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

A HIGH-LEVEL MINICOMPUTER

F. K. WILLIAMSON*, B. R. GAINES, J. A. MAINE* and P. V. FACEY†

* *Micro-Computer Systems, Boundary Road, Working, Surrey, UK*
† *Man-Machine Systems Laboratory, Department of Electrical Engineering Science University of Essex, Colchester, Essex, UK*

This paper describes the design considerations underlying the development of an advanced minicomputer (MINIC-S) now in commercial production. Emphasis is placed on programming/compiler and operating-system requirements on the one hand and engineering feasibility on the other. Microprogramming/trapping enables all machines of the range to offer the identical architecture and range of facilities. Descriptor-based data organisation enables a very wide range of operand types and lengths to be made available. Relocation/protection and a separate minicomputer I-O processor enable real-time process-structured operating systems to be implemented efficiently.

1. INTRODUCTION

The original design brief for the machine described in this paper was a replacement for an 8-bit mini-computer widely used in machine tool control - the replacement to have enhanced arithmetic capabilities on wider operands together with wider address scope. The technology was to be conventional and the over-all price range to be in the centre of the mini-computer market. It very rapidly became apparent, however, in the early stages of design that such a conception of minicomputer architecture had been overtaken by events. On the one hand micro-computers on a few chips were attaining the power of conventional minis - on the other hand standard circuit technology and costs were such that one could go way beyond present minicomputer architectures whilst still maintaining the cost objectives. Indeed 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 under compilers.

1.1 Increased hardware content for improved system performance

There are two main areas where increased hardware content may be used to improve system performance and take advantage of recent developments in computer science:

(i) Operating systems - the advantages of process-structured operating systems have been extolled as major aids to real-time system development and software reliability [1-3], both of which are of major importance in typical minicomputer applications. However, the protection and communication mechanisms necessary are not implementable on conventional machines without a high time overhead which in turn limits the potential of process-structuring in its most important applications - most real-time systems are inherently short of time. Additional hardware, appropriately organised, can provide a suitable environment for efficient process-structured operating systems.

(ii) High-level languages - the advantages of "high-level" systems programming languages have also been extolled as a basis for further and more manageable programming than can be achieved in assembly language but with the same level of detailed run-time space/time control.

Again, since most minicomputer programming is at machine-level, the availability of such high-level replacements is of potentially great importance. However, the successful implementations of such languages have been on large machines with wide address scope and a fairly uniform structure [4-5]. Most minicomputers have a variety of non-interchange-able address mechanisms as the scope changes, and non-uniform structures designed to allow the programmer to use a variety of specific techniques, or "tricks", to cope with common situations. The "high-level" languages implementable on the current generation of minicomputers are either partially interpretive at run-time, or, in fully-compiled form, require a substantial run-time system to cope with the addressing, missing operations, missing address nodes, etc. "Systems-programming languages" without these defects reduce to syntax changes in a conventional assembler.

If one analyses this situation further then it leads inevitably to the conclusion that to support a "high-level systems programming language" which is truly a replacement for an assembler one needs a "high-level" machine.

2. REQUIREMENTS FOR A "HIGH-LEVEL" MACHINE

It should be clear from the previous discussion that by a high-level machine we do not mean one that is specifically designed to hardware interpret an exist-ent high-level language (although that may be a side-effect and is also of interest). One reason for this is that from two points of view all existing languages are inadequate: practically - no language offers the full range of data types and constructs suitable for the majority of current applications - a language-specific machine is inherently restricted and, commercially, the more powerful languages such as SNOBOL and ALGOL 68 are not widely accepted where-as FORTRAN, whilst ubiquitous, is not an adequate competitor to assembly language; technically - all languages have their conceptual flaws, on the one hand features which are little-used but cause inherent run-time problems, e.g. dynamic own arrays in ALGOL 60, and on the other hand, features which are inconsistently provided and lead to arbitrary restrictions perhaps related to the original implementation, e.g. the form of array subscripts in ASA FORTRAN IV.

2.1 Main objectives

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 widely used existing languages intended for general use e.g. PL1, ALGOL 68, FORTRAN V.

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

From these one may extract the following main objectives:

(i) 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.

(ii) 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 "procedure variables", and generally to allow programs to be "data-driven" wherever appropriate.

(iii) 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/loader) without change in the instruction set and without users being aware of the change.

