SPRAWOZDANIA
INSTYTUTU INFORMATYKI
UNIWERSYTETU WARSZAWSKIEGO

Nr 130

P. Findeisen, P. Gburzynski,
E. Jezierska-Ziemkiewicz, A. Ziemkiewicz

A PROPOSAL FOR A MINICOMPUTER ARCHITECTURE

IInf UW REPORTS
INSTITUTE OF INFORMATICS, WARSAW UNIVERSITY

S P R A W O Z D A N I A

I N S T Y T U T U I N F O R M A T Y K I U. W.

IInf UW Reports

Nr 130

P.Findeisen, P.Gburzynski, E.Jezierska-Ziemkiewicz,
A.Ziemkiewicz

A PROPOSAL FOR A MINICOMPUTER ARCHITECTURE

W Y D A W N I C T W A
U N I W E R S Y T E T U W A R S Z A W S K I E G O
1 9 8 3

PL ISSN 0239 - 4359
Sprawozdania Instytutu Informatyki
Uniwersytetu Warszawskiego [IInf UW Reports].

Pracę zgłosił dnia 8 listopada 1983
doc. Antoni Kreczmar

Adresy Autorów:

P.Findeisen, P.Gburzyński:
Instytut Informatyki UW
00-901 Warszawa, PKiN p. 850

E.Jezierska-Ziemkiewicz, A. Ziemkiewicz:
Instytut Maszyn Matematycznych
02-078 Warszawa, ul. Krzywickiego 34

Wydawnictwa Uniwersytetu Warszawskiego

Wydanie I. Nakład 300 egz. Nr GP.II-441/2961/82. Cena 10 zł.
Wykonano w Zakładzie Małej Poligrafii UW. Zam 3/84.

LIST OF CONTENTS

1. INTRODUCTION

The proposal for a general purpose computer system architecture presented in this paper is not a revolutionary one. It rather well sticks to the von Neuman's ideology, with its occasional 'unorthodox' deviations being of a moderate character. We think however that there are at least a few genuine attributes of the proposed architecture which may be of some interest to the reader. They mainly reflect our opinion on what are the needs of an up-to-date system programmer. It seems that the existing computing machinery still fails to meet a substantial part of them.

The historical backgrounds of this design must be dated since 1977 when the idea of Loglan (c.f. [2]) as a base machine language was born in IIUW. The research on Loglan, being a strongly object-oriented general purpose programming language, covered several software and hardware problems, e.g. related to memory allocation and protection techniques. The solutions to those problems (some of them turned out to be quite non-trivial) produced a number of postulates for hardware designers, which contributed to the notion of a Loglan machine, as viewed by the participants of the project. Some of those postulates (particularly those of a more general nature) have been adopted in the presented proposal.

In 1981 a group of IMM research stuff directed by Ela Jezierska organized a number of meetings and discussions directed at the formulation of the fundamental ideas of a new polish minicomputer - the so called Solid (c.f. [1]). The proposed modern architecture was strongly oriented towards the C programming language, which was mainly reflected in the conception of a C-code (its design has never been finished) to be used as a basis of the user-extendable machine instruction set. A number of the postulated attributes of the Solid (e.g. functional specialization of the processors, interfaces, i/o ideology, e.t.c) have found their close relatives in the proposal. The idea of devoting the computer to C has been given up however.

In early 1983 a project team organized by prof. Andrzej Janicki started to work on the basic design of a new minicomputer (c.f. [4]). The design was ordered by the Foreign Trade Enterprise Metronex with intention to be used as a basis for an international cooperation. The contents of this paper may be viewed as the contribution of its authors to the project.

The authors do not mean to conceal the fact that however genuine, the design remains under some influence of the Vax11/780 architecture. This may be observed e.g. in the shape of the addressins modes. One of our intentions was to remove some disadvantages of this architecture.

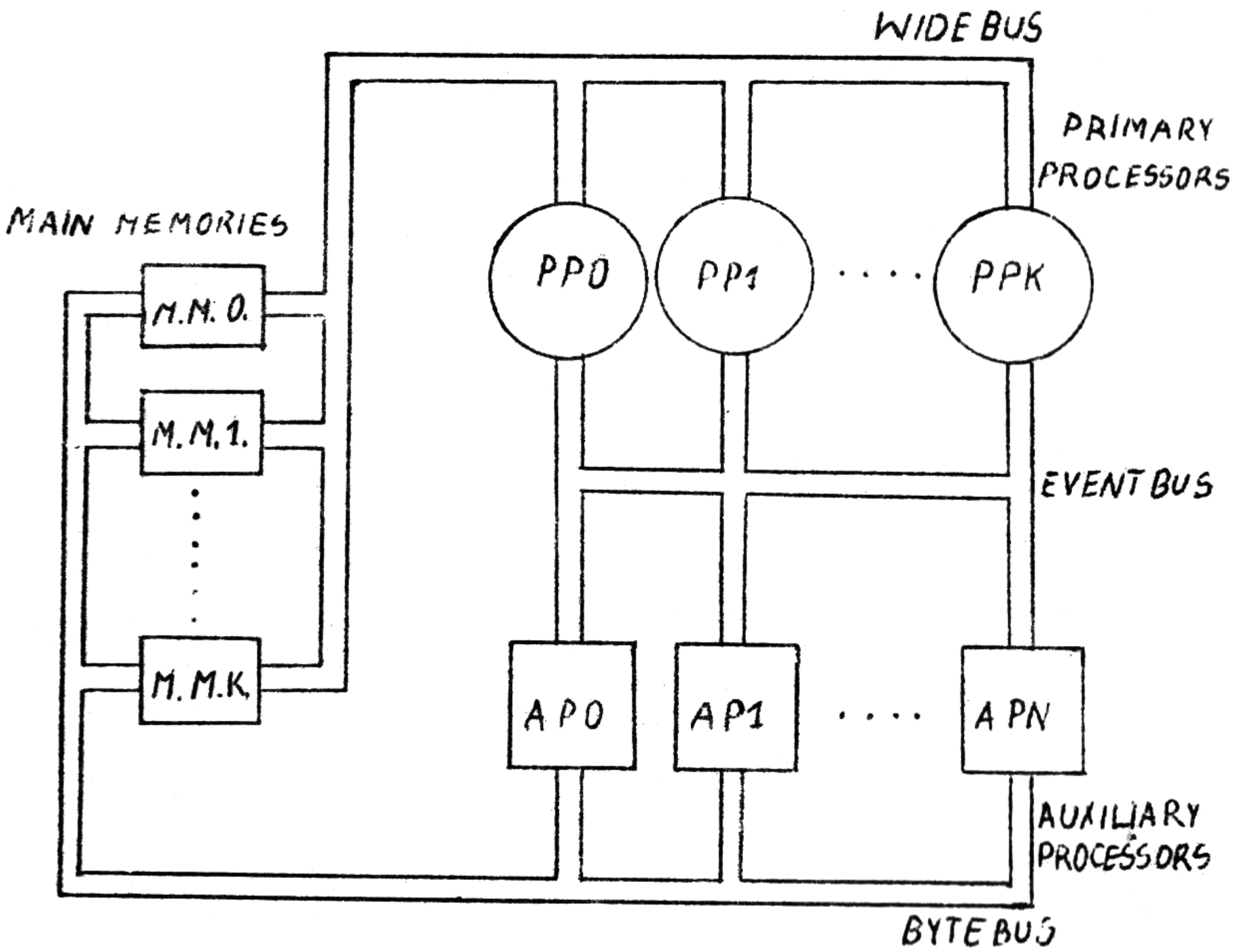
At least they appeared disadvantages in our opinion. Another computer architecture that rather significantly influenced the designers' work was the Mera-400 of our native origins. Although not all its elements can be now compared to the most recent achievements in computer technology, some of them certainly deserve to be further developed and applied.

A susbstantial contribution to the design from a number of our friends and coleagues - members of the project team - is to be highly appreciated. In particular we feel oblised to mention Jarek Deminet, Jerzy Dzoga, Andrzej Karczmarewicz, Antek Kreczmar and Andrzej Litwiniuk. The design would have never reached its present level of completness if not for the inestimable ideas of these gentlemen.

2. AN OVERVIEW OF THE SYSTEM ARCHITECTURE

2.1. Internal System

In the designed system two essential parts can be distinguished. The so called Internal System is composed of the Primary and Auxiliary Processors, main memory and interfaces. External System is constituted of mass memory, peripheral devices and communication lines.



Structure of the system kernel

The spatial structure of the Internal System is concentrated. However, due to its modular construction, it is susceptible for a flexible modification and, in particular, an incremental expansion.

2.1.1. Memories

The concept of memory organization in this design is derived directly from the solutions applied in Mera-400.

The main memory is logically divided into pages of fixed length (4KB). The pages can be organized into address spaces whose number is limited to 256. The zeroth address space is dedicated for the operating system. A single address space can consist of up to 2048 pages, i.e. its size is limited to 8 MB.

With each primary processor there is associated a "private" memory page. The page can be referenced as the first 4096 bytes of address space zero. Thus addresses less than 4096 (in address space zero) determine distinct bytes for distinct primary processors. The access to all the private memory pages in the system is provided only for the System Monitor.

In the system there occur memories of different types and of different access principles. The Main Memory (MM) is accessible to all active system elements, e.g. for all processors. A subset of Main Memory, corelated directly with the processors, creates the space of Local Memories.

To adjust the speed of the processors to that of the other elements of the system (especially memory), some fast, low capacity memories buffering the information transmitted from/to the processors are employed.

The Main Memory

Memory references are performed via virtual addresses. The main memory of the system creates a virtually addressable space of 256*8 MB. Up to 32 MB of physical memory may be implemented. The capacity of a physical memory module is estimated as 0.5 MB. Depending on the technology available the capacity may be enhanced to 1 MB. The physical modules are constructed of semi-conductor elements with as large capacities as reasonable (e.g. 64 Kb). Whatever their capacity, it is assumed that the elements with the decreased power consumption in power-down state will be used.

The main memory of the minicomputer will base on solid-state chips (bipolar chips are suggested) with the cycle of about 300 ns, and chip capacity not less than 64 Kb. There is a possibility of gradual switch to memory boards with greater capacities.

The Local Memory

The Local Memory of the system consists of the set of distinguished pages of the Main Memory. Local Memory cooperates directly with a processor and with one of the interface busses (Widebus or Bytebus) of the internal system.

The capacity of each particular Local Memory is 4 KB. Local Memory can be referenced by the processor being its "owner". For the diagnostic or bootstrap purposes an access to Local Memories from the interface is designed, but it is performed on special principles.

