Performance-Optimized Uniprocessor Architectures

Wolfgang Matthes
Franz-Mehring-Strasse 22
O-9006 Chemnitz
Germany

Introduction

The performance of state-of-the-art processors and the urge towards standardized open systems even at the binary code level (Application Binary Interface) make each attempt to develop new processor architectures highly questionable.
But the problem is not so obvious. Just the advent of open architectures should lead quite naturally to questions like:

- Which architecture is it worth to become a standard?

- Are innovative architectures really superior to well-introduced architectures implemented with the technologies of tomorrow?

- Which improvements in sheer performance or in performance-to-cost-ratio, respectively, would be possible? Will they be sufficient enough to justify the cost of software conversion?

- What is the optimum granularity of parallelism, or, with other words, the optimum size and structure of the processing elements if we want to build massively parallel systems (which is inevitable to satisfy highest-performance requirements)?

In this paper, we abstract from all non-scientific points of view, like market penetration, end-user acceptance, necessary investment capital etc. Obviously, all well-introduced architectures had been designed to earn money with. And this will be our final goal, too. To achieve this goal, we propose to take a first step: to try to reinvent the digital computer from the scratch, based on more than 40 years of experience and scientific work, relying on tomorrows technologies, like 50 million transistors on a single chip, 64 Mbit DRAMs etc., but with deliberate disregard of compatibility issues. A second step would be to evaluate the results and to consider the compatibility problems. Overwhelming as well as negligible improvements of performance-to-cost-ratio would solve the problem on the spot: the new ideas would find enough acceptance, venture capital etc., or they would be subject to the waste-basket, respectively. Without doubt, compatibility issues would be tackled, if worthwile. Perhaps it would be a strong motivation for creative people to invent something like hardware support, object code retargeting methods etc.

There is only one way to answer the questions: work hard and try. New architectures are to be developed, evaluated, and compared against the de-facto standards and other competitive approaches.

This paper outlines a tentative sketch of some related concepts and ideas.

Approaches to Performance Optimization

Two approaches were decisive for the development of recent popular computer architectures:

1. Optimization of machine instructions.

2. Exploitation of inherent parallelism in ordinary programs.

Machine instruction optimization is the core of the RISC-CISC debate. Instruction formatting, instruction timing, instruction pipeline structure, and register architecture are decisive for sheer performance as well as for the performance-to-cost-ratio of a processor architecture. Like in other fields of engineering, it has shown that near-optimum solutions will be well-balanced compromises; academic purism (e. g. RISC vs. CISC) is one thing, success on market is an other. To exploit inherent parallelism requires the capability to execute more than one operation at a time. This could be achieved by pipelining the operation unit, by providing multiple operation units, or by a combination of both (superpipelined/superscalar machines). To execute multiple operations in parallel requires appropriate principles to detect the parallelism and to control the hardware. This could be done by means of hardware support to detect parallelism during runtime and to issue more than one instruction in one machine cycle ("scoreboard" principles). An alternative approach provides extraordinarily long instructions which encode all operations to be executed simultaneously (Very Long Instruction Word Machines; VLIW). This principle requires the parallelism to be detected during compile time.

These approaches emerged mainly from measurements of instruction usage statistics, cache hit ratios, register allocation etc. In consequence, such architectures are already highly optimized. The optimization under the aspects of contemporary application programs, languages, and programming practice, however, will reach a point of saturation. Obviously, architecture development will reflect past and current programming habits, if essentially based on statistical data, thus leaving opportunities for further innovation untouched.

Hence the development of even better architectures requires to complement measurement-oriented approaches by completely new concepts. Architecture optimization should not be restricted to the machine instruction level but should also consider decisive application requirements directly.

This leads to an additional third approach:

3. A holistic all-embracing view from the application requirements (the problems to be solved) over the statements in a programming language and the compiled instruction sequences down to the register transfer structure of the hardware and the individual machine cycles.

Many important classes of application problems have a unsatiable demand for computing power. In such cases, the design of special hardware seems to be an obvious solution. Clearly, it is not feasible to provide special high-performance processors for each important application problem, but the performance superiority of properly designed dedicated hardware is really impressive. This leads to the

last approach:

4. Design principles and hardware structures of special processors should be used directly for the design of the universal computer. This does not mean to combine an universal computer with dedicated coprocessors in the obvious way. Instead, the principles and structures should be inherent in the architecture of the universal processor. Examples: dedicated circuitry for each kind of operation (multiple execution units), various dedicated storage means with parallel independent access paths, control and address calculation hardware adapted to typical innermost loops and access patterns.

