MU5 Project

UNIVERSITY OF MANCHESTER
Department of Computer Science

The MU5 Computer Complex

November 1974.

R.N. Ibbett.

1 The Multi-computer System

The MU5 complex has been designed as an integrated system in which the Operating System and other software is distributed in an optimum manner amongst several computers. Communications between different parts of software is via a formalised message-switching system and may involve the transfer of short control messages or pages of data between the computers in the complex. These transfers are implemented in the hardware by means of an Exchange "OR" Gate which allows single 64-bit parallel word transfers between up to a dozen Units, at a rate of one per 110 nanoseconds, i.e. at a rate of 80 megabytes per second. The devices currently linked via the Exchange are shown in Fig. 1. Transfers are initiated by a sending Unit, which sends to Exchange an address (consisting of the destination Unit Number and the address within the Unit) control information (read/write, etc.) and data where appropriate. Fixed priorities are assigned to Units attached to the Exchange so that if two or more requests arrive together from different Units, the request to or from the Unit with the highest priority is dealt with first. Having decided which request to service, Exchange checks that the destination Unit is able to accept the request, and if or when it is, routes the request to that Unit. In the case of a processor making a read access to a store, a further Exchange request will occur when the store has accessed the required data internally and is ready to send it back to the processor, which now becomes a destination Unit.

The MU5 Processor and its own Local Store are both connected as Units to the Exchange, as well as being linked directly. The Local Store consists of four stacks of Plessey Series 250 plated-wire storage having a cycle time of 260 nanoseconds.

Each stack contains 4K words of 8 bytes (plus an additional parity digit for each byte), and has its own access and read/write electronics. Under normal running conditions the stacks are interleaved to give a net access rate of one word per 65 nanoseconds or 128 megabytes per second. In the event of a hardware failure in one or more stacks, a fail-soft capability allows the store to be reconfigured, so that the best use can be made of the remaining stacks.

All peripheral handling in the MU5 complex is carried out by processors separate from the MU5 Processor, thus allowing the performance of the latter to be fully utilised in program execution. Communication of input and output between the MU5 Processor and the peripheral processors takes place mainly through the Mass Store, which acts as buffer area for data transfers between processors and also as a first level of backing store for the MU5 Local Store. The Mass Store Interface Logic allows up to four stacks to be connected (two are currently connected), with a fail-soft capability similar to that provided for the Local Store. Each stack consists of 128K words of 4 bytes (plus one parity digit for each byte) of Ferroxcube 2.5 μsecond cycle-time core store, giving a maximum transfer rate of 6.4 megabytes per second. Transfers of blocks of data between the Mass and Local Stores require appropriate requests to be made to alternate stores for each word in the block. Any one of the processors connected to the Exchange could generate the necessary sequences of requests, but this relatively simple task is better delegated to a dedicated Block Transfer Unit, so that the processors are free to operate on previously transferred data.

The Block Transfer Unit (BTU) connected to the MU5 Exchange contains four independent channels, each consisting of a number of registers and counters which can be addressed as store locations in "Vx" Store. Vx Store Locations are used in most of the Units attached to the Exchange to contain control information and are distinguished from actual store locations in a Unit by means of a digit in the real address sent through Exchange. When the MU5 Processor requires a block of data to be transferred from the Mass Store to the Local Store, for example, it sends the starting addresses for the transfer in each store to the BTU, together with the block size and the start command. The processor is then free to continue computation whilst the BTU channel generates the necessary requests, via Exchange, to the Mass and Local Stores to carry out the transfer. At the completion of a block transfer, a signal is returned to the Processor to inform it that the required data is now in its Local Store.
A second level of backing store is provided by the fixed head Disc Store which has its own block transfer unit capable of accommodating up to four devices. The two existing devices each have a capacity of 0.6 million 4-byte words contained on 64 bands.
Each band consists of 37 blocks of 256 4-byte words recorded at 1500 bits per inch. Transfers through the Exchange take the form of 8-byte words at a rate of one transfer per μsecond i.e. at 2 megabytes per second.

The ICL 1905E computer attached to the Exchange has served a number of purposes during the MU5 project. Initially it was used to simulate both the hardware and software of the MU5 Processor. Hardware simulation was carried out, at logic gate level, for separate groups of MU5 logic, using a suite of purpose-designed programs. Software simulation of MU5 was facilitated by the provision of a paging system which itself acted as a test bed for the associative circuits to be used in the MU5 Processor. The 1905E also acts as a peripheral handling processor and is fitted with standard peripheral devices such as a line printer, tape reader and punch, and exchangeable disc drives.

