Gaines, B. R., Facey, P. V., Williamson, F. K. and Maine, J. A. (1974). Design objectives for a descriptor-organized minicomputer. In European Computing Congress Conference Proceedings, EUROCOMP 74, pp. 29-45. Online, London.

Design objectives for a descriptor-organised minicomputer

B.R. Gaines, P.V. Facey,

Department of Electrical Engineering Science,
University of Essex, Colchester, Essex, U.K.

and

F.K. Williamson, J.A. Maine,

Micro-Computer Systems Ltd,
Boundary Road, Woking, Surrey, U.K.

Abstract

This paper describes the logic and motivation behind the design of a descriptor-organised minicomputer for commercial production. It is argued that the availability of low-cost integrated circuit families has made it simple to achieve the conventional minicomputer design objectives (low-cost, fast-response, high reliability, and ease of interfacing), and that additional objectives are now both possible and necessary. The most attractive lines of development appear to be those related to simple, secure and swift software engineering, and the requirements for the hardware support of process-structured operating systems and high-level languages are analysed. Finally a minicomputer design is outlined which combines microprogrammed multi-length operations, data descriptors and dynamic segmentation, linked to procedure calls, to satisfy most of the detailed objectives established.

1. Background

This paper describes the logic and motivation behind a design process which led to a descriptor-organised minicomputer for commercial production, and outlines the instruction set and organisation of the computer. The machine is a sister product to a small process-control orientated 8-bit minicomputer which is widely employed for machine-tool control, and the original design brief was for a machine using the same technology (TTL, 16-bit wide main memory and single-layer boards) but with increased computing power and range of applications. However, it became apparent in the early design stages that currently available MSI circuits would never be fully exploited in an extension of the conventional minicomputer architecture and a break was made with convention - the final design objective (with the same cost/technology constraints) was a machine that was both orientated to the real-time and reliability requirements of the minicomputer market and simple and natural to the programmer and compiler writer - a language and data-structure orientated machine. This paper outlines the basis for this re-appraisal, the hardware developments which made it possible, and the implications of some features of the machine for future minicomputer applications and for language development.

1.1 The Effective Use of Hardware

Historically the architecture of the digital computer and its range of applications have always seemed to be driven by the state of electronics technology. In recent years, however, we have arrived at the point where limitations of technology, size, speed, cost, power-consumption, ease-of-fabrication, etc., are very much less restrictive and very soon may become virtually non-restrictive. Certainly the classic von Neumann architecture places few demands upon current digital circuit technology and "anyone" can run off a minicomputer in a short time. Only a decision to push the technology to its limits involves major circuit engineering problems, and it is not clear that ultra-high-speed is the best long-term answer to providing high computing power.

Thus a basic problem for small computer designers nowadays is the effective exploitation of the capabilities of current electronic components. There is the danger on the one hand of designing a too-simple machine where the processor cost is negligible relative to power-supplies, cabinet and memory, whereas on the other hand a machine may be over-engineered with a diverse range of complex facilities that do not integrate well together and are difficult to utilise particularly in compiler-generated code. The distinction between the "hardware availability" of powerful features and the "software availability" of the same is highlighted in Brooker's (1970) paper on the influence of high-level languages on machine design. In an appendix to this paper Laski remarks, "unless the

logical designer understands all the software implications of the registers he is providing, the architecture of the computer that results will be such that high-level languages misfit rather than fully use the hardware provided".

These remarks are not intended to propagate a demarcation between hardware and software engineering. Both come together logically in the concept of deferred design - the practical importance of computers is best analysed by contrasting system development based on classical manufacturing techniques. Conceptually, the computer enables a major component of hardware production to be reduced to the manufacture of a general-purpose, mass-produced machine, so that specialist system design may be deferred to a later stage involving, perhaps, only "pencil-and-paper" software production and not hardware engineering. From this point of view the computer designer, the compiler writer, and the applications programmer are all part of the same team of system designers and there is a clear rationale for their activities and objectives to be considered together.

Conflicts of interest arise, not so much through hardware/software demarcation as through the examination of one part of the system implementation process in isolation from the others. The optimal overall system performance does not necessarily coincide with the local optima for each stage of the design, and, as a consequence, the relative priorities of different design considerations vary according to the ultimate application. For example, in a once-only system development software costs dominate and speed or core-occupancy can be well traded for ease of programming. The opposite consideration applies to a machine designed for use as a component in a standard system to be manufactured in quantity.