Computer architecture development today cannot rely on accidental revolutionary inventions. Instead, well-proven methods of scientific and engineering work had to be used to reduce the amount of "gut feel" decisions and thus to minimize the overall risk. This means: a systematic step-by-step approach, the reduction of complex problems to simpler ones, and the systematic experimental abandonment of todays conventions or de-facto standards, respectively. To achieve further improvements, it seems necessary to abandon current conventions (e. g. the byte structure, the UNIX-style file handling etc.) and to work out as many new concepts as possible. These concepts are to be evaluated and compared against current architectural principles and rather evolutionary approaches (e. g. the improvement of well-introduced architectures).

In this paper, four principles will be outlined which could complement other research activities and contribute to a generalizing framework of systematic computer architecture development:

1. the resources paradigm
2. the paradigm of object orientation
3. the principle of controlled cardinality
4. the principle of incarnated abstractions.

The Resources Paradigm

Each computer structure can be regarded as a collection of hardware resources, such as memory, processing means, data paths etc. These resources are program-controlled, i. e. a stored program determines how the resources are used. Thus the resources are to be completed by storage means for the program information (control memory) and by means which select the appropriate control information for each of the machine cycles and which control the remaining resources (for more details, see "The Generality of the Resources Paradigm").

Evidently, functional capabilities and sheer performance of the processing resources are decisive to the overall performance of the computer. For example, a computer with a hardware multiplication unit will perform multiplication faster than a machine which does multiplication by means of shift-add sequences, regardless of the particular architectural principle. Multiple processing resources will increase the performance proportionally, provided all these resources can be kept busy with useful work. The peak performance of an arbitrary computer structure cannot exceed the limit given by the simple formula:

$$\text{Peak\_Performance} = \frac{\text{Quantity\_of\_Appropriate\_Resources}}{\text{Duration\_of\_a\_Machine\_Cycle}}$$

If the ensemble of processing resources has been selected, performance will be determined by the connection (data path) resources together with the memory resources. These resources are decisive to feed the processing resources with data and control information.

The influence of instruction formats, addressing modes etc. is significant at the third place only. These architectural features must be designed with the main objective to keep most of the resources busy with useful workload the whole time.

To investigate these problems more in detail requires a closer look at the hardware structure (Figure 2).
Processing resources are made of flipflops or hardware registers and combinatorial circuitry. All flipflops and hardware registers of the processing resources are regarded as a resources vector. Parts of the resources vector need information from storage means, parts deliver selection information (addresses) or data to storage means, and parts of the resources vector are fed back one upon another. This simple paradigm applies to all hardware structures which are provided to do digital (binary) information processing, and it allows to investigate several design alternatives systematically (see "A Glimpse of the Register Transfer Level - the Resources Vector").

According to this paradigm, the objective of computer architecture development is to provide powerful, cost-effective, feasible, and versatile collections of resources. This problem may be tackled systematically according to the following steps:

1. An inventory of appropriate resources is to be defined (storage structures, control circuitry, operation units, address calculation circuitry, interconnection structures etc.). For each resource type, performance and cost are known. Each type can be specified by a set of data structures and a set of operations defined on them. Hence the resources may be described formally by some kind of algebra (see "The Resources Algebra Concept - an Informal Sketch").

2. To design a concrete machine requires to create a concrete ensemble of resources which are to be selected from the resources inventory. A common objective for optimization is to maximize performance within given cost constraints (e. g. silicon area or transistor count, respectively).

3. In the next step, the memory and interconnection structures are to be developed.

4. The last step is concerned with instruction set design, i.e. with the formatting of control information. If the application profile is known, this is also an optimization task which may be solved systematically.

Such an architecture is characterized by its resources algebra. The instruction set in the conventional sense is less important. This leads quite naturally to the idea to use the resource algebra as the primary compiler target.

The contemporary architectural [illegible] resources, too. Many years of work had led to cost-effective hardware resources which exhibit considerably high performance. Hence it is not easy to find sources of further improvements. Here are some obvious proposals:

1. New kinds of resources may be added to the inventory, e. g. more sophisticated (perhaps specialized) types of operation units.

2. In new architectures, the quantity of resources may be increased so that inherent parallelism could be exploited (superscalar architectures).

3. Each kind of resources is still subject of internal optimization, even if the improvements to be expected may be rather small.