2 The MU5 Processor

The MU5 Processor is designed around an order code which reflects the structure of high-level languages, e.g. the use of named and array quantities, and the structuring of programs into routines or proceedures. Instructions are basically of the single address format F/N, where F is the function and N represents the name of an operand associated with a specific routine within a program, the address of the operand being obtained by adding the name to a Name Base. (The value held in the Name Base register is unique to the data storage area associated with each routine, and is altered at each routine change). The operand obtained as a result of this "primary" access may be used directly as a variable or indirectly as a data descriptor which contains the type, origin (base address) and bound (limit address) of an array or string. The array element itself is obtained by a "secondary" access using an address generated by the addition of a modifier (held in a B register) to the origin address.

This distinction between primary and secondary operands is reflected in the hardware of the MU5 Processor by the provision of separate Primary and Secondary Operand Units (PROP and SEOP in Fig. 2). Instructions are accessed from store, via the Store Access Control Unit (SAC), by the Instruction Buffer Unit (IBU). PROP interprets each instruction and accesses the primary operand, and the resultant function and operand are then sent across a highway to the B-Arithmetic Unit (B-ARITH) to SEOP, or back to PROP. Instructions destined for the Accumulator Unit (ACC) all travel via SEOP.

All these sub-units operate independently, and each is divided into a number of stages which carry out parts of the total task of the sub-unit. Since each stage is designed to operate independently of its neighbours, the system as a whole operates as a pipeline in which many partially completed instructions are in progress concurrently. Although the time required to complete any one instruction is still limited by the sum of the times for the various activities, the rate at which instructions progress through the pipeline is only limited by the time for an individual activity, and is therefore increased by the overlapping of successive instructions.

2.1 The Store Access Control Unit

SAC co-ordinates interactions between the Processor, the Local Store and Units accessed by Exchange. It receives requests from IBU, PROP and SEOP, routes them to the appropriate store, and, for read requests, returns a 64-bit double-word or 128 bit quad-word to the appropriate sub-unit. The addresses received by SAC are normally virtual addresses containing a 4-bit Process Number (allowing up to 16 currently active processes to be resident in the virtual store at any one time), a 14-bit segment Number (allowing up to 16K segments per process) and 15 bits addressing a 64-bit word within a segment. (Although MU5 is nominally defined as a 32-bit machine, with 64K words per segment, data highways within the Processor are normally 64 bits wide to allow convenient handling of descriptors and floating-point numbers). These addresses are translated into real addresses by a set of 32 associatively (i.e. content) addressed Current Page Registers (CPRs). These have been developed from the Page Address Registers (PARs) used in the Atlas computer, but differ from the latter in that the size (and hence the number of pages in store) is variable, and the appropriate real addresses cannot be obtained as a result of a one-to-one correspondence between each associative register and a physical page in the store. Instead, each Current Page Register has a conventional, value, field as well as an associative field, so that when an equivalence occurs between the requested virtual block and the contents of one of the associative registers, the corresponding real page address is read from the value field Furthermore, since an Exchange Unit Number is included in the real address field, blocks of sparsely used data may be retained in the Mass Store and accessed directly without the need for a block transfer.

2.2 The Instruction Buffer Unit

The IBU performs two main functions. Firstly, it pre-fetches instruction quadwords (128 bits) from store and passes on half word instructions (16 bits) in their correct sequence to PROP; the Local Store can supply instructions at a sufficiently high rate to match the rate at which they are executed, but because the store access time as seen by the processor is much longer than the instruction execution time, instructions are actually requested far in advance of their being obeyed and are buffered in a number of stages in the IBU. Secondly, the IBU predicts the result of an impending control transfer instruction in order to make the correct early call even if the required instructions are not in sequential addresses.

