The MU5 instruction pipeline

R. N. Ibbett

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

MU5 is a high speed general purpose computer designed to meet the requirements of high level languages. The order code is such that several distinct operations are involved in accessing an operand. Operands specified directly by instructions are accessed by the instruction processing unit described in this paper. This unit is designed as a pipeline in which several instructions are in different stages of processing at any one time. A significant improvement in performance is obtained by this technique although detailed requirements of the order code deteriorate the performance in practice. An assessment of the expected overall performance is given.

(Received July 1971)

1. Introduction

The instruction processing unit described in this paper forms part of the central processor of the MU5 computer system currently being developed in the Department of Computer Science at Manchester University (Kilburn, Morris, Rohl, and Sumner, 1968). MU5 is a research computer intended to have a performance improvement of at least a factor of 20 over Atlas. A number of features of the system design contribute to this improvement, but particularly important amongst them are

1. the order code, and
2. instruction overlapping.

1. *The order code*

The current trend in computing is towards greater use of high level languages (ALGOL, FORTRAN, COBOL, etc.) rather than simple autocodes or basic machine codes. Thus there is considerable virtue, in terms of ease of compiling and economic running of the processor, in designing an order code with the structure of such languages in mind (Brooker, 1970; Lindsey, 1970). Important structural characteristics of high level languages are (i) the use of *names*, to identify the variables and arrays in a program, and (ii) the structuring of programs into independent sub-programs or *routines*.

A program therefore consists of one or more routines (each of which consists of a set of instructions which must be carried out to produce some required output data from the input data supplied) and forms part of a larger entity—a process. In MU5, a process consists of one or more programs, each of which may be a user program or a system program (i.e. one concerned with the running of the processor), together with their associated data. Addresses within a process consist of two parts—a 14-digit Segment Number, and a 16-digit address defining the required 32-bit word within a segment. (These addresses are virtual addresses and are translated into real addresses by a 'paging' mechanism developed from that used on Atlas (Kilburn, Edwards, Lanigan and Sumner, 1962).) Since several processes may occupy the available storage space at any one time, every virtual address also contains an additional 4-digit Process Number. Only one process may be in execution at any one time, however, and its process number is held in a special Process Number Register. All addresses generated within the central processor have the content of this register concatenated with them before any store access is made, thus preserving the uniqueness of processes. Processes may share segments of information with other, specified processes, however, and some segments are common to all processes.

Instructions in MU5 are basically of the single address format. In single address instructions in earlier generations of computers, i.e. in instructions of the form F/N, F represented the function to be carried out (in an implied arithmetic unit in the computer) and N represented the address in the computer store from which the operand was to be accessed. In the MU5 order code, N represents the name of a variable associated with a specific routine within a program, and the address of the variable is obtained by adding this name to a Name Base. The value held in the Name Base register is unique to the data storage space associated with each routine and is altered at each routine change. The advantage of this technique is that N referring to a name can be much shorter than N referring directly to an address anywhere in the computer store, so that instructions are kept short whilst the address field remains large, and the program itself requires minimal storage space.

N may be 6 or 16 bits long (requiring a 16- or 32-bit instruction) while NB, the Name Base register can address any word within one segment (usually zero) and is 16 bits long. Other segments are allocated to array quantities and to the program instructions, and access to variables in segments other than the 'name' segment is made either via a data descriptor (itself accessed by an array *name*) or via an Extra Name Base (XNB) which is used in the same way as NB except that it contains an additional 14 bits to indicate the required segment.

Studies of accesses for variables made by programs run on the Atlas computer indicate that the majority of these accesses are to the named variables within a routine, and that only a small number of these variables is in frequent use at any one time (Odeyemi, 1970). In order to reduce the time taken to access these variables in MU5, a high speed 'name store' has been incorporated into the central processor.

The name store is made up of 32 associative registers (Aspinall, Kinniment, and Edwards, 1968) containing the addresses (excluding segment number) of named variables, and 32 conventional registers containing the corresponding values. When the variable name has been added to the Name Base it is concatenated only with the Process Number, and the resulting address is presented to the associative field of the name store. If the address is identical with an address in one of the associative registers, an equivalence occurs and the value of the variable is read out of the conventional field of the store. If the required address is not in the associative store, an access is made to the main store, using the full virtual address including segment number, and the value obtained, together with the appropriate parts of its address, are written into an empty line of the name store. Any subsequent access for the same name does not then require an access to the main store. Although the main store itself has a comparatively short cycle time (250 nsecs), an access to it can take up to 800 nsecs due to address translation, priority circuitry, cable delays, etc., and this is an order of magnitude longer than the time required to access the name store.

2. *Instruction overlapping*

The execution of a single instruction requires various activities to be performed, e.g. instruction accessing, interpretation, operand accessing, arithmetic. If separate hardware units carry out these activities their operations can be overlapped to give an increased rate of completion of orders. This technique, first introduced in computers such as Atlas (Kilburn, *et al.*, 1962), and Stretch (Buchholz, 1962), has become known as 'pipeline concurrency'. In a pipeline computer, several partially completed instructions are in progress concurrently, and although the time to complete any one order 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.

In Atlas and Stretch the number of concurrent operations is of the order of four. More recently the pipeline concurrency principle has been extended to several tens of instructions in computers such as the CDC Star (Graham, 1970), which is designed principally as an array processor in which the same arithmetic operation is performed on many items of data in successive store locations, and in the IBM System/360 Model 195 (Murphy and Wade, 1970), which is a general purpose processor containing five overlapped functional units, each overlapped internally to give improved rates of execution.

In the MU5 processor, six functional units carry out the various activities involved in executing an order (**Fig. 1**). Accesses to store are made via the Store Access Control Unit (SAC), which is linked to the Local Store of the processor (16K words of 250 nsecs cycle-time plated-wire store), and also to an Exchange Unit (not shown) which can link several processors and mass storage units together. The SAC contains a set of Current Page Registers which translate the virtual addresses generated by the central processor into the real addresses required by the store. These registers are similar to the Page Addresses Registers used in Atlas. Instruction accesses are made by the Instruction Buffer Unit (IBU) in 128-bit groups, and these are then passed on in 16-bit groups to the Primary Operand Unit (PROP). This unit, which contains the Name Store, and is the one with which we are principally concerned in this paper, processes each instruction by (i) interpreting it, and (ii) accessing the operand specified directly by the instruction (i.e. the Primary Operand). This operand may then be used directly by the Primary Operand Unit itself or by the B (or Index) Arithmetic Unit (B-ARITH), or indirectly by the Secondary Operand Unit (SEOP). The latter treats the primary operand as a data descriptor to be used for accessing data structures, such as arrays, and consists of two parts. The first part interprets the descriptor, and carries out index modification using the content of the B arithmetic unit as the modifier, and then makes the necessary store request(s). The second part receives the requested word(s) back from the store and assembles the operand in the format dictated by the descriptor which caused the access. From the output of the Secondary Operand Unit the operand may be routed to the Accumulator Unit (ACC), the B-Arithmetic Unit, or back to the Primary Operand Unit. The Secondary Operand Unit may also use operands directly in order to carry out store-to-store functions. These functions perform various forms of data manipulation without involving any of the other computational facilities of the processor, e.g. 'table look-up', which scans through a table of data elements and identifies any with a value equal to the operand.