The Paradigm of Object Orientation

Object orientation is a necessity to master large software complexes. After the failure of iAPX432, hardware support for object orientation has gone out of vogue. But it is necessary to attack this problem. It seems possible to achieve a breakthrough if more hardware, i. e. an adequate resources ensemble, will be provided. (The expenditure of resources will be, in terms of gates and memory cells, or, with other words, in terms of silicon real estate, beyond the scope of customary microprocessors.) Such systems would be clearly superior compared to conventional architectures. For example, the access to descriptive information, e. g. object reference tables, object type descriptions etc., is decisive to overall performance. In such access sequences, an access is often dependent of information read in the previous access. A typical example is the access sequence according to Figure 4: selector in the instruction -> access reference table of the current program (occasionally referred to as capability table) -> object reference table -> content of the object. RISC-style overlapping principles are completely useless. The only effective solution is a structure of dedicated storage means together with appropriate access paths and control circuitry, i. e. the incarnation of the access scheme in dedicated hardware.

The Principle of Controlled Cardinality

This principle means to expose the finiteness of hardware resources explicitely in the architecture definition. Each ensemble of hardware resources is charcterized by a finite number of resources and by finite cost limits ("silicon budget"). Clearly, this finiteness should be hidden from the user; he should be in command of virtually unlimited resources. This is an important objective for each good programming language and operating system. Compilers, runtime environment, operating system, and hardware must work together to achieve this effect. At the instruction set architecture level, two approaches are applied to cope with the finiteness:

1. The quantity of resources is kept scarce, according to tight cost/feasibility constraints. A well-known example is the register file of the processor. To make efficient use of the resources is left to the programmer or compiler, respectively. To find compromises between cost, feasibility, and application requirements, a lot of

value cardinalities by measurements are available for evaluation (see "Adequate Cardinalities Proven by Experience").

2. The cardinality provided is so large that it can never go exhausted. This approach is typical of the storage address space (32-bit or even 64-bit addresses). To implement this principle means either excessive (in most cases prohibitive) expenditures or additional provisions which are effective during runtime to take special measures if some technically feasible limits have been exceeded. The most prominent example is the paging mechanism of the virtual memory.

The second approach has the obvious advantage of elegance. But it has also drawbacks with respect to hardware cost, machine cycle length, code size, and runtime behaviour. To exercise program control (i. e. to select one of the resources for use), the large cardinality must be coded somewhere (e. g. within the instructions) . This means considerably more bits per instruction and thus an increase in code size. More complicated hardware may lead to a longer machine cycle. The runtime behaviour may become unpredictable (example: page thrashing in virtual memory).

There are many good principles which could be implemented in a feasible scale, but which would require some compromises if implemented in large scale. Hence cardinalities in an architecture should be specified according to feasibility constraints. Example: A 16 by 16 crossbar network of fast 32-bit data paths is feasible, a 1024 by 1024 crossbar is not. Thus the architecture of the storage subsystem may specify modules of 16 memory banks each and a maximum of 16 nonblokking access paths. Hence each system programmer or compiler author could rely on the corresponding hardware properties or performance characteristics, respectively. An other example is hardware support for object orientation. A significant improvement in performance requires an appropriate structure of dedicated fast storage devices. The maximum capacity of these devices must be specified in the architecture so that performance-decisive access paths could be passed in one machine cycle. This could be achieved easily for object identifiers of 12 to 16 bits in length by application of fast static memories (4k to 64k), but obviously not for extremely large (e. g. 64-bit-) object identifiers.

The Principle of Incarnated Abstractions

This is the underlying principle for the selection of functions to be cast in hardware and for the selection of targets for further improvement.
Contemporary computer architectures had been optimized mainly on the instruction level. To that end, optimization data had been gathered by measurements and statistical evaluations based on existing programs. To be committed to existing programs only means to transfer conventional programming habits to new architectures, and may prevent from discovering opportunities for innovation.
Thus the holistic approach is an ultimative necessity. Application requirements are to be studied according to their intrinsic structures, not influenced by current programming habits. The base of experience may be found in sciences which correspond to important classes of application problems, like applied mathematics, formal logic, linguistics etc. An obvious source of experience are the well-

proven programming languages, operation principles, and machine architectures (see "Incarnated Abstractions - Three Examples").
These bases of experience are to be scanned systematically to find out widely accepted information structures and operations which are suitable to be cast in hardware structures (i. e. incarnated), or which provide a significant degree of usable inherent parallelism, respectively. From such systematic work, targets for future innovation may be identified. This is related to further improvement of already known resources (e. g. new operation units for numerical calculations) as well as to entirely new hardware structures. In other words: the principle of incarnated abstractions serves for selection and specification of features which are worth to be incarnated in dedicated hardware according to the approach to apply design principles of special processors.

