified and zero filled. Normally the modifier is checked against the BOUND (and that it is ≥ 0), an interrupt is caused if the check fails. Before the addition of the modifier and ORIGIN occurs, the modifier is scaled. This means that the displacement caused by modification is in units of element size. Special bits allow both the bound check and the scaling to be inhibited.

The introduction of arithmetic type into descriptors was considered, but it did not work out satisfactorily with the named operands which could not be dynamically typed except at the individual word level. Since the benefits were not tangible in a machine intended for high speed execution of the standard programming languages the idea was dropped. However a software escape is provided through a special descriptor type which forces a procedure call whenever it is used.

It has been a constant source of regret that only one bound could be fitted into the descriptor. To compensate for this facilities are provided for the XDR register to point to a "dope vector" whilst the address of an element in a multi dimensional array is built up in DR. This dope vector contains triples which are the two bounds and the multiplier for each dimension. Each subscript is computed in BM and a special function is then used which checks against both bounds and computes the displacement adding it into DR.

The Interpretation of Addresses

Addresses presented to the paging unit and hence the store contain 30 bits to specify a 32-bit word. This is interpreted as a two dimensional address containing a 14-bit segment number and a 16-bit word address within a segment. Segmentation was adopted for several reasons. For example:

1) It is a convenient way of rationalizing the sparse use of a large virtual store
2) Segments can be shared between programs as a means of economising on store usage and also as a means of inter-program communication
3) Some automatic detection of software errors is possible if selective access status can be set up for each segment (e.g. read only, obey only, etc.)

The MU5 segments do not correspond to logical entities in the programming languages such as arrays and procedures. They correspond more to larger subdivisions of the program space such as:

the code
the name segment
the segment of arrays
the input buffer
the output buffer

Performance

A prime objective of the design has been a high throughput of high level language programs. This has meant the removal of redundancies in the compiled code and the achievement of a fast execution rate. A peak rate of 20 MIPS (millions of instructions per second) was the target although an average rate of 10 MIPS or less was expected. To obtain these speeds with a store cycle time of 250nsec required the store accessing to be kept to a minimum. The hardware therefore incorporates high speed buffers, and it uses separate buffers and buffer loading strategies for

instructions
named operands
vector operands

A description of this hardware organisation has been given elsewhere (Ref.2). The degree to which it succeeds will soon be known.

References

1. Kilburn, Morris, Rohl, Sumner: A Systems Design Proposal IFIP 68. Vol.2 pp.806 - 812

2. Ibbett: Computer Journal, Vol.15 No.1 Feb.1972

SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 8, 131–136 (1978)

A Study of Machine-level Software Profile

AMNON B. BARAK AND MOSHE AHARONI*
Department of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel.

SUMMARY

The instruction mix of a CDC CYBER/74 computer in a university environment was monitored, and in this paper frequencies of execution for the most commonly used instructions are given. From these measurements we make a number of observations about several aspects of computing patterns. One observation is the fact that if we exclude the idle loop of the operating system, the percentage of occurrences for each type of instruction over various time intervals is constant. This fact is used to define a machine-level software profile (MLSP) for the type of machine operations in the given computing environment. It is shown that the MLSP could be used to find machine utilization and the extent to which software takes advantage of machine architecture, and as a consistent method to improve the performance of a machine configuration.

KEY WORDS Instruction mix Performance evaluation Software profile

INTRODUCTION

The question of what a computer really does in the course of running a stream of jobs is of great interest to software engineers as well as to designers of computer systems. A digital computer is a finite state machine. Any sequence of operations is described by the sequence of states through which the machine passes. Measurements of this sequence (the instruction mix) could be used to define a software profile of the measured computer, and the instruction mix could then be used to guide performance improvments of a given machine configuration.

There are several interesting studies of static and dynamic execution rates for high- and low-level languages. Knuth studied in Reference 5 various aspects of FORTRAN programs. A similar study of commercial PL/I programs was reported by Elshoff.[2,3] Wirth gave in Reference 6 statistics about execution of PASCAL programs on a CDC 6400 computer. A brief summary of instruction mixes for scientific and commercial computers was reported by Fuller.[4]

In this paper we study the instruction mix of a CDC CYBER 74 computer in a university environment. We define the instruction mix of a representative set of programs written over a subset of languages to be a machine-level software profile (MLSP). This is to distinguish it from a high-level software profile which is a dynamic frequency of execution of high-level language statements. We note that while a high-level software profile for a given language is machine-independent, MLSP is machine-dependent and therefore could be used to characterize the type of machine operations in a given computing environment.

* Moshe Aharoni is now with Elscint Electronics Inc. Haifa, Israel.

0038–6644/78/0208–0131$01.00

© 1978 by John Wiley & Sons, Ltd.

Received 23 August 1976
Revised 27 May 1977

The main result of this paper is that if we exclude the instructions that are executed by the idle loop, the percentages of the number of executions of the remaining instructions, over different periods of time, are constant. This fact enables us to define the MLSP for the measured computer.

Studies of MLSP for different computers might be useful for understanding patterns of computing. These patterns could be used to improve the design of software and hardware systems. In this paper we use the MLSP to find the rate of parallelism of the CYBER 74. The MLSP could also be useful in selecting a new computer or improving the configuration of an existing one.

Since we know that the configuration, the operating system and the job mix of our computer is similar to many other computer installations, particularly in universities, we predict that our results reflect the MLSP of most of these installations.

In the next section we give a brief description of the CDC CYBER 74 computer, the operating system and job mix at the Hebrew University. We then list the MLSP and draw conclusions about software and hardware utilization of this computer.

THE ENVIRONMENT

The CDC CYBER 74 is a highly parallel computer. The CPU has 10 independent functional units (2 Increment, 2 Multiply, Shift, Boolean, Floating Add, Integer Add, Divide and a Branch unit), 24 operating registers and an instruction stack which holds up to 27 instructions. The central memory has a capacity of up to 131K 60-bit words with an independent bank construction of 4K words and transfer rate of up to one word in each Minor Cycle (100 ns). For further details about the CYBER 74 and similar CDC 6000 Series computers, see the CDC reference manual.[1]

The CYBER 74 computer at the Hebrew University has 65K words of core memory, one 13·1 million words disk, model 808, eight 11·8 million words disks model 844 and four magnetic tape units. The operating system that is used is SCOPE 3.4.

Research and student jobs make up 95 per cent of all jobs run. The languages that are used are FORTRAN (87 per cent), PASCAL (5 per cent) and SNOBOL (3 per cent). Three different compilers are used for FORTRAN compilations, RUN (80 per cent), FTN (16 per cent) and MNF (4 per cent).

MACHINE-LEVEL SOFTWARE PROFILE FOR A CDC CYBER 74

In this section we give an MLSP and percentage of serial execution time for a set of instructions that are most frequently used by the CDC CYBER 74 computer at the Hebrew University.

Measurements were taken over a period of several months. The results are listed in Table I, where instructions are ordered according to their type. For each instruction we give the percentage of occurrences, average number of occurrences in one minute and average execution time (Minor Cycles). Also listed are some statistics about serial execution time.

Noticeable variations in the system load were reflected by the time spent in the idle loop. This loop is a system subprogram which is executed by the monitor whenever all in-core jobs are not active.

The main result of this paper is that if we exclude the idle loop, then except for some minor variations, the percentage of occurrences for each type of instruction remains unaffected. This fact enables us to define the MLSP as a representative instruction mix for the measured computer.

Table I. Statistics of instruction execution

Instruction type	MLSP (%)	Occurrence in minute (millions)	Execution time (MC)	Serial execution time (s)	% of serial execution time
Data Movements and Increment					
LOAD	29·0	40·14	8	32·11	35·05
STORE	7·2	10·00	5	5·00	5·46
Index increment	11·3	15·67	3	4·70	5·13
Total inc. unit	47·5	65·81		41·81	45·64
Arithmetics					
Add/Sub	4·9	6·85	4	2·74	2·99
Integer Add/Sub	2·5	3·45	3	1·04	1·14
Multiply unit 1	2·5	3·48	10	3·48	3·80
Multiply unit 2	0·4	0·49	10	0·49	0·53
Divide	0·05	0·07	29	0·20	0·22
Total Arith.	10·35	14·34		7·95	8·68
Control					
Uncond. branch	2·9	4·05	14	5·67	6·19
Condit. branch	9·8	13·70	8–15	16·44	17·94
Return. jump	7·2	10·08	13	13·10	14·30
Exchange jump	0·03	0·04	16	0·06	0·07
Total branch unit	19·93	27·87		35·27	38·50
Other					
Boolean	7·4	10·27	3	3·08	3·36
Shift	5·3	7·38	3–4	2·21	2·41
NOP	9·4	13·04	1	1·30	1·42
Total	100	138·71		91·62	100·00

We note that the number of occurrences for each instruction that is listed in column 2 of Table I represents a CPU run in which there are no idle loops. A practical method to measure the idle loop was to modify SCOPE's idle loop from

```
* EQ   BO, BO,*
```

which is a one instruction loop to

```
* CXi  Xk
EQ   BO, BO, *
```

and then monitor the number of occurrences of the CX instruction. This modification is necessary since EQ is a heavily used instruction, while CX, which is a population count of the number of one bits in X_k, is rarely used.

ANALYSIS

We now analyse certain aspects of computing which are reflected by the MLSP.

Idle Loop

As noted in the last section the Idle Loop (IL) is used by the monitor when there are no active jobs in the system. Every few milliseconds, the IL is interrupted, and a check is made whether any job can be resumed. The number of IL executions in a certain time interval gives a measurement of CPU utilization.

A theoretical maximum for the number of executions is 66·6 million IL per minute (MILM). This value results from the fact that a single IL is executed in 9 minor cycles (900 ns). In practice we measured 65 MILM during short intervals, when no active job was in the System. The minimum value for the number of IL executions is zero. This value was actually measured when a set of CPU bound jobs were executed.

The maximum and minimum values of the IL were measured over short intervals (a few minutes). Continuous measurements over longer periods (hours or days) are necessary in order to gain information about CPU utilization.

Our main conclusions are:

(a) a significant number of IL during peak hours could be a result of unbalanced machine configuration, e.g. lack of memory or insufficient number of I/O channels; further study of the causes of the IL is necessary in the second case;

(b) if the operating system is idling, the number of IL should approach the maximum; otherwise the system is inefficient;

(c) continuous measurements of the IL give an accurate percentage of CPU utilization; this information could be used to predict possible increases in computing capacity.

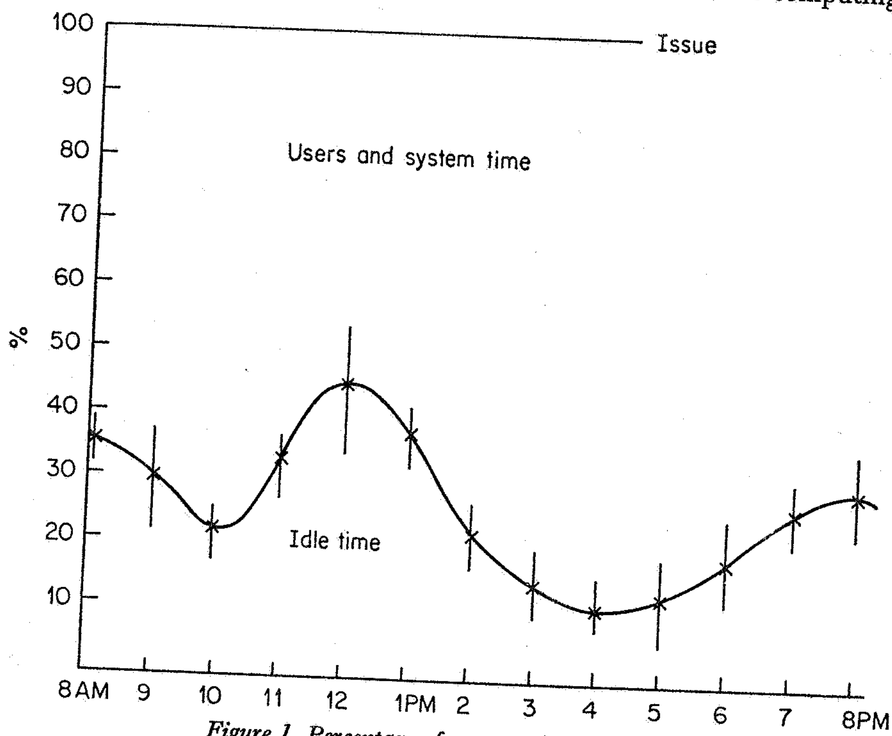


Figure 1. Percentage of average daily idle loop

In Figure 1 we give measurements of the average daily idle loop. Dataw were collected over a period of several weeks during the regular teaching term. We note that instructions are issued at the approximate rate of 138 million instructions per minute.

We believe that the idle loop never drops below 10 per cent because of lack of memory.

Software usage of machine architecture

The CYBER 74 computer is capable of executing several instructions in parallel. This is done by using different functional units where each unit can execute only one type of

instruction. For this reason several consecutive instructions of the same type might slow execution speed because they are performed by the same functional unit. The Multiply and Increment units are exceptions since these units are duplexed. In this case an instruction is sent for execution to the second unit if the first unit is busy.

An interesting observation about the extent to which software takes advantage of machine architecture can be made by comparing actual execution time to the serial execution time for a given code, particularly the MLSP.

By summing the number of occurrences of each instruction multiplied by the average execution time, we get the serial execution time. For the MLSP of Table I we have

(a) actual execution time: 60 s;
(b) serial execution time: 91·62 s.

We can now find the parallelism rate of the CYBER 74 by dividing the serial execution time by the actual execution time. For the MLSP of Table I this rate is 1·52.

Since the MLSP of our computer consists primarily of machine code that was generated by FORTRAN compilers, we became interested in finding the rate of parallelism for non-FORTRAN codes. Preliminary results show that this rate is 1·24 for PASCAL programs.