Minicomputers have been associated historically with small systems manufactured in quantity and other highly cost-sensitive applications, and in the past hardware constraints have been overwhelming for the minicomputer designer. For example, to keep a balance between logic circuit costs and core memory costs some ten years ago it was necessary to design simple instruction sets with order-codes that were easy to decode. Some five years ago logic costs dropped disproportionately to core costs and it made sense to increase the range of instructions and complexity of the order-code so that programs could be encoded with minimum memory requirements. This minimizes the hardware cost of the computer in a system but makes software support difficult since automatic code generation for such machines is virtually impossible and the only aid possible is a good assembler.

Presently memory costs are decreasing rapidly and, with the advent of semiconductor memories, will continue to do so - it makes sense to trade increased memory utilisation for better software support and go for instruction sets and order-codes that make

automatic code generation as simple and effective as possible. This corresponds to two commercial influences on the minicomputer designer: (a) that minicomputers on a few LSI chips are available for the highly hardware-cost-orientated single-function small systems market; (b) that the systems requirements for minicomputers are growing more complex and a degree of flexibility and continuous (and thus essentially one-off) in-service development is expected by most customers. Even if a minicomputer is originally justified as a part of a system purely on grounds of lower hardware cost than a specially manufactured item, it is also generally seen as a safety factor allowing future system modification and extension. Such an open-ended capability is more apparent than real in most systems to date. It requires a degree of intrinsic protection and modularity that is not easy to achieve, and may be impossible through software alone. A requirement for continuous system development also raises the level of programming activity and reduces control over it thus increasing the relative importance of software support.

In conclusion there is clearly no absolutely best possible machine design without specification of objectives and constraints. However, there is a distinction between a machine being optimal under given constraints and those constraints themselves being correct. In the following section we discuss the appropriate design objectives for a next-generation minicomputer and in the final section propose a machine organisation that satisfies many of them.

2. Design Objectives

It would be well at this juncture to review briefly the implicit mundane constraints upon the minicomputer designer so that the more recent objectives indicated in the previous section and discussed later are kept in proper perspective. The major considerations are:

(i) Low cost - useable configurations for a few thousand pounds not tens of thousands.

(ii) High reliability under fairly unconstrained industrial conditions.

(iii) Rapid response to external conditions - high-priority interrupts to be serviced in microseconds - there is a degree of nimbleness required which does not figure in EDP applications - maximum response time is often more important than throughput.

(iv) Ease of interfacing - anything may be hooked onto a minicomputer and both hardware and system software should be orientated to ease of interfacing.

(v) Modularity of configuration - buyers treat the minicomputer manufacturers' catalogues as a module supermarket and expect to be able to tailor configurations to their applications with the minimum of constraints.

35274

Given these basic requirements, we have argued that the minicomputer designer is being influenced both by technological advances and by market requirements to pay (relatively) less attention to hardware considerations and more to problems of software development. These may themselves be split roughly into two areas, one related to the technical operation of the system on a maximally protected basis despite software modification, and the other related to the problems of the programmer in generating software. In recent years there have been major developments in the science of both operating systems and high-level languages and, although these have been generally considered in the context of large machines, the experience gained and concepts generated seem to be immediately applicable to minicomputers.

2.1 Processes, Protection and Peripherals

The organisation of programs into maximally independent small modules with strict limitations on their access to the resources of other modules and well-defined inter-module communication is attractive on many grounds (Hansen 1970, Rossiensky and Tixier 1970, Dennis 1973, Horning and Russell 1973). In particular it is a major aid to real-time system development and software reliability, both of which are of major importance in typical minicomputer applications. However, if the protection and communication requirements of such "process-structured" systems are implemented in software on conventional machines the overhead in execution time rules them out in their most important role - most real-time systems are also inherently short of time. The provision of additional hardware facilities to support process-structuring as a basis for effective software engineering has been a major consideration in third-generation machines (Dennis 1971).

A wide variety of hardware resource-control schemes have been implemented (well summarized in Watson 1970 and Wilkes 1972), but the essential central feature is the integration of memory management and procedure calling so that the activation record for a procedure is a well-defined and protected entity. If this control of storage access is extended to cover access to other system resources, "privileged" instructions, priority of CPU utilization, and other "capabilities" (Dennis and Van Horn 1966), then possible system degradation through the introduction of new (faulty) software modules can be strictly limited, monitored and controlled.