An Example Architecture Proposal

Philosophy and Basic Concepts

Experience has shown the usefulness of designing "paper machines" at particular stages of research activities. In our case, we concentrated on high-end systems and intented to develop a "good" (i. e. universal, powerful and cost-effective) high-performance machine which could replace many of the current multimicroprocessor systems and which could be modified to be a superior building block for massively parallel systems.

The microprocessor in the true sense of the word was deliberately not chosen as the first research target. From the angle of the resources paradigm, microprocessors are simply some kind of an ensemble of resources. These resources had to be selected primarily under severe cost constraints (e. g. silicon real estate or transistor count, respectively) and under the constraint to provide off-chip interfaces which could be used by customers easily (regular address and data buses, timing schemes which fit for popular memory ICs etc.). Clearly, to design a whole system from the scratch, without the intention to sell the integrated circuits individually, allows for more degrees of freedom. Besides, contemporary superscalar processors could be characterized according to our terminology, too (see "How Far are Contemporary Processor Architectures from Our Principles?").

The objective of the proposal is to provide even more functionality and sheer performance. Supercomputer processing performance should be combined with the capability to manage a large database which may be upgraded to a true knowledge base. Moreover, it should constitute an implementation target for the IC technologies of the 90's, thus it should make efficent use of large RAM structures and of logic circuits comprising a few million transistors. This paper machine may be characterized by the following highlights:

- The architecture will be specified by a resources algebra.

- Orthogonality: the architectural specifications are independent of each other: data structures, descriptive structures, instruction formats, length of machine words, instruction issuing mechanism, selection mechanisms. A collection of Ada packages may be a good paradigm for such specifications.

- Hierarchical memory structure: archival/backup subsystem, knowledge base subsystem, execution environment memory, control memory, data/reference memory, memory structures within the operation units.

- Distributed processing with independently programmable devices in various functional units and levels of the memory hierarchy, respectively.

- The architecture is completely based on object-orientation, and comprehensive hardware support has been provided.

- Only one elementary data structure: a string of n bits in length (1 ≤ n ≤ 64k), called a binary vector. Binary vectors could be interpreted as numerical or non-numerical information, respectively. The basic numerical data type is the binary coded natural number. Integers are naturals extended by a sign bit (i. e. sign-magnitude representation as opposed to the usual 2's complement representation); floating point numbers are composite objects of an integer mantissa and an integer exponent; rational numbers (including decimal numbers) are represented as binary coded fractions. Floating-point numbers can be converted into extremely long integers and vice versa to facilitate implementation of high-accuracy algorithms.

- The logical representation of the objects is isolated from the physical packing of the bits which encode the object content. For sake of performance, there are some predefined formats of physical packing. The basic format is the bucket of 128 bits. Each bucket could be divided into bags of 8, 16, 32, or 64 bits. Buckets are packed together in containers of different sizes.

- The processing hardware comprises multiple high-performance operation units.

- The control principles are combinations of microprogram, VLIW, and dataflow concepts.

- instruction issuing, operand selection, and processing are independent of each other.

- I/O will be done via dedicated peripheral processors and standardized interfaces.

Overall System Structure

The main building blocks are shown in Figure 6. The peripheral subsystems are composed of state-of-the-art hardware, like disk arrays, optical disks, and various interface controllers. Thus a more detailed description can be omitted. The Knowledge Base Memory (KBM) is the heart of the memory hierarchy. It is not simply a large RAM array, but an active unit with various processing and memory resources. Its main purpose is to hold the data/knowledge base available for access, update, and processing. Processing will be done within the Execution Resources Ensemble XRE which comprises the Execution Environment Memory XEM and at least one processor. The processor is a superscalar/superpipelined machine comprising four universal high-performance operation units, independent address calculation hardware, memories, and control circuitry. To be processed, selected

information structures (complete or partial objects) from the KBM to the XEM which may be considered similar to the RAM in conventional architectures. Processors reference to the KBM via object identifiers and appropriate commands. Access to the XEM is via conventional addressing.