Exchange jump

The Exchange Jump (XJ) instruction is used to transfer control between tasks that are present in core. This is done when a job requests system interaction, mainly to perform I/O. We note that I/O operations are performed by a set of ten independent Peripheral Processors. When executed, the XJ instruction stores in core all CPU registers and reloads a new set of registers. Except for I/O, transfer of control is performed according to a certain priority function. This function depends on the available resources, job priority, job history and overall monitor activities.

Careful study of the MLSP could lead to an optimized priority function, thus improving the performance of the operating system.

Improvements of the configuration of an installation

Improvements of an installation performance can be done by using tables of MLSP for identical CPUs with different installation configurations. Similar information might be useful to guide the purchasing of a new computer with a desired configuration.

Benchmark models were traditionally used to compare workloads of systems. They are frequently used to aid in the selection of competitive hardware and software systems. However, there are many difficulties in constructing a real performance benchmark model for a desired configuration and only limited information is available. The method described in this paper combined with further studies which include input/output characteristics might lead to a more consistent approach to the subject.

Ordinateur

Our final observation is to compare the percentage of data movements and control instructions in the total number of instructions. As can be seen from Table I, data movements and control instructions constitute more than 67 per cent of the total number of instructions. By comparison, pure arithmetic instructions are executed in less than 11 per cent of the total time. We therefore conclude that our computer is actually an ordinateur (a French word for computer which literally means a device to manipulate data).

CONCLUSIONS

The concept of machine-level software profile was introduced. It was observed that if the idle loop is excluded, then for a given environment this profile is fixed. From this observation we drew some conclusions about aspects of computing, such as software usage of machine architecture. We also suggest a consistent method to gather information about the performance of a given machine configuration.

A suggestion for further study could be finding the MLSP for various high-level languages. Comparisons of MLSP for different computers might be useful in the study of performance evaluation.

ACKNOWLEDGEMENTS

The authors are grateful to Zvi Riv for his assistance. Thanks are also due to members of CDC and the Hebrew University Computing Center.

REFERENCES

1. *CONTROL DATA* 6000 *Series Computer System, Reference Manual* 1974.
2. J. L. Elshoff, 'An analysis of some commercial PL/I programs', *IEEE Trans. Software Engng*, **SE-2**, No. 2, 113–120 (1976).
3. J. L. Elshoff, 'A numerical profile of commercial PL/I programs', *General Motors Research Publication GMR-1927* (1975), also *Software–Practice and Experience*, **6**, No. 4, 505–526 (1976).
4. S. H. Fuller, 'Performance evaluation', in *Introduction to Computer Architecture* (*ed.* H. S. Stone), The SRA Computer Science Series, Chicago, 1975.
5. D. E. Knuth, 'An empirical study of FORTRAN programs', *Software—Practice and Experience*, **1**, No. 2, 105–133 (1971).
6. N. Wirth, 'The design of a PASCAL compiler', *Software—Practice and Experience*, **1**, No. 4, 309–333 (1971).

MU5 – AN ASSESSMENT OF THE DESIGN

F. H. SUMNER

The University of Manchester, Manchester, England

(INVITED PAPER)

The first proposals for the MU5 computer were presented at the IFIP Congress in 1968. The system is now built and this paper summarises the design. Attention is drawn to two of the most significant features, firstly the addressing of operands with the division into two classes, names and array elements, and secondly the fetching of instructions so that the gaps in processing caused by branch orders are minimised. The correctness of the structure is examined and possible extensions are suggested.

1. INTRODUCTION

At the IFIP Congress in 1968 a paper was presented [1] describing the initial ideas for the MU5 computer system. In the six years since 1968 this design has been completed and the hardware and much of the software for the system has now been implemented. The system is expected to be made available to users early in the summer of 1974 and the following period will be devoted to a detailed study of the behaviour and performance of the complete system.

Since freezing the design in 1971 subsequent work on implementing the hardware and software has increased our belief in the correctness of many of the design features and has also revealed areas where changes would lead to improved performance. It is the purpose of this paper to assess the design of the MU5 in the light of the experience gained in bringing it to a working system.

2. SUMMARY OF SYSTEM DESIGN

There have been many papers describing the system in detail (e.g. [2 and 3]). In this paper a brief description will be given which covers the significant features of the design.

The complete system is a complex of several processors and stores linked by an exchange as shown in fig.1. It is expected that stores and processors will be added and removed as the project develops. Each main processor has direct access to its own store as well as the common accesses through the exchange; this is essential if they are to operate at high speed.

The aim in the design of the MU5 central processor was to produce a machine whose structure is well suited to the needs of modern high level languages. It was also hoped that the system would have a power of some twenty times Atlas.

Following our experience on ATLAS in which relatively inefficient use was made of the fast registers it was decided that in MU5 there would be no explicitly addressed fast registers, all operands would be addressed as though in the virtual address space of the current process. The address space for each process is quite large and is divided as follows.

14 bits to define the segment
16 bits for page/line
2 bits for the byte address

The page/line division is not fixed, the page size is variable in powers of 2 from 16 to 64K. The management of the variable page size has not proved

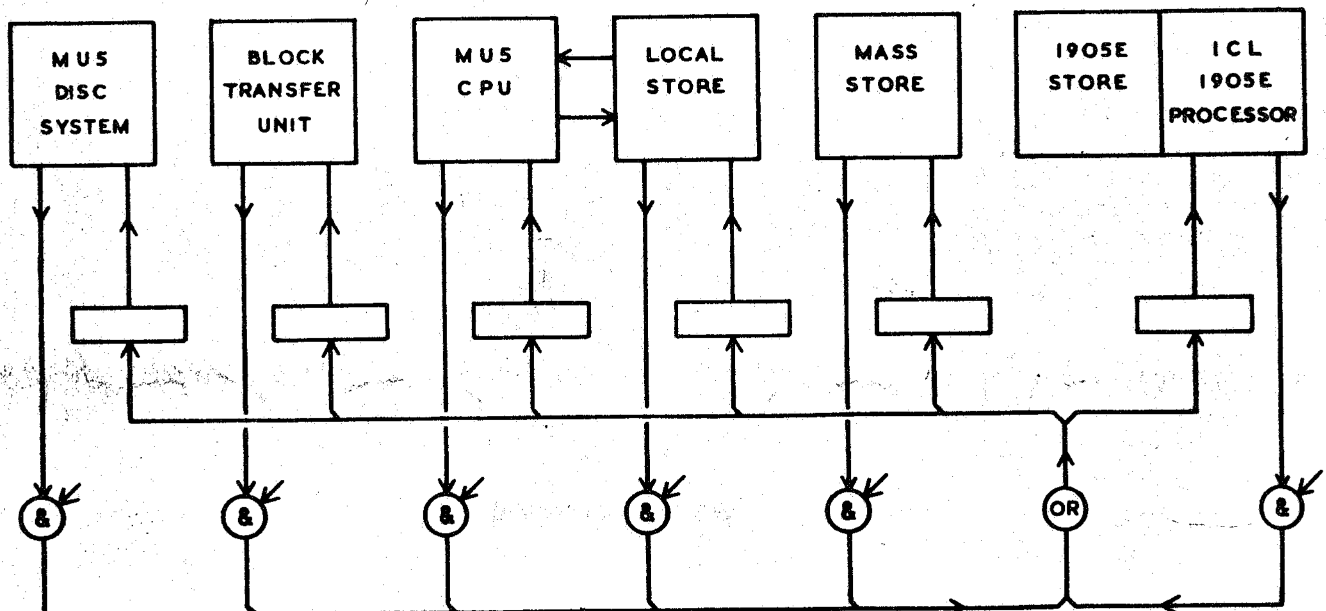


Fig. 1. The MU5 Multicomputer System.

too difficult and the additional overheads beyond normal fixed length paging should not be significant. The page size may be different for each segment and one of the major programs of research over the next two years will be the investigation of the advantages and disadvantages of this flexibility.

High level languages generally demand from the programmer information about the types of operand he intends to use. In the MU5 design, this information is not thrown away but is made available to the computer hardware so that it can increase the efficiency of the store organisation.

The types of operand commonly in use in high level languages may be listed as follows:-

(a) Literals consisting of fixed-point and floating point constants.
(b) Primary operands consisting of:-
  (i) integer and real variables
  (ii) data descriptors
  (iii) names of functions and routines
(c) Secondary operands consisting of:-
  (i) elements of arrays and vectors
  (ii) strings of elements of structured data sets.

Only primary operands, which we have called named quantities, can be addressed directly. The instruction format of MU5 has two fields the function and an address; this address is of a primary operand and defines the operand within the current procedure the full virtual address being obtained by adding a name base to the "name" N. All named quantities are kept in segment zero and the remaining segments are used for instructions (in pure procedure segments) and for array elements etc. These latter are accessed indirectly by means of a named descriptor, which defines the origin, and a precalculated indexed quantity which gives the dispalcement from the origin. The system of operand accessing is shown schematically in fig.2. It is expected that the majority of instructions will only be 16-bits long though of course longer ones of 32, 48 and 80-bits are provided for long literals or for long addresses.

The component parts of the MU5 processor are shown in fig. 3. It was stated above that a design aim was a performance of 20 times ATLAS and to this end the instruction unit is designed to prefetch instructions and to supply these to the primary operand pipeline (PROP) at a maximum rate of one every 40 nsec. This rate is impossible within PROP if all operands have to be brought from the store as the access time including address translation and cables is about 500 nsec. To overcome this there exists within PROP a 32 line associative memory with an access time of 40 nsecs. which traps up to 32 named quantities. If a name is in the fast memory a rate of one instruction per 40 nsec. can be maintained by PROP but the penalty for a miss will be a gap of at least 500 nsecs. The performance of the entire system is therefore critically dependent on the performance of this associative memory. A hit rate in the region of 99% is expected.

Instructions concerned with index arithmetic are completed within PROP but instructions which result in accessing an array element, or which require use of the main arithmetic units, proceed to the secondary operand pipeline (SEOP).

3. OPERATION OF THE SYSTEM - OPERAND ACCESSING

The operation of the system is best illustrated by a simple example, consider the evaluation of the following simple sequence.

1) a := a + b(i+3)

   i = i + 1

   go to 1) if i<n

(a and the elements of b are floating point numbers)

The first instruction loads the value of the named real quantity into the floating point arithmetic unit; as "a" is a named operand it should be read out of the name store in PROP and then sent via SEOP to the arithmetic unit. Then come two orders to form i + 3; i is read from the PROP name store; 3 is a literal; and the arithmetic is done in the index arithmetic unit.

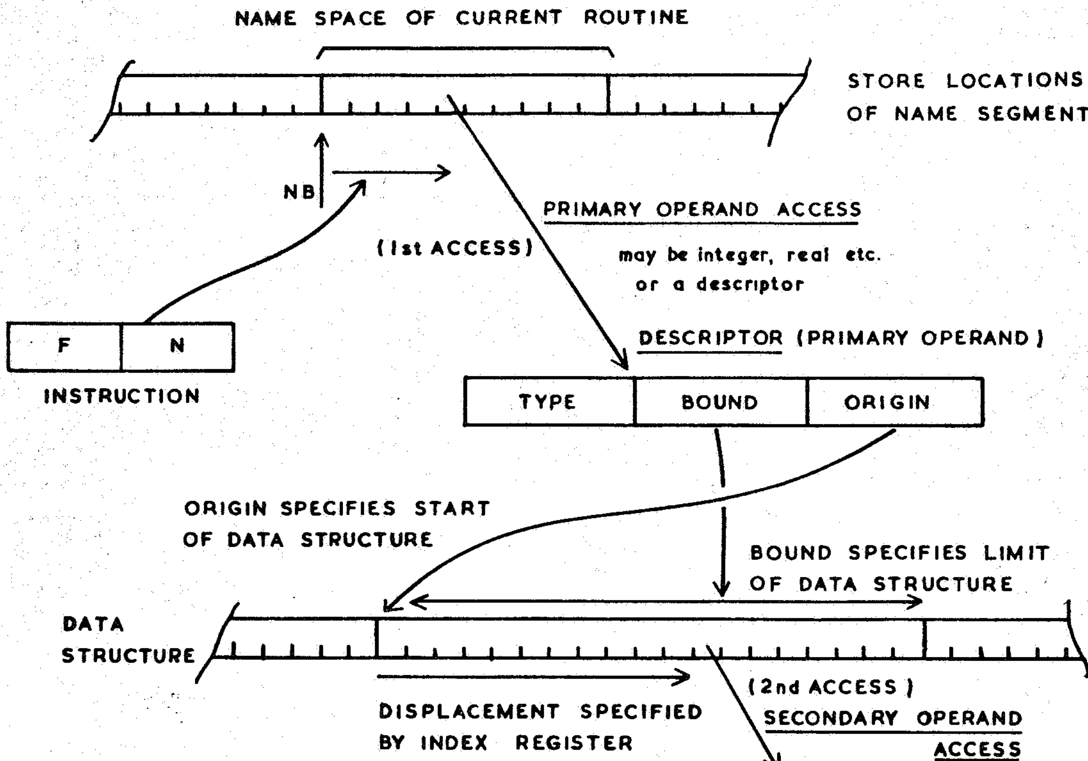


Fig. 2. Operand Address Development

B-UNIT
index arithmetic
instruction accessing
initial instruction processing
INSTRUCTION BUFFER UNIT
PRIMARY OPERAND PIPELINE
DESCRIPTOR UNIT
SECONDARY OPERAND PIPELINE
MAIN ARITHMETIC UNIT
address modification
operand manipulation
request for instructions
processor's dedicated store
instructions
STORE ACCESS CONTROL
LOCAL STORE
interface with store system
ADDRESS ROUTES
DATA ROUTES
TO EXCHANGE

Fig. 3. The MU5 Central Processor.