The prediction mechanism makes use of an eight-line associatively addressed 'Jump Trace' store. Whenever an instruction address is generated within the IBU it is presented to the associative store before being sent out to SAC; if an equivalence is found, the content of the corresponding value register is used in place of the original address and subsequent requests are for instructions following those at the predicted address. The Jump Trace is only loaded for control transfer instructions and when an instruction which finds address equivalence in the Jump Trace is sent to PROP for execution, it is marked as being followed by instructions which are 'out of sequence'. Thus, when a control transfer instruction is executed by PROP, the 'out of sequence' digit is examined. If the following instructions have been correctly predicted, execution of instructions continue uninterrupted. If the instructions are not out of sequence, but should have been, a request is made to SAC for the instruction at the 'Jumped-To' address , and at the same time, a line in the Jump Trace is loaded with the 'Jumped-From' address on the associative side and the 'Jumped-To' address on the value side. When the Jumped-From address re-appears within the IBU, the instructions at the Jumped-To address are automatically pre-fetched.

2.3 The PrimaryOperand Unit

PROP is concerned with accessing the operand specified directly by the instruction and routing the instruction, together with its operand, to the appropriate following sub-unit for execution or further processing. If the primary operand is a named variable or a literal, for example, instructions can be executed immediately,

whereas an instruction specifying a data structure must be sent to SEOP, where the primary operand is itself interpreted as a descriptor specifying the data structure element.

The distinction between named operands and other classes of operands is made because most store accesses made in high-level language programs are for the named variables, and a comparatively small high-speed buffer store dedicated to holding named variables can trap around 99% of such accesses and thereby avoid the long access time to the Local Store. Thus PROP contains a Name Store made up of 32 associative registers containing addresses of names in current use and 32 conventional registers containing the corresponding values. Associative addressing is used since this relieves the programmer of the onus of allocating registers to operands, and therefore allows code generated by compilers to run as efficiently as directly written code.

The Name Store is accessed by presenting the required operand address as an input to its associative field. If the address is identical to an address in one of the associative registers, an equivalence occurs and the value of the operand is read out of the conventional field of the store. If the required address is not in the associative store, i.e. a non-equivalence occurs, an access is made to the Local Store, and the value obtained, together with the address, is written into an 'empty' line of the Name Store. Any subsequent access for the same operand does not then require an access to the Local Store. The 'empty' line for overwriting is selected by a cyclic replacement algorithm, the previous contents of this line being written back to the Local Store only if they have been altered by the action of a write-to-store order.

PROP is split up into five independent stages, each of which carries out part of the instruction processing. The first stage receives instructions from the IBU and carries out the decoding and interpretation of the instruction required to deal with multi-length orders and to select the appropriate base register to which the name part of the instruction is added in the second stage. The resulting address is presented in the third stage to the associative field of the Name Store, and the value contained in the value field of the Name Store is read out in the fourth stage.

In the final stage the operand is assembled in its correct format ready to be sent the appropriate following sub-unit. Each of these sub-units sends a control signal to PROP indicating whether or not it can accept an instruction. Once accepted by the sub-unit, the instruction is guaranteed to go to completion, so that the Control Register can be incremented for the instruction, and instructions in earlier stages of the PROP pipeline can each move on to the next stage. Instructions therefore proceed through PROP in a series of beats, the minimum time between beats being 50 nseconds.

2.4 The B-Arithmetic Unit

The B-ARITH carries out logical and signed arithmetic functions between an incoming operand and the contents of the 32-bit B Register. Instructions sent to the B-ARITH are received on an input buffer and proceed to the arithmetic unit proper if or when the latter has completed any previously accepted function. Most functions are actually completed in 45 nseconds, the main exceptions being multiply and shift which take a variable time according to the value of the operands involved.

One of the principle uses of the content of the B Register is as the modifier in a data structure access. When an instruction is sent from PROP to SEOP as a result of a modified data structure operand specification, a modifier request is sent, in parallel, to the B-ARITH. No function is executed as a result of this request, but the value in the B Register is sent, via a separate dedicated highway, to SEOP, as soon as the request is accepted by the arithmetic unit within the B-ARITH. SEOP always waits for the signal from B-ARITH before commencing the modification, since any previously received modifier value may have been invalidated by the execution of a B function.

2.5 The Secondary Operand Unit

SEOP is divided into three major sections. The first and last sections (Dr and Dop) constitute the Descriptor System and contain the essential hardware for carrying out the requirements of the order code. The accessing facilities in the Descriptor System can be invoked directly by the use of store-to-store orders which implement data processing facilities in languages such as COBOL, or indirectly by the

specification of a data structure element as the operand to be used by a computational or organisational function. In either case Dr generates operand addresses, whilst Dop selects the operand from the appropriate part of the store word received as a result of the corresponding store access, and either routes the operand to the appropriate execution unit for a computational or organisational order, or processes it internally in the case of a store-to-store function. Interposed between Dr and Dop is the Operand Buffering System (OBS) which, like the Name Store in PROP, is invisible to the programmer and is incorporated for the purpose of improving the instruction execution rate.