The Accumulator Unit contains a number of accumulators to accommodate different types of arithmetic. The structure of the order code allows all of the functions associated with these accumulators to be implemented in hardware but some, principally decimal arithmetic functions, are carried out by software in MU5. When a primary operand is to be used directly by the Accumulator Unit it is routed through the Secondary Operand Unit without affecting the data structure accessing mechanism.

2. The order code

The order code of MU5 uses two basic instruction formats, one for computational and store-to-store functions and one for organisational functions. These are distinguished by the value of the 'type of function' bits of the instruction half word (**Fig. 2**) and use 4 and 6 bits respectively to specify the function. The remaining 9 or 7 digits specify the primary operand. The main consequence of this difference in format is that for computational functions 16-bit orders are generally sufficient, whereas organisational functions must use the extended operand specifications, and therefore 32-bit orders, for all except short literal operands. In either case, however, considerable flexibility is allowed in operand accessing methods.

Amongst these methods are the use of named 32-bit (e.g.

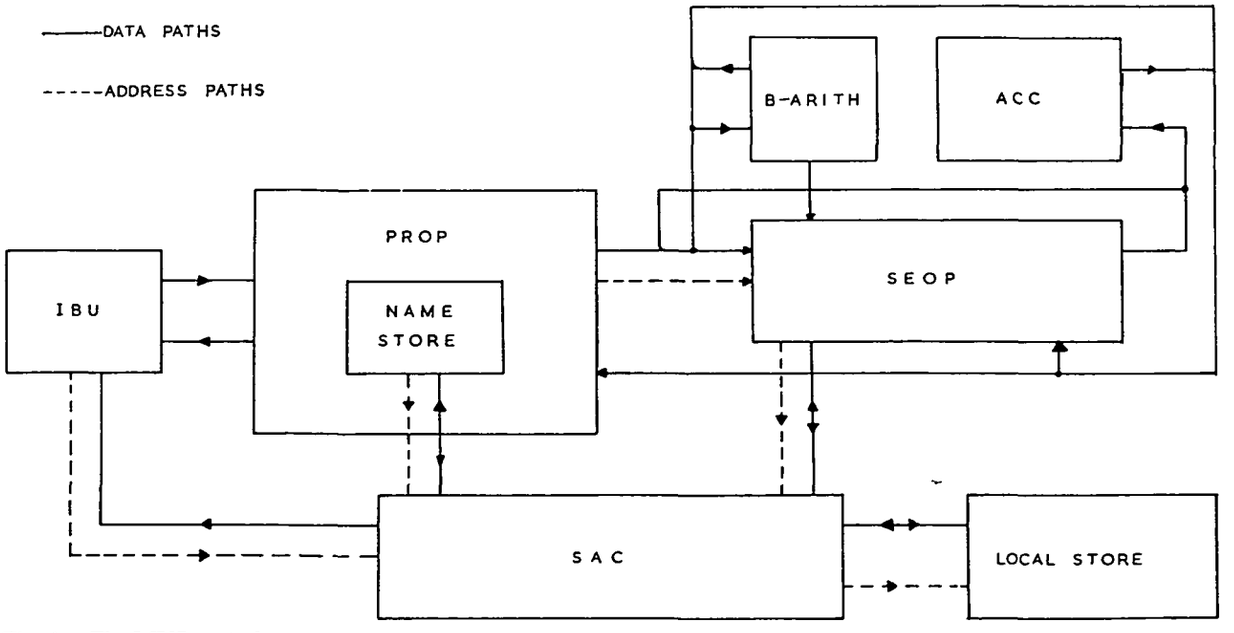

Fig. 1. The MU5 central processor

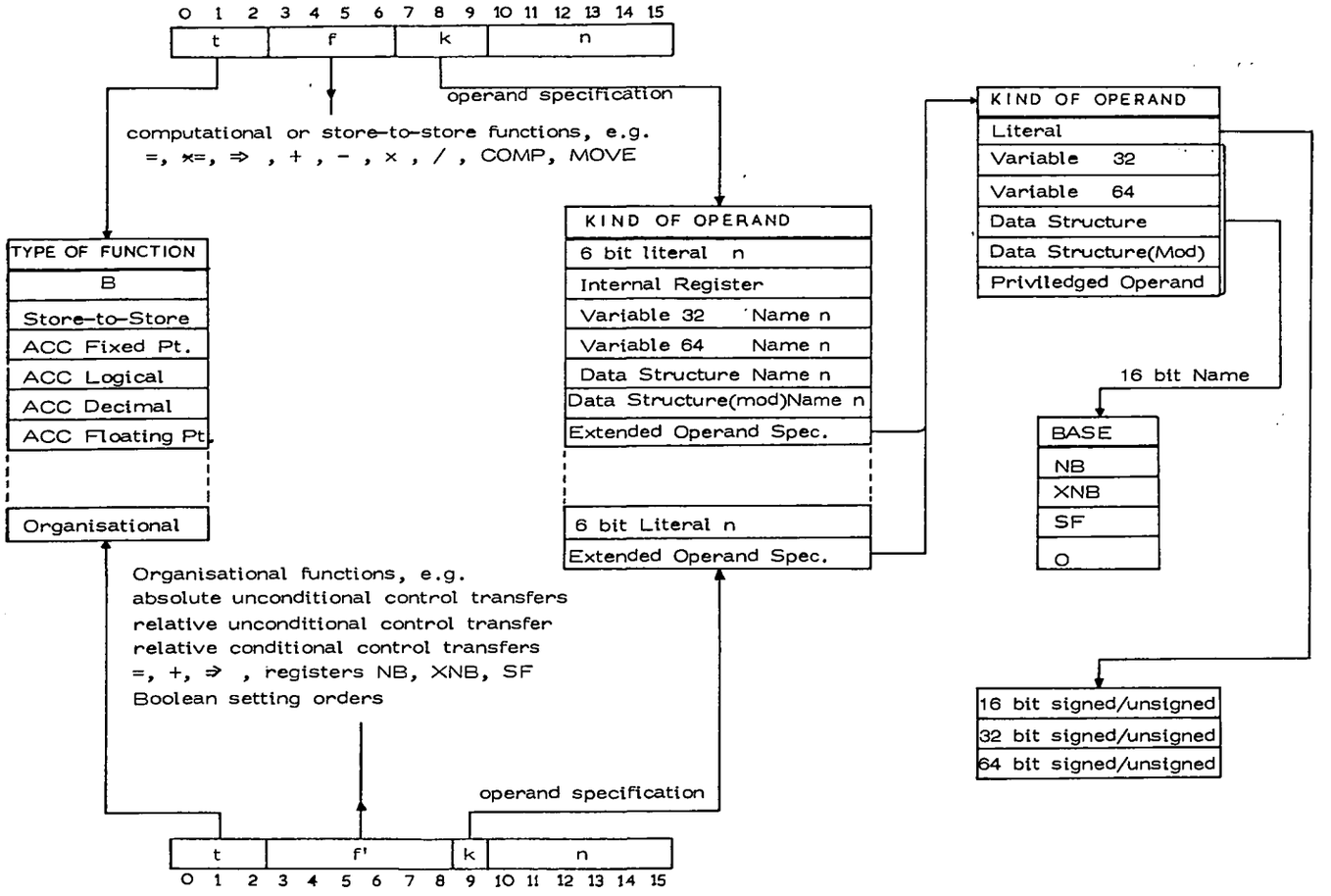

Fig. 2. The MU5 order code

integer) and named 64-bit (e.g. floating point) variables, the specification of a data structure by means of a named descriptor (which may or may not be modified), and the use of an internal register within the central processor or a literal quantity. A facility also exists for stacking partial results in the store. This is achieved by means of a function 'STACK and LOAD' (*=) which is interpreted as two orders:— (i) 'STACK'—increments the content of a special Stack Front register and then stores the content of the arithmetic register at the address indicated by the Stack Front; (ii) 'LOAD'—loads the arithmetic register with the operand specified by the instruction. Operands are unstacked, and the Stack Front register decremented, when an operand is accessed using SF as the base register in the extended operand set.

The extended operand specification also allows the use of XNB or zero as base register (as well as NB and SF), and the use of literals of length 16, 32 or 64 bits. Thus, in general, the use of an extended operand specification also implies the use of additional groups of 16 bits in an instruction. The privileged operands are available only to the system programs and consist mainly of registers concerned with the Interrupt facility (Section 3) and storage hierarchy management.

The computational functions include normal 'load' (=) and 'store' (=>) orders as well as the double function 'stack and load' (*=). Arithmetic functions +, −, ×, /, and logical operations are provided, together with a test instruction 'COMP' which compares the context of the central register with the incoming operand (by subtraction) and records the result (zero, negative, or overflow) in a single test register held in the Primary Operand Unit. The store-to-store functions are carried out by the Secondary Operand Unit and fall into three classes:— string-string, byte-string and table-string orders. The string-string orders operate on a source string and a destination string and the strings can be moved, compared or logically combined. The byte-string orders are basically the same as the string-string orders but use a byte (8 bits) repeated as often as necessary, as the source string. The table-string orders make it possible to translate the characters of a string into a different code specified in a table, or to check a string to see if it contains any of the characters specified in a table.

The control transfer functions are implemented in the Primary Operand Unit and the conditional transfers use the contents of the test register in determining the jump/continue condition. Separate functions are provided to test for $=0$, $\neq 0$, >0, ≥ 0, <0, ≤ 0 and overflow in the test register, so that any of the conditions stated in a high level language program can be compiled directly into a machine order. An additional test digit, the 'Boolean' digit may also be used as a test condition, and may be set either directly as a logical combination of the operand value and the previous Boolean value, or as a result of one of a set of conditions, identical with the conditional control transfer conditions, being satisfied. This facility allows multiple conditions, used for a single test in a high level language program, to be combined directly in the hardware before the appropriate control transfer function is obeyed.