Buffer Memories

Each primary processor is equipped with very fast memory of small capacity which buffers the stream of instructions and data on the way from/to the Main Memory. The Auxiliary Processors are equipped with buffer memories to obtain the continuity of information transmission and to make the error correction in the transmission (e.g. in Mass Memory Processor) feasible.

Internal Memories of the Processors

The processors of the system have internal memory resources. They are microprogram and program control memories of ROM and partly RAM type (for dynamic extension or redefinition of processors' functions) and auxiliary RAM type memories for storing internal variables, working buffers and stacks organizing the processors' work.

2.1.2. Processors

The number of primary processors, which are fundamental processing units of the system, is limited by 8. This restriction comes from the estimated throughput of the main interface (the so called Widebus).

Increasing the number of primary processors above this limit would not result in a significant improvement of the system performance.

The number of auxiliary processors is not subject to such severe restrictions as in the case of primary processors (auxiliary processors would cause much less interface contention). It is only confined by the total number of all processors in the system which cannot exceed 16. The particular configuration of processors in any given system is composed according to the presummed and user-specified assortment of applications.

2.1.3 Interfaces

The system organization constrains very high requirements for the interface between primary processors and main memory. It is assumed that the processors will be able to execute about 700 thousands instructions per second with each instruction operating on 6 bytes on average. Taking into account also the peak transfer rate for disks, which for nowadays disks is 1.2 MB/s we arrive at the conclusion that the throughput of the interface must be at least of the value of 10 MB/s.

Apart from the Widebus, which is the fastest and widest internal interface of the minicomputer, there are two more "buses" dedicated to service less critical transfers. The so called "Bytebus" is used to communicate the auxiliary processors with main memory, and the Eventbus to communicate the processors between each other. All the system interfaces work asynchronously with confirmations (handshake).

The interface is equipped with a special bit **IB** (Interlock Bit) which may be set and cleared with the rule of exclusivity. This mechanism allows for mutual time exclusion of selected instructions and operations. The processor executing such an instruction (for example inserting an element into a queue) must set IB for the time of its execution. However, if IB was already set then the instruction execution is delayed until the processor is able to set the bit. The instructions that set IB for the time of their execution are called **indivisible instructions.**

The internal system interfaces: Widebus, Bytebus and Eventbus have no possibility of extension in the sense of space. Because of the requirements for the transmission speed the interfaces are assumed to be limited in length. The space extension of the system is done with the help of external interfaces.

The Widebus interface transmits: data of the width up to 8 bytes

with the possibility of double transmissions (2*8 bytes), 31-bit virtual addresses, control, checking and handshake signals. The interface data lines are two-way, the other lines are one-way.

Three-state elements are suggested to be used for the implementation of the interface. The circuits controlling the access to the interface should be build in the bi-polar technology of very high speed.

The working cycle of Widebus is 750 ns (1350 ns for double transmissions) on average. In this way the throughput rate of about 10 MB/s is achieved.

Bytebus facilitates the Auxiliary Processors' access to main memory. The technology is similar to that applied in the Widebus case. The assumed throughput rate of Bytebus amounts in about 2.5 MB/s.

The Eventbus interface has the width of information line up to 40 bits, 24 addressing lines, 4 lines for identifying both the sender and the receiver of the message, control and handshake lines and service requirement lines. All lines of the Eventbus are two-way. The Eventbus transmission rate has no significant influence on the resulting system performance.

2.2. External System

The minicomputer communicates with its devices and surroundings through separated interfaces conforming to the commonly known and accepted standards. Among those interfaces are:

- interfaces to mass memory devices (the suggested standard capacity of a single disk unit should not be less than 30Mb);
- interfaces to peripheral devices and connections with intelligent terminals;
- object connections (Camac, Inteldigit PI, IEEE-488, Proway);
- serial transfer interconnections, with several speeds and protocols;
- interfaces to specialized local nets (i.e. Nokia, Ethernet and others);
- interfaces to associated computer systems (Unibus, Q-bus, Multibus and others).

All i/o transactions in the minicomputer are performed or assisted by the Auxiliary Processors. Any primary processor can initiate such a transaction, examine its status and/or terminate it.

Whenever an i/o operation is initiated on a specified auxiliary processor the invoking primary processor supplies a 40-bit argument which specifies the type and parameters of the requested action. In particular, this argument may include an address of a table in memory where some more complex and sophisticated information may be included. In general, the format and expected contents of the arguments of an i/o operation may depend on the kind of the auxiliary processor involved as well as on the operation type. After the desired action is finished the auxiliary processor sends the pertinent specification along with an interrupt to the system. The basic concepts of the interrupt mechanism are presented in 3.1.5.

3. THE PRIMARY PROCESSOR

3.1. The conception

3.1.1. Functional Attributes of the Primary Processor

The primary processor is equipped with 16 general purpose 40-bit registers denoted **R0-R15** which are homogenous (have equal rights). Some slight and immaterial exceptions from this statement are delineated in 3.1.4. (addressing modes). The instruction counter (IC) **does not count** to the pool of those registers (which is a difference to PDP-11 and Vax-11).

Some of the machine instructions (e.g. double precision floating point arithmetic) may operate on register pairs. Such a pair is always constituted of two consecutive registers (note that R0 is the successor of R15).

There is a special one-bit mark M associated with each register. Whenever a quantity of data is written into a register its mark becomes cleared. There is however one exception from this rule. A special instruction (c.f. 3.2.) can be used to load something into the specified register and light its mark. The marks reflect a part of the hardware support for high-level object-oriented languages.

Apart from marks there exist four other one-bit flags associated with the processor rather than particular registers. Those flags, denoted by **Z, N, V, C**, correspond to the analogous flags of PDP-11 or Vax-11 and usually indicate several arithmetic events.

Among the special registers of the primary processor the most important is the so called **Status Register**. It is composed of a number of fields which are listed below (bit-length of each field is specified in parentheses):

IC (24) - Instruction Counter which contains the address of the next instruction to be executed. The instruction is fetched from that address according to the so called Code Space Index.

CSI (8) - Code Space Index - this field of the status register specifies the current memory space for program code. Whenever CSI contains zero (i.e. the zero code space is assumed) the privilesed instructions are enabled and legal.

DSI (8) - Data Space Index - this field contains the index of the current memory space for program data.

SSI (8) - Secondary Data Space Index. The user may simultaneously access two different data spaces (see below). This field specifies the index of the second data space.

WREN (1) - Write to Secondary Data Space Enable. When this bit is set to one the secondary data space may be written into. Otherwise an attempt to write anything into this space would cause an interrupt.

PP (3) - Processor Priority - the contents of this field specify the level of interrupts which are allowed to be accepted and serviced in a given moment.

T (1) - Trap. If this bit is set to one an internal interrupt (trap) is triggerred after each execution of a instruction.

(3) Reserved.

Marks (16) - The M-flags (marks) associated with the general purpose processor registers.

CC (4) - Condition Codes - the indices of arithmetic events (ZNVC).

IOV (1) - Integer Overflow Trap Enable. Setting this bit to one causes that whenever an overflow happens during execution of an integer arithmetic operation it will result in an internal interrupt (trap).

FUN (1) - Floating Overflow Trap Enable - if this bit is set to one a floating underflow triggers an internal interrupt. Otherwise the execution continues with the zero result.

(2) Reserved.

The last 24 bits of the Status Register are logically encapsulated into the so called **Program Status Register** (PSR) which is directly accessible to the user. There exist (c.f. 3.2.) non-privileged instructions that can read the PSR as well as change its contents.

The State Register of a primary processor includes, among others, indices of code and data memory spaces. The instructions to be executed (the program code) are fetched from the current code space. Most of arguments for the instructions are located in data spaces (c.f. 3.1.4.).

All the primary processors of a given system configuration are indifferent and have equal rights and privileges in accessing the global resources of the system. Some logical concepts of the solutions relevant to the Primary Processor are implied by the fact of the potential coexistence of a non-trivial number of its cooperating copies.

In particular, each primary processor is equipped with a private clock, i.e. a separated 24-bit register which can be switched between the following two states:

- clock stopped - the contents of the register are freezed and they do not change in time;
- clock running - the value in the register is decremented by one each fixed interval of time (e.g. 0.1 ms); when zero is reached a special interrupt is triggerred.

The processor clock may be started, stopped, read and preset by suitable instructions. Besides the presented private timers of primary processors there exists a global day-time clock combined with a calendar.

3.1.2. Data Representation

After getting acquainted with the well known architectural aspects of the existing 16 or 32-bit minicomputers (Pdp-11, Vax-11), and after having listened to the opinions of their miscellaneous users, the authors came to the conclusion that these solutions are tainted with the folowing two basic defects:

First, the 32-bit address size certainly adds to a significant waste of memory space. This size was calculated on the basis of estimations which anticipated that demanded capacity of the minicomputer address field grows by three bits each two years. It seems, however, that one should not put too much faith in such predictions. After the significant extensions of the long-lived 16-bit address size have come into reality,

those demands become seriously accommodated. The authors are profoundly convinced that 16 Megabytes of a virtual address space would be a satisfying quantum for a long time.

Second, the floating point arithmetic based on 32-bit representations (with 64 bits in double precision) is highly criticized by many numerical analysts. Most of complaints are due to the small exponent size which imposes too severe restrictions on the range of represented numbers. Any attempts to extend the exponent by borrowing a few bits from mantissa result in an intolerable lost in precision, being anyway too poor for many typical applications.

In a determined pursuit to ameliorate the disadvantages mentioned above the authors gave in the principle that the number of bits occupied by an elementary data quantity should be a power of two (e.g. 8, 16, 32, 64 - for Vax-11). Nevertheless, the notion of byte has been fully retained together with all its consequences. It means that each address in memory is an address of a byte. Any addressable object must start from a byte boundary and be composed of an entire number of bytes. The notion of word is dispensable: in practice there is no such thing.

The assumed sizes of the elementary objects resulted from thorough and deep considerations around the following premises:

> It frequently happens that the 16-bit size of an integer number appears too short. On the other hand, it feels that relatively slight extension of this size would substantially improve the situation without need to recall to a multiple precision. This originates the idea of the 24-bit integer format which is advised to be used in system as well as standard software.
>
> In authors' opinion the 40-bit size of reals is a well balanced compromise between the defective 32-bit floating point formats and double precision (being usually a needless expense). The experts in Numerical Analysis who consulted this proposal asserted that it well covers all typical applications of the floating point arithmetic.
>
> The 40-bit format of integer numbers is mainly constrained by the ANSI requirements for Fortran-4 and Fortran-77. These requirements specify that the single precision floating point number should be of the size of integer. The need to obey (at least optionally) this

specification leads usually to some waste - wherever reals and integers are of different sizes (e.g. Pdp-11). One of the non-neglisible intentions of the authors was to facilitate an effective and compatible implementation of Fortran. This motivates the 40-bit integer format which is also recommended as the standard format of integers for HLL compilers.