The allocation of dynamic storage to a procedure is naturally associated with a stack in block-structured languages. However, the protection of private storage, and the control of access to shared storage, require that the store be divided into more general "segments" (Organick 1972). The allocation of real storage to such logical segments can be complex particularly when multiple asynchronous processes are being activated and deactivated in an

indeterminate fashion (Cleary 1969) or where procedures have more equal standing (generalised co-routines) so that control may pass between them without necessary loss of activation records (Bobrow and Wegbreit 1973).

These considerations lead to the following design objective:

(vi) Flexible storage relocation/protection linked to procedure calls Tentatively we took the storage protection/allocation requirements of a multi-job operating system with shared procedures on the one hand, and the run-time procedure entry/exit requirements of FORTRAN IV and ALGOL 60 on the other, as "benchmarks" against which to test any proposed scheme. Subject to efficient operation in these cases and no increase in the basic cost, it was to be generalized to be, at least potentially, capable of handling wider requirements such as the deferred binding of MULTICS (Organick 1972).

This objective covers one aspect of the operating system - its relationship to the user. However, its relationship to the peripheral hardware of the system is of equal consequence. Although it is true to state that peripheral devices can be treated as "processes" in their own right and their operation and communication with other processes can be subsumed under the general procedures and disciplines established for any process, in practice users expect to be able to hang devices requiring arbitrary communications protocols on minicomputers and not be forced to buy special interfaces to support them. More generally also the processes required by many peripherals are activated very frequently compared with user initiated activities and may have real-time requirements that make it impossible for them to obey normal queuing and synchronisation disciplines. These considerations lead to the objective of:

(vii) Decoupling peripheral device requirements from the user environment We took it as a basic requirement that the uncontrolled user job mix on the system should not be able to affect real-time peripheral service requirements, and conversely that all peripheral transfers should be brought within the disciplines established for process operation without special hardware prerequisites in peripherals or interfaces.

2.2 Language Requirements

It is clearly not the prime objective of a minicomputer designer to support languages such as COBOL and PL1, if only for historic market reasons. Most minicomputer programming is still at machine level and any aids to assembly language programming are of major importance. Particularly relevant is the increasingly popular transition to "high-level" systems programming languages which offer

faster and more manageable programming than can be achieved in conventional assembly languages but with the same level of detailed run-time space/time control. However the exemplary implementations of such languages to date have been on large machines with wide address scope and a fairly uniform structure (IBM360 - Wirth 1968, PDP10 - Wulf, Russell and Habermann 1971) whereas most minicomputers have a variety of non-interchangeable address mechanisms as the scope changes, together with non-uniform structures designed to allow the programmer to use a variety of specific techniques, or "tricks", to cope with common situations.

These considerations suggest that the work on "high-level" systems programming languages provides a useful guide to machine design - possibly a more important one in terms of minicomputer applications than those of the classic EDP languages developed in a more machine-independent environment. From this point of view the main directions of development can be seen to be:

(a) Improvement of assembler syntax This is desirable in itself but also has relevance to the machine designer in that a syntax allowing the formation of larger constructs than single-instructions will, if tailored to the actual machine, allow expressions which may, or may not, appear natural to the programmer - it seems reasonable to aim for naturalness. For example, within the constraint that all storage allocation must be explicit, only a stack-organized machine will allow infix expressions to be accepted without requiring specification of temporary storage locations. Conversely, if constructs which are natural to the programmer generate a large amount of code, or, worse, widely differing code according to circumstances, or worst, are impossible to execute under certain circumstances, then it seems to indicate a mis-match between machine and user. For example, Wirth (1968: section 9) has criticized the lack of instructions for treating the 360 arithmetic condition codes as normal data items.

(b) Reduction of housekeeping A major incentive for PL360 was to take care of the base register housekeeping; such a requirement again reflects on the machine design that makes it necessary. More generally relevant is the removal of the housekeeping associated with various types of program control structures, loops, conditional execution, etc., and with data structure access. These control and data structures are implicit in all programs but the maintenance of the necessary control variables, pointers, etc. is a chore prone to error. The explicit forms adopted in languages such as BCPL (Richards 1969) and BLISS (Wulf, Russell and Habermann 1971) provide indication of mechanisms which could be incorporated in the basic instruction set.