3. The primary operand unit

The Primary Operand Unit is an instruction processing unit which interprets the various operand and function specifications and produces an appropriate operand in each case. **Fig. 3** shows the basic sub-units required to process a typical instruction specifying a named variable as the operand, and also the various stages of operation involved—initial decoding, addition of name to base, association of address, reading of value, and assembly of operand. In addition, this unit is concerned with carrying out the organisational orders and it also contains

much of the hardware concerned with the 'Interrupt' facility of MU5.

An interrupt signal occurs whenever some operation in a process, or in part of the processor, cannot continue without the intervention of a supervisory system program. The occurrence of an interrupt signal causes the processing of instructions in the Primary Operand Unit to be inhibited and a pair of fixed instructions to be obeyed instead. These instructions preserve the contents of the Control (instruction address) Register, and overwrite it with one of eight addresses referring to the first instruction of an appropriate system program. Each of these system programs corresponds to a specific type of interrupt and once entered, determines the exact cause of the interrupt and takes some appropriate action.

The four interrupt types arising from the operation of the processor (system malfunction, address non-equivalence in the Current Page Registers, end of an Exchange block transfer, peripheral servicing request) are of higher priority than the four types of interrupt arising from the running of a process (end of time allocation, illegal instruction, arithmetic overflow, system software interaction). A program entered as a result of one of the latter can be interrupted by the occurrence of one of the former, but not vice-versa. Similarly, an interrupt signal of one priority cannot interrupt a program previously entered as a result of an interrupt of the same priority, except that a processor based interrupt program can be interrupted by a system malfunction signal.

At the end of a program entered as a result of an interrupt signal the Control Register is restored to its former value and the previously inhibited instruction is re-accessed and processed.

All instructions are received from the Instruction Buffer Unit into registers DF(function) and DN(name) and the first sub-unit carries out the decoding and interpretation of the instruction required to deal with multi-length orders and double functions. It also selects the appropriate base (Name Base, (NB), Extra Name Base (XNB), or stack front (SF)) and the appropriate name part of the instruction. For accesses to a 32-bit variable the name is shifted down one place relative to the base and the least significant digit is then used by the fifth sub-unit to select the appropriate half of the 64-bit word obtained from store.

The second sub-unit is simply a 16-bit parallel adder which adds the name to the base to give the 15-bit address of a 64-bit operand within the Name Store segment. NB, etc., are also 15 bits long in MU5, rather than 16, since the order code is designed for a range of processors and assumes a basic 32-bit word. The 16th bit in the adder is used as an interrupt condition indicating overflow from the Name Segment. The adder is also used to increment and decrement the Stack Front register for orders which stack or unstack operands, and to carry out the organisational orders which load or increment the Name Base, Extra Name Base and Stack Front Registers (Fig. 2).

In the third stage of processing the 15-bit operand address is concatenated with the 4-bit Process Number (PN) and presented to the content addressable (associative) 'virtual address field' of the Name Store. Segment Number is not used since this is constant for all names in a process. A logic '1' appears from the line which has a content identical with that presented at the input and this signal is then used in the fourth sub-unit to access the operand value from the 64-bit wide conventional 'value field' of the Name Store. In parallel with this operation a check is made to determine whether an equivalence actually occurred in the associative field. In determining this equivalence an additional digit associated with each line indicates whether or not the content of the line is meaningful, i.e. whether the line is in use or empty. If no equivalence is found, i.e. the required named operand is not in the Name Store, an access is made to the Local Store and the Name Store is updated.