Under the above premises the sizes of particular elementary objects are looking as follows:

byte	- 8 bits,
address or short integer	- 24 bits,
integer	- 40 bits,
floating point number	- 40 bits,
double floating point number	- 80 bits.

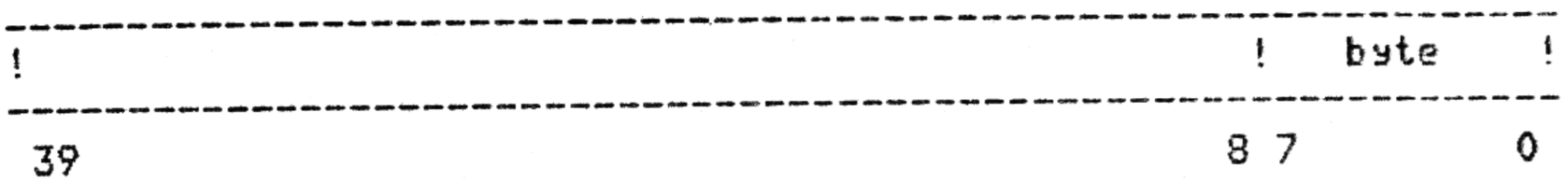
The objects belonging to the above classes will be denoted further by B, A, N, F, D respectively. In some very special cases (c.f. 3.2.) there appears a pair of 40-bit integer numbers with certain properties of a singular object. Such an object will be denoted by L. Occasionally there may happen an addressable data quantity which does not fall into any of the presented categories. All such exceptional cases, which are rather few, are detailed in 3.2. (instructions list).

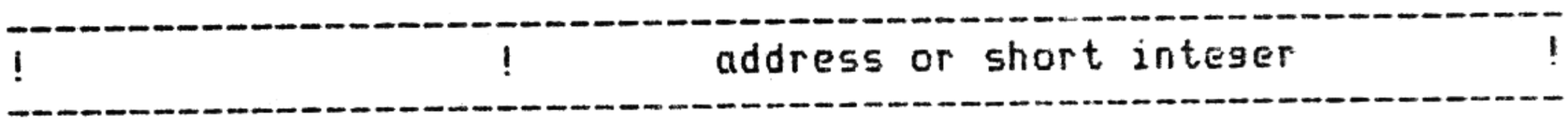
For an addressable object, its address is the address of its last byte. Consequently it is the least significant byte (which notion is obvious e.g. for an integer number). The order of bytes is thus the same as in Pdp-11 and Vax-11.

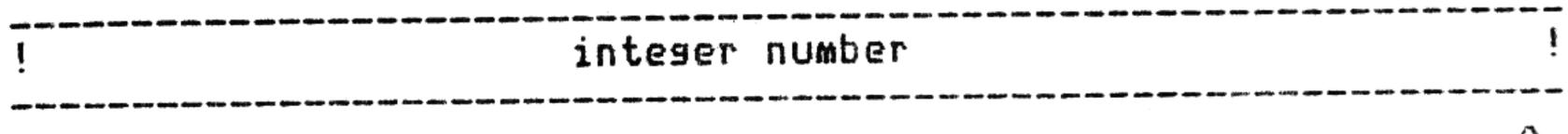
Despite obvious differences in object types and lengths there is a complete arithmetic (addition, subtraction, multiplication and division) provided for each category. Whenever sign problems are relevant one should remember that addresses are always positive, i.e. the leftmost bit of an address is significant.

Below is presented the arrangement of the particular types of data in the general purpose registers of the primary processor. Register bits are numbered from zero: the lower numbers correspond to the less significant bit positions. The presented schemas also reflect the arrangement of data in memory: the lower memory locations (lower

addresses) correspond to the less significant bytes of the structure.

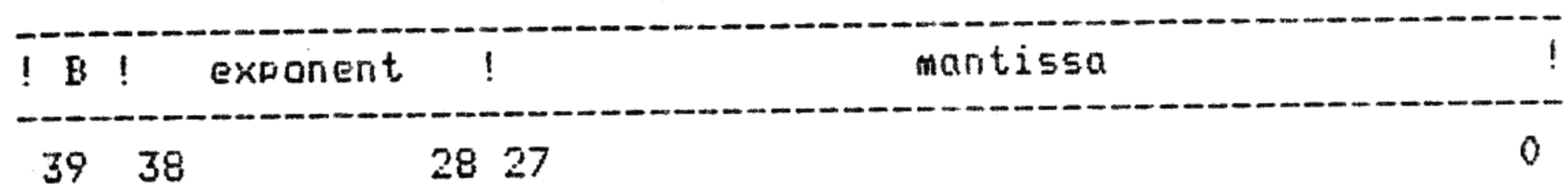






Floating point numbers are represented by pairs: exponent, mantissa. The encoded number reads as:

mantissa * 2 ** exponent



The two-complement binary exponent of a floating point number is biased by 1024 (decimal). Thus, encoded values of exponents are always viewed as non-negative integer numbers. For example: exponent 0 is encoded as 1024, i.e. 400 hexadecimal; analogically, the lowest possible pattern, i.e. 0, represents the lowest acceptable exponent which is -1024.

The mantissa also conforms to the standard two-complement notation and, when normalized and different from zero, it represents a fraction which falls into [1/2,1) or [-1,-1/2). The normalized zero value is represented by five consecutive zero bytes.

A double precision floating point number is encoded in two consecutive registers (or in 10 consecutive bytes in memory). The exponent together with the most significant part of the mantissa (the single precision part) are located in the register with the lower number (modulo 16). The second register includes the 40-bit mantissa extension.

B	exponent	mantissa
39	38 28	27 0

mantissa extention
79 40

The data formats presented above are retained when the encoded entities are located in memory rather than resisters. One should remeber, however, that the address of a data item addresses its last byte, i.e. it corresponds to positions 0-7 of a register.

3.1.3. Structure of the instructions and arguments

Each machine instruction, whatever the number and types of its arguments, occupies an entire number of bytes and starts with an 8-bit operation code (opcode). In some particular cases the opcode may be continued on the next four bits.

The collection of all machine instructions may be classified with respect to the expected number of arguments and their acceptable formats. The notion of the so called **standard argument** plays an important part in classification of the instructions. The standard argument is described by the so called **leading byte** which is usually followed by a further specification. The format of the leading byte is presented below:

am	rd
7 3	0

where:

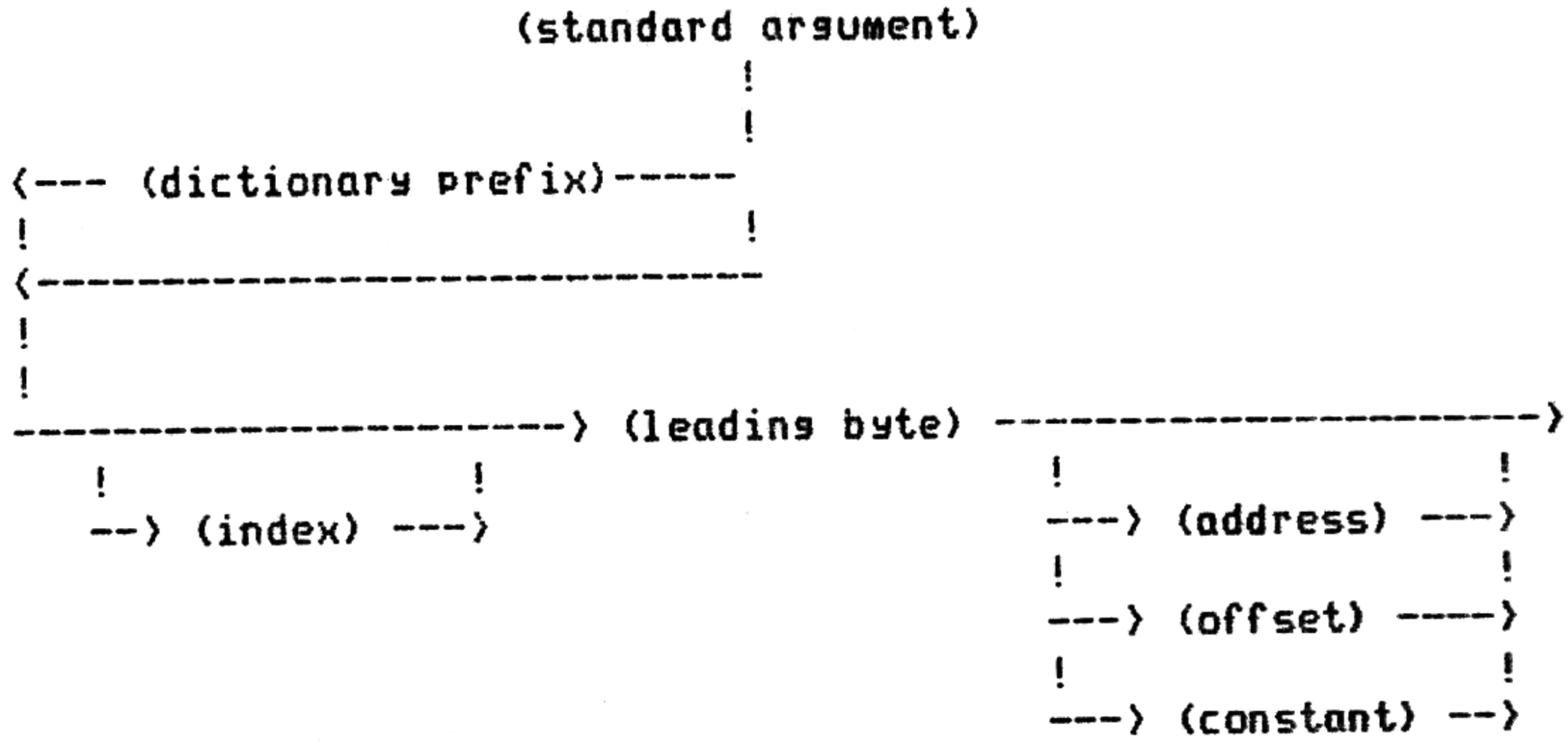
am - addressing mode ordinal (see below);

rd - number of a general purpose register; in a singular exceptional case (c.f. 3.1.4.) rd is an extension of am.