Knowledge Base Memory

The KBM is a heterogenous configuration which comprises a structure memory, an array of data memories, and the necessary interfacing provisions. The structure module provides the hardware support for the object orientation. Each information structure is considered to be an object which can be accessed by its identifier. The identifier-to-address mapping is done by specialized hardware which includes dedicated memory structures (here, the principles of incarnated abstractions and controlled cardinality have been applied). Figure 7 illustrates the flow from the access control information (which may be part of an instruction) to the data address of the object content (this scheme is complemented by Figure 8 and Table 5).
Memory space is allocated in form of containers. In the architecture, the container sizes are standardized (if only one container size were provided, the concept would be similar to the conventional paged virtual memory). Complex allocation principles are implemented to guarantee that the data will be stored according to the particular use (for high-speed processing, an object should be stored as an entity, for retaining in a backup storage, data may be scattered, and even data compression may be applied).

Processor module

The components of the processor module will be explained according to Figure 6:

- The Control Memory CM contains the last recently used instructions. It may be compared to a conventional instruction cache, but according to the principle of controlled cardinality, software can exercise a tighter control (there are explicit instructions to load programs into the CM, and hardware provisions to access these programs directly which allows for a shorter machine cycle than the usual set-associative access mechanism). In the example, the CM is composed of four banks of 4k buckets (128 bits each).

- The References/Data Memory RDM may be compared to the conventional data cache, but it is, similar to the CM, subject of tight software control. Essentially, it is used to hold access descriptors (capabilities; see Figures 7,8) and temporary variables of the active programs (it resembles somewhat to a scratchpad memory or to the on-chip RAM of the Transputer, respectively). In the example, the RDM is composed of four banks of 4k buckets (each bucket may contain two access descriptors or some temporary variables, according to the particular data type).

- The Selector/Iterator Resources Ensemble SIRE is essentially a collection of address calculation means similar to the example shown in Figure 5. Adequate hardware is provided to ensure that the proces-

many pipelines should be kept busy in typical performance-decisive loops.

- The Processing Resources Ensemble PRE consists of four processing units, called Processing Resources Collections PRC. Figure 9 illustrates the structure of a PRC. There are resources for numerical and non-numerical (logical, graphics) processing as well as for the detection of various conditions. Internally, a Multipurpose Memory MPM is provided. It may be used as a scratchpad, a stack cache, or as a collection of vector registers, respectively. In the example, it is composed of four banks of 256 buckets each. Multiple data paths are 64 or 128 bits wide, and all resources are designed for maximum performance. Some of the resources could be operated in parallel. The PRC may be considered somewhat similar to state-of-the-art processors, like i860 or TMS 34082. (For a glimpse of more details, see MAT91.)

System expansion

The system could be expanded according to an "inverted tree" scheme (Figure 10): The KBM subsystem could be extended by more KBM units. Multiple XRE units could be attached to the KBM. Up to two processors could be attached to one XEM. Thus memory capacity and processing performance will grow together adequately if the architecture is extended to true parallel (medium-grain MIMD-type) processing.

References

CHI86 W.-M. Ching, "An Extended von Neumann Model for Parallel Processing," Proc. FJCC 1986, pp. 361-371.

CO89 R. Cohn et al., "Architecture and Compiler Tradeoffs for a Long Instruction Word Microprocessor," SIGARCH Computer Architecture News Vol. 17, No. 2, pp.2-14, April 1989.

COL87 R.P. Colwell et al., "A VLIW Arcitecture for a Trace Scheduling Computer," Operating Systems Review Vol. 21, No. 4, pp. 180-192, October 1987.

DA89 W.J. Dally, "Micro-Optimization of Floating-Point Operations," SIGARCH Computer Architecture News Vol. 17, No. 2, pp. 283-289, April 1989.

DI87 D.R. Ditzel, "Design Tradeoffs to Support the C Programming Language in the CRISP Microprocessor," Operating Systems Review Vol. 21, No. 4, pp. 158-163, October 1987.

GI81 W.K. Giloi. Rechnerarchitektur. Springer, 1981.

IN88 INMOS Ltd. Transputer Reference Manual. 1988.

INT90 Intel Corporation. i860 64-bit Microprocessor Programmers Reference Manual. 1990.

JE88 Y. Jegou, "Access Patterns: A Useful Concept in Vector Programming," Supercomputing. 1st International Conference Athens, Greece, June 1988 Proceedings, pp. 377-391 (Springer 1988, LNCS Vol. 297).

JO88 N.P. Jouppi, "Superscalar vs. Superpipelined Machines," SIGARCH Computer Architecture News Vol. 16, No. 5, pp. 71-80, November 1988.

JO89 N.P. Jouppi, D.W. Wall, "Available Instruction-Level Parallelism for Superscalar ans Superpipelined Machines," SIGARCH Computer Architecture News Vol. 17, No. 2, pp. 272-282, April 1989.