The fifth stage of processing is the assembly of the operand into its correct format, e.g. a 32-bit integer may be taken from either half of the 64-bit value held in the store, but must always appear at the least significant end of the data highway when presented to a succeeding unit in the central processor. The register HI forms the input to this highway and the register HO is connected to one of its outputs to receive store order operands returned from the succeeding units and also operands used by organisational orders.

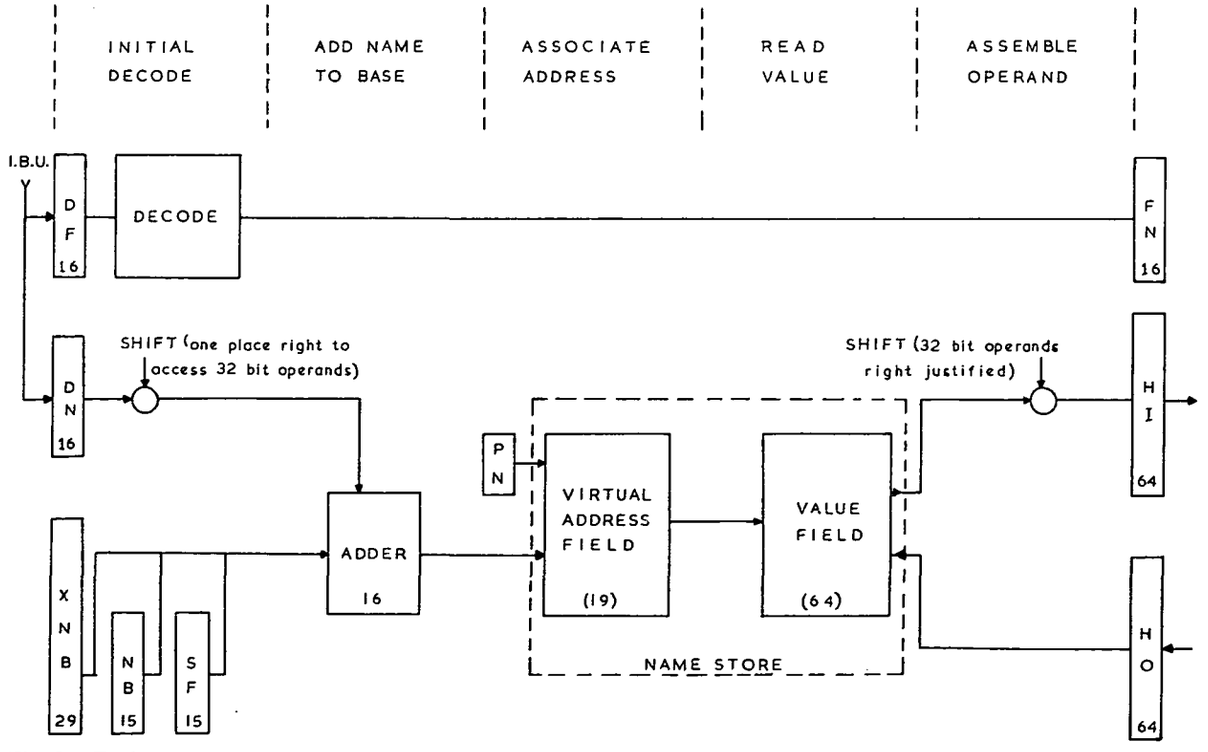

Fig. 3. Basic component parts of the primary operand unit

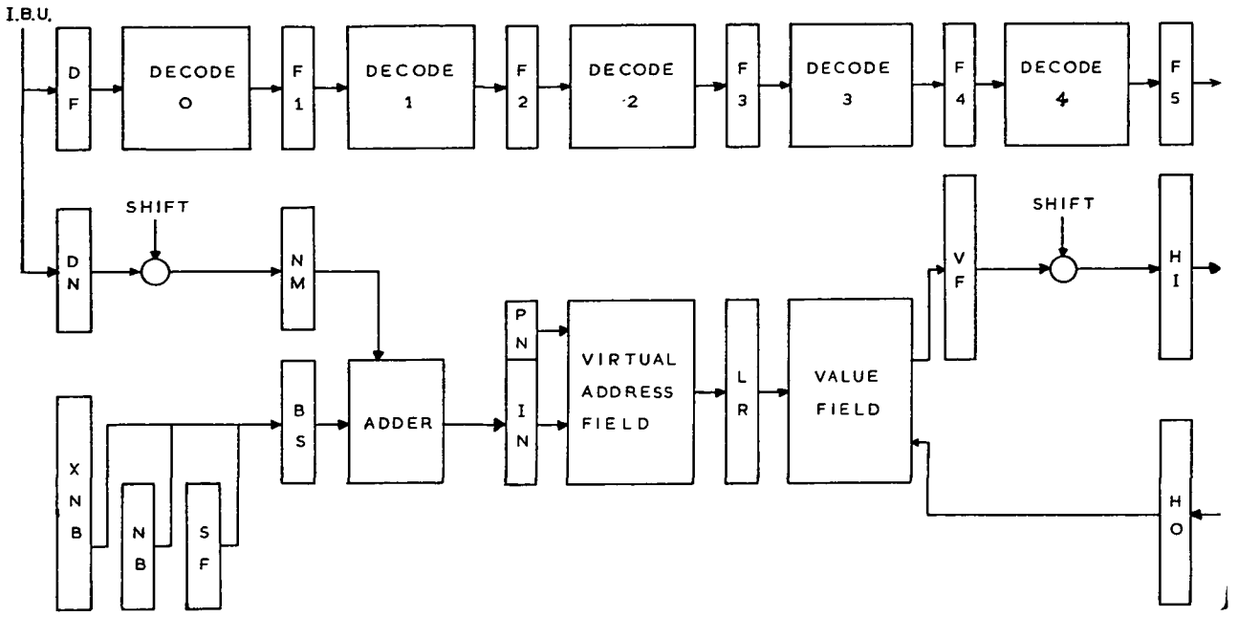

Fig. 4. Overlapped version of the primary operand unit

Each of these five stages involved in accessing an operand can be made independent of the others, and each requires a time comparable with that required for the execution of simple functions in the arithmetic units. Thus, not only can the accessing of an operand for one instruction be overlapped with the execution of another, but the various activities within the Primary Operand Unit itself can also be overlapped for different instructions. The Primary Operand Unit then becomes a 'pipeline', with five instructions proceeding through it at any one time. The maximum rate at which instructions are processed within it is accordingly increased by a factor of five (theoretically), although in practice the method of implementing the overlap limits this factor to 4·1 (Section 4).