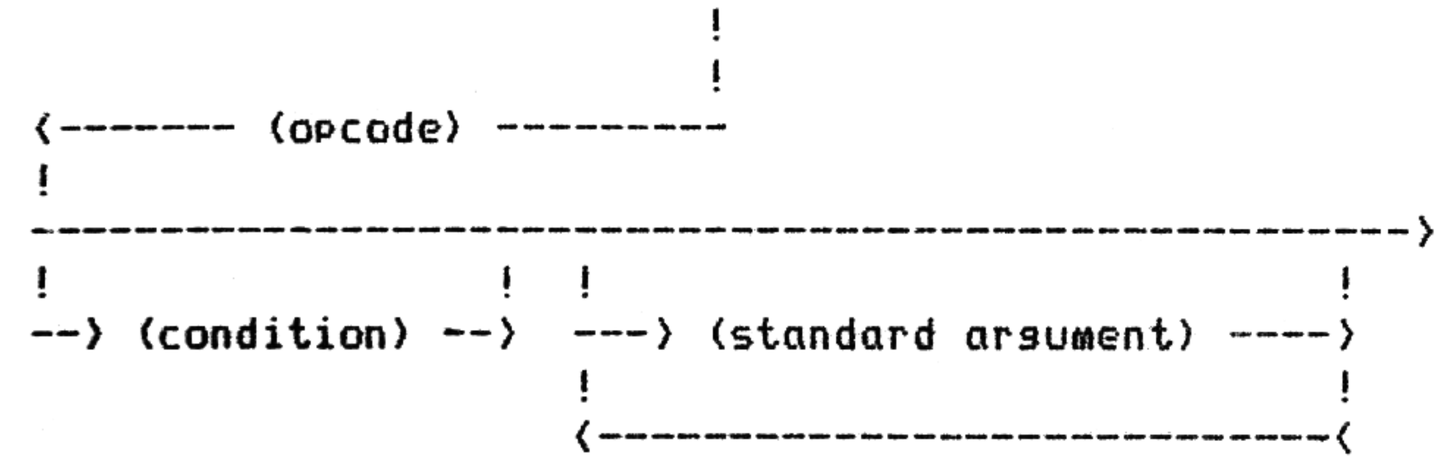
For some addressing modes the leading byte carries the complete specification of the argument. For the other modes it is followed by an address, offset (8 or 24 bits) or an immediate constant (8, 24 or 40 bits) whose length depends on the opcode.

Regardless of the general principle expressed above there are three values of am which are treated in a special way. One of those special values announces the **index.** The leading byte of such a kind may (once) precede any standard argument. The second reserved value of am denotes the so called **dictionary prefix** which, as in the previous case, may once precede any standard argument. An index may be also preceded by a dictionary prefix; a dictionary prefix, however, cannot be preceded by an index. The last of the non-standard values of am represents the so called **condition.** The only place where such a kind of leading byte may occur is immediately after the opcode of an istruction (before arguments). The semantics of index, dictionary prefix and condition is described in 3.1.4.

The general format of the standard argument is presented on the following diagram:

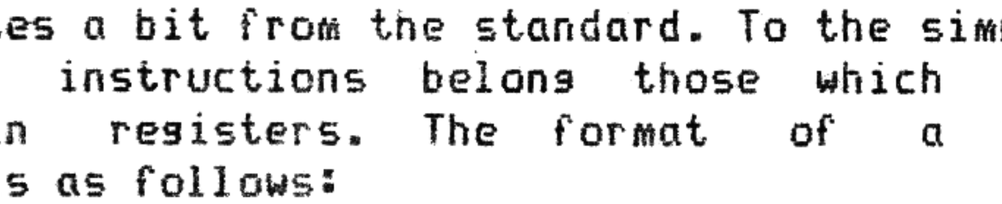


The most important, from the viewpoint of the system architecture, instruction class comprises the instructions whose all arguments conform to the standard format. Below is presented the syntax diagram of such an instruction:

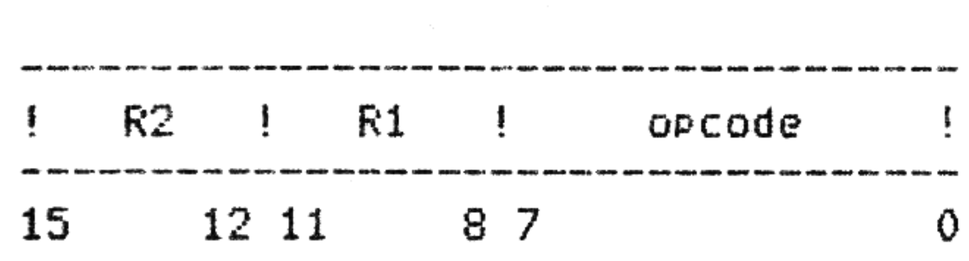


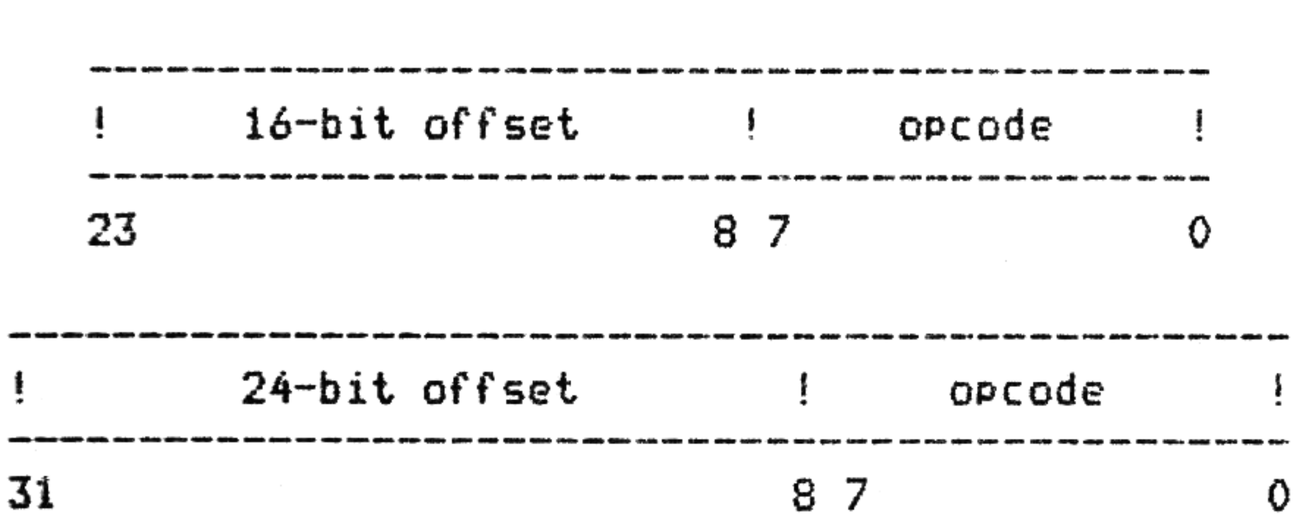
The addressing modes of the Vax-11, which in some sense inspired our solutions, have been recently subject to a serious criticism. The main reason for objections is in relatively severe code redundancy caused by the uniform and wasteful format of arguments. For instance, to use a byte constant as an argument it always requires an extra byte. Analogically, in the situation when the arguments of an instruction are exclusively located in registers (which is not a seldom case) it takes a full byte to specify each of them.

In order to eliminate such disadvantages and to reduce the code length in many typical cases we have designed some instructions, whose syntax deviates a bit from the standard. To the simplest class of such (abbreviated) instructions belong those which only accept the arguments in registers. The format of a "register-register" instruction is as follows:

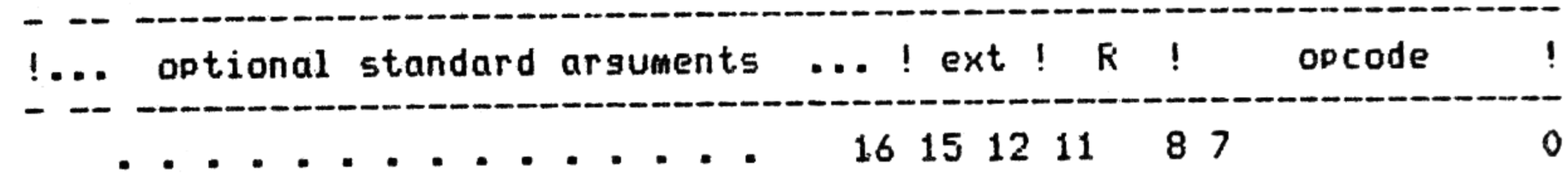


Another class of abbreviated instructions includes some jumps. Within this class the following formats are accepted:





The last interesting class collects instructions with first argument in register. The format of such instructions is presented below:



Let us note that in the last presented format the opcode is extended by four additional bits (field ext).

The standard argument, depending on the operation code, represents one of the objects B, A, N, F, D, L, which usually (if it is not an immediate constant) needs be fetched from a data memory space. Some exceptions from this rule are in standard jumps whose target arguments represent locations in the code space. The offset argument occurring in a jump instruction specifies the so called **displacement**, i.e. the difference between the target memory address and the current value of IC.

3.1.4. Addressing Modes

Before presenting the addressing modes of the new minicomputer let us say a few words about motivations which directly influenced our decisions and finally brought us to the presented solutions.

The most recent world-wide trends in software are strongly attracted in direction of unified programming emvironments based on object-oriented high level programming languages (e.g. Ada, Loglan, Smalltalk and many others). Among these base languages especially interesting and promising are those which also provide for parallel computations. It seems obvious that, whatever the future of Ada and Loglan, the efforts towards developement of the programming environments of such a kind will not be stopped soon. The presented basic design remains under strong influence of this conviction. In particular, we do not attach too much importance to the notion of a stack pointer (Pdp-11, Vax-11) located in a dedicated general purpose processor register, as this notion has a much limited range of applications when used in an object-oriented parallel computing environment.

In any given moment of its activity a primary processor can access three address spaces described by the contents of three address space registers:

- code space (CSI),
- data space (DSI),
- secondary data space (SSI).

The contents of the address space registers need not be different; in particular, data and code of a program may occupy one common address space.

Data addresses ended with zero (on the most significant bit position) reference the primary data space. Analosically, any data address whose most significant position contains one references the secondary data space (modulo 2**23). Thus the maximum size of a data space becomes 8 megabytes. Both data spaces, however, may be mersed into a singular continuous area of 16Mb.

The size of the code space is strictly limited to 8Mb. Whenever an attempt is made to reference an address with its leftmost bit set to one, it is treated as an error which causes an internal interrupt.