For accesses to array or vector elements the required address is formed in Dr by adding the content of the B Register to the content of the Origin Field of the descriptor. The resulting address is sent to OBS and the adder is then used to carry out a bound check in which the modifier value is subtracted from the value in the Bound Field of the descriptor. An interrupt is generated if the result of this subtraction is less than or equal to zero, or if the modifier is itself negative.

The Dop section contains masking facilities which permit the selection of left or right hand masks to any bit position over the full 4-bit width, and a shifting mechanism which allows for any shift from 0 to 63 bits in single bit increments. Thus operands down to a single bit size can be selected from any position in the 64-bit store word. Dop is controlled by a combination of a control field generated by Dr (at the same time as the address) and the original function code which accompanies the instruction as it passes through the system.

The OBS, although serving the same end as the Name Store in PROP, i.e. overcoming the speed limitation of the Local Store, acts in a very different manner. This difference reflects the essential difference between the usages of named variables and data structure elements in a process - named variables are generally used randomly from amongst a small group, whilst data structure elements are generally selected sequentially from a large group. OBS does not, therefore, attempt to buffer large amounts of data; instead it effectively incorporates the SAC and Local Store into the Processor pipeline. SAC and the Local Store cannot be

used alone as a pipeline since, due to the interleaved addressing of the Local Store stacks, operands may be returned to OBS out of sequence. Thus, in order to maintain the correct sequence, OBS sends tag information with each request to SAC, and also enters the same tag information, together with the corresponding function, into a six-entry function queue when a request is made. An operand returning from SAC is copied into one of a set of buffer registers, as selected by its tag information, and the buffer is marked as being full. When the corresponding function reaches the end of the queue, and is ready to be sent to Dop, the tag information selects the appropriate buffer, and the operand is read out. If the buffer is not marked as full, then the function must wait for the reply from SAC before being sent to DOP.

Since OBS must contain at least as many operand buffer registers as there are positions in the function queue, it may frequently contain some of the operands for which Dr makes requests. This possibility is enhanced by the fact that data structure elements tend to be accessed sequentially, and also that although the size of data structure elements varies from 64 bits down to 1 bit, the buffers are actually 128 bits wide. Thus the total number of store accesses is considerably reduced by only a small amount of buffering, with a consequent improvement in store availability for the remaining requests, not only from OBS, but also from PROP, IBU and Exchange.

Two other forms of buffering are also contained in OBS. The first is for literal operands - since all Accumulator functions must pass through the OBS function queue (in order to maintain the correct sequencing of instructions) literal operands supplied by PROP must be buffered with the corresponding function. The second form of buffering consists of an extension of the Name Store. This additional Name Store, consisting of 24 lines, is needed for two reasons. Firstly, since the PROP Name Store is separated from the Accumulator Unit by the instruction stages in Dr, OBS and Dop, then if a named variable held in the PROP Name Store were used to accumulate a total calculated by a program loop using Accumulator orders, up to a dozen orders would have to separate the order storing the total and the order re-accessing it if the overlapping of instructions

was not to be held up. Secondly, in languages such as Fortran, where the programmer is not obliged to declare names at the start of a routine, many more names are created (and rapidly discarded) than in a language such as Algol where declarations are mandatory. Most of these additional names are used with Accumulator functions, so by incorporating a Name Store in OBS for holding Accumulator names, and thereby relieving the PROP Name Store of this task, the efficiency of the latter can be maintained.

The action to be taken in the event of a non-equivalence in either the PROP Name Store or the OBS Name Store now becomes dependent on whether the order is destined for the Accumulator Unit or not, and whether the required operand is already in the "wrong" Name Store. For an Accumulator order the normal situation is for a non-equivalence to occur in the PROP Name Store and equivalence to occur in the OBS Name Store. If equivalence occurs in the PROP Name Store, however, then for a "load" (i.e. non-store) order, the operand is carried through OBS as a literal, whilst for a store order it is returned from the PROP Name Store to the Local Store and is then treated as a normal non-equivalence by the OBS Name Store. When a PROP Name Store non-equivalence occurs for a non-Accumulator order, the OBS Name Store is checked before the operand is accessed from the Local Store, and OBS actually makes this access on behalf of the PROP Name Store if it does not itself contain the required operand. If OBS does contain the operand it returns the value to PROP via the normal internal highway.