4. Pipeline design

The overlapping of the various operations in the Primary Operand Unit is achieved by staticising the information obtained at the end of each stage in an appropriate flip-flop buffer register, the output of the register at the end of any one stage forming the input to the next stage. The flip-flops used in the buffer registers are 'D' type devices, which take up a state determined by the signal on the 'information' input when a pulse is applied to the 'strobe' input. **Fig. 4** shows the basic Primary Operand Unit with the additional buffer registers incorporated. The function itself is also staticised at each stage and the decoding circuits are distributed to the various stages in which they are required.

In order to prevent interference of one instruction with another the result obtained at the end of any one stage can only be presented to the following stage when the result of that stage has itself been staticised. The strobes used to copy information into the buffer registers must therefore be staggered, the overall timing diagram being as shown in **Fig. 5.** The shaded portions show the progress of one instruction through the Primary Operand Unit. It is first copied into DF and DN (Function and Name respectively) and after sufficient time has elapsed for the outputs of the stage to have settled, the registers F1 (function), NM (name), and BS (base) are strobed. The addition of name and base now takes place and after the appropriate time has elapsed the result is copied into IN, the Interrogate Register. The output of IN is concatenated with PN, the Process Number to form the input to the associative field of the Name Store. The result of the association is then copied into the Line Register, LR, the output of which accesses the appropriate line in the value field of the Name Store. The value field output is copied into the register VF and thence, after assembly, into HI, the Highway Input register.

The output of HI is propagated through the highway linking the Primary Operand Unit to the subsequent units of the MU5 Central Processor. Each of these units sends a control signal to the Primary Operand Unit indicating whether or not it can accept an instruction. Once accepted by the Unit the instruction is guaranteed to go to completion, so that the Control Register, also contained within the Primary Operand Unit, can be incremented for the instruction. An adder is associated with the Control Register and the addition of the increment to the value held in it occurs in parallel with the transfer of the instruction across the highway. (These operations form a sixth stage of the pipeline, so that the theoretical performance ratio is increased to six.) The actual strobing of the Control Register occurs at the same time as the strobing of the input buffer of the following unit. The register HI, and hence all other registers in the Primary Operand Unit, cannot be re-strobed for a succeeding instruction until this action has occurred. Instructions therefore proceed through the Primary Operand Unit in a series of beats, the rate at which these beats occur being determined by the acceptance rates of the succeeding units, i.e. when an instruction is accepted by the appropriate following unit a beat is initiated. Each instruction in the Primary Operand Unit is then moved onto the next stage by a pulse which propagates back along the pipeline (heavily drawn in Fig. 5).

The Primary Operand Unit itself imposes a minimum on the time between pulses propagating along the pipeline. This time is that required for the operation of the stages and must clearly be equal to that required for the longest stage. It is given by:

$$t_{\text{stage}} = t_r + t_l + t_s \qquad (1)$$

t_r is the time required for the information presented at the input of a buffer register to appear at its output (typically 5 nsecs), and t_l the time required to complete the logical operation carried out by the stage. t_l is principally determined by the time required for the associative store to operate (typically 25 nsecs). Once the output of a stage has settled it must then be maintained steady during the strobing of the following buffer register, i.e. for time t_s. t_s must be longer than t_r for reliable operation of the flip-flops (typically 10 nsecs), and since it must be assumed that t_{min}, the minimum time for a signal to propagate through a stage, could in principle be zero,

the strobe applied to any one register cannot begin until the strobe applied to the succeeding register has ended.

t_{stage} is thus typically 40 nsecs, and represents the minimum time between the completion of successive orders by the Primary Operand Unit. In practice, this time is affected by two conflicting factors: (i) t_{min} is not zero and some overlapping of strobes is acceptable; (ii) each stage requires several strobe generator circuits and these actually have pulse widths and inherent delays which are not all equal, i.e. the time allowed for t_s must be the maximum expected value.

The total minimum time required for an order to be completely processed by the Primary Operand Unit, i.e. the time from entering the Primary Operand Unit to acceptance at the far end of the highway by the following unit, is given by

$$T_{\text{PROP}} = 6.t_r + 6.t_l + t_s \qquad (2)$$

i.e. typically 190 nsecs. Without overlap within the Primary Operand Unit, i.e. without the internal buffer registers, this time would be reduced in principle by $5t_r$, i.e. to 165 nsecs, although in practice some of the registers provide fan out between stages which would otherwise require additional logic gates incurring similar delays. The advantage gained from overlapping the stages of processing within the Primary Operand Unit is thus an increase in execution rate of at least 165/40, i.e. 4·1.

In practice, the maximum rate cannot be maintained due to (a) the limited capacity of the Name Store and (b) the detailed requirements of the order code. The limited capacity of the Name Store means that the required operand is not always available within it, and an access must be made to the Local Store of the MU5 Central Processor. When the operand is returned from the Local Store it is written, together with its address, into an empty line of the Name Store and is then available for use after a delay of approximately 800 nsecs. In general, all lines of the Name Store are in use and a line must therefore be chosen for emptying. Various algorithms could be used for this purpose but any that is used has to be implemented in hardware, otherwise the time required to select the line would seriously affect the overall effectiveness of the store. The simplest algorithm to implement is a cyclic one in which lines are chosen in rotation and this is the one actually used. In addition, however, a digit associated with each line indicates whether or not its content has been altered by the action of a store order since it was copied from the Local Store. When the line is selected for emptying it is simply over-written if it has not been altered, but the address and value are read out and sent back to the Local Store if it has.

Names contained in the Name Store are actually B/D-names, i.e. names of integer variables or arrays, destined for the B-Arithmetic Unit or base registers, or the Secondary Operand Unit. Simulation studies carried out on Atlas (Odeyemi, 1970) indicate that with a 32-line B/D Name Store an access to the Local Store for a B/D-name must be made for approximately 2% of all accesses. The average time per order is therefore increased to [98 × 40 + 2 × 800]/100 nsecs, i.e. 55 nsecs.

A-names, i.e. names of real variables (floating point, decimal, etc.) destined for the Accumulator Unit are held in a separate name store associated with the Secondary Operand Unit. All accumulator orders pass through this unit and up to 12 orders can be overlapped within it. This means that if a name held in the B/D store were used to accumulate a total calculated by a program loop using accumulator orders, at least 12 orders would have to separate the order storing the total and the order re-accessing it if the overlapping of orders was not to be held up. This situation is unacceptable, whereas holding the A-names in the Secondary Operand Unit reduces the minimum separation to two orders and involves considerably less effect on the overlap.

Certain effects of name store usage which would deteriorate overall performances have thus been avoided. An additional effect, that of store orders involving names which are held in the B/D name store, is discussed in the next section, together with other effects on performance which arise from the detailed requirements of the order code.

5. Requirements of the order code

The description of the pipeline action so far has assumed that the order being processed is a simple 16-bit order specifying a named variable. Whilst this type of order is expected to occur comparatively frequently in practice, the order code has provision for considerable variations, and the effectiveness of the pipeline varies according to the type of order being executed.