(c) Remedying machine defects It is a common criticism of all machines that the hardware designer has made inessential differences

in the treatment of various operations, inconsistencies in the treatment of status information, unwanted side-effects on accessing certain registers, etc., etc. It is bitter experience that hardware "options" in particular require far bigger drivers than originally envisaged, 90% of which look after exceptional conditions which rarely occur (and hence may seem of lesser importance) but which must be properly treated when they do. It is easy to dismiss each individual defect of a machine as a design fault having no general implications. However, there is the general lesson that such faults readily creep in and need positive preventative measures.

One may well ask if the machine is to become "higher-level" in itself why not make the transition completely and support a language such as FORTRAN or ALGOL as completely as possible ? This is feasible - the B6500 may be thought of as an ALGOL machine and FORTRAN machines have been proposed (Bashkow, Sasson and Kronfeld 1967). It is particularly attractive when the language supported is not well suited to conventional architectures (EULER - Weber 1967, APL - Hassitt, Lageschulte and Lyon 1973). However if one is competing with assembly language then any one existing language is inadequate. No language offers the full variety of data types and constructs suitable for the range of current applications - the more powerful languages such as SNOBOL, PL1 and ALGOL 68 are not widely accepted and the established languages such as FORTRAN lack many facilities. Additionally all languages have conceptual flaws, on the one hand features which are little-used but cause inherent run-time problems, and on the other hand features which are inconsistently provided and lead to arbitrary restrictions perhaps related to the original implementation. In these circumstances one is tempted to invent a new language such as that for the SYMBOL machine (Chesley and Smith 1971) and optimize the language and the machine together - whilst technically attractive this is commercially unacceptable.

However, whilst no one high-level language is adequate as the basis for machine design, in total the objectives of a range of languages provides a foundation. Two sources in particular seem important:

(a) The extension of existing languages intended for general use, notably: ALGOL 60 in the light of Wichmann's (1973) analysis; FORTRAN in the light of the ANSC X3J3 proposals (FORTREV 1973); and ALGOL 68 (Lindsey and van der Meulen 1971) in the light of Lindsey's (1971), and other implementation comments (Peck 1971).

(b) The extension of existing machine facilities established in "systems-programming languages" e.g. PL360, BLISS and BCPL.

From these considerations one may extract the following main objectives:

(viii) Consistency and uniformity of treatment of all facilities - We took it as a basic design requirement that every operation be available with every address mode and with every operand type, and that status information should be treated as a normal data item.

(ix) A wide range of operand types - most machines provide one integer and one floating-point format with a means of "double-precision" - the provision of a wider range of integer and real lengths seems a common objective to all extensions. We took it as a basic design objective to provide a full range of operand types covering all normal applications.

(x) More explicit data structures - data-structuring on most machines is primitive, typically up to the level of a combination of indirect and indexed addressing to allow easy access to 2-dimensional arrays - the data-structures made available by languages are most often implicit to the program generated by the compiler and do not exist as run-time attributes of the data, e.g. the structure of FORTRAN COMMON cannot be determined except by the way it is used, although in more dynamic languages "dope vectors" and "thunks" make structure more explicit - "reference-variables" go even further in allowing structural information to be passed at run-time - it is particularly in the manipulation of pointers and packed structures containing mixed data and pointers that assembly language generally scores over higher-level languages. We took it as a basic design objective to provide explicit data structuring through hardware-interpreted "reference variables" and "dope vectors", and generally to allow programs to be "data-driven" wherever appropriate.

(xi) Separation of instruction set and order-code - many of the programming problems of machines, particularly minicomputers, arise because the order-code seems to have been designed before the instruction-set and then instructions have been "fitted in" - problems such as: "paging" (limited address range); the need to change address modes, say from direct to indexed, as the scope changes; side-effects from basic operations where several functions have been crammed into one operation (e.g. increment and skip if zero as the only increment operation); all these arise from order-code constraints on instruction-set availability - whilst it is clear that the word-size must affect the range of instructions that can be encoded in one word, it should not be allowed to place arbitrary limits on the instruction set. We took the design criterion to be that the instruction set should be designed first (with "architectural" not "order-code" considerations in mind) and that the "order-code" should be designed thereafter to minimize program storage requirements in "typical" applications on the one hand, and to permit rapid instruction-decoding on the other. It is even possible to envisage that the

order-code of the machine will change in the light of experience (and the assembler or, preferably, loader) without change in the instruction set and without users being aware of the change.