KU74 D.J. Kuck et al., "Measurements of Parallelism in Ordinary FORTRAN Programs," Computer Vol. 1, No. 1, pp. 37-46, January 1974.

KUL81 U.W. Kulisch, W.L. Miranker. Computer Arithmetic in Theory and Practice. Academic Press. 1981.

MAH86 M.J. Mahon et al., "Hewlett-Packard Precision Architecture: The Processor," Hewlett-Packard Journal, August 1986, pp. 4-22.

MAL84 R.E. Matick, D.T. Ling, "Architecture implications in the design of microprocessors," IBM Systems Journal Vol. 23, No. 3, pp. 264-280, 1984.

MAT90 W. Matthes, "Hardware Resources: A Generalizing View on Computer Architectures," SIGARCH Computer Architecture News, March 1990.

MAT91 W. Matthes, "How Many Operation Units are Adequate?," to be published.

MUL88 D. Mueller-Wichards, "An Algebraic Approach to Performance Analysis," Parallel Computing in Science and Engineering, pp. 159-185 (Springer 1988, LNCS vol. 295).

ORG73 E.I. Organick. Computer Systems Organization. The B 5700/B 6700 Series. Academic Press, 1973.

ORG82 E.I. Organick. A Programmers View of the Intel 432. McGraw-Hill. 1982.

SM89 M.D. Smith et al., "Limits on Multiple Instruction Issue," SIGARCH Computer Architecture News Vol. 17, No. 2, pp. 290-302, April 1989.

SO89 G.S. Soni, S. Vajapeyam, "Tradeoffs in Instruction Format

Design for Horizontal Architectures," SIGARCH Computer Architecture News Vol.17, No. 2, pp. 15-25, April 1989.

THO64 J.E. Thornton, "Parallel Operation in the Control Data 6600," AFIPS Proc. FJCC, Part 2, Vol. 26, pp. 33-40, 1964.

WAF87 S.P. Wakefield, M.J. Flynn, "Reducing execution parameters through correspondence in computer architecture," IBM J. Res. Dev. Vol. 31, No. 4, pp. 420-434, July 1987.

BOX 1

The Generality of the Resources Paradigm

The simple paradigm, sketched roughly in Figure 1 [MA90], covers even the extremes in computer architecture: the "puristic" v.Neumann as well as the dataflow concept.

In the v.Neumann architecture, the control memory is accessed sequentially. The sequence is determined by the control means according to hard-wired algorithms of instruction address counting or branching, respectively. Besides, in a genuine v.Neumann machine there is only one memory; i. e. control and data memory are unified. v.Neumann machines are performance-limited since the processing resources are controlled sequentially. Hence inherent parallelism during runtime cannot be exploited.

In a dataflow architecture, the control memory is essentially an associative memory which delivers control information according to the availability of resources and data. Obviously, this allows for maximum performance, but such an completely associative control memory is cost-limited. Implementations of practicable size will require compromises between associative and sequential access, i. e. between performance and cost, respectively.

BOX 2

A Glimpse of the Register Transfer Level - the Resources Vector

The resources vector paradigm (Figure 2) means essentially to study and evaluate architectural principles at the register transfer level. Evidently, all architecture proposals which pretent to become implemented must have this level in common. Our approach is not to start architecture design by deciding between CISC, RISC, VLIW etc. principles, but to consider an ensemble of resources which has been assembled together to fulfill specific cost/performance objectives, and to provide for the most efficient use of these resources. We want the resources to do useful work in each machine cycle, thus "auxiliary" cycles which contribute nothing to solve the application problem (like instruction fetches, address calculations etc.) have to be avoided. Figure 3 a) illustrates the obvious solution: in each machine cycle, all the necessary control and data information from memory must be delivered in parallel. VLIW and dataflow architectures come close to this ideal. But cost is high, and some concepts are not feasible, at least in the near future. The common compromise is to select only a part of the resources vector in each machine cycle and to encode this selection within the control information (i. e. within the instructions). This scheme (Figure 3 b)) applies to most of the contemporary computers. In RISC machines, the control circuity is merely combinatorial, and each machine cycle is controlled by a particular instruction. To implement CISC architectures requires rather complex sequential control circuitry (i. e. a state machine) or even microprogram control.