The most important types of order to be considered in terms of their effect on pipeline performance are the following:

1. Long orders.
2. Double orders.
3. Store orders.
4. Base manipulation orders.
5. Control transfers.

1. A short order can only specify one of 64 named 64-bit or 32-bit variables or a literal of value ±32. To access more names an extended operand type (Fig. 2) is specified, and the name used is then specified by an additional 16 bits. These additional

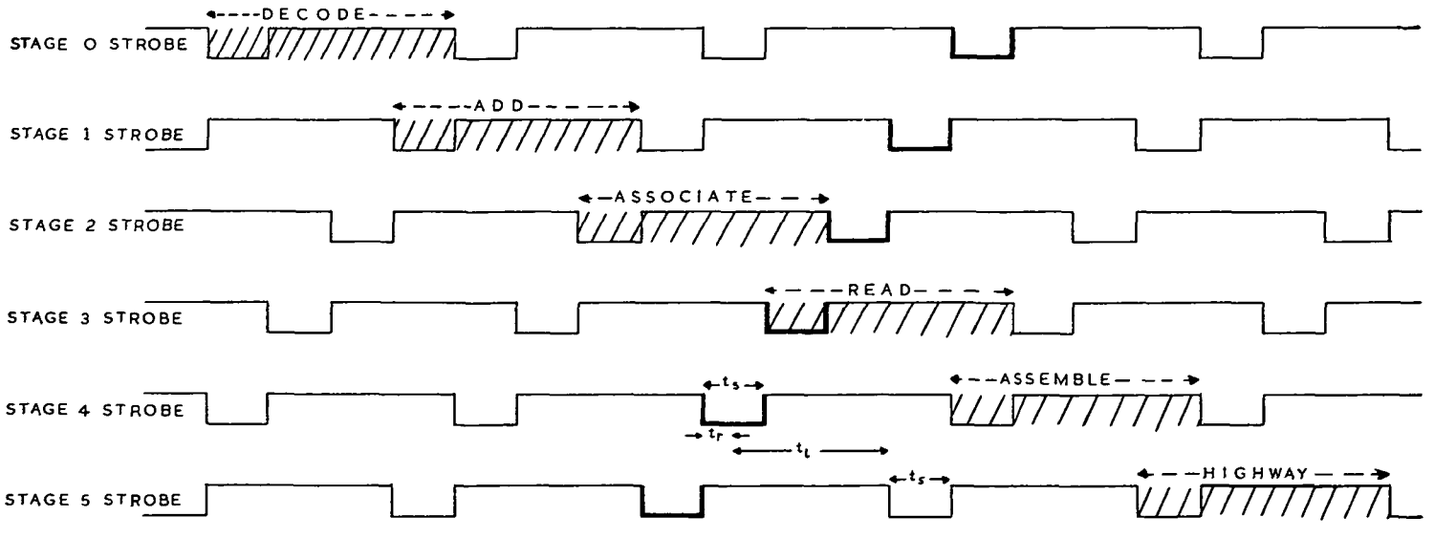

Fig. 5. Basic timing diagram of the primary operand unit

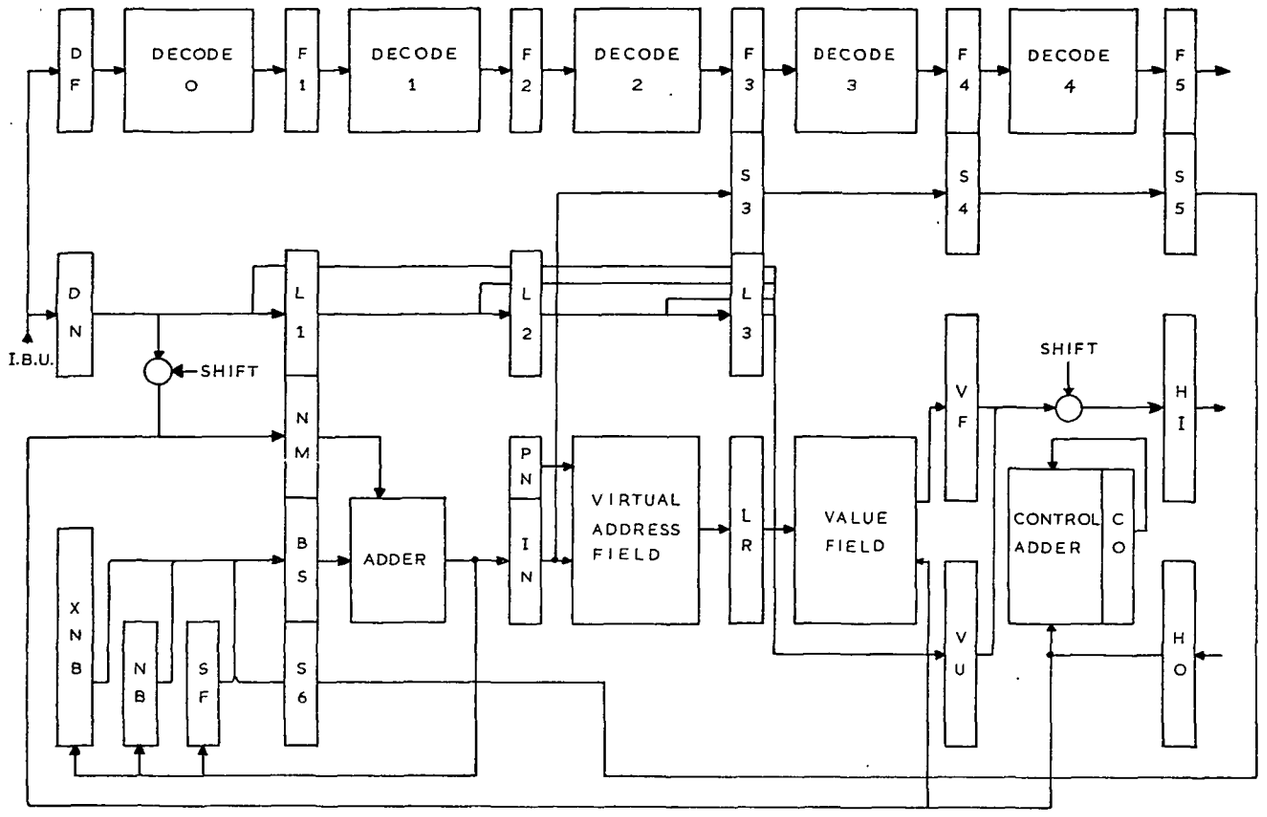

Fig. 6. Complete primary operand unit

16 bits are obtained from the Instruction Buffer on a second beat of the pipeline after the long order has been decoded, and are loaded into DN. (DF and DN are both 16-bits long, and for a short order contain the same information. DF is always used for decoding, however, and the appropriate number of name digits is selected from DN according to the length of the order.) As the long order cannot proceed to the second stage of the pipeline until the name becomes available, a dummy order must be created in the second stage, ahead of the long order. This is achieved by setting an additional function digit—the 'Valid Digit'—to zero. On the next beat of the pipeline the long order proceeds to the second stage of operation as for a short order, accompanied by a Valid Digit set to 1.