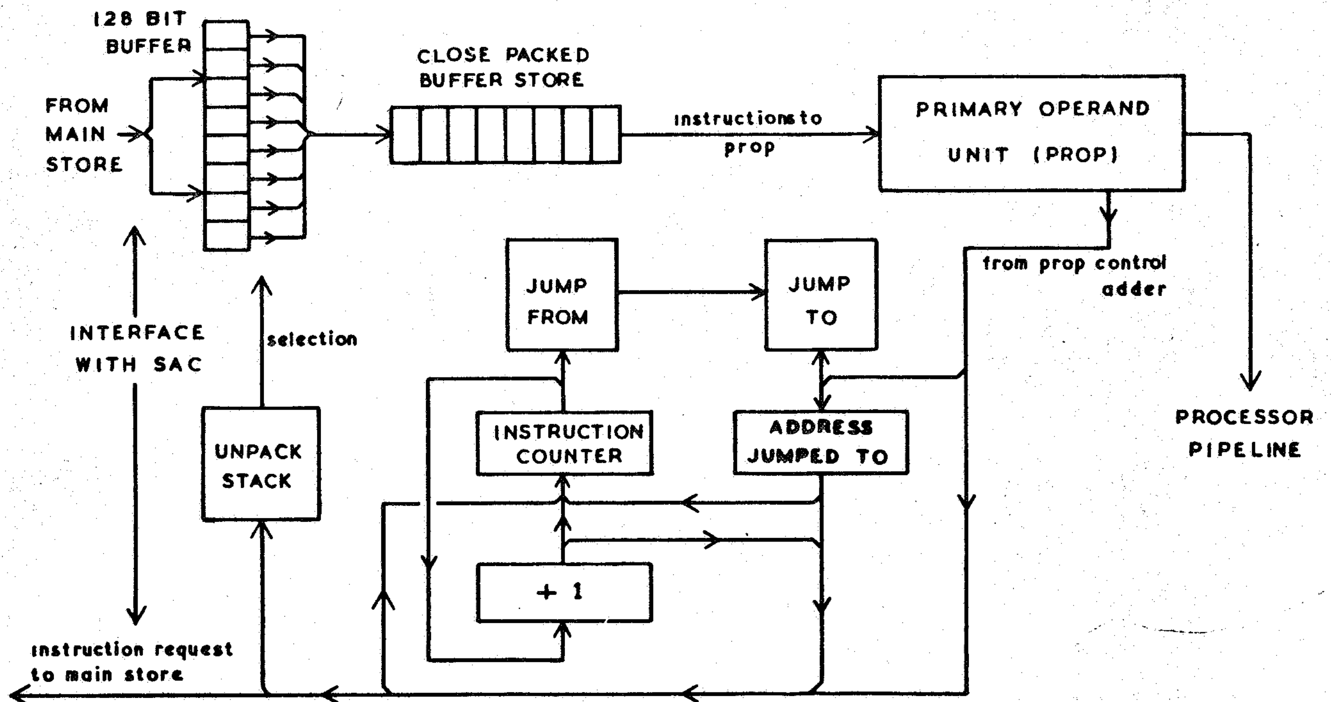


Fig. 4. Instruction Fetching.

These orders will be completed long before A gets through SEOP to the floating point arithmetic unit. The next order reads the descriptor for the vector B from the name store, forms the address of B(i+3), reads the value of this element from the store and sends it via SEOP to the floating point arithmetic unit for addition to a. The final order of this group writes the contents of the floating point arithmetic unit back into the value of a in the name store. This writing obviously presents many problems; there will be a long delay before the information is available and any subsequent reference to the name store for a will have to be detected and held up. These and other arguments led to the decision to split the name store into two parts, one remaining in PROP to trap integers and array descriptors and the other part for reals being placed at the end of the SEOP near to the floating point arithmetic unit. All the names still remain in the same segment of the virtual space and the hardware is therefore required to perform detailed checks on where any particular virtual address is currently living. A better solution may be to extend the concept of matching the address structure with the language structure and to define a distinct segment of virtual space for reals. The associative memory to trap elements of this new segment could then be placed near the arithmetic unit which manipulated reals and the organisation and management of this associative memory could be arranged to match the pattern of usage for reals. A further major difficulty with the present design is illustrated if B(i+3) in the above problem is replaced by B(i,j). A commonly used way of accessing an element of a two-dimensional array is to use the first descriptor and the value of i to read a descriptor for the ith row from a vector of descriptors; this is then used together with j to access the jth element of this row. The access for the second descriptor will go via SEOP as it itself is an array element. Whilst a separate path is provided so that the access is not held up by operands destined for the floating point arithmetic unit, the access for the element B(i,j) is still a very lengthy process. A further extension of the above argument is to place both single descriptors and vectors of descriptors in the same segment and to have a third associative memory placed near to the address calculation unit to trap elements of this segment.

4. OPERATION OF THE SYSTEM - INSTRUCTION FETCHING

Returning to the program given above the next orders increment i and jump if i is less than n. In any pipeline system discontinuities in the instruction stream caused by branch orders give severe degradation in the performance and an average pipeline efficiency of 20 to 25% is generally considered to be acceptable. When a conditional branch is obeyed the worse delay is A + B where A is the access time for the jumped to instruction and B is the length of the pipeline. For small loops all the instructions may be trapped in a fast buffer reducing the loss to something approaching B. However, B is approximately the same magnitude as A and the losses are therefore still large. Furthermore, for large or complex loops, trapping of the loop instructions within a small buffer is not possible.

The MU5 instruction fetch unit(IFU)is shown in fig. 4. A small associative memory is used to remember previously obeyed pairs of jumped from and jumped to addresses so that on a subsequent pass through the same part of the program the early call system will see the jumped from address and cause the orders from the jumped to address to be prefetched. Thus if the jump occurs again there will be no pipeline gap at all. This simple system will have gaps on the first obeying of a jump and also when an order which has previously jumped decides to go straight on. However, it is expected to correctly predict about 60% of all jumps and this approach appears to be far more cost effective than a large buffer of more conventional design. Only branch orders with literal operands can be predicted in this way so many orders such as return links still cause gaps of A + B. The system could easily be expanded to include branch orders with operands from store locations but other more ambitious improvements are also possible. If the associative memory in the IFU could be written to by the object program it would be possible for the compiler to plant instructions which cause the appropriate address pairs (jumped from/jumped to) to be set in the buffer before the first occurrence of the jump and also to be deleted after the last occurrence. These extensions would prevent many of the remaining pipeline gaps and the overall efficiency could be greatly increased. This does not appear to be an unreasonable demand on the compiler-writer particularly for code which is frequently obeyed such as that within the compiler library.

5. POSSIBLE EXTENSIONS TO THE SYSTEM

The problem above of accessing vectors of descriptors is a particular example of the more general problem of list processing, where, for the present purposes this means any situation in which the next operand cannot be accessed until the present one has been brought back to the CPU and possibly processed in some way. A pipeline processor is unsuited to this type of problem as little or no overlap of instruction execution is possible. The evidence we have to date is that the object code of FORTRAN and ALGOL 60 will be executed efficiently but the compilers themselves will be less efficient and furthermore the code executed in the operating system will be even less suited to the structure of the hardware. The optimistic way of looking at this is that the operating system orders will be obeyed as efficiently as in conventional computers and the compilers will be better and object code will be very good. As shown in fig. 1, MU5 is a multi-computer system and it was always intended that all the input/output would be done by the 1905E rather than the MU5 processor. Perhaps the correct extension of the design is to have other computers in the system which are designed to be good at obeying compiler code or operating system code. For an overall system ratio of say 10% operating system, 15% compiling, 75% execution it would be easy to design processors for these particular jobs. The drawback of this approach is that if the balance of activities varies one or other of the special purpose processors will be overloaded, and even if this does not happen over long periods there are bound to be short bursts of activity for individual processors with the others remaining idle. These extra processors will not however be expensive and it seems reasonable to investigate the possibilities of extending the basic MU5 concept of the structure matching the problem to the structure of the whole system matching the structure of the complete work load.

ACKNOWLEDGEMENT

MU5 is the work of a large research group at the University of Manchester under the general direction of Professor T. Kilburn.

REFERENCES

[1] T. Kilburn, D. Morris, J.S. Rohl and F.H.Sumner, A system design proposal, Information Processing 68, North-Holland, Amsterdam, 1969.

[2] D. Morris, G. Detlefsen, G.R. Frank and T. Sweeney, The structure of the MU5 operating system, The computer journal, May 1972.

[3] R. N. Ibbett, The MU5 instruction pipeline, The computer journal, February 1972.

Computer Systems

G. Bell, D. Siewiorek, and S. H. Fuller, Editors

The Manchester Mark I and Atlas: A Historical Perspective

S. H. Lavington
University of Manchester

In 30 years of computer design at Manchester University two systems stand out: the Mark I (developed over the period 1946–49) and the Atlas (1956–62). This paper places each computer in its historical context and then describes the architecture and system software in present-day terminology. Several design concepts such as address-generation and store management have evolved in the progression from Mark I to Atlas. The wider impact of Manchester innovations in these and other areas is discussed, and the contemporary performance of the Mark I and Atlas is evaluated.

Key Words and Phrases: architecture, index registers, paging, virtual storage, extracodes, compilers, operating systems, Ferranti, Manchester Mark I, Atlas, ICL

CR Categories: 1.2, 4.22, 4.32, 6.21, 6.30

1. Introduction and Overview

In the period 1946–76 five computer systems have been designed and implemented at Manchester University. A general account of the prototypes and their industrial derivatives has been given elsewhere [6], along with a comprehensive list of some 60 references to their hardware and software. The main purpose here is to highlight two of the more significant of these five designs. The latest computer in the Manchester series, MU5, is described fully in a companion article [4].

As far as active University research is concerned, Manchester's involvement with digital computers dates from December 1946 when F. C. Williams and Tom Kilburn joined the University from their wartime posts at the Telecommunications Research Establishment. The computer projects, first in the Department of Electrical Engineering and then since 1964 in the Department of Computer Science, have followed the pattern summarized in Table I. This table relates primarily to hardware development; associated system software activity has naturally spanned similar periods, beginning in a small way in 1949 but gathering momentum with the release of the first compiler (1952).

The five prototype computers in the table are the Mark I, the Meg, an experimental transistor computer, the Muse (later Atlas), and the MU5. The names of the five industrially produced derivatives are respectively the Ferranti Mark I, Ferranti Mercury, Metropolitan-Vickers MV950, Ferranti Atlas, and the ICL 2980. The ICL 2980 is not in fact a direct derivative but its architecture owes much to, and has a great deal in common with, MU5. As may be inferred from the table, the cooperation between industry and university has been a fruitful and continuous process since the autumn of 1948. The only one of the five Manchester projects to receive direct government funding was the MU5, which in addition had significant help from ICL in the form of production facilities and engineering support.

The Mark I and Atlas have been chosen for closer study not only because they contain significant innovations, but because they convey an evolutionary progression with respect to the following design themes: i) Instruction format, ii) operand address-generation, iii) store management, and iv) sympathy with high-level language usage. The evolution is continued in MU5 [4]. Whilst all three machines were conceived as general-purpose computers, the internal architecture has tended to favor high-speed scientific applications.

Of the two Manchester computers omitted from detailed analysis in this paper, the Meg (precursor of the Ferranti Mercury) has been passed over because it was essentially an updating of the Mark I concept. By changing the technology and providing parallel access to the main store, the Meg became faster, more compact and easier to maintain. Apart from the incorporation of hardware floating point arithmetic, the instruction format and repertoire were similar to that of the Mark I. The market area of the Ferranti Mercury was much the same as that of the IBM 704, though the 704 was faster and considerably more expensive. The other Manchester computer to be omitted, the experimental point-contact transistor machine, was designed as a small and economic system using a drum as the main store. To help avoid the consequent latency problems a pseudo two-address (or 1 + 1) instruction format was used, in which the address of the next instruction was contained within each instruction. The transistor computer was in this respect untypical of the

Copyright © 1977, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

Author's Address: Department of Computer Science, University of Manchester, Manchester, M13 9PL, United Kingdom.

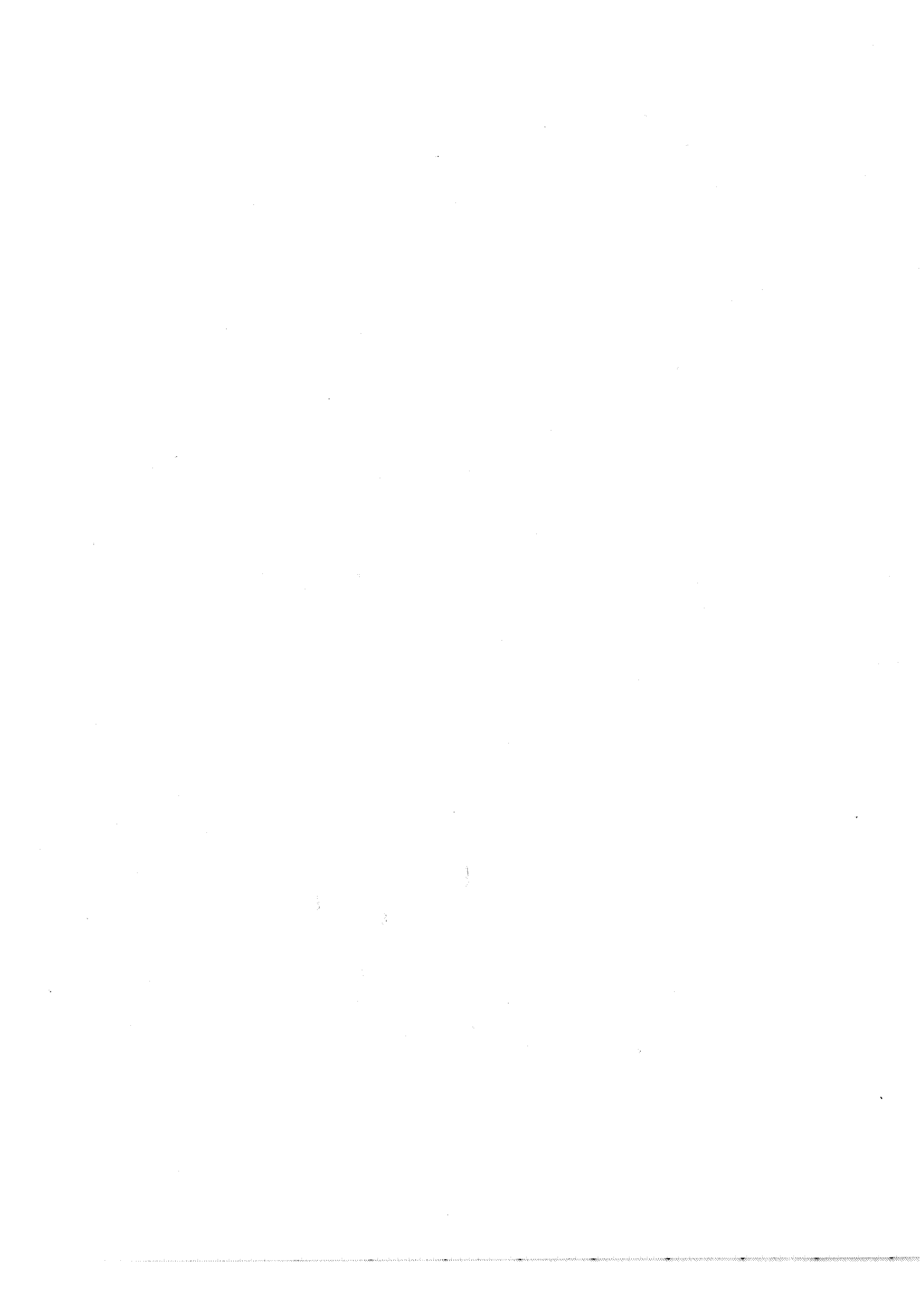


Table I. Summary of Manchester University computer projects and their industrially produced derivatives.