BOX 3

The Resources Algebra Concept – an Informal Sketch

An algebraic structure is characterized by a set of data structures $|D| = \{D_1 \ldots D_n\}$, a set of operations $|O| = \{O_1 \ldots O_m\}$, and a mapping $MD \subset \{|D| \times |O|\}$ which assigns to each operation $O_i \in |O|$ the corresponding data structures $D_j, D_k \ldots \in |D|$.
Obviously, it is suitable to provide two kinds of algebraic structures: an inventory algebra IA and an architecture algebra AA.
An inventory algebra IA may be represented like

$$IA = \{|D|, |O|, |R|, MD, MR\} \text{ where}$$

$|D|$ is the set of data structures (fundamental data types),

$|O|$ is the set of operations provided,

$|R|$ is the set of hardware resources (the inventory proper),

$MD \subset \{|D| \times |O|\}$ is the mapping which assigns data structures to operations,

$MR \subset \{|O| \times |R|\}$ is the mapping which assigns operations to resources (it has been introduced because some hardware resources are capable of performing more than one operation; the conventional arithmetic-logic unit may serve as an example).

An architecture algebra AA may be used to describe a particular computer architecture:

$$AA = \{|RA|, |RC|, IS\} \text{ where}$$

$|RA| = \{RA_1 \ldots RA_r\}$; $RA_i \in |R|$, $i = 1 \ldots r$, is the set of resources selected from the inventory,

$|RC| = \{c_1 \ldots c_r\}$; $c_i \in \{1,2,\ldots\}$, $i = 1 \ldots r$, is the set which describes the quantity (cardinality) of each selected resource,

IS describes the interconnection structure.

These simple algebraic structures may be complemented by other formalized descriptions of cost, performance etc. (the performance algebra in MUL88 may be considered a stimulating example).

BOX 4

Adequate Cardinalities Proven by Experience

Many experimental and analytical results are available which could be used to define cardinalities in innovative architectures. Important topics are:

- the size of a register file
- the size of instruction, data, and stack caches
- the size of memory pages and segments
- the number of parameters to be passed in subroutine invocations
- the frequency of particular operations
- the rate of usable inherent parallelism.

Two examples will be described more in detail:

1. The size of a stack cache

In the experimental CRISP microprocessor [DI87], a stack cache of 32 words of 32 bits had been provided. This capacity was chosen according to a comprehensive hit ratio analysis. Some of the results are given in Table 1, showing the significance of comparatively small fast on-chip memories.

2. The correspondence between selection fields in instructions and the number of data objects to be selected

Comprehensive investigations [WAFS7] have shown that there is virtually no application program which accesses to more than 4096 different data objects (variables and constants). Most frequently, programs access to 17...64 data objects. Obviously, massive reductions in code size could be expected if address field length in instructions would correspond directly to the number of data objects which are accessed by the particular program (i. e. the field in the instruction would contain not an address but the ordinal number of the corresponding data object, and some mapping hardware would be required). Some results are shown in Table 2. Bit-variable field length is difficult to implement, thus some fixed-length formats (e. g. 6 and 12 bits) will provide a feasible compromise, and sufficiently fast memories (e. g. 4 kbits, 10 ns) are available for performance-efficient implementation of the mapping hardware. Refer to Figures 7 and 8 for an example of a hardware structure in the case of a totally object-oriented concept, where two mapping stages are provided (ordinal -> access descriptor -> object descriptor). A more conventional approach would require only the mapping from the ordinal to the object address, thus requiring only one memory.

BOX 5

Incarnated Abstractions - Three Examples

1. Data types and elementary operations for numerical computations

Mathematical research has shown that the following data types are necessary to perform numerical computations [KUL81]:

1) natural numbers
2) integer numbers
3) real numbers
4) complex numbers
5) intervals over 2) and 3)
6) vectors and matrices over 2), 3), and 4).

The necessary operations are:

- addition
- subtraction
- multiplication
- division
- dot product.

For the data types 3)...6), structures and operations are shown together in Table 3. Each mathematical operation on mathematical structures is to be mapped to a machine operation which processes machine-internal data representations. These mappings must guarantee that the obtained machine-internal representation of the result will be as close as possible to the accurate result (there must be no other value of the internal representation between the internal and the accurate result), and the rounding direction must be controllable by program. These mathematical realizations may lead to high-performance/high-accuracy resources for numerical calculations based on only one elementary data type: the binary coded natural number.