Literals greater than ± 32 may be specified as being 16-, 32- or 64-bits long in an extended order, the additional bits being obtained in units of 16. The actions in the first and second beats are identical to those for long orders, but in the third beat the content of DN is copied into LI (**Fig. 6**), the first of three special literal registers. The function proceeds through on this beat, and after five beats appears in F3. The literal is now in L3 if it is 16-bits long, L3 and L2 if it is 32-bits long or L3, L2, L1 and DN if it is 64-bits long. On the following beat the literal is assembled into VU, the function now being in F4. In the next pipeline stage the appropriate register, VU for a literal, or VF for a name, is selected for routing into HI. The literal does not go directly into VF since this would require additional logic gates at the input to VF, thereby increasing the logic time between LR and VF to longer than the design figure of 25 nsecs.

The time taken to complete any long order is clearly increased according to the number of beats required, and the net completion rate of orders correspondingly decreased.

2. The *= function has been described in Section 2. It requires access to two distinct operands, and hence must involve two separate pipeline beats. The stacking operation also involves an additional complication: the Stack Front register SF is incremented for an order while there are still several orders ahead of it not yet completed (i.e. the Control Register CO, at the end of the pipeline, has not yet been incremented for these orders). Any one of these orders may be a control transfer order requiring that the partially processed orders in the pipeline be abandoned in favour of a different sequence. Should this situation occur, the SF register will contain an incorrect value. The problem of maintaining the correct SF value can be overcome by preventing overlap in such situations, but this would seriously deteriorate the pipeline performance, and the alternative solution adopted is to allow the SF register to be incremented when required and to carry along with the stacking order the value of SF created by it (registers S3, S4, S5 in Fig. 6). When the control register is incremented for that order, S5 is copied into S6. Now when a control transfer occurs the value in SF may have been altered by orders which are abandoned, but the value in S6 is correct and is used instead of that in SF for the first stack access of the new sequence.

3. In the case of store orders using B/D-names, the order does not reach the unit from which the operand is to be stored until some time after the required access has been made to the Name Store, i.e. the operand is not available at the appropriate time. In the case of store orders within the organisational function set this is not important since these orders occur infrequently and simply causing a hold-up does not seriously affect the pipeline performance. Store orders within the computational function set occur quite frequently, however, and causing a hold-up in the pipeline to await the return of the operand would involve a delay of around 240 nsecs. Since this delay is long in comparison with the basic order time of 40 nsecs, a different technique has been adopted.

To avoid the need for a hold-up in the operation of the pipeline, the content of the Line Register is preserved, for a B store order, in an additional register BW (not shown in Fig. 6). A special digit is set in F2 and the order proceeds unimpeded. The special digit in F2 prevents any further B store orders

from proceeding beyond F2 until the first store order has been completed.

When the operand required by the store order becomes available from the central register, it is returned to HO and the Primary Operand Unit interrupts its normal instruction sequence. The information held in BW is then used to re-access the appropriate line in the value field of the Name Store and the contents of register HO is written into it. (The information in BW is also used to hold up any further orders accessing the address for which a store order is outstanding.) The additional time required to execute a store order is approximately 80 nsecs, in this case, making a total of 120 nsecs per store order.

4. When a base manipulation order is decoded in stage 1 the overlapping of instructions in the pipeline must be abandoned, since the next order in sequence must use the correct base value. The base operations are performed by copying the operand (received from the highway in HO) into NM, and the base (for an addition) into BS. The adder is then activated, and the result routed back to the appropriate base register. The delay involved for such orders is of the order of 400 nsecs, but since these orders occur infrequently no special action has been taken to reduce the delay.

5. The execution of control transfer orders requires a hold-up in the pipeline operation until the appropriate operand is is received in HO. When it is available, this operand is routed to one input of the adder associated with CO and CO itself (for a relative transfer) is routed to the other input. After such an order has been executed the succeeding orders in the pipeline will be incorrect if the transfer is (a) unconditional, or (b) conditional and the condition is satisfied. The 'Valid Order' digits associated with each function register are all re-set to zero under these conditions and a gap of at least 940 nsecs is created, i.e. 750 nsecs for the store access for the new instructions and 190 nsecs for the time through the pipeline.

Since control transfers occur as about 10% of all orders (Sunderland, 1970), a delay of this order is unacceptable. The Instruction Buffer Unit is therefore designed to reduce the number of occasions on which this delay is incurred by predicting the result of the control transfer (Taylor, 1969). The prediction technique is based on the use of an associative store containing addresses of instructions which have previously caused control transfers to occur and a corresponding conventional store containing the addresses of instructions to which control was actually transferred. Before an instruction pre-fetch cycle is initiated by the Instruction Buffer Unit a check is made in the associative store to determine whether any of the instructions currently in the buffer has caused control to be transferred. If equivalence is found in any of the associative registers the address of the next instruction to be pre-fetched is read from the corresponding register in the conventional store rather than being evaluated by incrementing on from the previous pre-fetch address.

Now when a control transfer is obeyed the orders in the pipeline are correct if either (a) the transfer condition is satisfied and the following order is the first order of the new sequence, or (b) the transfer condition is not satisfied and the following order is in normal sequence. In either case the Control Register is set to the appropriate value and after a delay of 120 nsecs in excess of the basic 40 nsecs the normal pipeline action is re-started and the succeeding orders in the pipeline are obeyed normally. An additional digit accompanies each order from the Instruction Buffer Unit to indicate to the control transfer logic that an order is the first order of a new sequence of instructions rather than a continuation of the previous sequence.

If the transfer condition is satisfied and the following order is in normal sequence, i.e. the prediction has not been made, then provided that the operand used in the control transfer order is a literal (i.e. is invariant), the Primary Operand Unit signals the Instruction Buffer to set up the appropriate addresses in its store for future prediction. At the same time the actions of re-setting the 'Valid Order' digits and accessing the store for the next instruction occur as before. These actions also occur if the control transfer condition is not satisfied and the following order is the first of a new sequence, i.e. the transfer has been incorrectly predicted. Simulation studies show that correct prediction occurs in approximately 75% of cases where it can be applied.

Other detailed requirements of the order code also affect the performance of the pipeline, but these are expected to occur relatively infrequently in practice and do not need to be taken into consideration here.