University Project	Industrial Derivative
Manchester Mark I hardware development period: 1946–49 prototype operational: June 1948 enhancements: April and Oct. 1949	Ferranti Mark I first installation: Feb. 1951 last one delivered: 1957
Meg hardware development period: 1951–54 prototype operational: May 1954	Ferranti Mercury first installation: Aug. 1957 last one delivered: 1961
Transistor computer hardware development period: 1952–55 prototype operational: Nov. 1953 enhancement: April 1955	Met-Vickers MV 950 first installation 1956 last one delivered (?) 1958
Atlas (formerly Muse) hardware development period: 1956–62 first installation operational Dec. 1962	Ferranti Atlas first installation: Dec. 1962 last one delivered: 1965
MU5 hardware development period: 1966–74 computer operational Oct. 1974	(ICL 2980) 2900 range officially announced: Oct. 1974

other Manchester designs. The use of a drum for primary storage made the transistor computer slower than the Mark I. Perhaps the most important impact of this machine on the Manchester group was the early experience it provided in transistor circuit techniques. The Meg and the transistor computer are described more fully in [6].

In the following account of the Mark I and Atlas each system is presented in three parts. First the objectives of the project are given, during which the motivation and evolutionary starting points are outlined. Secondly the principal features are given, in describing which, some of the original terminology has been replaced by its nearest modern equivalent for the sake of readability. It should be stated, however, that any serious further study of the designs should start with the original papers quoted in [6]; a useful selection of these is [1, 2, 5, 8, 9, 10]. Finally, each computer system is assessed according to its immediate and long-term impact.

2. The Mark I

2.1 Objectives of the Project

The initial aim was to build a realistic test environment for a novel digital store. The store was the electrostatic Williams Tube [9], and the prototype Mark I simply consisted of a 32×32 bit Williams Tube store plus elementary computational facilities. Nevertheless, when it successfully ran a 52-minute factoring program on 21 June 1948 it became the first general-purpose stored-program computer to work. Thereafter the machine underwent intensive engineering development so that by April 1949 a realistic computer had resulted. The objectives by 1949 were to provide sufficient memory and computational facilities to solve the number-theory problems that were provided by early Manchester users [6].

The computer design activity in 1949 was mainly concerned with the engineering aspects of Williams Tubes and drum memories, from which work some elementary "one-level store" ideas began to emerge (see below). The team throughout the Mark I period averaged about four people, working in relative independence from other groups in England and America. Being basically an engineering project, innovation and improvement were more or less continuous processes up to about October 1949.

2.2 Principal Features

a) Technology. The Mark I logic was implemented with EF50 (CV1091) and EF55 pentodes and EA50 vacuum tube diodes—these types being readily available owing to their extensive use in military equipment. The production Mark I comprised 4050 thermionic tubes and consumed about 25KW of power. The digit period was 8.5 microseconds (extended to 10 microseconds in the Ferranti production version). The Williams store was at first based on a standard CV1131 cathode ray tube, but specially-manufactured CRTs were used later.

Williams Tubes were used not only for the main memory but also for the accumulator and other central registers because this was cheaper than providing flip-flop registers. When compared with the mercury delay line which was the other common form of digital store in the late 1940's, the Williams Tube had the following advantages: i) It was random access (not serial access), and ii) it was cheaper to build and required no special temperature control. Williams Tubes did, however, require electrostatic shielding.

The Mark I backing store was a nickel-alloy plated drum, of 30 milliseconds revolution time. The drum was servo-synchronized to the main CPU clock, thus allowing extension to multiple drums without special buffering. Phase modulation recording was used.

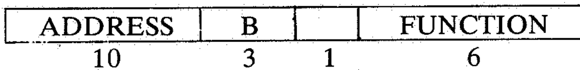
b) Architecture. The Manchester Mark I was a serial-ALU, fixed-point, binary computer employing a single-address instruction format. The original word length was 32 bits, but this was increased to 40 bits in 1949 for the sake of greater computational accuracy. A double-length (80-bit) accumulator facility was also provided. Two 20-bit instructions were packed to a word and addressing was to 20-bit boundaries.

The April 1949 version of the Mark I had a repertoire of 26 functions (op codes) in its instruction

set, including hardware multiply. It also had two 20-bit modifier (index) registers called B-lines, 128 words of random access main store and a 1024-word drum backing store. The Ferranti production Mark I was essentially the same architecture but with the following enhancements: i) An instruction set of 50 op codes, ii) eight modifier registers (B-lines), iii) 256 words of main store (Williams Tubes), iv) 4K (extendable to 16k) of drum store, and v) faster multiply time (2.16 milliseconds).

The characteristics of the production Mark I are now described in greater detail, since they form the definitive expression of ideas contained in the series of University prototypes developed during the period 1946–49.

The 20-bit instruction format was as follows:

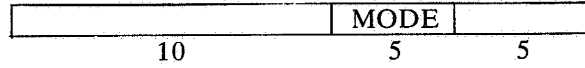


The three B digits specified one of eight B-lines and the specification of BO was normally used to indicate no modification. There was a separate B-arithmetic unit and associated eight-line B-store for carrying out modifier-register manipulation. Normal operands were 40-bit words, the bits being treated either as a two's complement number or as an unsigned quantity depending on the instruction. The full instruction set is given in Appendix 1, and it may be seen that considerable help was given with multilength arithmetic. There is also a population count or "sideways add" order (denoted in Appendix 1 by the mnemonic SADD), a facility requested by the Manchester mathematicians for their number theory problems. A similar instruction is provided on some modern computers, e.g. the CDC 7600, for nuclear physics applications programming, etc. The Ferranti Mark I also had a hardware random-number generator, available via mnemonic RNDM in Appendix 1. This somewhat unusual facility was included mainly at the request of the mathematician A.M. Turing, who was at Manchester from September 1948 until his death in June 1954.

Transfers to and from the drum and other peripheral equipment were carried out via 20-bit control words. These had two formats, distinguished by one of the mode bits. For *drum* transfers the format was:

Three of the mode bits then specified reading/writing, read-checking/write-checking, single page/double page transfers. The main store was arranged as eight 32-word pages on eight Williams Tubes, backed by the drum(s). The track-address was stored along with each page of information on the drum, and when a page became resident in main store an extra 20-bit line was assigned on each Williams Tube to hold the track-address of that page. This page-address line was normally invisible to the programmer, but could be accessed via the special LDAD instruction (Appendix 1). This was the germ of an idea which later led to page-address registers and virtual-to-real address translation on the Atlas computer.

For input/output transfers the control word format was:



For the early Ferranti Mark I's input was via a 250 character/sec 5-track paper tape reader using the 5-bit teleprinter code, and output was to a tape punch and printer. Four mode bits in the control word specified: Output a character, check output buffer, input a character, send a control character (equal to carriage return, linefeed, figure shift, letter shift) to the output device. Two input/output commands were provided (see Appendix 1): one took its control word from a main store address and the other used a 20-digit pattern set on the console switches—useful during bootstrapping.

c) System Software. In 1949 there was no Mark I system software, except for basic utilities such as input routines. Coding was normally carried out using the symbols of the 5-track teleprinter code. Once the Ferranti Mark I had been installed software development increased, with the emphasis being on embryonic high-level languages. After one earlier effort at compiler writing (1952), the Mark I Autocode [1] was available from March 1954 as an easy-to-learn scientific programming language for users having small or medium-sized problems.

An additional Autocode implementation objective was to simulate a one-level store so that the user had no need to organize his own drum transfers. It was possible to simulate this one-level store on the Mark I in a reasonably balanced way because the access time for reading an operand from the drum happened to be about the same time as a floating-point addition via an interpretive library routine. When running Autocode programs 128 tracks on the drum were reserved for instructions and 128 tracks for variables. Individual routines were transferred to the fast CRT store as they were required. To gain access to a variable an interpretive routine in fast store first determined on which track it lay, then transferred that track or "page" to the fast store, and finally selected the particular line within the page. Since successive operands were quite often located on the same page, steps were taken to avoid unnecessary drum transfers.

Arithmetic in the Autocode system was normally performed on floating-point variables $v1$, $v2$, . . . , etc. with provision for integers $n1$, $n2$, . . . , to be used as indices and counters. Simple conventions also existed for control transfers, intrinsic functions, input/output, and simple job control using symbols from the five-bit teleprinter code. An impression of the neatness of the system may be gained from the following Mark

I Autocode sequence which prints the root mean square (rms) of the variables $v1, v2, \ldots v100$. (Note that the symbol $*$ causes printing of a variable to ten decimal places on a new line, and $F1$ signifies the intrinsic function 'square root'):

$n1 = 1$
$v101 = 0$
$2v102 = vn1 \times vn1$
$v101 = v101 + v102$
$n1 = n1 + 1$
$j2, 100 \geq n1$
$v101 = v101/100.0$
$*v101 = F1(v101)$

2.3 Evaluation

The Mark I, in common with most early computers, was first applied to scientific and engineering problems. Measurements performed on a sample of Mark I jobs estimated that 16% of computing time went on drum transfers, 28% on multiplication and 56% on other arithmetic operations. Multiplication took 2.16 milliseconds and other accumulator orders took 1.2 milliseconds.

A contemporary benchmarking exercise rated the Mark I at about the same raw power as the National Physical Laboratories' ACE computer, even though the ACE had a digit period ten times shorter. The favorable performance of the Mark I was attributed to its random access main memory (ACE had a delay line store) and its relatively fast multiplier. Ferranti delivered nine Mark I and Mark I star machines between 1951 and 1957, three of them being exported (to Canada, Holland, Italy).

The long-term significance of the Manchester Mark I project is threefold. Firstly, it proved the viability of a digital storage technique (the Williams Tube), at a time when the successful implementation of the stored-program concept awaited the development of a suitable storage device. Williams Tubes were adopted by several computers in England, Russia, and America—including the IBM 701. Secondly, the project inspired the British government to give financial support to Ferranti Ltd., thus laying one of the cornerstones of the British computer industry. Thirdly, and of perhaps wider significance, the Mark I project was the first to focus attention onto the problems of linking fast random-access main memory to slower sequential-access rotating memory.

It was in the light of these problems that B-lines were first conceived of as relocation registers. It soon became clear that B-lines could also be used for general address-modification purposes, and so with the inclusion of a B-test facility the modern index register was born. The problem of automating backing store transfers ("overlays") still remained a challenge, but two Mark I facilities were later to suggest a solution to the Manchester team. First there was the fact that every page resident in the Mark I fast store carried with it the corresponding drum address in a special "page-address line" (see above). Second, there was the way in which the Autocode system handled the *drum* address of a user's variables. Out of these two facilities grew the concept of allowing the user always to program in a virtual (or "drum") address space and then providing system hardware and software to achieve automatic translation into the real (or "fast") address space, using information held in a set of associatively interrogated page-address registers. Thus the automated "one-level store" was conceived, and the realization of the other programming advantages to be gained from separating virtual and real address spaces followed shortly. These ideas were implemented in the Atlas computer.

3. Atlas

3.1 Objectives of the Project

By 1956 it was clear that Britain was falling behind the United States in the production of high-performance computers. The MUSE ("micro-second") project, started by Kilburn at Manchester in the autumn of 1956, was a conscious effort to remedy the situation. From January 1959 Ferranti Ltd. officially became involved and a joint University/Ferranti team under Kilburn continued the development of the computer, which was now known as Atlas.

Initial discussions with potential users of high-performance machines, both scientific and commercial, had produced a requirement for instruction times approaching one microsecond, the ability to attach a large number of i/o devices of various types and a main store size approaching 100k words. High computing speeds and rapid turnaround of user jobs became the keynotes of the Atlas design. The principal difficulties in achieving these goals arose from the wide differences in operating speeds between the various types of peripheral equipment and the CPU, and between transistor logic circuits and available core stores. Efficient and economic utilization of equipment was also very much a design-objective, since Atlas was intended to be sold on the open market. The somewhat conflicting requirements for high-speed and relative economy led to the incorporation of many techniques which were not extant when the project started in 1956. Amongst these were multiprogramming, job scheduling, spooling, extracodes, interrupts, pipelining, interleaved storage, autonomous transfer units, virtual storage, and paging. Although not all of these ideas originated in Manchester, they combined to make Atlas probably the most powerful machine available in the early 1960's.