Two of the addressing modes presented below (5, 6) may seem rather intricate and unnatural to the user of a traditional computing system. These two modes (together with Marks) considerably help to implement advanced access methods to generally viewed structural objects organized in dictionaries. Being equipped with these addressing modes the processor architecture provides for an effective and convenient solution to the extremely hard goal which is an organization of a fast and safe access to the objects in the situation when:

- objects with variable sizes are dynamically created and removed which results in a need for a periodic or continuous (on-line) compactification and garbage collection;

- the dictionary as well as a single object may be simultaneously accessed from many different processors which action must be safely interlocked without causing too much resion contention.

List of Addressing Modes

0-2 **short literal.**
The argument is an immediate constant directly described by the contents of the leading byte. For argument types B, N and A this constant is interpreted as a nonnegative integer number from range 0-47. For a floating point argument (single and double precision) the constant in question represents the so called **short floating point constant** whose value is fetched from a ROM table indexed by the integer value of the literal.

3 ?R? **condition.**
This kind of argument may exclusively appear immediately following the operation code. If the contents of the least significant byte of register R are nonzero the instruction is skipped (but its arguments are decoded and interpreted). In the opposite case the instruction is executed as if the condition mode would not occur.

4 [R] **index.**
This addressing mode may only be used to prefix other modes rather than directly describe an argument. The address indicated by the prefixed mode is treated as an address of a table in memory which then is indexed by the value in register R to obtain the desired data.

5 **dictionary prefix.**
If the contents of register R are nonzero then the following action is performed: The lower (less significant) 3 bytes of register R are assumed to represent the so called **Virtual** (intermediate) **Object Address** (VOA). The fourth and fifth bytes from this address are then compared to the upper (more significant) two bytes of register R. If the comparison produces a negative result or if register R includes zero (all bits cleared), an interrupt is triggered and the execution of the current instruction is abandoned. In the opposite case the initial three bytes from VOA are used as a base address (in the suitable data space) for the standard addressing mode following the dictionary prefix. This base is used as a bias to all addresses which occur in evaluation of the succeding argument.

6 **dictionary mode with a byte offset.**
If the contents of register R are nonzero then the

following action is performed: The lower (less significant) 3 bytes of register **R** are assumed to represent the so called **Virtual** (intermediate) **Object Address** (VOA). The fourth and fifth bytes from this address are then compared to the upper (more significant) two bytes of register **R**. If the comparison produces a negative result or if register **R** includes zero (all bits cleared) an interrupt is triggered and the execution of the current instruction is abandoned. In the opposite case the argument's location is determined by the sum of the address fetched from VOA and the byte which immediately follows the leading byte.

7 **R register.**
The contents of register **R** are used as an argument.

8 **(R) address in register.**
The argument is located in memory at the address occupying the lower (less significant) 3 bytes of register R.

9 **(-R) decremented address in register.**
The argument's length is subtracted from the contents of register **R** and then these contents (lower 3 bytes) are used as the address of the argument in memory.

A **(R+) address in register incremented.**
The contents of register **R** (lower 3 bytes) are used as the address of the argument in memory and then those contents are incremented by the argument's length.

B **((R+)) intermediate address in register incremented.**
The memory location of the argument is determined as the contents of three consecutive bytes from the address represented by the lower 3 bytes of register **R**. Then the contents of register **R** are incremented by 3.

C **(R+b) byte offset.**
The contents of register **R** augmented by the contents of the byte immediately following the leading byte are used as the address of the argument.

D **((R+b)) intermediate address with a byte offset.**
The sum of the contents of resister **R** and the byte immediately following the leading byte is used as the address of the argument's address.

E **(R+a) offset.**

The contents of register R augmented by the address immediately following the leading byte are used as the address of the argument.

F **((R+a)) intermediate address with an offset.** The contents of register R augmented by the address immediately following the leading byte are used as the address of the argument's address.

For modes C-F, if the specified register number is zero then the arguments are evaluated in a different manner:

C **constant.** The leading byte is immediately followed by the argument.

D **displacement.** The short integer immediately following the leading byte is added to the instruction counter (IC) to produce the argument's address. Note that the resulting argument is located in code address space.

E **address.** The leading byte is immediately followed by the address of the.argument.

F **intermediate address.** The leading byte is followed by the address of the argument's address.

3.1.5. Interrupts

Internal interrupts are raised within each primary processor. They may be grouped into two classes, depending on classification of their reasons. The first group includes the so called **Program** Traps which are raised due to some conditions caused by currently executed machine instructions. The occurrence of program traps is controlled: there is no notion of a pending program trap. The interrupts of such kind may be caused by the following reasons:

- an attempt to execute a non-implemented instruction,
- an attempt to execute an instruction which is illegal in the current mode,
- errors (division by zero, overflow and underflow, memory errors and so on),
- memory management exceptions (page fault, illegal dictionary reference etc.),

♦ extracodes (System Call).

The interrupts presented above cannot be masked or locally disabled: they are accepted and serviced as soon as they occur.

The second group includes interrupts which are independent of time: they may be raised in any moment, regardless of the current mode and status of the processor. Among them are: power failure, power restart, clock interrupt, hardware or firmware malfunction and so on. The clock interrupt may be masked and thus temporary disabled, the alarm interrupts cannot be masked however.

Inter-processor interrupts can be viewed as some messages sent between processors. Such messages can be divided into the following types:

♦ request to perform a specific action,

♦ confirmation, i.e. a message which informs that a requested action has been completed,

♦ exception raised within the sender processor i.e. failure, suspension or termination requested by the operator etc.

Such interrupts-messages are sent via Eventbus and are raised within the receiver processor.

Interrupt Service

The interrupt service can be divided into the followins stages:

♦ identification of the reason that caused the interrupt,
♦ preservation of the Task State Vector,
♦ loading of the new Task State Vector,
♦ execution of the service routine,
♦ return from the interrupt.

The first three stages constitute a single compact action which is performed in hardware. The service routine, which is entirely programmable, may be executed with (partial) admission for higher-priority interrupts to be received and serviced. Thus there may be a number of pending interrupts stacked in a single processor.

Details of the presented stages of the interrupt service may differ depending on the type and purpose of the processor. Primary processors, being completely unified, are also homogenous with respect

to the interrupt service.

Dynamic Reconfiguration of the Hardware

Any failure or malfunction in a system unit detected by hardware causes an internal interrupt in the relevant processor. In consequence, this processor may send pertinent interrupts to the other processors in the system. The collection of tools provided by this mechanism is sufficient to enable an operating system to update the system configuration, if only the severity of the damage permits it. Then the system may continue its processing, although possibly with a lost in performance and/or in the assortment of functions.

Priorities of Processors

The system interface includes a four-bit internal register, the so called Supervisor Processor Index (SPI), whose contents specify the index of the primary processor being authorized to receive all interrupts from the Auxiliary Processors.

Such a concept allows to treat each primary processor in the same manner. There is no need to definitely assign particular functions to a given processor, which substantially contributes to the system reliability. In particular, a damage in one primary processor neither has to crash the entire system nor even disable one of its functions.

On the other hand, the possibility to specialize the primary processors is still allowed. Let us note that even the external interrupts may be distributed among them. Namely, while the contents of SPI are fixed, the processor receiving an interrupt may pass it over to another processor by executing a special machine instruction (c.f. 3.2) which interrupts all the primary processors in the system. All irrelevant receivers of this passed interrupt will ignore it, so that it will be picked up and serviced by its legitimate addressee. The contents of SPI are set and changed by the System Monitor, which may be also possible upon a request from a primary processor.

The hardware stages of an interrupt service in a primary processor may be stressed as follows:

1 - The first element from the special list (whose address is fixed) is removed, its address is put into a special processor register **SA** (State Address). This operation is indivisible.

2 - The Task Status Vector including the 16 general purpose registers followed by the processor Status Register is written into memory starting from the address in SA.

3 - PP is set to the interrupt priority, CSI and DSI are cleared, the contents of SA are moved to R0.

4 - The interrupt specification and other specific information (if any) are successively put into the general purpose processor registers starting from R1.

5 - The new value of PC is fetched from a special table indexed by the interrupt number.

As all the processors in the system have an access to the Main Memory, the interrupts become a sufficient means for the communication between them. Each primary processor can send interrupts to all the remaining processors in the confisuration.

Presented below is the list of interrupts of the Primary Processor grouped with respect to their priorities:

priority 7 (interrupts always accepted)

- processor power failure

priority 6

- clock interrupt (c.f. 3.1.1)

priority 5

reserved

priority 4

- upper interrupt from a primary processor

priority 1-3

- interrupts from auxiliary processors

priority 0

- lower interrupt from a primary processor
- program interrupt

The upper interrupt from a primary processor may be used to signal an emergency event, e.g. a failure of its sender, the lower is recommended as a means of communication between processors.

Apart from the interrupts presented in the list above there may occur internal interrupts (traps) which are always accepted and serviced, regardless of the current processor priority. For a trap service being initiated the processor priority does not change, except in the situation when the previous priority was 0 - then the new priority is 1. The list of the internal interrupts is presented below:

- floating point overflow
- floating point underflow
- floating point division by zero
- unnormalized floating point operand
- integer division by zero
- illegal instruction or illegal contents of IC
- unauthorized write into Secondary Data Space
- illegal format of argument
- argument out of range
- illegal dictionary address
- index out of range
- System Call (extracode)
- Breakpoint Trap (c.f. 3.2)
- instruction trap (on T bit set)
- page fault
- memory parity error

3.2. Preliminary Instruction List

The proposed instruction list comprises preliminary instructions considered in the primary processor design. The instructions were intended to provide for relatively simple code generation for the compilers of high level programming languages, and also to offer an effective tool for the assembly language programmers (see [6], [7]).

The instructions are divided into thematic classes. Since most of the operations performed by the instructions exist in several versions depending on their argument(s) type, some abbreviations have been introduced - to address such groups by single mnemonics. In such cases the arguments are described by **S** (for standard) the possible postfixes to the instruction mnemonics (specifying argument types) are given.

--- Move S, S
postfixes: B, A, N, F
The first argument is copied to the second one.

--- Negate S
postfixes: B, A, N, F
The argument is negated arithmetically.

--- Reverse Bits S
postfixes: B, A, N, F
The argument is negated logically.

--- Add S, S
--- Sub S, S
--- Mul S, S
--- Div S, S
postfixes: B, A, N, F
The instructions perform the four standard arithmetic operations, both for the fixed-point and single-precision floating-point arguments. The first argument is added to (subtracted from, multiplied by, is a divisor of) the second argument and the result is located in the second argument.