(iv) Consistency and uniformity of treatment of all facilities - Wirth [4] (section 9) criticises certain features of the 360 architecture which make for both user and implementation difficulties in PL360 - it is the most common criticism of all machines that the hardware designer has made inessential differences in the treatment of various operations, inconsistencies in the treatment of condition codes, 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 to the engineer are of little importance) but which must be properly treated when they do. 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.

2.2 Auxiliary objectives

The objectives of the previous section have been programmer/language derived. There are also a number of objectives which stem from real-time operating system and commercial/manufacturing requirements.

(v) Decouple real-time requirements from user programs - the requirement to service an interrupt within a specified maximum time can be problematic if the instruction-set of the machine includes operations which can take a long time, e.g. long floating-point arithmetic, and which are arbitrarily available to all users at all times - it becomes essential to allow instructions to be interrupted within their execution.

We took a basic design requirement to be that real-time system programs could be run in a separate environment having priority over the normal, uncontrolled user environment - the systems programs handling real-time interaction could be written under constraints guaranteeing a maximum response time whilst the user programs were free to utilise the full range of facilities of the machine.

(vi) Flexible relocation/protection - many schemes (well-summarized in [6-7]) have been put forward for memory management and no one seems to have a clear advantage over all others. We were concerned to adopt a basic scheme that gave high-speed hardware relocation/protection that could be optimally utilised for speed in specialised dedicated systems but which could also be used to support general-purpose operating systems of the MULTICS [8] type without excessive overheads.

(vii) Microprogramming - 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 microprogram memory. This capability is an important machine feature in its own right and we wished to retain it in MINIC-S.

(viii) Extensibility through trapping - one objective in any machine design is lifetime - there is no sustainable argument for freedom from obsolescence of any current design - 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. A trapping system of this type was an inherent part of the design specification for MINIC-S.

(ix) Customer-image of the 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-time sharing, 16-bit only, etc., user has been a constant benchmark. Similarly we have not sought to impose 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.

3. THE DESIGN OF MINIC-S

In section 2 we have tried to express clearly the logical foundations for our design objectives. However, there is another side to the story which, in retrospect, one tends to omit but may be more important in practice. These design objectives did not arise as a whole and the original objectives were more mundane - the final objectives are a technical translation of basic marketing/production requirements. However, in practice 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 Operands and operations

The first minicomputers had as their operands the basic "words" of the memory, i.e. generally 12-bit or 16-bit operands. Primitive "half-word" operations aided character-handling, and a "link" or "carry" status bit aided multi-length arithmetic. In recent machines a range of operations on both 16-bit and 8-bit operands has become common together with hardware "options" and concomitant instruction sets for floating-point operands, generally in a 32-bit format. Alternatively, some machines have had a single operand size which is set by a status word to be 8/16/24/32 bits.

There are four basic criticisms one may make of many of these schemes:

(i) The different types of operand are not treated consistently, e.g. in the PDP11 only a limited range of operations are made available for 8-bit operands, and integer arithmetic is in a 2-address instruction format whilst floating-point arithmetic is in a 1-address instruction format. This makes it difficult for the compiler writer to treat the different operands as interchangeable data types. The assembly-language programmer finds that identical calculations have to be programmed in quite different ways for different types of operand.

(ii) The status information from comparisons, errors, etc. is rarely treated consistently for all types of operand.

(iii) Virtually no matter what operand lengths are made available in the ranges above there will be programs that require something longer, e.g. financial calculations often require very long integer arithmetic, or intermediate results in machine-tool calculations may exceed 32-bits. As soon as the basic hardware-recognised lengths are exceeded the software problems of multi-length operations return - a 32-bit divide does not readily extend to 40-bit divide at software level (on the other hand it may well do so at hardware level for certain organisations of the arithmetic unit) - a signed 16-bit multiplication does not readily extend to a signed 32-bit multiplication.

(iv) Where the additional operands and operations are made available through hardware "options" it is rare for the software utilities of a system without options to emulate them exactly, if at all. The addition of the option generally radically changes the machine architecture and requires extensive re-programming. It is not possible to trade speed and cost over a range of configurations without re-programming.

3.2 Descriptors

These considerations lead us to propose that MINIC-S should have a far wider range of operand types and lengths than had any previous machines and that these should be intrinsically available on all machines in the range. This did not in itself present major engineering problems - in particular, for the middle-range, micro-programmed machines, 8/16/24/32 arithmetic would be implemented by iteration of basic 8-bit operations and the only change required for greater lengths was extension of the accumulator scratch-pad registers and the iteration counters. There was no point in extending the operand length beyond useful bounds and we took 64-bit operands as a reasonable working limit. The actual maximum was set at 128 bits to go well beyond this for floating-point working and to allow double-length intermediate integer results to be handled as simple operands rather than in "upper" and "lower" parts.