2.6 The Accumulator Unit

The ACC is the main arithmetic unit of the MU5 Processor, capable of performing fixed-point and floating-point arithmetic, logic and shifting. Functions and operands are received from Dop into buffer registers, and when any previous function is complete the new function is performed between the incoming operand and the content of whichever register is specified. ACC contains a number of registers, the principle ones being a 32-bit fixed-point accumulator, a 64-bit floating-point accumulator and a 64-bit accumulator extension register. The order code also allows for decimal arithmetic, though this has not been implemented in the current version.

Two's complement representation is used for fixed-point numbers and for the floating-point mantissa. This gives a unique representation of zero and allows multiplication to be implemented in a straight forward manner. The 11-bit floating-point exponent is represented in excess 1024 and is to a hexadecimal base. Since the operands are of a comparable order of magnitude in very many calculations, the use of a large base means that pre-arithmetic and normalising shifts are required relatively infrequently and addition/subtraction can be started immediately on the assumption that no alignment is necessary. The check for alignment proceeds in parallel, and where a pre-arithmetic shift is found to be necessary the first addition is abandoned and a new one begun.

The adder is a parallel sequential state adder based on the ECL circuits used throughout most of the MU5 design, and involves no more than 5 gate delays for an addition over the full width of the mantissa. Multiplication is carried out using two carry-save adders activated alternately. The inputs to each carry-save adder are the output of the other carry-save adder and a multiple (in the range -4 to +4) of the multiplicand selected by taking a group of three multiplier digits at a time and the most significant digit of the previous group. Negative multiplier values are dealt with automatically by this technique and only one pre-addition (to form three times the multiplicand) is needed before the multiplication is started.

2.7 The Operating Console

The Console provides for direct control of the MU5 Processor and for communication between an operator and the hardware/software of the system by means of control switches, indicator lamps, program readable handkeys and a teletype. The Console also allows the execution rate of instructions, normally limited only by the processing rate of the Processor itself, to be limited to the rate of a variable frequency clock for test and diagnostic purposes, and the instructions themselves to be taken from a set of handkeys or the Console teletype instead of from the IBU. Thus the teletype can be used in ON-LINE or OFF-LINE mode; in ON-LINE mode it acts as a conventional 1900-type operators' teletype, whereas in OFF-LINE mode it acts as an instruction source in which successive characters on the tape are treated as octal or hexadecimal characters, as appropriate, to build up 16-bit instruction half-words for presentation to PROP. The Console also contains a real-time digital clock which provides one second and one-tenth second interrupts to the Processor and which can be read by the Operating System for accounting purposes.