--- Clear S
postfixes: B, A, N, F
Clearing (i.e. assigning zero to) the argument.

--- Compare S, S
postfixes: B, A, N, F
Arithmetic compare.

--- And S, S
--- Or S, S
--- Xor S, S
--- Erb S, S
postfixes: B, A, N

The set of elementary logical operations on a single byte, a 3-tuple and a 5-tuple of bytes.

--- Exchange S, S
postfixes: B, A, N

Simultaneous exchange of the arguments. This is an indivisible instruction.

--- Add Double D, D
--- Sub Double D, D
--- Mul Double D, D
--- Div Double D, D
--- Move Double D, D
--- Compare Double D, D
--- Negate Double D

The complement of the above set of instructions for double-precision arguments.

--- Add Numeric R, R
--- Sub Numeric R, R
--- Mul Numeric R, R
--- Div Numeric R, R

Shortened instructions (register arguments only) for 40-bit arithmetic operations.

--- Add Floating R, R
--- Sub Floating R, R
--- Mul Floating R, R
--- Div Floating R, R

Shortened instructions for single-precision floating-point operations.

--- Shift Arithmetically B, N

40-bit number is shifted arithmetically (i.e. with sign extension and overflow test). The first argument specifies the shift count.

--- Rotate B, L

The second argument is rotated by the bit count specified by the first argument.

--- Extended Multiply N, N, N, L

The first two arguments are multiplied and added to the third one. The 80-bit result is stored in the fourth argument.

--- Extended Divide L, N, N, N
The 80-bit integer number (first argument) is divided by the second argument. The result is located in the third argument, and the reminder in the fourth one.

--- Add With Carry N, N
The first argument and the C (carry) bit are added to the second argument. The result is located in the second argument.

--- Modulo1 B, A
--- Modulo2 N, N
The second argument is divided by the first one. The reminder is located in the second argument.

```
--- Lt      B                    ; N = 1
--- LtU     B                    ; C = 1
--- Le      B                    ; (N + Z) = 1
--- LeU     B                    ; (C + Z) = 1
--- Gt      B                    ; (N + Z) = 0
--- GtU     B                    ; (C + Z) = 0
--- Ge      B                    ; N = 0
--- GeU     B                    ; C = 0
--- Eq      B                    ; Z = 1
--- Ne      B                    ; Z = 0
```

These instructions determine the specified relations for the signed and for the unsigned (postfix U) arguments basing on the arithmetic flags Z, N and C. The **false** value is represented by 0 and **true** by all-ones.

--- Test Bit S, B
--- Set Bit S, B
--- Clear Bit S, B
--- Test & Set Bit S, B
postfixes: B, A
The second argument is used to determine a memory address. It may be represented in an arbitrary form (with the exception of modes 0-4 and 7). The first argument constitutes the bit index related to the specified memory address. The operation is performed on the specified bit. Testing means locating the negated bit value in the Z-flag. The instructions Clear Bit and Test & Set Bit are indivisible.

--- All Bits Set B, B, B
--- All Bits Clear B, B, B
--- Any Bit Set B, B, B
--- Any Bit Clear B, B, B

All the arguments of the above instructions are bytes, determined by the standard addressing modes. Each of the above instructions computes the logical relation (on the first arguments) described by the mnemonic. The logical value, according to the principles described above, is located in the third argument.

--- Find First Set B, A

The instruction searches for the first non-zero bit starting from the memory location described by the first argument. The index of the found bit (related to the starting location) is placed in the second argument.

--- Round D, F

The conversion from the double-precision form to the single-precision form with rounding.

--- Move Address S, A
postfixes: B, A, N

The address of the first argument is placed in the second argument.

--- Move2 arg, arg
--- Move4 arg, arg

The transmission of two and four bytes respectively.

--- Truncate to Address N
--- Truncate to Byte A
--- Truncate Numeric to Byte N

The instructions test if the argument lies within the specified range. If not then a trap is generated.

--- Re-cache

The instruction emits a signal to all the primary processors, which have their code address space register equal to the contents of the (data or code) address space register of the processor executing the instruction. The signal causes reloading of the fast memory for the code, which is in the primary processors' equipment.

--- Nop

Do-nothing instruction.

--- Index A, A, A, A

It is tested, if the first argument lies within the range defined by the lower and upper limit. If not then a trap is generated. Succesively, the sum of the third argument and the product of array element size by the difference of the first argument and the lower limit is located in the fourth argument. The second argument locates the "dope vector", containing succesively lower limit, upper limit and array element size.

--- Set PSR A
--- Fetch PSR A

The PSR manipulation instructions.

--- System Call B, A

The instruction invokes an operating system function. The first and the second arguments are passed similarly as the interrupt specification.

--- Breakpoint Trap

The request to change the program mode for debugging.

--- Extended Function Call

The instruction (after its decoding) initializes a microprogram supplied by the user (if an extension to the instruction list is implemented). The functions of the microprogram, i.e. further identification of the instruction, the form and evaluation method of the arguments, updating IC and the arithmetic flags depend entirely on the user.

--- Jump B

Unconditional jump to the address specified by the argument.

--- Insert A, A
--- Remove A, A

The lists manipulating instructions. An list element is represented by two addresses: the former element address and the next element address. The first of the instructions above inserts the second argument as the list element after the element pointed by the first argument. The Remove instruction deletes the list element specified by the first argument and stores its address in the second one. These instructions are indivisible.

Jump instructions

The Jump instructions exist in three versions: with single byte offset, with two-byte offset and with three-byte offset. The offset appears directly after the operation code or after the first argument (with no leading byte) and is denoted in the following list by **X**. In the case when the branch is performed the instruction counter is incremented by the offset value. For the following instructions (if they have X-argument only) the conditions (addressing mode 3) cannot be applied.

--- Branch X
--- Branch on Lt X
--- Branch on LtU X
--- Branch on Le X
--- Branch on LeU X
--- Branch on Gt X
--- Branch on GtU X
--- Branch on GeU X
--- Branch on Eq X
--- Branch on Ne X
--- Branch on V set X
--- Branch on V clear X
--- Branch on C set X
--- Branch on C clear X
--- Branch on False B, X
--- Branch on True B, X

The unconditional and conditional branches. The last two instructions test the conditions stored directly in the first (byte) argument.

--- Branch to a Subroutine A, X

The branches with a trace which is stored in the first argument.

--- D.bnz S, X
--- I.bnz S, X

postfixes: B, A, N

The instructions help to control loops. The first argument is decremented (or incremented) by 1. If the result is non-zero then a branch by the specified offset is executed.

Instructions with extended operation code

The first argument of these instructions is always in a register. In the list below it is denoted by **R**.

--- Add R, S, S
--- Sub R, S, S
--- Mul R, S, S
--- Div R, S, S
postfixes: B, A, N, F

--- And R, S, S
--- Or R, S, S
--- Xor R, S, S
--- Erb R, S, S
postfixes: B, A, N

The three-argument analogons of the instructions already described. The difference is that the result is transmitted to the register being the first argument rather than to the last argument.

--- Load & Convert R, S
postfixes: B, A, N, F

Data loading to the register with conversion depending on the argument type: B -> N, A -> N (sign extension), N -> F, F -> N (conversion).

--- Load & Set Mark R, S
postfixes: B, A, N, F

These instructions set the M-flag connected with the register. The flag will be cleared when any other instruction which can alter the register's contents is executed.

--- Load & Negate R, S
postfixes: B, A, N, F

The negated (arithmetically) value of the second argument is placed in the register.

--- Load & Complement R, S
postfixes: B, A, N

Similarly as above, but with the logical complement.

--- Jump on None R, A
--- Jump on not None R, A
--- Equals None R, B

--- Not Equals None R, B
The instructions allow to test if the object referenced by the L-address (see 3.1.4.) located in the register exists. The first two instructions allow to branch when the appropriate situation occurrs, the last two compute the logical value of the relation and place it in the second argument.

System (privileged) instructions

--- Set Interrupt
The instruction generates the program interrupt.

--- Emit Interrupt
The instruction emits interrupt signal to all the primary processors in the configuration.

--- Exit A
An indivisible instruction for return from interrupt service. The argument is stored in the special register **SA**. The 16 general purpose registers and processor state register are reloaded from a memory table addressed by SA. Succesively the element specified by SA is inserted to the list whose head is located at a fixed address. The list is the same used by the hardware interrupt (or trap) service.

--- Exit From Trap A
The effect of this instruction is identical to that of the above case with the only difference that directly after its execution the T bit in PSR is not tested.

--- Halt

--- Allocate Memory A, A
--- Test Memory Page A, A
The first argument describes a physical memory frame, the second one the new contents of its allocation register (for Allocate), or fetched contents of this register (for Test).

--- Set DSI B
--- Set SSI B

--- Read Clock A

--- Set Clock A
--- Stop Clock
--- Run Clock

The instructions for local clock service. The 24-bit clock register in equal time intervals decrements its value by 1, and in case of the zero result forces an interrupt. The Stop and Run operations allow to passivate and re-activate the clock.

--- Initiate Processor B, N
--- Examine Processor B, N
--- Stop Processor B

The instructions for inter-processor communication, in particular for input/output requests and emulation in auxiliary processors.

Special Operations

With the special operations, an argument describing the object location in memory is denoted by **B**. It must be represented with the aid of addressing modes: 5-6 and 8-F to define memory location.

--- Move Bytes B, B, A

The first argument specifies the start of block to be copied, the second one - destination address and the third one - number of bytes to be transmitted.

--- Move Bytes Until Character B, B, B

The first two arguments are as above. The third argument specifies the byte value which terminates (inclusively) the string to be copied.

--- Translate Bytes B, B, A

The first argument describes the starting address of the input text, the second one points to the 256-byte translation table, and the third one is the length of the input text to be copied.

--- Compare Bytes B, B, A

Comparison of two character strings. The arithmetic flags are set as for the other comparing instructions.

--- Scan Bytes B, A, B, B
--- Span Bytes B, A, B, B

The first argument defines the start of the input text,

the second one the length of the text, and the third one the 256-byte translation table. The fourth argument is a mask, which is logically and-ed with the translated bytes. The instruction stops when the logical product is zero (for Scan) or non-zero (for Span), or the input text is exhausted.

--- Evaluate polynomial F, N, B, F
The third argument describes the polynomial coefficient table located in memory. The second argument is the polynomial degree. The polynomial value for the fourth argument is assigned to the first argument.

--- Case
--- Call procedure
The proper semantics of these instructions will be described in a more detailed paper. However, the need for their implementation is already marked here.