2.3 Extensibility and Microprogramming

One objective in any machine design is lifetime - there is no sustainable argument for freedom from obsolescence of any current design since computer science and user requirements are both in a state of flux and we do not have firm foundations for tomorrow's needs. In these circumstances it is probably a better strategy to aim for a "clean" simple basic design giving facilities which appear to be universally required and leave a large-part of the order-code unallocated. Undefined codes can then cause traps which can be software-interpreted initially with later moves into microprogram and then hardware if required. This requires a well-defined trapping system that can partially execute instructions; trap if an unknown address-mode, operand-type or operation is specified; pass information to a software routine in an appropriate format (pointers, operands, etc); and continue instruction execution when the routine exits having interpreted the undefined part of the code. This capability is a substantial answer also to Brooker's (1970) suspicion that high-level language machines may be "a prison for the thoughts of a language designer or compiler writer".

(xii) Extensibility through trapping was taken as a basic design objective.

Microprogramming may itself be seen as a means rather than an end - it is probably the only technique by which one of our basic objectives might be realised: the availability of a range of machines with variable cost/speed trade-off but with identical architecture and facilities. However, our past experience with MINIC I, a small microprogrammed machine had also demonstrated the capabilities of microprogramming to provide high-speed specialist facilities and speed-up key operations with no hardware change save the addition of micro-program memory. This capability is an important machine feature in its own right and we wished to retain it in MINIC-S.

(xiii) Dynamic microprogramming was taken as a fundamental requirement.

2.4 Overall Image of Machine

From all that has been said before most readers with experience of computer manufacture will see the abyss of customer education yawning before the proposed machine. As noted previously the original requirement was a marketing one and the technical objectives arise logically out of commercial and production considerations - the facilities provided are not expensive in hardware and are a necessary

foundation for current software techniques. However, there is one criterion fundamental to any design - a user should never incur penalties for complex facilities when he only needs simple facilities. This is all too often forgotten and the question as to how the machine appears to a non-multi-processing, 16-bit data only, etc., user has been a constant benchmark. Similarly we have not sought to impose conceptual constraints upon the user - e.g. stack-operations are important but so is the capability to use the machine in the single-accumulator mode to which most minicomputer users are accustomed - both modes should be available. Thus a final overall objective is:

(xiv) No penalty when the more advanced facilities are not used. This applies both technically to space/time overheads, and commercially to an unnecessarily complex image of the machine.

3. The Design of MINIC S, A Descriptor-Organized Minicomputer

In the previous sections we have tried to express clearly our design objectives and their logical foundations. Much of the discussion has been retrospective in the sense that hardware considerations make certain facilities simple and economic to implement and many of the features of the machine arose as logical extensions of quite simple basic requirements and are justified as much by pragmatic arguments as by the preceding rationalisations. In the ensuing discussion we shall present such arguments also since they indicate that engineering/economic considerations are equally strong in forcing machine design along the lines put forward.

3.1 Overall Organisation

Figure 1 shows the overall structure of MINIC S. It will be noted that we have achieved objectives (iii), (iv) and (vii) by the simple device of including a MINIC M (the most recent version of our first 8-bit minicomputer MINIC I) as an I-O processor. This was originally a hardware proposal in that it was cheaper to use a MINIC M CPU as an interface card rather than produce a new one that integrated into the complex MINIC S segmentation and variable-length operand scheme. However it has a number of ancillary advantages that have proved equally compelling:

(a) No new peripheral controllers The complete range of MINIC M peripherals are now also MINIC S peripherals.

(b) Device-independent I-O software exists MINIC M has a device-independent, buffered, multi-job, data-transferring and scheduling package. The device handlers are table-driven and can cope with a wide variety of peripheral devices, character codes, control codes, etc. This can be transferred virtually completely to MINIC S.