3.2 Principal Features

a) Technology. The Atlas logic circuits were based on an OC170 germanium junction transistor used as an inverter, preceded by germanium OA47

diodes for the logic gating. This gave a typical gate-delay of 12 nanoseconds. Care was taken to avoid saturation (low collector-base volts), since the OC170's response became slow in the saturation region. The parallel adder employed a special symmetrical transistor (the SB240) as a switch in the carry-path, which resulted in a basic add-time of 200 nanoseconds for 48 bits—a significant achievement in 1959. There were about 80,000 transistors in the entire computer, mostly mounted on 8-inch by 5-inch printed-circuit boards.

The main store was 2 microsecond cycle-time core four-way interleaved, backed by drums having a 12 millisecond revolution time and capable of transferring one 512-word block every 2 milliseconds. Two other "private" storage units were provided: A high-speed read-only "fixed store" of 0.35 microsecond access time made from small slugs of copper or ferrite inserted in a woven wire mesh; and a system working store of 2 microsecond cycle-time core, which served as working space for the operating system routines (many of which resided in the fixed store). The size of all these stores varied between production versions of the Atlas. The Manchester prototype had the smallest capacity, expressed in 48-bit words as follows:

i) main store: 16k core, backed by four drums each of 24k
ii) fixed store: 8k
iii) system working store: 1k (later increased to 4k)

The largest production Atlas, installed at the Science Research Council's computing laboratory at Chilton (Harwell), had a main core store of 48k.

Bulk storage was provided by eight (expandable to 32) tape decks on eight channels, each having a transfer rate of 90k characters per second. Preaddressing and fixed 512-word blocks were used, thus allowing a tape to be written to nonsequentially when required. A 16-million word file disk was added later.

The Manchester Atlas had 17 conventional i/o devices, two high-speed data links, an on-line x-ray crystallographic diffractometer and an experimental speech input/output unit. The interrupt structure allowed for the connection of up to 512 peripheral units, with hardware assistance for determining the source of an interrupt.

b) Architecture. Atlas was a 48-bit word parallel computer with a one-address instruction format as follows:

FUNCTION	Ba	Bm	ADDRESS
10	7	7	24

The repertoire of functions or op codes, summarized in Appendix 2, was divided into two groups: normal instructions and extracode instructions. Generally speaking an extracode was a commonly used but relatively complicated function which it was not economic to implement directly as hardwired logic. Instead, an extracode consisted of a sequence of normal instructions (a "macro routine") held in the fixed store. Entry to these macro routines was very rapid and involved no preservation of central registers since there was a dedicated extracode program counter (or control register) and reserved B-lines, and any extracodes needing working space used a private area of the system working store. Amongst extracode instructions available to the user were ones for carrying out the common intrinsic functions such as square root, log, cosine, etc.

Of the normal instructions, Appendix 2 shows that they divide into three subgroups: main accumulator (A) orders, index register (B) orders, and test-and-count orders. There were independent A and B arithmetic units. The instruction set and the Atlas pipeline architecture assumed there would normally be no interchange of operands between the A and B ALUs.

This design philosophy, also to be seen to some degree in MU5, works most effectively for computations such as forming the scalar product of two vectors. By careful pipeline design and by using tricks such as assuming that the next instruction usually occurred in the same page as the last instruction, Atlas could overlap the execution of three A-instructions and then any associated B-instructions were normally executed concurrently with minimal additional time penalty.

The Atlas instruction could be double address-modified, according to the specification of the Ba and Bm bits. There were 127 24-bit B-lines (index registers) for this purpose, mostly held in a 0.7 microsecond cycle-time core store. The top three B-lines, B125–B127, were implemented as flip-flop registers and were reserved for use as independent program counters respectively for interrupt, extracode, and main program control. This explains why no explicit jump (branch) instructions appear in Appendix 2.

Of the 24 address bits in an instruction, 20 were used to cover the virtual address space of one million words, three specified a 6-bit character position within a full word, and one bit distinguished between a normal address and a "*V*-store" address. The *V*-store was the collective name given to all central registers and peripheral device registers which needed to be accessible to a (system) program. Into this category came such things as interrupt registers, page-address registers, and the data and status registers of all i/o units. Since normal instructions could, with suitable protection checks, use *V*-store addresses for operands there was no need for explicit op codes for the control of i/o equipment etc. The incorporation of peripheral devices into the total address space has since been used on other computers such as the PDP11.

The Atlas paging system used 512-word fixed-size pages, with a page-address register for every 512-word section of main core store. Each register contained a lock-out digit, so that pages of more than one program could be resident in core concurrently. The address-translation time, i.e. associative interrogation of the

page-address registers, was 0.7 microseconds, which represented about 40 percent of the total operand fetch time (including cable delays, store access time, etc.). Considerable effort was spent in ensuring efficient page-turning, and the replacement algorithm—contained in the fixed store—used a learning program which attempted to identify pages in main store which had fallen out of use [5]. No copy was normally kept on drum, and each drum had a rotational position indicator to speed the transfer of a replaced page to the first available space on drum. (Many modern paging computers, including MU5, now keep copies on drum and arrange not to write back pages which are unaltered.) The Atlas virtual address space was sufficiently large for the compilers to arrange simple segmentation conventions during the compilation process.

c) System Software. The Atlas Supervisor (operating system) fully exploited the following concepts: i) Multiprogramming (of up to 16 jobs concurrently), ii) on-line spooling of input and output, and iii) job scheduling (according to user-indicated job characteristics such as use of magnetic tapes, volume of output, or priority request). The aim was to keep all of the computer equipment busy while minimizing the turnaround of individual jobs. Since in addition the Supervisor produced comprehensive logging statistics for the user, the operator, and the (daily) accounting program, the computer room operators had little to do except feed in work—an exceptional state of affairs in the 1960's. The programmer was provided with straightforward job control conventions which were simple for simple tasks, while allowing scope for more complex requirements such as multiple input or output streams, or the use of special "private" compilers.

The standard Atlas compilers were mostly implemented using the Compiler Compiler [2], a formal language for the transparent description of syntax and semantics. While compilers for all the common high-level languages such as Algol, Cobol, and Fortran were eventually written for Atlas, the Manchester users programmed extensively in Atlas Autocode to begin with. Atlas Autocode was a block-structured language specified at about the same time as Algol 60, but implemented sooner. Except for a more limited concept of compound statements and *for* clauses, Atlas Autocode was very similar to Algol 60. As an example the root mean square calculation given in Section 2.2 might be programmed in Atlas Autocode as follows:

```
RMS = 0
cycle i = 1, 1, 100
RMS = RMS + X(i) * X(i); repeat
print (sqrt (RMS/100), 6, 4)
```

3.3 Evaluation

Some typical Atlas instruction times were as follows:

fixed-point B addition 1.59 microsec.
floating-point add, no modification 1.61 microsec.
floating-point add, double modification 2.61 microsec.
floating-point multiply, double modification 4.97 microsec.
floating-point division 10.66 to 29.80 microsec.

The overall average instruction time when executing Fortran scientific programs was measured to be about 3.35 microseconds per order. For comparison, the corresponding overall average instruction rates for typical present-day computers executing similar programs range from about 110 nanoseconds per order (CDC 7600) to about 6.6 microseconds per order (IBM 370/135).

In terms of contemporary machines Ferranti salesmen equated one Atlas to four IBM 7094s as regards work throughput. It is disappointing that only three full Atlas's and two scaled-down versions ("Atlas II") were sold between 1962–65. To some extent the marketing of Atlas was overtaken by events: Ferranti was currently developing other systems besides Atlas; Ferranti sold its large computer interest to ICT (later ICL) in 1963; ICT afterwards introduced its 1900 range of computers. By the mid-1960s the direct market competitor to Atlas was the faster CDC 6600, whose designers have said they learned some useful lessons from Atlas. In 1967 a benchmarking comparison for 22 compute-bound Fortran scientific jobs was performed on an Atlas with 32k core, a single-processor Univac 1108 with 64k core, and a CDC 6600 with 64k core [3]. The compilers used were respectively the London nonoptimizing Atlas Fortran V, the Univac F4012 Fortran IV, and the CDC Chippewa Run. Under these conditions the average CP computing speeds for Atlas, Univac 1108, and CDC 6600 were measured to be in the ratio 1: 2.1: 5.9 respectively. By the start of the 1970s ICL had introduced the 1906A computer, which has a work throughput of the order of three times that of Atlas, depending upon the configuration.

The long-term significance of the Atlas project lies in the design-concepts which it introduced. The more important of these are in four general areas: Pipeline techniques for high instruction throughput, paging and virtual storage, operating system features, and extracodes. The first area is now well-defined in respect of single instruction streams. The second has had far-reaching consequences. It has been the subject of much subsequent analysis and development (e.g. for MU5), in the light of which it is interesting to observe that the inspired guess of 512-word pages for Atlas proved to be about right. Programs on Atlas generally produced about one page-exception (page-address register nonequivalence) per 5×10^4 accesses. With regard to operating system features, the Atlas Supervisor would seem to the present-day user to have a very limited concept of file manipulation and no time-sharing (interactive) facilities. However, in all other

respects, especially in job throughput and ease of use, the Atlas Supervisor set a high standard which is still relevant today. Atlas was also one of first computers in which specific hardware facilities were provided at the design stage to aid the operating system—e.g. in the area of peripheral control, interrupt handling, and store management. With regard to extracodes, the concept of providing easy access to commonly required software has been taken up by several other designers. On Atlas the existence of a specially constructed fixed store for extracodes and certain Supervisor routines was partly determined by nonavailability of suitable fast core. It was generally observed that over half of all Atlas user-programs spent more than half their run time in executing common software such as the extracodes and i/o routines, so there was ample justification for having speeded up the access to much of this standard software.

As for the influence of Atlas on the design of its successor at Manchester, MU5, an important lesson was learned concerning the B-lines. The Mark I had had eight such lines and on Atlas about 90 of the 127 B-lines were available to the general user. This was indeed a lavish provision for the *assembler* programmer, but it was observed that the compiled code of the Atlas high-level language programmer could not easily make use of more than a few of these. A compiler writer has a potential need for registers such as B-lines for two main object-code purposes:

i) as bases and pointers for address-generation;
ii) as fast storage for frequently used operands when attempting run-time optimization—(this applies not only to integers for loop-control etc., but also to frequently-used floating-point variables).

For the former application the registers should be capable of reflecting the usage of local and nonlocal name spaces in block-structured languages—though there is no such special requirement for languages such as Fortran. For the latter application it would be advantageous if identification of frequently used operands was automatic at run time, thus saving on compiler complexity. The MU5 design attempts to satisfy these high-level language requirements with specific naming registers and associatively accessed buffers, and from Manchester's viewpoint Atlas marked the end of the road for the general-purpose B-line. The vindication of the MU5 approach lies in the more efficient compiled code which it produces [7].

4. Conclusion

This paper and its companion [4] reveal a developing view of computer design over a period of 30 years. At the beginning of this period the task of inventing the basic functional units and then keeping them running for long enough to obtain useful computation, dictated a spartan engineering approach to machine architecture. As technology advanced and successive Manchester computers were implemented and evaluated, so the designers were able to observe and incorporate in hardware more of the requirements of the system software and the users. These requirements themselves evolved over the years. Although the emphasis of the Mark I, Atlas, and MU5 has been on large high-performance systems, it is evident that the designs have not only kept abreast of the requirements of the general user, but in some cases (e.g. paging and virtual storage) the architectural innovations have been in advance of the facilities expected by normal programmers. In time, with the decreasing cost of logic and main storage, many of the Manchester "high-performance" devices have come to be adopted in succeeding middle-range computers.

Appendix 1

The Instruction Set of the Ferranti Mark I

In order to relate to modern terminology the following notation is used when describing the action of each order:

ACC: the contents of the double-length main accumulator (80 bits)
AM: the most-significant 40 bits of ACC
AL: the least-significant 40 bits of ACC
S: the contents of a store line (40 bits), except that B orders use the least significant 20 bits and control-transfer orders the least significant 10 bits.
B: the contents of a B-line (index register)
D: the contents of the multiplicand register (40 bits)
H: the digits set up on 20 console handswitches.

a) Main Arithmetic and Logical Orders

Mnemonic	Description
LDA	load AL (AM cleared)
LDAS	load AL, sign-extend into AM
LDN	load AL negatively
STA	store AL
STM	store AM
STMC	store AM and clear AM
SWAP	interchange AM and AL
STAM	store AL, move AM to AL and clear AM
STAC	store AL and clear ACC
CLR	clear ACC
ADD	ACC := ACC + S (signed S)
ADDU	ACC := ACC + S (unsigned S)
SUB	ACC := ACC − S (signed S)
ADDM	AM := AM + S
LDDU	load D (unsigned multiplicand)
LDDS	load D (signed multiplicand)
MADU	ACC := ACC + D × S (unsigned S)

MADS	ACC := ACC + D × S (signed S)
MSBU	ACC := ACC − D × S (unsigned S)
MSBS	ACC := ACC − D × S (signed S)
AND	ACC := ACC & S (S sign-extended)
ORA	ACC := ACC or S (S sign-extended)
NEQ	ACC := ACC ≢ S (S sign-extended)
SHLS	ACC := 2 × S (arithmetic shift)
ORS	S := AL := AL or S
ORSC	S := AL or S, then clear ACC

b) B-line (Index-Register) Manipulation

Mnemonic	Description
LDB	load a specified B-line
STB	store a specified B-line
SUBB	B := B − S
LDBX	load a B-line (without modification)
STBX	store a B-line (without modification)
SBBX	B := B − S (without modification)

c) Control Transfer Orders

Mnemonic	Description
JMPA	absolute indirect unconditional jump
JMPR	relative indirect unconditional jump
JGEA	if ACC ≥ 0, absolute indirect jump
JGER	if ACC ≥ 0, relative indirect jump
JGBA	if (last-named B-line) ≥ 0, absolute indirect jump
JGBR	if (last-named B-line) ≥ 0, relative indirect jump

d) Peripheral and Miscellaneous Orders

Mnemonic	Description
IOTH	i/o transfer using H as a control word
IOTS	i/o transfer using S as a control word
NORM	add to AM the position of the most significant one in S
SADD	add to AM the number of 1's in S − (population count)
RNDM	load a random number into AL
LDAD	load a page-address word into AL
DST1	debugging stop (1)
DST2	debugging stop (2)
TIME	S := clock
HOOT	pulse the console hooter
STH	S := console handswitches H
NULL	no operation

Appendix 2

Abbreviated Summary of the Atlas Instruction Set

In describing the action of the orders the following notation is used:

AM: the contents of the main 48-bit accumulator. (For floating-point working a 40-bit mantissa, 8-bit exponent and an octal base is used. For fixed-point working only 40 bits are used.)