The incremental unit in which operands could vary was difficult to determine - we were finally debating 1-bit or 8-bit units (bytes) and decided upon 8-bits. Either can be made self-consistent, e.g. the type/length of an operand may be encoded in one 8-bit unit in an 8-bit organisation. The advantage of 1-bit variability is that it places no constraints on data-packing - one disadvantage is the requirement for no data-unit boundaries in the main memory. This has been implemented in the B1700 [9] and may become generally feasible with low-overhead, low-cost in future memory technologies. The extra 3-bits required for bit-addressing rather than byte-addressing are probably less important since a bit-organised machine could readily have, for example, 19-bit rather than 16-bit pointers.

For the immediate future we concluded that the advantages of a bit-organised machine did not outweigh the practical problems of its design and implementation, and the marketing problems of such a radical innovation. However, bit-organisation is one probable facet of the shape of things to come and we have attempted to structure MINIC-S in such a way that program for the byte-organised machine could be directly transferred to a future bit-organised machine.

The range of types of operands was easier to determine - unsigned integer, signed integer, floating point (real) were obvious - complex as a basic operand was desirable - reference variables (pointers) and procedure calls, automatically decoded to fetch operands were also desirable. We were particularly concerned to leave the range of types open for future expansion - 16 types together with 16 lengths could be encoded in an 8-bit descriptor and seemed to give ample scope for expansion.

The major problem which arises when a range of operands is made available is how they are to be specified by the programmer. Two schemes have been adopted in previous minicomputers:

(a) Specify the operand type in the instruction - when there are few types this is attractive because it minimizes the amount of code generated - it does not allow type-independent functions to be written or types to vary dynamically at run-time - however, its main disadvantage is that there are few bits to spare in the typically 16-bit instruction-formats of minicomputers and using some for the type still further restricts address scope and is generally not possible consistently for all operations and address modes.

(b) Specify the operand type is a status word - this is advantageous if the machine is to be operated in major sections of code as 8-bit, 16-bit or floating-point, etc. - changes of type require housekeeping operations and mixed-mode operations may require more housekeeping code than actual program - to avoid these problems status-word manipulation has to be fully-integrated into the instruction set, particularly in subroutine/procedure calls.

There are two other possibilities which have been suggested previously only for much larger machines [10-11]:

(c) Specify the operand type in a pointer to it - this is particularly attractive for structured systems of operands, such as arrays, where a pointer to the base of the array may additionally contain a descriptor of the type of operands in it and bounds on its dimensions (such a pointer complex has itself been termed a "descriptor" for the array).

(d) Specify the operand type as part of the operand e.g. as the first byte of a multi-byte item - this is clearly attractive for the implementation of a language, like EULER [12] or ALGOL 68 (united modes) [13], which allow type to be dynamic at run time - it is also the optimum solution, however, in situations where mixed-mode expressions are common and it becomes cheaper to specify the operand type once with the operand rather than on each occasion it is used. Since a major extension of standard languages seems to be in the direction of more data types and lengths the use of descriptors with operands is likely to become of increasing interest.

In MINIC-S we took the logical step of allowing all four means of specifying the type of an operand, with the rule that the last specification found during the instruction decode took precedence so that, for example, the default status word could be over-ruled by access through a pointer containing a descriptor. Coercion of operands to a common type is automatic for the straightforward cases of mixed lengths, mixed integer/floating-point, etc., reference variables (pointers) which are followed to access the operand location, and function variables which cause code to be executed to access the operand location (trapping within an instruction and continuing its execution is a feature of the machine); other type clashes are resolved by traps to either monitor code for error messages or user-code for programmer-defined coercion.