(c) Maintenance and fault diagnosis The diagnosis of a machine as

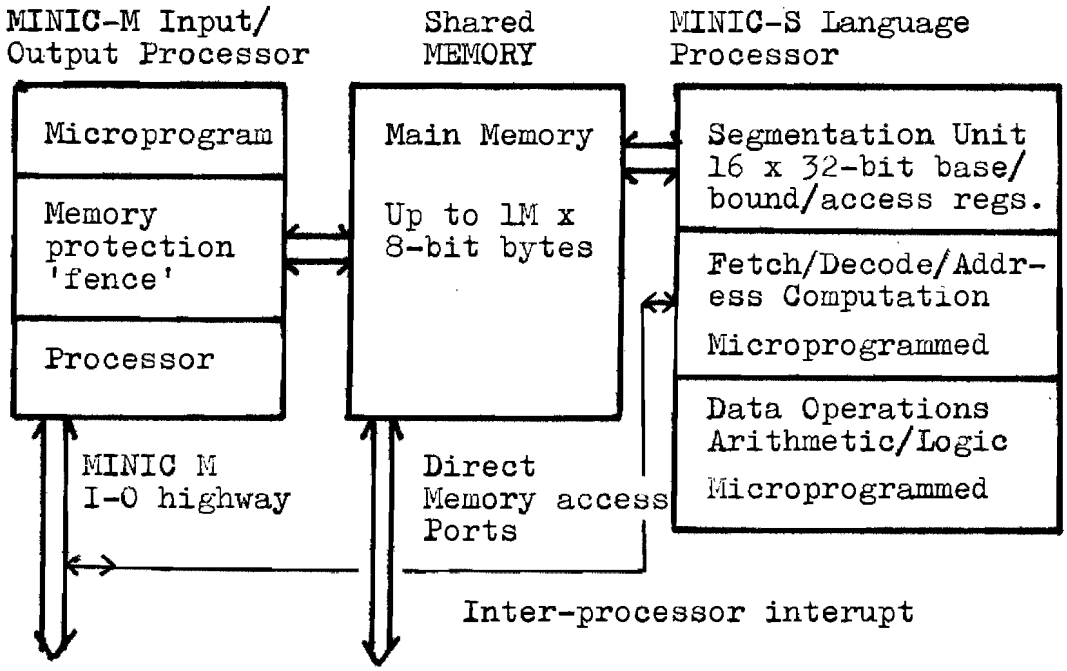

Figure 1 MINIC S

complex as MINIC S is difficult. MINIC M is simply checked out and may then be used as diagnostic exerciser of the MINIC S hardware. The software already exists as a MINIC S emulator for MINIC M which was developed in order to enable MINIC S systems software to be generated in advance of hardware manufacture. The routine availability of a minicomputer as a tool to the service engineer is very attractive.

(d) Special-purpose real-time software There are both programs and special microprograms for MINIC M in critical activities such as machine-tool servo loops. These low-level high-speed routines can continue to be utilised as part of the I-O environment.

The overall effect is that the MINIC S operating system can rely on all peripheral transfers taking place in a uniform, disciplined manner, and need take no account of differences in devices, communications, protocols, etc. The two processors communicate through the memory with a single interrupt line to direct attention. MINIC M is able to monitor the activitities (in terms of procedure currently executing) in MINIC S and hence act as an intelligent priority interrupt system. MINIC M has elementary protection that allows a program to be trapped on attempting to execute privileged instructions or to access memory above a "fence". Hence secure software development on MINIC M is possible whilst the whole configuration is in use, although users will only need to regard this machine as a computer when interfacing new peripheral devices.

Looking now at the MINIC S processor, it consists of three separate units:

(a) Segmentation through an address mapping unit consisting of a 20-bit adder and 16 x 32-bit base registers giving access to 16 segments of up to 65K bytes, variable in units of 16 bytes and with individual read/write/execute pre-trap/post-trap control.

(b) Fetch/decode/address computation performed by a microprogrammed unit with its own (integer) arithmetic facilities.

(c) Data operations unit giving arithmetic and logic operations on operands varying in length from 8 to 128 bits in units of 8 bits. This unit contains 4 x 128 bit accumulators and performs integer, real and complex variable-length arithmetic under microprogram control. It is asynchronous with unit (b) so that fetch and execution can be overlapped.

3.2 Instruction Set - Operands and Descriptors

MINIC S is organized as a single-accumulator machine with one-address operations between accumulator and memory and zero-address operations between accumulator and top-of-stack. The majority of code commonly required for expression-evaluation does not require stack operations, and generating mixed zero- and one-address instructions presents no compilation problems. The accumulator length was chosen to be longer than any likely to be required so that all problems of multi-length arithmetic could be avoided. 64-bit integers and the equivalent 16-digit reals seemed adequate and 128 bits were allowed for double-length integer results and packing two reals as one complex.