3.3. Structure of the Primary Processors

On the way between a primary processor and main memory, connected to each other by Widebus, there occurrs a fast cache storage with relatively low capacity. The cache storage, provided for both data and code, reduces the resulting main memory cycle by decreasing the waiting time of the processor caused by the limited speed of interface and memory. The capacity of the cache storage (with a "transparent write") may be 2 or 8 Kb.

A primary processor executes instructions fetched from memory and communicates with other processors via Eventbus. It may respond to the events occurring in that bus and/or send/receive messages to/from the other processors of the system.

The anticipated speed of a single primary processor executing instructions from the standard instruction set of the minicomputer averages on 700000 operations per second.

The processor is built of a number of blocks which are working simultaneously. Each block is equipped with private autonomous control memory for microprograms (fixed and changeable), autonomous microprogram control, private registers and working storages and executive modules.

I-block fetches subsequent instructions from the cache storage, decodes them, determines the operands and controls the fetching of arguments. It is equipped with registers which describe the state of the processor, its status, the virtual memory space etc.

E-block constitutes an executive section of the processor. It includes the sixteen 40-bit general purpose operating registers together with appropriate indices and the arithmetic-logical processing unit (ALU). The ALU is intended to be built of suitable segments to provide for variable length of operands. The maximum length of operands serviced in ALU is 80 bits.

A-block is an optional section of the processor used for implementation of additional non-standard machine instructions. The structure of A-block leaves it open for implementation of vector instructions, specialized functions for signal processing (e.g. Fast Fourier Transform, filter operations etc.). This block may be easily exchanged.

C-block communicates the processor with the other processors of the system via Eventbus. It is equipped with facillities which organize the interrupt service, determine the processor priority related to the

priorities of other processors and allow to diagnose hardware malfunctions of the processor. C-block also undertakes emergency actions caused by exceptional events occurring in hardware (i.e. power failure).

The main processors will be implemented mainly in the Schottky TTL technology (2 - 3 ns propagation delay) using 4 or 8 bit-wide bit-slices (e.g. AMD 2900). Fast internal memories (register file, scratchpad memory) will employ static bipolar memories with access time of 50 - 70 ns. For cache, about 100 ns RAMs will be used.

At least several customized chips will be employed, mainly programmed logic arrays. Those chips will perform the interprocess communication, main memory control units, interface units, associative memories used by the virtual memory circuits and by the cache.

4. AUXILIARY PROCESSORS

Next to the notion of the Primary Processor presented in the previous section, but nonetheless as much important, is the notion of the Auxiliary Processor which is a substantial component of the system architecture. The basic design elaborates the following representatives of this component:

- supervisor and controller of system activity (**System Monitor**),
- memory manager and virtualizer (**Virtual Memory Processor**),
- emulator (**Emulation Processor**),
- rotating mass storage and tape controller (**Mass Memory Processor**),
- external interface controller (**Interface Processor**),
- input/output request handler (**I/O Processor**).

It is intended that most of the auxiliary processors, in particular those which are used to organize the cooperation of the system with several character-oriented peripheral devices or specialized measurement and industrial connections, be highly, if not completely, unified.

The auxiliary processors will be built around fixed-instruction-set microprocessor chips (e.g. INTEL 8080/86, INTEL 8088/89, Z8000) coupled with various supporting LSI chips and peripheral control chips. Whenever the speed of data transfer exceeds 100 Kb/s, the auxiliary processors are suggested to be built of fast bipolar chips.

Some other types of auxiliary processors, not mentioned in the list above, may be designed, put into production and distributed, if a need for them develops in the future.

The auxiliary processors cooperate with the Main Memory of the system via the Bytebus interface. Data width depends on the processor type and varies between 1 and 8 bytes per one interface cycle.

After having received an request the auxiliary processor initiates the demanded action on the pertinent device. While this action is in progress, the auxiliary processor supervises it, detects any faults and, if possible, attempts to correct them. It may also perform some preliminary preprocessing of the transferred data, as decoding/encoding, computing ECC or BCC, packing or unpacking (line

protocol frames, disk records, etc.). When the requested action is finished, unexpectedly terminated or suspended the auxiliary processor sends a suitable return information (interrupt) to the primary processors via Eventbus.

The precedence of auxiliary processors in accessing interfaces is determined by their relative priorities.

An auxiliary processor may act either in a selector mode (e.g. for fast rotating media) or in a multiplexer mode (for most of typical devices).

Local memory of an auxiliary processor, wherever it is present, may be used as an intermediate buffer storage or as an area to keep some status information (e.g. for processors which control some complex and sophisticated interfaces: Camac, IEEE-488, etc.).

The standard assortment of operations of an auxiliary processor may be easily extended due to the fact that most of its functions are defined in exchangeable software.

The assignment of functions to the particular types of auxiliary processors is presented below:

System Monitor

This auxiliary processor supervises the entire computational activity of the system. It initiates the operating system (bootstrap), validates system consistency and undertakes suitable emergency actions (e.g. system restart and recovery). System Monitor may also be able to execute (and report) some performance tests which could detect occassional bottle-necks and other critical situations in the system. Having a privileged access to all system resources, the System Monitor plays an important part in the situations when a system reconfiguration has to be performed. In particular, the contents of a special interface resister **SPI** (Supervisor Processor Index - cf 3.1.5) are exclusively changed by this auxiliary processor. As the System Monitor needs to have an immediate access to entire central memory of the system, it seems desirable to use a 32-bit microprocessor as a base of its implementation. Nonetheless, some other solutions (e.g based on bit-slice microprocessors) may also be taken into account.

System Monitor is interfaced (via some adaptative layer) with all primary and auxiliary processors of the system. That makes it capable of accessing, in a

privileged manner, the "private" memory area of any particular processor.

Virtual Memory Processor

Virtual Memory Processor is a kind of monitor which intercepts and fullfils page faults signalled by primary processors. Whenever an activity of a primary processor is interrupted by a page fault, the page request is passed to the Virtual Memory Processor. Then the interrupted primary processor can either wait for this request to be satisfied or continue the execution of another process. A number of page requests (possibly from different primary processors) may be queued in the Virtual Memory Processor which performs the appropriate algorithm of page exchange for each of them. It is also suggested that the Virtual Memory Processor would be equipped with some functions of optimization for the assumed stratesies of virtualization. An important point is that the action of resolving page faults is exclusively performed in the Virtual Memory Processor. No assistance of a primary processor, except when the action is initiated, is necessary. This kind of solution has two major advantages. First, the primary processors are not burdened by the problems of memory virtualization; second, due to the fact that those problems are managed by one specialized unit, they are much easier to be solved (synchronized) than in the case when each primary processor would be responsible for itself.

Emulation Processor

This kind of auxiliary processor serves the purpose of emulating one of alien computing systems as Pdp-11, Mera-400, Vax11/780 or others. Emulating alien program is seen by a primary processor similarly as an i/o operation.

Mass Memory Processor

This auxiliary processor controls the activity of disk drives by initiating and supervising device-memory transactions. An important property of the Mass Memory Processor, which makes it a bit different from other auxiliary processors in the system, is its wider (possibly 8 bytes) access to the Main Memory. Such an access is

necessary, due the the special nature of the transfers performed in this processor, to reduce the interface load. Buffer memory of the Mass Memory Processor allows to avoid the danger of breaking a disk transfer.

The Mass Memory Processor may also control other devices, e.g. magnetic tapes.

I/O Processor

This auxiliary processor interfaces the system with slow character-oriented peripheral devices, as display monitors, printers, readers, plotters and floppy-disks.

Instrumentation Processor

This local auxiliary processor supervises a configuration of a few IEEE 488 interface busses (e.g. up to 4 buses). It is also used to control the send and receive operations in the IEEE 488 interface. The Instrumentation Processor interprets a service program which implements an algorithm of measurement or communication. It supervises the execution of the measurement program and interprets the instrument data.

Communication Processor

The Communication Processor provides for remote cooperation of the desigsned minicomputer with other computers and devices via telex and phone lines as well as via specialized connections of data transmission (e.g. local networks).
The Communication Processor provides for a link-level message transfer with error detection and correction.

5. SOFTWARE

Employing Auxiliary Processors leads to considerable unification of cooperation principles between the system and environment, at system level and user software level as well. Charging the Auxiliary Processors with the i/o operations causes that the operating system layer including the set of device service modules (handlers or drivers) will be almost entirely executed by the microprocessors. This fact forces the specific structuralization of the operating system and necessity of accurate elaboration of the interconnection between primary and auxiliary processors. The advantage of this solution (after fixing the principles of cooperation) is the feasibility of easy, even intrusive division of goals at the system software.

5.1. Hardware support for synchronization

A programmer writing parallel programs (e.g. operating system) must employ some synchronization tools to assure the corectness of the program execution. These tools acquire some particular importance in multiprocessor environment, where there is a necessity of synchronizing several actions performed by a processor, that in single-processor systems go without any synchronization (e.g. interrupt stack service).

From the software point of view the design provides a usage of the unified hardware synchronization mechanism.

Due to the special bit IB in the main interface, which is on during the execution of the operations whose mutual exclusion should be guaranteed, none of the "indivisible" operations can start with the IB set and is delayed until the bit is off. Then the processor performing the indivisible action sets IB (the proper mechanism provides that only one of the processors at the moment is authorised to do so), continues the activity and after instruction's completion clears the IB flag.

The advantage of this solution is in the feasibility for other operations to run along with the indivisible operations, without any delay.

On the other hand the synchronization is required by only a few instructions or actions in the system, such as:

- queue instructions,
- Test & Set Bit and Clear Bit instructions,
- Exchange instruction,
- Exit instruction and hardware interrupt service in a primary processor,
- a change in memory allocation (mapping the virtual addresses into the physical ones).

Because synchronization consisting in delaying causes always a decrease in system performance, the design provides only inconspicuous but sufficient hardware mechanism for synchronization. One should note, that the set of indivisible instructions is extendable.

5.2. Operating System OOPS

The software of the proposed minicomputer is based on the operating system OOPS (Object Oriented Parallel System). A preliminary version of this system is now debugged and tested on the Mera-400 minicomputer. The philosophy of this system allows for its simple implantation to the designed computer, whole in assembly language. In this definite case we can hope that the efficiency of this operating system will be joined with the short period of its completion.