AL: for double-length working AL forms a 39-bit mantissa extension.

S: the contents of a store line (normally 48 bits)

BA, BM: the contents of 24-bit B-lines (as addressed by the Ba and Bm fields).

BT: the contents of a B-test register.

N: a 24-bit literal ("immediate operand"), specified by taking the value of the instructions' address field as a two's complement number.

a) Main Accumulator Arithmetic Orders

(Note that several arithmetic orders were repeated with minor differences concerning accumulator standardization, rounding, clearing of AL, etc. Such orders are asterisked).

Mnemonic	Description
LDA	load AM (* four versions)
LDN	load AM negatively (* three versions)
LDL	load AL (* two versions)
LDDL	load AM and AL double-length
LDDLN	load double-length negatively
STA	store AM (* two versions)
STL	store AL (* two versions)
STDL	store AM and AL double length
ADD	fixed-point add
ADFL	single-length floating point add (* two versions)
ADFD	double-length floating point add
SUB	fixed-point subtract
SBFL	single-length floating point subtract (* two versions)
SBFD	double-length floating point subtract
RSUB	fixed-point reverse subtract
RSBFL	single-length floating point reverse subtract (* two versions)
RSBFD	double-length floating point reverse subtract
MPY	fixed-point multiply
MPFL	single-length floating point multiply (* two versions)
MPFD	double-length floating point multiply
NMPY	fixed-point multiply and negate
NMPFL	single-length floating point multiply and negate (* two versions)
NMPFD	double-length floating point multiply and negate
DIV	fixed-point divide
DVFL	single-length floating point divide
DVDL	double-length floating point divide

There were an additional 17 orders for performing miscellaneous minor operations on the accumulator such as negating, taking the modulus, etc.

b) B-line ("Index Register") Manipulations

(Note that orders asterisked used the 'read-pause-write' (split cycle) technique.)

Mnemonic	Description
LDB	load a specified BA (BA' = S)
LDBN	load negatively a specified BA
LND	load literal (BA := N)
LNN	load negatively a literal (BA := − N)
STB	store a specified BA (S := BA)
STBN	store negatively a specified BA
ADB	add (BA := BA + S)
ADN	add literal (BA := BA + N)
SBB	subtract (BA := BA − S)
RSBB	reverse subtract (BA := S − BA)
SADB	add into store (S := S + BA) *
SSBB	subtract from store (S := S − BA) *
RSSBB	reverse subtract from store (S := BA − S) *
SBN	subtract literal (BA := BA − N)
RSBN	reverse subtract literal (BA := N − BA)
AND	BA := BA & S
ANDS	S := BA & S *
ANDN	BA := BA & N
ANMN	BA := BM & N
ADMN	BA := BA + (BM & N)
NEQ	BA := BA ≢ S
NEQS	S := BA ≢ S *
NEQN	BA := BA ≢ N
ORB	BA := BA or S
ORBN	BA := BA or N

There were a further four miscellaneous simple B orders. More complex B operations, including multiplication, were peformed by extracodes—see Section d below.

c) Test and Count Orders.

i) Six orders of the form: BA := N *IF* BM is: odd, even, = 0 ≠ 0 ≥0, <0.
ii) Four orders of the form: BA := N *IF* BT is: = 0, ≠ 0, ≥0, <0.
iii) Four orders of the form: BA := N *IF* (AM, AL) is: = 0, ≠ 0, ≥0, <0.
iv) Four orders which set BT according to the result of: (S − BA), (BA − S), (N − BA), (BA − N).
v) Four orders of the form: *IF* BM ≠ 0, BA := N *AND*: (BM := BM + $\frac{1}{2}$), (BM := BM + 1), (BM := BM − $\frac{1}{2}$), (BM := BM − 1).
vi) Four orders of the form: *IF* BT ≠ 0, BA = N *AND*: (BM := BM + $\frac{1}{2}$), (BM := BM + 1), (BM := BM − $\frac{1}{2}$), (BM := BM − 1).

Note that BA was thought of as the "arithmetic" B-line and BM as the "address-modification" B-line. Adding $\frac{1}{2}$ to BM allowed halfword boundaries to be accessed.

d) Extracodes. These caused automatic entry to and return from fixed-store routines. Extracodes intended for general use divided into the following groups:

i) 14 extracodes for operating on B-lines, providing multiplication, division, shifting, etc.
ii) 46 extracodes giving additional (AL, AM) facilities such as:
   arithmetic using literals
   evaluation of standard trigonometric functions
   evaluation of other standard functions such as log, exp, sqrt, reciprocal, etc.
iii) Three extracodes for subroutine entry (return link stored in BA).
iv) 20 user-orientated extracodes for the control of magnetic tape, i/o stream selection, etc.

The rest of the fixed store was filled with system software such as the drum learning program, i/o device routines, and standard test programs.

Acknowledgments. The author would like to thank Professor Tom Kilburn and many of the staff in the Department of Computer Science, University of Manchester, for the helpful discussions concerning systems described in this paper.

Received March 1977; revised July 1977

References

1. Brooker, R.A. An attempt to simplify coding for the Manchester electronic computer. *Brit. J. Appl. Physics 6* (1955), 307–311.
2. Brooker, R.A., MacCallum, I.R., Morris, D., and Rohl, J.S. The compiler compiler. *Ann. Rev. in Automatic Programming, 3* (1963), 229ff.
3. Hughes, P.H. University computer benchmark report. Atlas Computing Service, U. of London, July 1967.
4. Ibbett, R.N., and Capon, P.C. The development of the MU5 computer system. *Comm. ACM 21*, 1 (Jan. 1978), 14–25.
5. Kilburn, T., Edwards, D.B.G., Lanigan, M.J., and Sumner, F.H. One-level storage system. *IRE Trans. EC-11*, 2 (1962), 223–235 (Reprinted in Bell, C.G., and Newell, A. *Computer Structurers: Readings and Examples.* McGraw-Hill, New York, 1971).
6. Lavington, S.H. *A History of Manchester Computers.* Nat. Comptng. Ctr. Publications, Manchester, England, 1975 (also published in the U.S. by Hayden, Rochelle Pk, N.J.)
7. Lavington, S.H., and Knowles, A.E. Assessing the power of an order code. Proc. IFIP Congress 77, Toronto, Canada, 1977, pp.
8. Morris, D., Sumner, F.H., and Wyld, M.T. An appraisal of the Atlas Supervisor. Proc ACM Nat. Meeting, 1967, pp. 67–75.
9. Williams, F.C., and Kilburn, T. A storage system for use with binary digital computing machines. *Proc. IEE*, Vol. 96, Pt. 2, No. 30, 1949, p. 183ff.
10. Williams, F.C., Kilburn, T., and Tootill, G.C. Universal high-speed digital computers: A small-scale experimental machine. Proc. IEE, Vol. 98, Pt. 2, No. 61, 1951, pp. 13–28.

Synchronisation and arbitration circuits in digital systems

D.J. Kinniment, M.Sc., Ph.D., C.Eng., M.I.E.E., and J.V. Woods, M.Sc., Ph.D.

Indexing terms: Multi-access systems, Time-sharing systems

Abstract

Synchronisation of two independently clocked processor units, or arbitration between two asynchronous units requesting access to a common resource, can cause serious time losses in a computer system. The ways in which these problems arise are considered, and a theoretical basis for calculation of the time losses is presented. The theory is then correlated with measurements on practical devices, and currently available methods for minimising the time loss are evaluated. Conditions necessary for prediction of the performance of synchronisers and arbiters are established and it is shown that design principles exist which allow the construction of systems with known reliability.

1 Introduction

In an asynchronous computer system it is possible for two or more autonomous units to request access to a common resource within a short time of one another. This situation occurs typically when a number of independent processors request access to a shared memory as shown in Fig. 1*a*. If the requests are sufficiently widely spaced in time for each to be recognised and dealt with before the next request occurs, then the resource can be allocated as requests occur. However, in the general case, requests may occur at any time relative to one another, and there may be any number of requests outstanding at a given time. Here the resource must be allocated to each of the requesting units in some order which may depend on a predetermined or a dynamically variable priority. An arbitration network now decides the order in which requests are serviced, and this is shown in Fig. 1*b*, which also indicates the request and acknowledge signalling required between the units.

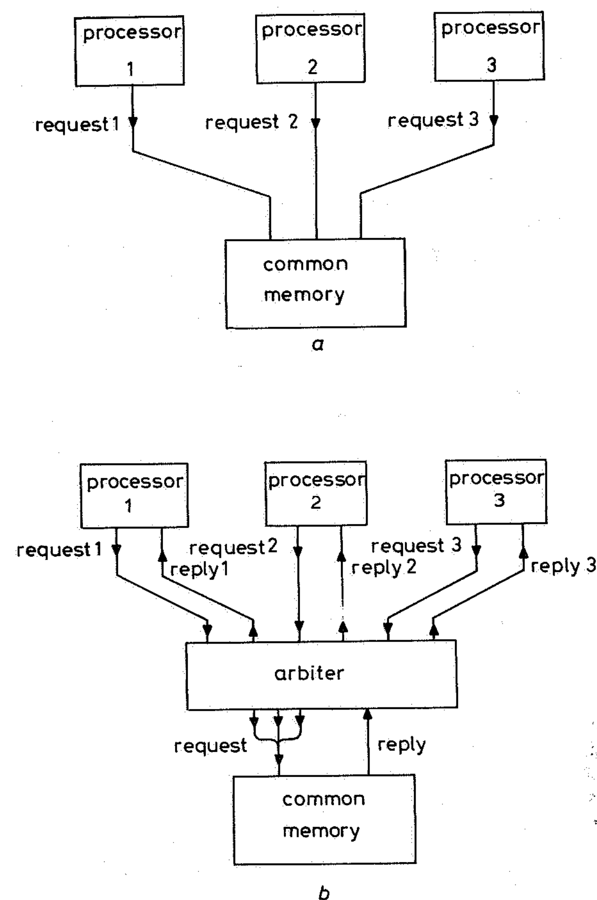


Fig. 1
Access request

a Clashing of requests
b Arbitration and signalling to avoid clashes

Paper 7751E, first received 4th February and in revised form 25th June 1976

Dr. Kinniment and Dr. Woods are with the Department of Computer Science, University of Manchester, Manchester M13 9PL, England

To achieve a high level of performance in systems where a memory is shared between processors, it is important to perform the arbitration function as quickly and simply as possible, since a long decision time adds directly to the memory access time, and may also increase the effective memory cycle time. Serious problems can arise in fast asynchronous systems because of the number of decisions which have to be taken, and the time required in some cases to decide whether or not a request exists.[1-3,5-8] In such a system several units may be actively requesting a particular resource, or may be changing state from idle to active at the time the decision is to be made. At that time the unit having priority must be determined on the basis of the request signals outstanding, which may change, and some other external data determining priorities. Any electronic circuit used to select one request out of many will take a finite time to operate, and the request signals must therefore remain steady for the period of time necessary to make the resource allocation.

The design of the arbitration network thus requires the solution of two problems:

(i) the resolution of the time-varying request pattern into a digital vector, constant over the decision period
(ii) the allocation of the resource to a particular request and organisation of the timing.

The solution of the first problem in an asynchronous environment requires special circuit techniques, since it involves the resolution of an infinitely variable quantity (the time of the request) into two discrete states (occuring before or after a particular instant). This problem is closely related to the synchronisation of an external interrupt with the internal clock of a synchronous processor. Here again, the time of an interrupt must be determined as occuring before or after a clock pulse. In either case the time of occurrence of the external event is infinitely variable, and it will always be possible to find a value for the time that causes long response times in any practical decision circuit.

Once a solution to the fundamental decision problem has been proposed within the overall design constraints, the organisation of resource allocation and time can be done by a network of conventional digital circuits designed to suit the particular environment. The design of such arbitration networks for digital systems has received increasing interest[3,4,9,10] with the growth of multiprocessor systems and large fast systems comprising a number of asynchronous units. The purpose of this paper is to present and compare some of the alternative decision circuits available to the system designer, and to examine the design principles involved in the organisation of resource allocation.

2 Decision circuits

2.1 Flip-flop decision times

The function of the decision circuit in an arbitration network is to examine a time varying set of request signals at one particular point in time, and to hold them steady long enough to establish the request with the highest priority. The most commonly used method of achieving this objective is to strobe the request signals into a register, which it is assumed will not change value after the removal of the strobe signal.

This is shown in Fig. 2 which indicates the situation that may occur if a request status changes just before the termination of the strobe. Here the effective drive pulse to change the corresponding flip-flop within the register may be infinitesimal, and, at the termination of the pulse, the flip-flop output voltage may be at some point between 0 and 1 logic levels.