3.3 Data controllers

One possible use of extra hardware content is to increase the number of index registers available. However, this leads to problems in its own right: a large environment that has to be retained and restored under process changes; a limited set of registers that have to be appropriately allocated without conflict. For MINIC-S we have adopted the view that semiconductor main memories have a short enough cycle time compared with the speed of appropriate CPU logic families for indirect addressing to be used as a basis of all address modification. Any 16-bit word in memory may be used as an "index register" and the instruction can specify that an offset (of bytes or of operands, in the accumulator, the instruction, or both) is added or subtracted, that the modified or unmodified value points to the operand, and that the modified value of the "register" replaces the previous value. The wide range of possible address modes is encoded into a variety of instruction formats so that, for example, pre-increment by one operand length, and other common forms of modification are available as single-length (16-bit) instructions. Similarly, a short-form address field enables the first 256 index locations to be utilised by single-length instructions. However, every logically possible address mode is available with every possible operation in double-length format - the instruction-set and order-code are fully decoupled as discussed earlier.

To further increase the power of the addressing system, multiple-length "index registers" are allowed (we use the term data controller for these more general accessing mechanisms). A normal indirect address is 15-bits and the top bit being set to 1 indicates a 15-bit data controller - this is made up of: 3-bit function code; 8-bit operand descriptor; and 4-bit segment number (see next section) - followed by one or more 16-bit words which may be addresses or bounds according to the function code. Apart from extending the address scope, the provision of multiple-length data controllers as generalized indirect-address and indexing mechanisms with some undefined function codes leaves the machine architecture open-ended in the variety of hardware/trap-interpreted data structures possible. The basic range of address modes and data controller functions makes the handling of most common structures, stacks, queues, rings etc., simple and automatic.

3.4 Input-output, protection and segmentation

Figure 1 shows the overall organisation of MINIC-S. Requirement (v) of section 2.2 has been achieved by using a MINIC-M processor to provide an independent I-O environment. The MINIC-S processor communicates with it only through the memory and an interrupt line. This has a number of ancillary advantages: all peripherals connect only to MINIC-M which already has the appropriate interfaces and software drivers; MINIC-S does not have to provide very short-delay interrupt-handling which simplifies its design (e.g. multi-precision real arithmetic does not have to be interruptable within an instruction); MINIC-M may be used as an exerciser and fault-diagnoser for MINIC-S; the MINIC-S processor becomes an optional upgrade to a MINIC-M configuration, providing segmentation and high-level language facilities; real-time software for MINIC-M, for example machine tool servos, can be used as part of the I-O package of a MINIC-S configuration.

MINIC-M has a simple memory protection "fence" restricting the accesses of non-privileged programs. An independent and far more elaborate scheme is provided for the MINIC-S processor to allow a ring-structure based on logical segments [14]. 16 active -segment registers provide a bounds/limit and status specification giving an active environment of up to 16 segments per process, each up to 64k bytes in length (in units of 16 8-bit bytes). The status specification allows each segment to be placed in one of 8 rings with three types of access right. A logical segmentation system allows for up to 128 shared program segments, together with up to a further 128 unshared logical segments per job. Logical segment boundaries, and hence also rings, may only be crossed by procedure calls. These are microprogrammed for normal transfers and trap to executive code only if the required segment is non-resident.

The procedure call mechanism supports both the ring-structured protection system and normal language requirements in a co-ordinated form. Each process has two local segments, a stack frame (dynamic local) and an own frame (static local) which are automatically relocated on a procedure call. These are actually full segments which are dynamically mapped within the other segments of a process. Since they are protected independently of the surrounding segment, it is possible, for example, for a call on a procedure in an outer ring to pass read-access to the whole of a segment but write-access to only part of it. In particular this overcomes the problems associated with outward calls and inward returns since the return information is not modifiable by a procedure called by an outward call. The two forms of local segment have natural interpretations in most languages as holding local variables whose scope does (own frame) or does not (stack frame) extend outside the procedure itself, and this enables the associated protection mechanisms to be utilised naturally within the language. Kernel or supervisory programs may be written in FORTRAN, ALGOL, or any other language which makes the procedure call available and gives access to the local segments.

This integration of segmentation, protection, and procedure calls into a single framework supported by hardware and microprogram enables an effective modular supervisory and program development system to be established on a minicomputer intended for real-time applications programmed in high-level languages. The desirability of maximally protected program development in multi-user real-time systems is obvious. The merits of structured programming in high-level languages may nowadays be taken as equally apparent. However, providing the appropriate support facilities on a minicomputer, and without unacceptable overheads, is a difficult problem which we feel the architecture of MINIC-S goes a long way towards solving.

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.