Bit-variable operand lengths were considered but rejected because of the housekeeping involved. Byte-variation in units of 8 bits was selected as including all normally used lengths. An 8-bit status byte can indicate the 16 possible lengths together with 16 possible data types which again seemed adequate.

Hardware considerations and market requirements make multi-length operands attractive. Then comes the question of how the housekeeping associated with the type/length status information is handled. Four possibilities have been allowed:

(a) Default status in the processor status word - this allows for zero overhead on the user of only one type/length (in line with objective (xiv)).

(b) Status in instruction - allowed for the odd case of one different operand type known at compile time.

(c) Status as tag (Feustel 1973) - necessary for the stack and allowed for generality in passing and using parameters of unknown type.

(d) Status in indirect address - as a "descriptor" (Iliffe 1968). This allows the efficient tagging of operands in arrays and is used for type-passing in calls by reference.

Having introduced a system of descriptors it has been natural to expand it to encompass other facilities. Automatic type/length coercion (both de-referencing and widening) is simple if the rule adopted is that result has greatest length and most complex type (COMPLEX > REAL > INTEGER > LOGICAL, with REFERENCE types always dereferenced and FUNCTION-CALL types always evaluated). Keeping data on a stack always tagged enables traps to be placed on it to ensure that the number of arguments passed to a routine is correct (types do not matter if automatic coercion is used).

3.3 Instruction Set - Load Segments, Addressing and Procedures

The order-code of MINIC S is such that the basic 1-address instruction family allows: 64 operations on 16 types of 16 lengths with 32 address modes into 16 segments of 65K bytes. Adding up the bits makes it clear that some structuring is necessary if the majority of instructions generated are not to encode into many bits. In practice the objective has been for the majority of instructions to encode into 16 bits with 32-bit and 48-bit forms for exceptional cases.

This compression is achieved by using two of the base registers to establish local segments on entry to a procedure. One is a dynamic area within a "stack" segment and the other a static area within a "heap" segment. The dynamic local segment is a conventional "stack frame" - the static local segment was incorporated originally for FORTRAN locals (those regarded as SAVEd (FORTREV 1973)) and gives an "own" area to a procedure. As well as allowing short-code addressing of static and dynamic locals, the mapping of these regions through segment registers allows for protected interprocess communication in that the surrounding segment can be write-disabled whilst the local regions are write enabled only to the procedure owning them.

The two local segments are set up automatically by microprogram on procedure calls and, for ALGOL, the appropriate display is also transferred to the base of the dynamic local segment - the ALGOL run-time environment is virtually identical to that of Gries (1971: Ch.8). Short-code addressing is also available to a global segment. This structure gives efficient access to FORTRAN COMMON, locals and parameters. Combined with the descriptor system it copes effectively with call by name, reference and value.

In hardware terms it was apparent that with semiconductor main memory the inclusion of special index registers gave little speed advantage, and the software opportunity was taken to do away with index register housekeeping by using a variety of indirect address computations. Any 16-bit operand in memory can be used as an indirect address with specification of pre- or post-modification and replacement by the modified value (modifier in instruction or on top-of-stack). In short-codes (16-bit instructions) this gives access to 512 "index locations" (three "displays" in the local and global segments). The indirect format is such that 48K bytes are accessible by a single-word. Multiple-word "indirect addresses" (called data controllers) contain segment and type/length information and allow a variety of structure specifications, e.g. dope vectors, which are decoded by microprogram when they are referenced.

3.4 Operating System and Segmentation

The architecture and facilities of MINIC S are such that it is able to support a wide variety of operating systems, e.g. with process-structuring and deferred segment-binding if required. Our initial operating system is aimed at small real-time applications and treats the machine in a simple manner. Exec holds tables of up to 63 shared and 192 unshared logical code segments and mediates all inter-segment procedure calls through these. Inter-segment references are bound at load time in these tables. No provision is made for similar access to data segments since scattered references are prevalent and hardware paging is not incorporated.

Each program sees an environment of several segments, including his own code, exec code, global and local data segments and exec tables. He has potential access through all 16 segment registers but some will be disabled to him for certain modes. Different jobs may share data segments (all code segments are intrinsically shareable).