The operating system OOPS is oriented first of all for interactive work. However, the job priorities structure allows for simultaneous batch processing. The virtual computer as seen by the user can be briefly characterized by the following points:

- the user is independent of the environment elements, not directly connected with him (her);
- the size of the memory field occupied by the user's program is subject to the limitation of the address space size only;
- the system covers the actual number of processors in configuration up, simultaneously giving the possibility of creating concurrent processes (the number of processes is not limited a priori); it means that in configurations with more than one primary processor the user has the feasibility of executing true parallel programs;
- the user can conveniently create classes of problem-oriented files and exposes the software for distributed system structure, among others due to hierarchical, tree-like File System structure.

Many solutions of OOPS are based on the concepts applied in the UNIX operating system. UNIX was developed by Bell Laboratories, Inc. and runs on many computers (PDP-11, VAX-11, IBM-370 and others). In particular the File System's organization and the principles of interface with the user are borrowed from UNIX. However, the internal system structure is developed originally for supporting parallel computations in multiprocessor environment.

Standard mechanisms offered by the system

The user jobs processed in the system are called **tasks.** Each task may consist of a unlimited a priori number of **processes.** The mechanism for creating new processes and tasks is available to any user program. The allocation of system resources for particular tasks or processes

is entirely dynamic. The processes belonging to a single task form a tree structure of an unlimited a priori height. There is a feasibility for parallel (or semi-parallel) execution of several processes in the system, in particular parallel execution of a common task's processes.

Separate processes of a single task communicate by means of common memory. The system provides some natural, simple and efficient tools which allow to organize critical sections.

The standard synchronizing tools provided by the system and designed for organizing critical sections are:

- **lock**(u), where u is a binary variable (semaphore). The operation enters the critical section controlled by u. It should be emphasized, that the operation never causes a passivation of the process, even in a case when the given critical section is occupied. In this case the process trying to execute lock is abandoned by the processor without any change in its status.

- **unlock**(u), u as above. The operation exits critical section controlled by u. Formally the operation resets the semaphore u only. The processes abandoned by unsuccesful execution of lock operation gain automatically the possibility to enter the section.

- **stop**(u), u as above. This indivisible operation passivates the process executing it and simultaneously unlocks the semaphore u.

The purpose of the operations listed above is in organizing primary critical sections designed for synchronizing short-time sequences of instructions (e.g. counter modification). More sophisticated synchronizing mechanisms as: monitors (e.g. Hoare's or Loglan's), rendez-vous (Ada), conditional critical sections, semaphores (Dijkstra) or event queues are expressible in simple way in terms of theese operations and the **run**(P) operation, where P is a process identifier, that activates the given process after its possible passivation.

The system includes an unified i/o system based on the file conception. A **file** is a separated and defined finite sequence of bytes usually located in mass storage. All files accessible through the system have unified, transparent structure void of the notion of "logical record". In particular it means that the bytes within a file can be accessed randomly. The peripheral devices seen by the system are also treated as files, with specific limitations (e.g. a sequential nature of data medium). The system does not interfere in the contents of files, in particular binary and coded files are treated identically.

The unified i/o structure embodies also pipe-lines, allowing for information flow between different tasks. Due to this facility it is possible to aggregate many tasks into a single processing line. The following example of such a line can be given: preprocessor -- lexical analyser -- parser -- semantic analyser -- code generator -- assembler (possibly a two-pass one) -- link-editor. All compilers expected in the system form processing lines similar to the above line.

In many applications the usfulness of intercepting certain interrupts by user's processes is seen. The system allows to do it in a convenient, resilient and effective way. This property is especially useful in real-time applications. In case of particular requirements about fast and reliable program reaction it is provided that the whole code and data of the task are resident in the Main Memory. Such a task does not take the advantages of the virtual memory mechanism.

The system provides for an efficient service of power-alarm event turning to auxiliary power supply and allowing safe disconnection and turning off the controlled devices and objects. It is to emphasize, that the system architecture allows to continue the work in case of any single processor break-down.

File System

The File System of the OOPS organizes files into a tree-like structure, allowing among others to simplify the protection mechanisms, and giving facilities to reference groups of files. This structure is adaptable to distributed structure of a computer system. There are special files in the system, called directories, containing lists of file names (including also directory names). When a reference to a file is being made, it is specified by the identifier composed of a chain (may be empty) of directory names terminated by the proper file name. It allows for easy addressing of files existing even outside given computing environment (access through a network).

Files are preliminary objects in the system and a file is one of the most significant conceptions for the user. In particular executable program, subprograms library or source module are files and are identified by their descriptions in directories. Suitable, unified protection mechanisms provide for a safe usage of the File System in the sense, that is impossible, for example, to execute a file which does not contain a ready-to-go (executable) binary module. The same mechanisms are employed also for determination of user's ability to access (actively or passively) particular files.

Dividing files into different directories is user's job. Usually it

reflects applicative hierarchy of elaborated by him (her) problems. It allows the user to concentrate exceptually on aspects that are interesting him (her) at the moment.

The file system due to to its hierarchical construction, provides the possibility of simple insertion of the system into an existing computer network and creation of own, properly oriented networks supported by the adequate, fitted for applications, geometry. In particular the possibility for implementation of distributed, hierarchical data bases was considered.

5.3. Standard software

The standard software for the minicomputer is concentrated around the operating system OOPS. The ideas of that software are strictly connected with the system structure, in particular with the File System. The standard software, along with the tools offered directly by OOPS comprise the base for software environments of the new minicomputer. The most significant utilities are:

- User's Directives Interpreter (Job Control) being the only system interface with the interactive user;
- text editor EDM, based on editors ED and EX which are accessible under UNIX operating system;
- stream editor, compatible with EDM (regular expressions), usable as a filter for text files;
- text files preprocessor which allows including the contents of specified files into processed text and extending macro calls. The preprocessor should be automatically invoked before each compilation of a programming language used. It allows to give a library-like nature to a certain pieces of source text and to the usage of globally defined objects, as constants dependent on the actual configuration or application. The macro system of the preprocessor allows to apply parametrized macros.
- high-level languages compilers:
  Loglan-79, Fortran IV, Pascal, C, Basic
  and in the future:
  COBOL, APL, PROLOG, FORTRAN-77, LISP, LOGLAN-82;
- GASS macroassembler, based on existing assembler Gass-400 for minicomputer Mera-400;
- link-editor;
- set of debuggers, suitable for particular programming languages. The debuggers are supported by the possibility of step-by-step program execution;
- universal screen editor;
- text formatters for publishing, compatible with NROFF, TROFF and TEX;
- on-line calculator;

♦ universal sort/merge program;

♦ standard subprogram libraries, they are automatically searched when binary programs are linked. The link-editor, invoked by a compiler (assembler) after translation completion, searches the libraries specified by invoking program. The main system library includes input/output-, binary/character conversion- and programming languages standard functions subprograms.

The tools for creating software

Among means designed for software creating first of all there are some tools for compiler writing. Lexical analysers generator LEX and parsers generator YACC will be employed here. YACC provides also the possibility for suitable creation of semantic analysers based on its output. These tools are borrowed from another computer's software and are well known in the world. They have earned good opinion already. In particular YACC is very popular, not only among compilers writers, but also among regular users. Especially for the new minicomputer a unified code generator will be provided. It will be grounded on the inference-tree supplied by a YACC's parser and independently on the source language used it will apply unified code optimization principles. This approach to the compilers production allows to minimize the efforts in introducing new elements to the set of programming languages available on the new minicomputer.

The interface between OOPS and the programmer

A programmer who works interactively communicates with the system by means of command interpreter, invoked aotomatically when he (she) logs in at his (her) terminal. The interpreter reads user's commands, executes them and writes out messages. A single command consists of a name of file containing the program to be executed and of a chain of parameters, which are passed to the invoked program. A command may also include a few parts of that shape connected in a special way. Such a command is interpreted as a request for invoking specified programs and linking them with the aid of the pipe-lining into single processing line.

The user may also request to run the program in the background, concurrently to the further command processing. The user is able to stop such a program, to test its state and to make up his (her) mind about its nearest future (continuation or abortion). The number of

tasks run at the background level is unlimited a priori.

In a particular case, when the user-specified file doesn't contain any executable program (information about it is located in the file description in the File System), the file is treated as a script-file containing commands for the interpreter. It should be noted here, that the command interpreter is same program like another utilities and can be called recursively. The above mechanisms together with the possibility for task creation from a user's program eliminate the necessity of employing macros at the interpreter level. Indeed, the means obtained in this way are incomparably more expressive than the most sophisticated macro-system.

BIBLIOGRAPHY

[1] Jezierska-Ziemkiewicz, E. (ed.): Opracowanie ramowej koncepcji i okreslenie mozliwosci realizacji minikomputera wykorzystujacego dotychczasowy dorobek PRL w systemach minikomputerowych; IMM works - int. publ. (1981) (in Polish).

[2] Kreczmar, A. (ed.): Report on the programming language LOGLAN-79; int. publ. UW (1982).

[3] Eckhouse, E. and Levy, A.: Vax 11/780 Manual; Digital Equipment Corporation (1981).

[4] Janicki, A. (ed.): Basic design of a new minicomputer (the so called "New Mera-400"); TKP Consultants Ltd (1983), unpublished.

[5] Jezierska-Ziemkiewicz, E. and Ziemkiewicz, A.: Struktura systemu New Mera 400; preprint (1983) (in Polish).

[6] Findeisen, P. and Gburzynski, P.: Koncepcja procesora podstawowego (wraz z lista instrukcji) i systemu operacyjnego OOPS; preprint (1983) (in Polish).

[7] Litwiniuk, A.: Komputer a kompilator. Postulowane wlasciwosci sprzetu; preprint (1983) (in Polish).

S P R A W O Z D A N I A

I N S T Y T U T U I N F O R M A T Y K I U. W.

No 130 1983

P.Findeisen, P.Gburzyński, E.Jezierska-Ziemkiewicz,
A.Ziemkiewicz

PROPOZYCJA ARCHITEKTURY MINIKOMPUTERA

Streszczenie

W niniejszej pracy przedstawiono koncepcję minikomputera przeznaczonego m.i. do realizacji języków programowania zorientowanych obiektowo (np. Loglan-82). W projekcie szczególnie uwzględniono możliwość wykonywania programów współbieżnych, kładąc jednocześnie nacisk na możliwość efektywnej implementacji takich języków programowania jak Pascal i Fortran. Wprawdzie istota proponowanej architektury pozostaje w duchu tradycji maszyny von Neumana, tym niemniej zastosowano kilka nowszych rozwiązań, jak np. oddzielenie przestrzeni danych od kodu, wieloprocesorowość, zastosowanie mikroprocesorów do wspomagania operacji peryferyjnych.


SIGILLUM UNIVERSITATIS VARSOVIENSIS