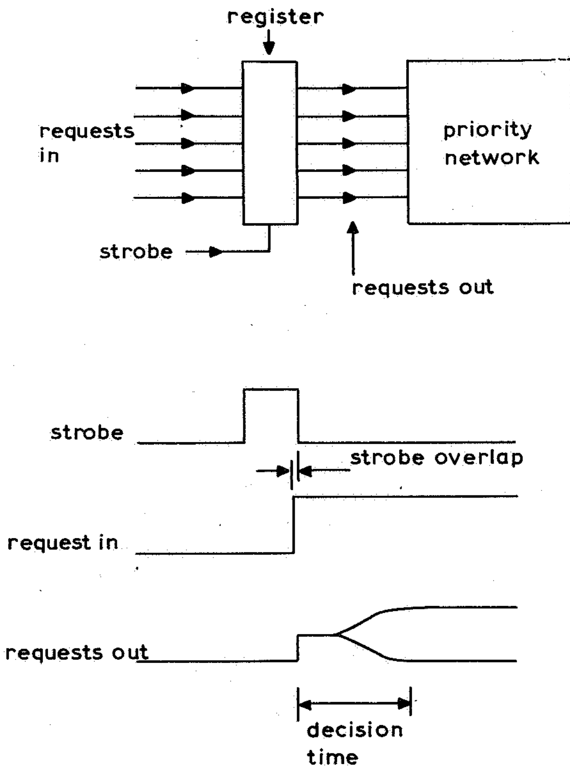


Fig. 2
Decision using flip-flop

Recovery from these points depends on the initial conditions and the characteristics of the flip-flop circuit, but in general the rate of change of the output is proportional to initial displacement from a position of unstable equilibrium midway between the 1 and 0 levels. Recovery from an initial condition exactly on the point of equilibrium can take a relatively long time.

Analysis of the decision time required has been undertaken by several authors,[3,6,9] and the results are summarised and extended here. It is usual to simplify the problem by assuming that the positive feedback of the flip-flop at its position of unstable equilibrium causes a rate of change of output voltage V_0, proportional to the actual output voltage displacement from the equilibrium or zero position. This is true of simple circuits such as a tunnel diode flip-flop which can be represented by a negative resistance in parallel with a capacitance, and approximately true of an e.c.l. flip-flop, a simplified circuit of which is shown in Fig. 3. Here the end of the strobe pulse causes current to be removed from the pair of transistors T_3 and T_4 and to appear in the cross-coupled pair T_1 and T_2. In this situation the base-collector

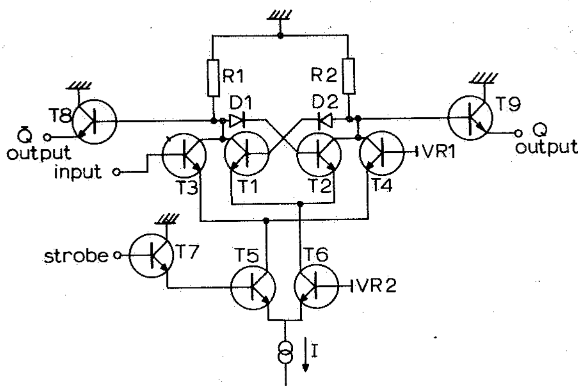


Fig. 3
Simplified e.c.l. latch flip-flop

capacitances of T_1 and T_2 will charge at a rate depending on the voltage difference between the two collectors, and the rate of change of the output voltage difference V_0 will be given by

$$\frac{dV_0}{dt} = \frac{V_0}{\tau} \tag{1}$$

A response characteristic of a positive exponential form will then result in

$$V_0 = Ke^{t/\tau} \tag{2}$$

where the constant K is the initial displacement and τ is a time constant which can be associated with the circuit itself. This model is accurate only in the region where the flip-flop can be represented by a linear system, but as the situation which causes difficulty is one in which the output voltage remains near zero for a long time, the model can give a sufficiently accurate prediction of the response of simple circuits as will be shown later.

In the situation indicated by Fig. 2, the input is represented by logic levels $\pm V_L$, whose overlap in time x with the strobe may have any value. The effect of this on the flip-flop can be evaluated by noting that the feedback loop is broken, but for small values of V_0 the circuit capacitances are now charged at a rate proportional to the input voltage (as the input here is compared with a reference of value 0).

The rate of change of V_0 under these conditions is given by

$$\frac{dV_0}{dt} = \pm\frac{V_L}{\tau} \tag{3}$$

thus the initial condition at the end of a strobe overlap of duration x is given by

$$K = \pm\frac{V_L}{\tau} X \tag{4}$$

From eqns. 2 and 4 it is now possible to predict the time t taken for the flip-flop to respond to a strobe overlap of x seconds to give a total voltage difference of $\pm 2V_L$ at the outputs:

$$t = \tau \log_e \frac{2\tau}{x} \tag{5}$$

To calculate the time which must be allowed for the decision in a given situation, the strobe frequency and the time between successive requests which may change the state of the flip-flop must be known. Assume that the time between strobes is T_s and the time between requests is T_R. The number of times a request occurs x seconds before a strobe termination during a long period of N seconds is given by

$$P = \frac{Nx}{T_s T_R} \tag{6}$$

so that the decision time of the flip-flop will exceed

$$t = \tau \log_e \frac{2\tau N}{T_s T_R} \tag{7}$$

on average only once during N seconds. It should be noted here, that τ is the time taken to move from the zero, or centre point of the output levels to one logic level under the influence of an input and a strobe; it will be approximately half the normal rise time, which is the time taken to move the full logic swing.

2.2 Effect of noise

It can be deduced from eqn. 4, that if each of the range of input request times has an equal probability, then each of the initial condition states will also have an equal probability. Thus the probability P_i of the initial conditions being within the range $\pm V_L$ at some time during N is given by putting $x = 2\tau$ in eqn. 6.

$$P_i = \frac{2N\tau}{T_s T_R} \tag{8}$$

and is a constant with respect to output voltage. The subsequent output voltage trajectories are positive exponentials as shown in Fig. 4.

The probability of the initial state being within the range ΔV between points K_2 and K_3 is given by

$$P_V = \frac{2N\tau}{T_s T_R}\frac{\Delta V}{V_L} \tag{9}$$

and in the absence of disturbance due to noise will be equal to the probability of the output arriving between K_2' and K_3' after a time Δt.

As

$$K_2' = K_2\, e^{\Delta t/\tau}$$

and

$$K_3' = K_3\, e^{\Delta t/\tau}$$

the probability density is thus independent of output voltage, and the addition of a random noise voltage contribution between 0 and Δt will not alter its characteristics as the noise is just as likely to move a trajectory nearer to zero as it is to push one away.

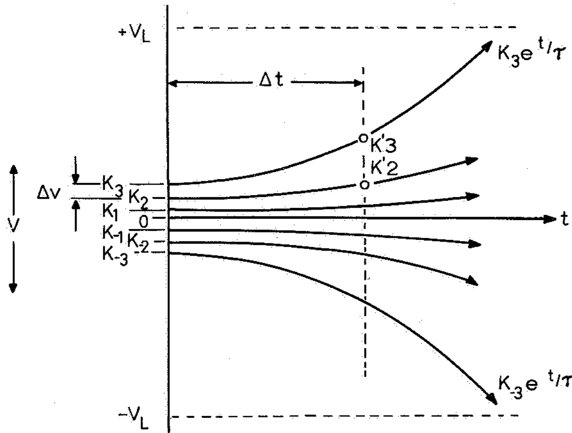


Fig. 4
Output voltage trajectories

The probability of finding a trajectory still within the logic levels after a decision time t will therefore be

$$P = \frac{2N\tau}{T_s T_R}\, e^{-t/\tau} \qquad (10)$$

or alternatively, during a period N seconds the decision time will exceed t on average once if

$$t = \log_e \frac{2\tau N}{T_s T_R} \qquad (11)$$

2.3 Experimental

Experimental results illustrating that the probability of a tunnel-diode flip-flop escaping from the metastable region follows the relation

$$P = 1 - e^{-t/\tau}$$

have been published by Couranz and Wann.[9] Results published earlier by one of the authors[6] were obtained on two types of e.c.l. flip-flop by means of the experimental apparatus shown in Fig. 5. This apparatus feeds the clock input of a flip-flop with a strobe signal from an oscillator at a period T_s, and the information input from a separate asynchronous oscillator at a period T_R. After a decision time t the output of the flip-flop is stored and compared with its output a long time later. Selected outputs from an e.c.l. flip-flop still in the metastable region after different decision times are illustrated in Fig. 6. Note here that the discrimination level of FF2 will always be consistently to one side of the equilibrium output level of the flip-flop under test. The apparatus therefore only detects half of the

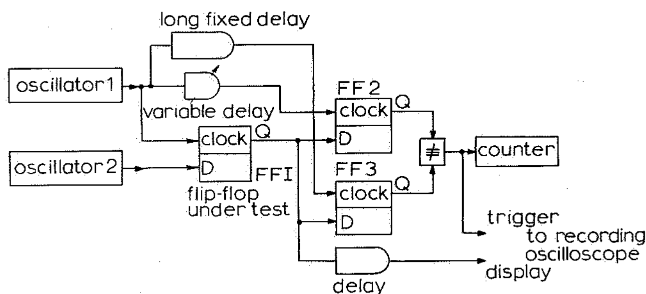


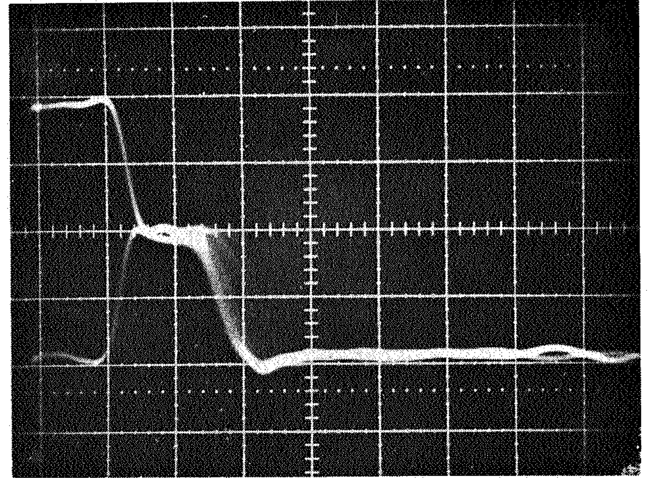
Fig. 5
Experimental set up

cases where the flip-flop is in the metastable region, and only those output trajectories which finally end on the opposite side of the discrimination level will be shown.

Results are presented in Fig. 7 for $T_s = T_R = 200$ ns and two different e.c.l. flip-flops. The propagation delay and rise time of these flop-flops are comparable and the experimental results can be compared to the theoretical predictions by putting

$$\tau_1 = 1{\cdot}1 \text{ ns}$$

$$\tau_2 = 0{\cdot}9 \text{ ns}$$



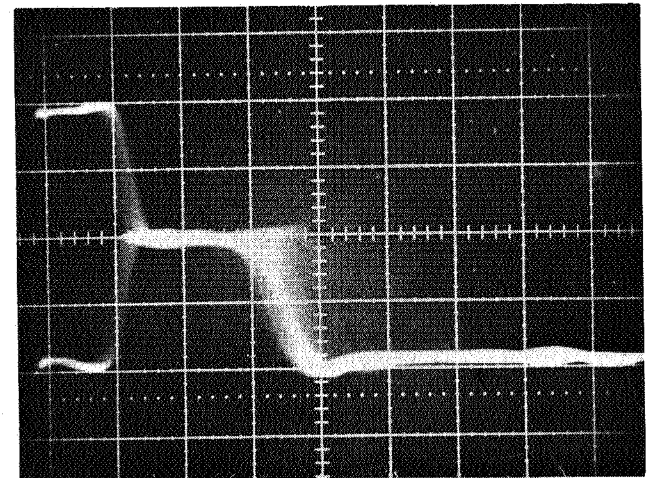


Fig. 6
E.C.L. flip-flop responses

a Delay time at 10 ns
b Delay time at 15 ns

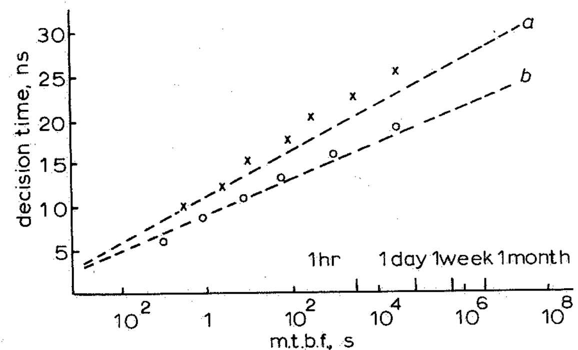


Fig. 7
M.T.B.F. against decision time – e.c.l. flip-flops

a Flip-flop A $\tau = 1{\cdot}1$ ns
b Flip-flop B $\tau = 0{\cdot}9$ ns
x, 0 measured values
---- theoretical characteristic

and m.t.b.f. = $2N$, as nonequivalence in the experiment will only occur once on average in a period of $2N$ seconds. Good correlation is shown between the theoretical and experimental results thus enabling design calculations to be carried out in a practical situation.

2.4 T.T.L. circuits

The development of the formulas in Section 2.1 depends on the assumption that the small-signal response of the circuit under investigation is dominated by a single time constant. This assumption does not hold for a flip-flop constructed from a pair of cross-coupled t.t.l. gates, however. To predict the performance of these gates under conditions where the data and strobe are asynchrnous, it would be necessary to investigate their small-signal characteristics. However, these characteristics are still not enough to predict performance on their own as it is possible for the open-loop gain at high frequencies to be greater than unity, and in some devices even greater than the zero-frequency gain. This leads to output trajectories with a large component of an oscillatory nature superimposed on the positive exponential, a characteristic demonstrated by Chaney and Molnar[7] in 7400 gates. If the oscillation is sufficiently large the operation of the circuit becomes nonlinear, and analysis correspondingly complex.