4. Conclusions

We have attempted in this paper to illustrate the direction of current trends in minicomputer architecture in the light of our own experience in the design of MINIC S. It is probable that the architecture of MINIC S goes far beyond what many readers will regard as expected minicomputer facilities. However, there is great force in the argument that the logical extension of current practice through the incorporation of greater hardware content leads to such enhanced facilities. The real design problem is to keep the new facilities under control, to make them programmer/language orientated - not a diverse repertoire of special tricks but instead an integrated structure of operand types, operations, data structures and accessing mechanisms. If for nothing else we should take advantage of low-cost hardware to free ourselves from the burden of programming around hardware.

5. References

Bashkow, T.R., Sasson, A., Kronfeld, A. (1967) A System Design for a FORTRAN Machine, IEEE Trans. EC-16 485.

Bobrow, D.G., Wegbreit, B. (1973) A Model and Stack Implementation of Multiple Environments, CACM 6(10) 591.

Brooker, R.A. (1970) Influence of High-level Languages on Machine Design, Proc. IEE 117 1219.

Chesley, G.D., Smith, W.R. (1971) The Hardware-implemented High-level Machine Language for SYMBOL, AFIPS 38 SJCC 563.

Cleary, J.G. (1969) Process Handling on Burroughs B6500, Proc. 4th Australian Comp. Conf.

Denning, P.J. (1971) Third Generation Computer Systems, ACM Comp. Surveys 3(4) 175.

Dennis, J.B., Van Horn, E.C. (1966) Programming Semantics for Multiprogrammed Computations, CACM 9(3) 143.

Dennis, J.B. (1971) Third Generation Computer Systems, ACM Comp. Surveys 3(4) 175.

Dennis, J.B. (1973) Modularity in Advanced Course on Software Engineering (ed. Bauer, F.L.) Springer Lecture Notes in Econ. & Math. Syst. 81 128.

Feustel, E.A. (1973) On the Advantages of a Tagged Architecture, IEEE Trans. C-22(7) 644.

FORTREV (1973) Working Document of ANSC X3J3 (73-06-09) Bell Labs, Holmdel, N.J., USA.

Gries, D. (1971) Compiler Construction for Digital Computers Wiley.

Hansen, P.B. (1970) The Nucleus of a Multiprogramming System, CACM 13(4) 238.

Hassitt, A., Lageschulte, J.W., Lyon, L.E. (1973) Implementation of a High Level Language Machine, CACM 16(4) 199.

Horning, J.J., Randell, B. (1973) Process Structuring, ACM Comp. Surveys 5(1) 5.

Iliffe, J.K. (1968) Basic Machine Principles, Macdonald.

Lindsey, C.H. (1971) Making the Hardware Fit the Language, in Peck (1971).

Lindsey, C.H., van der Meulen, S.G. (1971) Informal Introduction to ALGOL 68, North-Holland.

Organick, E.I. (1972) The MULTICS System, M.I.T. Press.

Peck, J.E.L. (ed.) (1971) Algol 68 Implementation, North-Holland.

Richards, M. (1969) The BCPL Reference Manual, Computer Lab., Cambridge, U.K.

Rossiensky, J.P., Tixier, V. (1970) A Kernel Approach to System Programming: SAM, in Software Engineering, Tou, J. (ed.), Academic Press 205.

Watson, R.W. (1970) Timesharing System Design Concepts, McGraw Hill.

Weber, H. (1967) A Microprogrammed Implementation of EULER on IBM 360/30, CACM 9(9) 549.

Wichmann, B.A. (1973) ALGOL 60 Compilation and Assessment, Academic Press.

Wilkes, M.V. (1972) Time-Sharing Computer Systems: Second Edition Macdonald.

Wirth, N. (1968) PL360, A Programming Language for the 360 Computers, JACM 15(1) 37.
Wulf, W.S., Russell, D.B., Habermann, A.N. (1970) BLISS: A Language for Systems Programming, CACM 14(12) 780.

6. Acknowledgements

We would like to acknowledge the comments, criticisms and contributions of colleagues at Micro-Computer Systems, particularly David Hill, Mike Haines and Malcolm Herd, and also of Peter Madams, Department of Electrical Engineering, Essex University. It is also appropriate to make a general acknowledgement to the Department of Computer Science, University of Manchester, for their all-pervading influence on computer concepts and developments in Britain.