6. Conclusions

The principal aim of designing the Primary Operand Unit of the MU5 computer as a pipeline is to increase the rate at which orders can be passed on to succeeding units (and to match this rate to the operating rates of those units), and a factor of 4·1 has been achieved for simple orders. In assessing the overall effectiveness of the pipeline technique, however, its performance for all types of order, and the additional equipment and complications involved, must be considered.

The amount of equipment required for operand accessing by the Primary Operand Unit (apart from the Name Store) is approximately 1,500 MECL integrated circuits, and some 500 of these are incorporated to allow the various stages of operation within the unit to be overlapped. The Name Store itself is made up of more complicated integrated circuits than those used in the rest of the Primary Operand Unit, and the total number required is equivalent to 1,000 MECL circuits. Thus the additional amount of equipment required to achieve an increase of 4·1 in the basic Primary Operand Unit performance is one quarter of the amount of equipment required for the basic unit itself. The inclusion of this equipment clearly involves additional complication in the timing and control logic, but this factor is offset by the fact that the unit can be easily partitioned into its various sub-units, thereby allowing the detailed design and manufacture of the sub-units to be carried out independently.

The overall performance of the Primary Operand Unit pipeline is determined by the relative frequency of occurrence of the types of orders described in Section 5. Information about

Table 1 Overall performance table

TYPE OF ORDER	EXCESS TIME	% OCCURRENCE	NET TIME ADDED
Long Store	40 nsecs	10	4 nsecs
	80 nsecs	15	12 nsecs
Organisational (except Control Transfers)	360 nsecs	1	3·6 nsecs
Control Transfer (Predicted)	120 nsecs	6	7·2 nsecs
Control Transfer (Unpredicted)	940 nsecs	4	37·6 nsecs
Name Store‡	800 nsecs	2	16 nsecs
	Total net time added		80·4 nsecs

The types of order listed in column 1 require the times shown in column 2, in excess of 40 nsecs, for their execution.

Column 3 shows the expected percentage occurrences for these orders, and column 4 the net time added to the execution time of an average order due to each type. The overall average execution time is thus expected to be approximately 120 nsecs/order.

these frequencies of occurrence is difficult to obtain since (a) no existing computer has an order code structure comparable with that of MU5; and (b) tracing simulated MU5 code on an existing conventional computer requires an excessive amount of computing time in order to obtain meaningful statistics. Such figures as are available have therefore been obtained by tracing existing programs on the Manchester University Atlas computer and assessing their meaning for MU5 by a comparison of the corresponding compilers. These figures are summarised in **Table 1.** Column 2 indicates the extra time in excess of the basic 40 nsecs required for the various types of order, and column 4 the net time added to the order time due to each cause. Thus the net time for the completion of an average order is 40 nsecs plus the sum of the figures in column 4 of the table.

The excess times are principally incurred by long orders, store orders, organisational orders and orders which give non-equivalence (‡) in the Name Store. Long orders are mainly used for named variables (a) with organisational orders, and (b) when the number of named variables within a routine exceeds the number addressed by a 6-bit name, i.e. 64. These situations occur infrequently, however, since (a) organisational orders generally use short literal operands, and (b) routines tend either not to use that many names, or else to use the names declared first (and therefore allocated to 6-bit names) most frequently. The other use of long orders is for literals, and these incur a 40 nsecs penalty for each 16 bits used. The net effect of these factors is estimated at 10% of all orders requiring an additional 40 nsecs for their execution. The overall average length of an order may also be assessed from these figures as $[90 \times 16 + 10 \times 32]/100$, i.e. 18 bits approximately.

Store orders to the B/D Name Store can be expected to occur as about 15% of all orders. Store orders to the A Name Store occur rather more frequently (around 25%) but this does not directly affect the performance of the Primary Operand Unit.

Organisational orders other than control transfers occur as about 1% of all orders. Control transfers account for 10% of all orders and are either predicted or not predicted or are non-predictable. Assuming that 80% of control transfers are of the predictable type then 75% of these, i.e. 60% of all control transfers, will be predicted and thus 6% of all orders will be of this type. Similarly 4% will be unpredicted control transfers.

Some orders will be combinations of these types, but this does not affect the final figure since the individual excess times are additive. Thus the total time taken to complete an 'average' order in excess of the basic 40 nsecs is, from column 4 of the table, 80·4 nsecs, giving an average time per order of approximately 120 nsecs.

This figure is necessarily speculative, but since the MU5 computer is a research machine, attention is being given to hardware monitoring techniques for determining the frequencies of occurrence of events which affect the system performance during normal running, and to methods of increasing the overall completion rate by modifying the existing design for dominant types of order. This information will then allow a better assessment to be made of the relative importance of incorporating particular features of design into the system and therefore allow the best use to be made of central processor equipment in future computers.

Acknowledgements

The MU5 Project is supported by SRC and ICL. The author would like to thank all members of the joint University-ICL design team for many helpful discussions, but particularly Professors T. Kilburn, D. B. G. Edwards, F. H. Sumner and D. Aspinall (now at University College, Swansea) for their advice and encouragement and Mr. R. B. Lee for his valuable assistance throughout the project.

References

ASPINALL, D., KINNIMENT, D. J., and EDWARDS, D. B. G. (1968). Associative memories in large computer systems, reproduced in *Information Processing 68*, Vol. 2 (Morrell, A. J. H., editor), Amsterdam: North Holland Publishing Co.

BROOKER, R. A. (1970). Influence of high-level languages on computer design, *Proceedings I.E.E.*, Vol. 117, pp. 1219-1224.

BUCHHOLZ, W. (1962). *Planning a Computer System*, New York: McGraw-Hill Book Co.

GRAHAM, W. R. (1970). The parallel and the pipeline computers, *Datamation*, Vol. 16, pp. 68-71.

KILBURN, T., EDWARDS, D. B. G., LANIGAN, M. J., and SUMNER, F. H. (1962). One level storage system, reproduced in *Computer Structures—Readings and Examples* (Bell, G. G., and Newell, A., editors), 1971, New York: McGraw-Hill Book Co.

KILBURN, T., MORRIS, D., ROHL, J. S., and SUMNER, F. H. (1968). *A system design proposal*, reproduced in *Information Processing 68*, Vol. 2 (Morrell, A. J. H., editor), Amsterdam: North Holland Publishing Co.

LINDSEY, C. H. (1970). Making the hardware suit the language, reproduced in *Algol 68 Implementation* (Peck, J. E. L., editor), Amsterdam: North Holland Publishing Co.

MURPHY, J. O., and WADE, R. M. (1970). The IBM 360/195, *Datamation*, Vol. 16, pp. 72-79.

ODEYEMI, I. A. (1970). Experiments on Operand Buffer Stores, Ph.D. Thesis, University of Manchester.

SUNDERLAND, E. (1970). Automatic Program Tracing, M.Sc. Thesis, University of Manchester.

TAYLOR, L. A. (1969). Instruction Accessing in High Speed Computers, M.Sc. Thesis, University of Manchester.