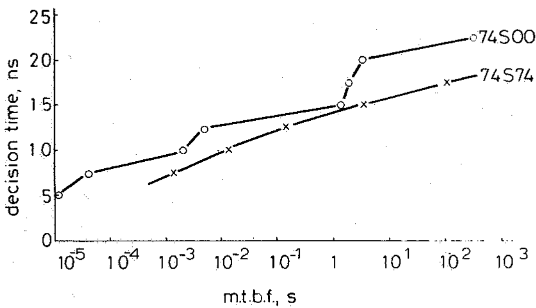


Fig. 8
M.T.B.F. against decision time – t.t.l. flip-flops

The effect of oscillation is shown in Fig. 8 which indicates the performance of a 74S00 cross-coupled gate system and that of a 74S74 flip-flop in which the basic decision element is a simple pair of transistors cross-coupled on the surface of the integrated circuit chip. The cross-coupled gate system clearly shows the presence of an oscillation at a period of 6 ns (twice the gate propagation delay) which significantly deteriorates the performance, whereas the 74S74 circuit performance curve is much smoother. In comparing the time constant indicated by this curve with the propagation delay of the circuit element, it should be noted that the 74S74 comprises the decision flip-flop in series with a second flip-flop which effectively adds to the propagation delay. The performance of the decision flip-flop is thus comparable to that of the 2·2 ns e.c.l. flip-flop.

3 Improvement of the decision time

3.1 Tunnel diode decision circuit

The foregoing results show that, to achieve adequate reliability in a digital system, the decision time allowed for a simple flip-flop is long compared with the normal propagation delay, and is strongly dependent on the switching time of the flip-flop under small-signal conditions. As a relatively small number of arbitration networks may have a disproportionately large effect on the access time of several processors to a memory, and therefore the performance of the system as a whole, it is worthwhile considering the use of

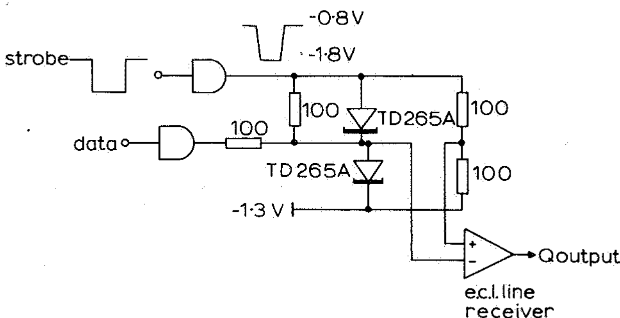


Fig. 9
Tunnel-diode flip-flop

high-performance high-cost units to replace the simple flip-flops. A device which allows fast switching as a flip-flop is the tunnel diode, and units are available with switching times as low as 35 ps.

The circuit of an e.c.l. compatible flip-flop using TD265A tunnel diodes in a Goto pair configuration is shown in Fig. 9. The total decision time is now increased by the e.c.l. input gate for the strobe, and the line receiver used to convert the Goto pair output back to e.c.l. levels. The total delay through these gates is approximately 7 ns, and experiments on the same test apparatus used for the e.c.l. flip-flops give the results indicated in Table 1.

Table 1
TUNNEL DIODE FLIP-FLOP PERFORMANCE

variable delay	m.t.b.f.
ns	s
7·5	$\approx 10^{-2}$
10·0	$> 10^{6}$

Results presented here are somewhat restricted as a small change in the delay has a considerable effect on the m.t.b.f. in the region of interest. However, a value of τ is likely to be approximately 100 ps, giving an m.t.b.f. of 136 years for a decision time of 3 ns or a total delay through the circuit of 10 ns. Each extra 1 ns allowed would then increase the m.t.b.f. by a factor of e^{10} or 22 000.

3.2 Indicator or 3-output flip-flop

An alternative solution to the decision problem has been independently suggested by several authors,[5,8] in which the presence of the flip-flop output trajectory within two voltage limits $\pm Vt$ is detected by comparators, and indicated as a signal on a separate output which holds up further action until the output trajectory has escaped from the metastable region. A circuit of this type is shown in Fig. 10. With this circuit the probability of the trajectory returning to within the measurement band for a significant length of time after it has left depends on the probability of a random noise pulse approaching the magnitude of Vt. Though this is more likely than a change of state of the flip-flop under quiescent conditions, as the width of the measurement band is by definition less than the total logic swing, it remains an extremely unlikely event.

The majority of decisions do not cause entry into the metastable region as shown in eqn. 8, and hence only a few will hold up the internal clock sequence to any extent. This system therefore allows most decisions to be relatively fast but has some disadvantages:

(i) the decision time is now variable and can theoretically approach infinity
(ii) the indication of a metastable state takes some time, typically 7·5 ns in e.c.l. compatible with the 2·2 ns delay flip-flop, and it is necessary always to wait for this time before examining the flip-flop outputs
(iii) it cannot be used if the flip-flop output has any pronounced oscillatory component.

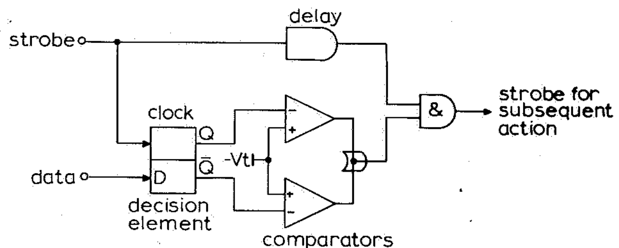


Fig. 10
Indicating flip-flop

3.3 Common clock

If all the requesting units and the resource itself are timed from a single clock source, it is not possible to have a continuous spectrum of time differences between the request and strobe. Thus, the problem does not occur in a synchronous system, and one method of reducing the delays introduced by the decision circuits is to reduce the number of independent time frames within a system. This is not always possible, for example in the obvious case where human interaction with the machine is necessary. Further problems may be caused by the difficulties of accurate clock distribution in a large machine, which can itself cause system delays greater than those introduced by the decision circuits.

4 Arbitration

If a request pattern can be guaranteed stable to an acceptable reliability, it is possible to assign the common resource to one particular request by means of a network of conventional digital circuits and various methods for the design of such networks have been proposed.[4,10]

These methods usually allow only a fixed priority to each request and often assume the arbiter is free to make another allocation as soon as a reply has been received from the resource. The reply, however, may be only an indication that the input data to the resource has been recognised and that the requesting unit is free to remove the data from its output buffer. If a further request is now selected before the processing of the first request is complete, it cannot be processed immediately and a subsequent request from a different source at a higher priority may be locked out for an unacceptably long time. More flexibility in the choice of time to make the next decision is thus required.

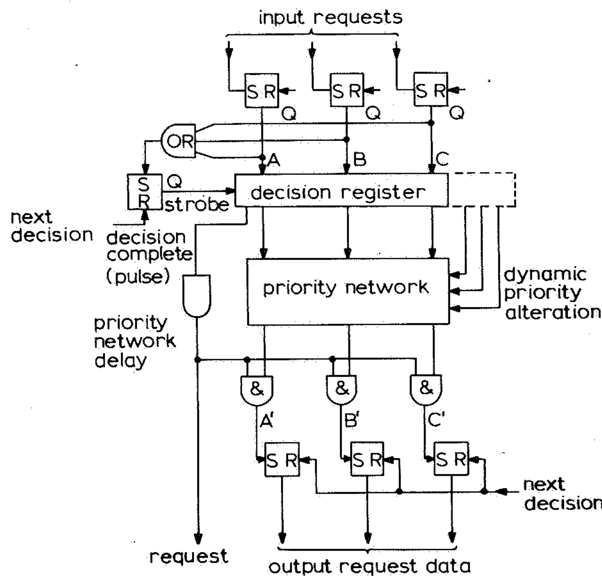


Fig. 11
Arbiter

A block diagram of an arbiter intended to overcome these difficulties is shown in Fig. 11. Here the incidence of one or more requests causes a semaphore flip-flop to be reset, whose function is similar to the *P*/*V* semaphore described by Dijkstra[11] in relation to software systems. Resetting this flip-flop causes the strobe to be removed from the decision register which, either after a fixed delay or by means of a combination of indicating outputs, produces a pulse signalling completion of the decision process. At this point the request status is stable until the semaphore is set again later in the control sequence when a further decision is required. The selected request can now be chosen by a simple combinational network, so that only one output of the network is active for any input pattern. Here a fixed priority algorithm could be implemented, or one which is alterable according to the status of the control, for example. If the common resource is a memory with an output register dedicated to read requests, it may be desirable to allow write requests when the register is full, but block further read requests until the data has been transferred to its destination. In this case the request type must be indicated to the priority network as well as the control status, and any information likely to change during the arbitration process must be staticised in the decision register.

After a time determined by the maximum delay through the priority network, the request will have been selected, and data concerning its nature can be passed to the resource at the same time as the request input flip-flop is reset. The reply to the requesting unit may be made at this time, or later when the acknowledgement from the resource is available. A new cycle of the arbiter is initiated by a pulse which sets the semaphore flip-flop, and allows the latest request status to be admitted to the decision register. This flip-flop must be of a type which allows the output to go to a 1 if both inputs are present, as one or more requests may already be outstanding which will have already reset the semaphore thus initiating a new decision process.

This design allows the next decision to be taken at a point determined by the control itself in releasing the semaphore, and the priority algorithm alone determines the order in which decisions are taken if there is a clash.

5 Comparison of decision circuits

Three basic methods have been presented for the improvement or elimination of the long decision times encountered when two or more asynchronous digital systems interact, and these are

(*a*) use of a common time frame for all interacting systems
(*b*) use of a decision element with a fast switching time
(*c*) linking the timing of the interacting systems by means of an indicating or 3-output flip-flop.

Each of these methods is appropriate to certain areas of digital design, for example, a system in which the central clock has a fixed period which cannot be shared by other units such as a fast disc memory with a pre-recorded clock track, cannot use an indicating flip-flop because of the variable nature of the decision time. In general, however, the choice of method depends on the time loss incurred by the use of that method and the system reliability obtainable.

Table 2 shows the time loss involved using different methods in one particular large computer system and illustrates the factors involved in the choice of method. The use of a simple flip-flop for the decision element with an appropriate control delay to give an acceptable reliability results in a time loss depending on the reliability. (Note here that an m.t.b.f. of 10^{12} seconds is much greater than the normally accepted component reliability.) Replacing this flip-flop with a state-of-the-science tunnel-diode decision element reduces the time loss to 10 ns at an m.t.b.f. of 10^6 seconds and the rate of increase of m.t.b.f. with delay time is much improved.

Indicating flip-flops can only be used in an environment where the decision flip-flop has a well defined, nonoscillatory response and a significant time delay may be experienced in an arbiter situation where many requests are competing. Several factors contribute to the time loss as follows:

(i) typical flip-flop delay	2·2 ns
(ii) comparator and gating delay to produce indication of metastable output	5·0 ns
(iii) time cost in combining several indicating outputs for a register	5·0 ns
(iv) clock hold-up gate	5·0 ns
	17·2 ns

Table 2
ARBITER TIME LOSS

	m.t.b.f.* = 10^6 s (11·6 days)	m.t.b.f.* = 10^{12} s (31 700 years)
	ns	ns
Simple flip-flop and delay (normal device delay 2·2 ns)	30	45
Tunnel-diode flip-flop and delay (normal device delay 35 ps)	10	11
Indicating flip-flops (normal device delay 2·2 ns)	minimum of 17 ns	
Common clock	depends on system size	

* Clock rate = request rate = 5 × 10^6 per second

Other factors, such as fan out of the subsequent clock to reset the input requests and set the output request data, may add extra time which can be taken into account in the delay setting of the simpler system; 17 ns then represents a minimum delay to which any settling time must be added.

The time loss involved in extending the boundaries of a common clock depends on the characteristics of the clock distribution system and the physical size of the machine involved, but the feasibility of synchronising as much of a system as possible to the central clock is one which should be considered at the outset of any design.

6 Conclusions

There is at present a lack of information on the characteristics of flip-flops usable in the decision element situation, and in particular some circuit elements specifically intended for the synchronisation operation are not sufficiently well specified. It has been shown in

this paper that a flip-flop circuit to be used as a synchronisation or arbitration element must have a response which is fast and non-oscillatory. In addition, information on the circuit's small-signal response in the metastable region is required to predict its behaviour. If these conditions are fulfilled, design principles exist that allow the construction of systems of known reliability.

7 Acknowledgments

The work described here arose as a direct result of the MU5 project. The authors would like to thank all members of the University and ICL involved in the project for many helpful discussions, and in particular K.C. Johnson of ICL for suggestions in connection with the indicating flip-flop.

8 References

1 CATT, I.: 'Time loss through gating of asynchronous logic signal pulses', *IEEE Trans.*, 1966, **EC-15**, pp. 108–111
2 LITTLEFIELD, W.M., and CHANEY, T.J.: 'The glitch phenomenon'. Computer System Laboratory Technical Memo. 10, Washington University, St. Louis, Mo., Dec. 1966
3 COURANZ, G.R.: 'An analysis of binary circuits under marginal triggering conditions', Computer System Laboratory Technical Report 15, Washington University, St. Louis, Mo., Nov. 1969
4 PLUMMER, W.W.: 'Asynchronous arbiters', *IEEE Trans.*, 1972, **C-21**, pp. 37–42
5 CHANEY, T.J., ORNSTEIN, S.M., and LITTLEFIELD, W.H.: 'Beware the synchroniser'. Presented at the IEEE Computer Society Conference Comp CON–72, San Francisco, Calif., 1972
6 KINNIMENT, D.J., and EDWARDS, D.B.G.: 'Circuit technology in a large computer system'. Presented at the joint IERE–IEE–BCS conference on computers – systems and technology, London, 1972
7 CHANEY, T.J., and MOLNAR, C.E.: 'Anomalous behaviour of synchroniser and arbiter circuits', *IEEE Trans.*, 1973, **C-22**, pp. 421–422
8 PATIL, S.S.: 'Bounded and unbounded delay synchronisers and arbiters'. MIT Computation Structures Group Memo. 103, June 1974
9 COURANZ, G.R., and WANN, D.F.: 'Theoretical and experimental behaviour of synchronisers operating in the metastable region', *IEEE Trans.*, 1975, **C-24**, pp. 604–616
10 CORSINI, P.: 'Self-synchronising asynchronous arbiter', *Digital Processes*, 1975, **1–1**, pp. 67–73
11 DIJKSTRA, E.W.: 'Co-operating sequential processes' *in* GENUYS, F. (Ed.): 'Programming languages' (Academic Press, New York, 1968), pp. 43–112