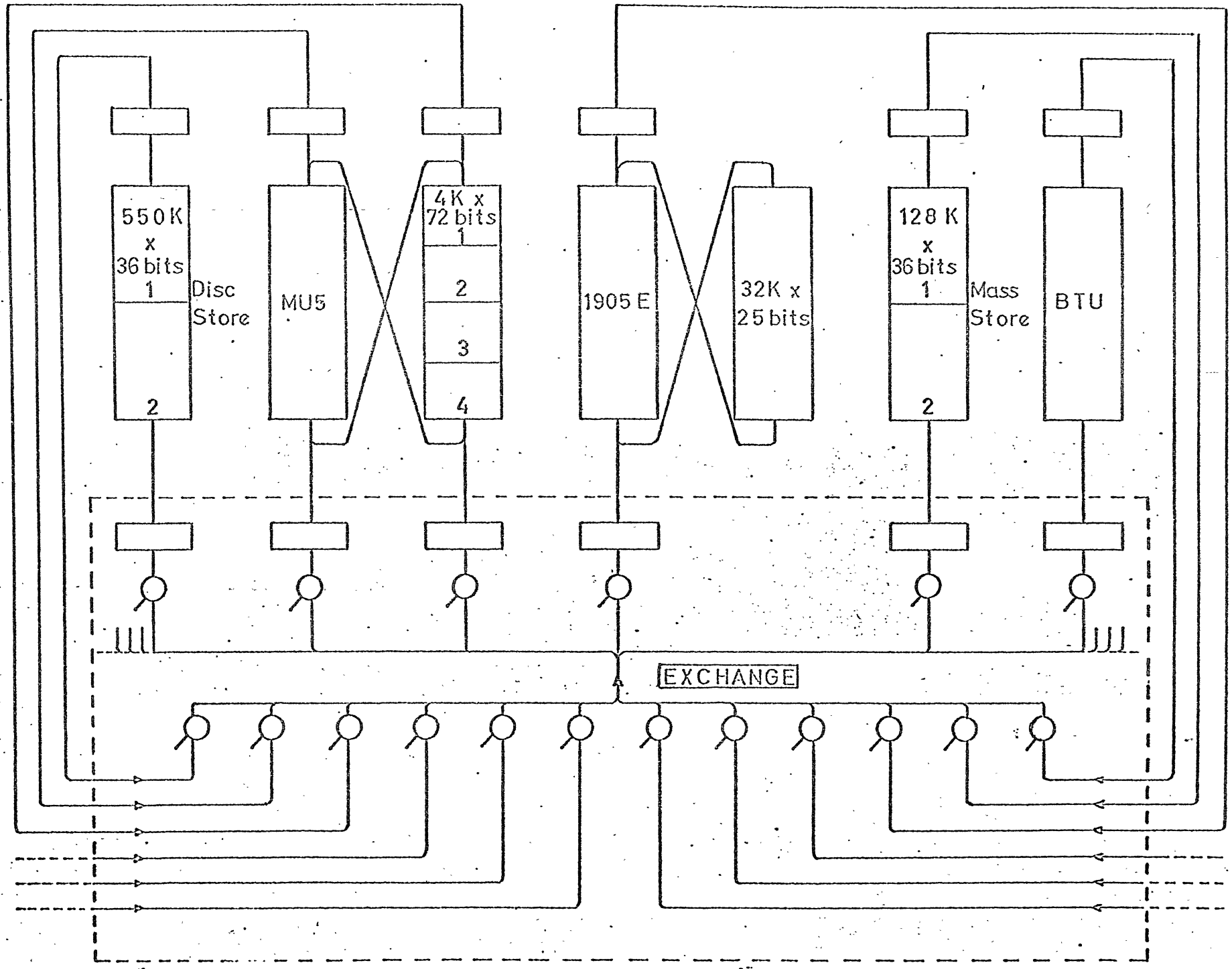

Fig.1. THE MU5 MULTI-COMPUTER SYSTEM

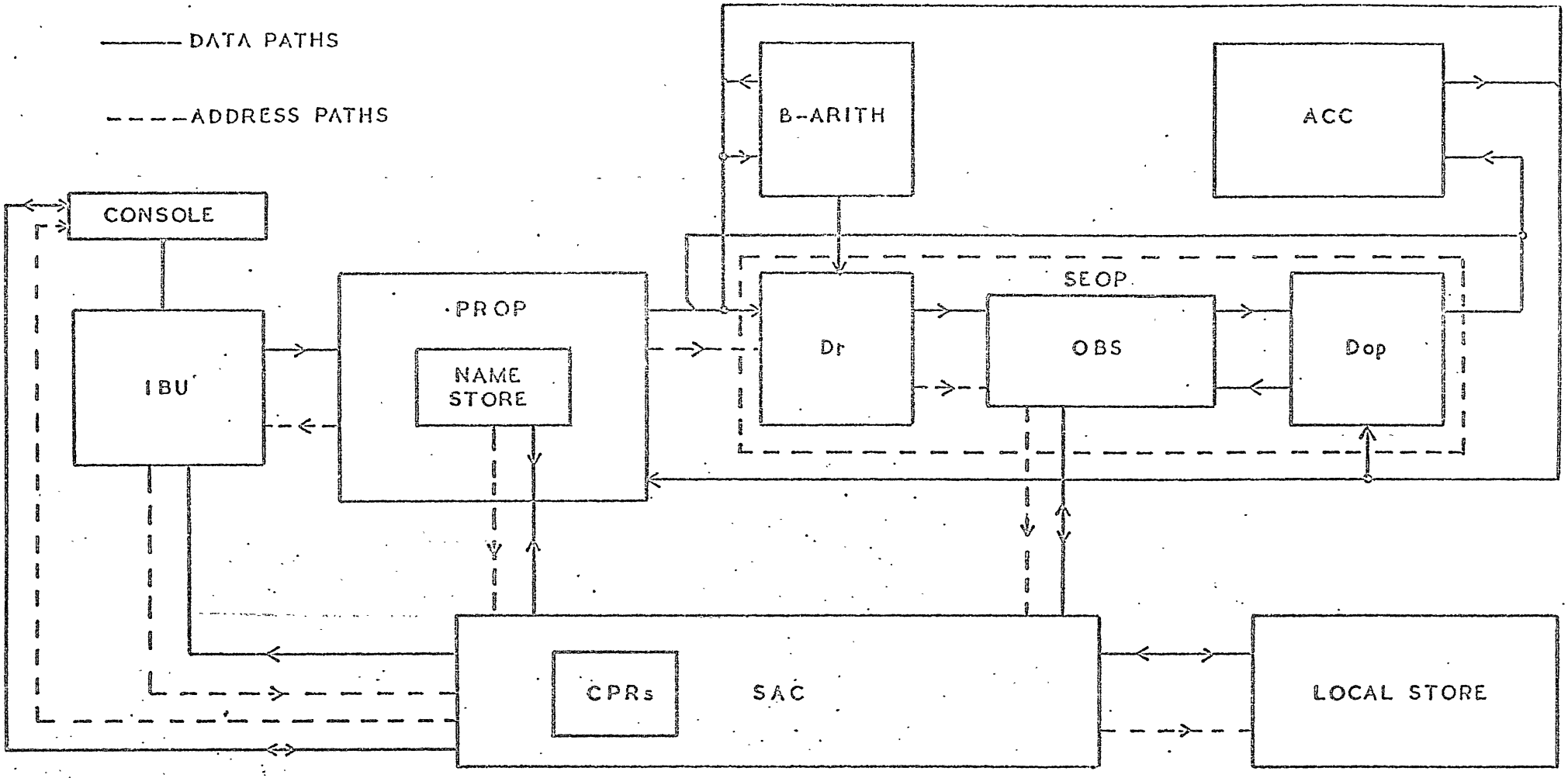

FIG 2 THE MU5 CENTRAL PROCESSOR

3. References

This is not a complete list of publications relating to MU5 but covers most of the aspects described above.

1. "A System Design Proposal": T. Kilburn, D. Morris, J.S. Rohl & F.H. Sumner
2. "Associative Memories in Large Computer Systems": D. Aspinall, D.J. Kinniment & D.B.G. Edwards

} IFIP Congress 1968

3. "Sequential-state Binary Parallel Adder": D.J. Kinniment & G.B. Steven - Proc. I.E.E. Vol.117, No.7, July 1970
4. "Instruction Fetching in High Speed Computer": F.H. Sumner INFOTECH Conference 1970, Giant Computers, published 1971
5. "Design of Large High-speed Binary Multiplier Units": J.R. Gosling - Proc. I.E.E. 1971, Vol.118, pp 400-506
6. "The MU5 Instruction Pipeline": R.N. Ibbett Vol. 15, No.1, Computer Journal, February 1972
7. "The Use of Logic Simulation in the Design of a Large Computer System": H.J. Kahn & J.W.R. May, pp 19-30
8. "Communications in a Multi-Computer System": D. Morris, G.R. Frank & T.J. Sweeney, pp 405-414
9. "Control of the MU5 Instruction Pipeline": R.N. Ibbett, E.C. Phillips & D.B.G. Edwards - pp 415-428
10. "The MU5 Secondary Operand Unit": J. Standeven, S.M.B. Lanyado & D.B.G. Edwards - pp 429-440

} IERE Conference on "Computer Systems and Technology" October 1972

11. "The Implementation of Record Processing in MU5": P.C. Capon, R.N. Ibbett & C.R.C.B. Parker
12. "The MU5 Disc System": D.B.G. Edwards, A.E. Whitehouse, L.E.M. Warburton & I. Watson
13. "Operand Accessing in a Pipelined Computer-System": J.V. Woods & F.H. Sumner
14. "The MU5 Exchange": S.H. Lavington, G. Thomas, D.B.G. Edwards

} I.E.E. Conference on "Computer Systems and Technology" October 1974