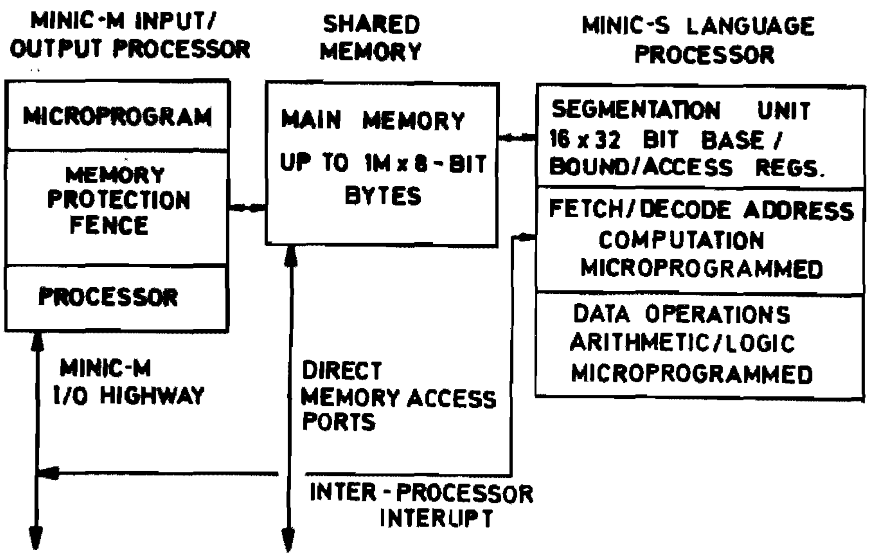
MINIC-M INPUT/ OUTPUT PROCESSOR
SHARED MEMORY
MINIC-S LANGUAGE PROCESSOR
MICROPROGRAM
MEMORY PROTECTION FENCE
PROCESSOR
MAIN MEMORY UP TO 1M x 8-BIT BYTES
SEGMENTATION UNIT 16 x 32 BIT BASE / BOUND/ACCESS REGS.
FETCH/DECODE ADDRESS COMPUTATION MICROPROGRAMMED
DATA OPERATIONS ARITHMETIC/LOGIC MICROPROGRAMMED
MINIC-M I/O HIGHWAY
DIRECT MEMORY ACCESS PORTS
INTER-PROCESSOR INTERUPT

Figure 1 MINIC-S Organisation

The final question may be - "What is a minicomputer?" - is there not some sense in which minicomputers are by-passing supposedly "larger" machines ? One answer is to go by price and memory-width - we have described a few-thousand pound, 16-bit machine. More profoundly, the computer industry, as ever, is in a state of flux and it is already apparent that manufacturers must seek their identity more in software, marketing and application support than in hardware characteristics. The machine we have described fairly reflects the current position in the continuum of minicomputer development.

REFERENCES

[1] P.B. Hansen, The nucleus of a multiprogramming system, C.A.C.M., vol. 13, 1970, 238.
[2] J.P. Rossiensky and V. Tixier, A kernel approach to system programming: SAM, in Software Engineering, J. Tou (ed.), New York: Academic Press, 1970, 205.
[3] J.J. Horning and B. Randell, Process structuring, A.C.M. Computing Surveys, vol. 5, 1973, 5.
[4] N. Wirth, PL 360, A programming language for the 360 computers, J.A.C.M., vol. 15, 1968, 37.
[5] W.S. Wulf, D.B. Russell and A.N. Habermann, BLISS: a language for systems programming, C.A.C.M., vol. 14, 1971, 780.
[6] M.V. Wilkes, Time-sharing computer systems: second edition, London: Macdonald, 1972.
[7] R.W. Watson, Timesharing system design concepts, New York: McGraw Hill, 1970.
[8] E.I. Organick, The multics system, Mass. M.I.T. Press, 1972.
[9] Burroughs B1700 Reference Manual, Detroit: Burroughs Corporation, 1972.
[10] J.G. Cleary, Process handling on Burroughs B6500, Proc. 4th Australian computer conference, Adelaide: Griffin Press, 1969, 231.
[11] J.K. Iliffe, Basic machine principles, London: Macdonald, 1968.
[12] N. Wirth and H. Weber, EULER: a generalization of ALGOL and its formal definition, C.A.C.M. vol. 9, 1966, 13(Pt. I) and 89(Pt. II).
[13] C.H. Lindsey and S.G. van der Meulen, Informal introduction to Algol 68, Amsterdam; North Holland, 1971.
[14] M.D. Shroeder and J.H. Saltzer, A hardware architecture for implementing protection rings, C.A.C.M. vol. 15, 1972, 157.