2. Access Patterns

Access patterns may serve as abstractions for data selection in performance-decisive DO-loops. Examples are shown in Table 4 [JE88].
Besides, data structures and operations of the APL language are well-suited as a base of experience, too (this language has already inspired architecture research: e. g. CHI86, GI81). Dedicated hardware could be provided to facilitate rapid access to elements of data structures according to such principles. An example is the iterator hardware shown in Figure 5 which can deliver address values to access array structures from one to three dimensions. To formulate such an access pattern in a common

programming language required nested DO-loops, e. g.:

```
for AD3 = 1 to EC3 do
    for AD2 = 1 to EC2 do
        for AD1 = 1 to EC1 do

...calculations using variables Vi(AD1,AD2,AD3)...

    end;
  end;
    end;
```

Usually, the address of an array element Vi(AD1,AD2,AD3) will be calculated according to the formula

ADDRESS(Vi) = ARRAY_BASE + AD1 + (AD2-1)*EC1 + (AD3-1)*EC1*EC2

(EC1,2,3 represent the element count of the correponding dimension.)
The iterator hardware avoids address calculation in the loop body. Especially the multiply operations are omitted; they will be needed only for the set-up of the hardware (offset calculations) prior to loop execution.

3. Inherent Parallelism

The rate of usable inherent parallelism depends on the semantic level on which the problem has been investigated. A lower level means less usable parallelism. Investigations at instruction level show a rate disappointingly low (e. g. not more than two instructions are recommended to be executed in parallel; e. g. JO89, SM89). On the contrary, investigations of Fortran source code had promised that from 16 to more than 128 independent operation units could be kept busy [KU74]. Hardware implementation of APL-like structures and primitves allows for even more parallelism; dedicated high-performance hardware (special processors optimized for the particular data structures and operations) have shown to be superior even to (hypothetical) dataflow machines. These facts emphasize the importance of considering higher semantic levels for architecture design (the usefulness of skipping intermediate semantic levels in the process of architecture optimization has been stressed in MAL84, for example).

BOX 6

How Far are Contemporary Processor Architectures from Our Principles?

Characteristic features of contemporary high-performance processors are completely adequate to the principles described, or, from another point of view, these principles are merely generalizations of and extrapolations from the state of the art which show evolutionary ways to still better architectures. Here are a few examples (mainly based on i860, i960, TMS 34082, Transputer T 800, Motorola 88000 and DSP 960002):

Multiple and specialized operation units may be considered to exemplify the resources paradigm and the principle of incarnated abstraction. There is separate integer and floating point hardware which can be operated in parallel, thus providing parallel floating point computations and integer address calculations. Besides, some universal architectures provide operations which had been typical of special processors (e. g. elementary graphics operations, like Clip, Draw, z-buffer moves etc.). Further directions of evolution:

- more units
- other special operations (e. g. to facilitate knowledge processing)
- improved internal chaining of operations (see the multiply-add chaining in the FP unit of the i860)
- specialized units for universal high-accuracy arithmetic as well as for address calculations (Figure 5 may serve as an example) may perform better than the combination of FP and universal integer units.

The principle of controlled cardinality may be exemplified by on-chip memory which is directly controllable by software (i860 cache used as vector register, Transputer on-chip RAM). Further directions of evolution:

- heterogenous distributed memories (multiple vector registers, dedicated memories to support object-oriented addressing; see Figures 7,8)

- on chip multipurpose scratchpad, accumulator, or stack cache memories.

On-chip memory management, protection, segmentation, and paging hardware (i860) may be considered a predecessor for fully-fledged support of object orientation (these features provide already some degree of abstraction for data and instruction access).

Stack cache size (32-bit words)	Hit ratio (on-chip references)
0	0%
4	42%
8	50%
16	73%
32	81%
64	82%
128	82%
256	82%

Table 1: Empirical results: stack cache size vs. hit ratio.

Length of address (selection) field in instruction	relative code length
variable to the bit	1.0
4/8/12 bits	1.18
6/12/18 bits	1.13
8/16/24 bits	1.43
only 12 bits	1.78
only 16 bits	2.12
like S/370	3.30

Table 2: Empirical results: address (selection) field length vs. code efficiency.

Data types	Structure	Operations
- real numbers, - real intervals, - complex numbers, - complex intervals.	Scalar	+ - * / *
	Vector	+ - *
	Matrix	+ - *

Note: All data types are valid for all structures.

Table 3: Overview: elementary data types, structures, and operations for numerical calculations.

1. a row of a matrix
2. a column of a matrix
3. the main diagonal of a quadratic matrix
4. the row section of a quadratic matrix according to the upper triangular matrix
5. the row section of a quadratic matrix according to the lower triangular matirx
6. a vector consisting of the odd elements of another vector
7. a vector consisting of the even elements of another vector
8. the transposed matrix
9. the "planes" of a cube
10. a matrix consisting of the odd elements of the odd rows of another matrix
11. submatrices

Table 4: Examples of typical access patterns in numerical computing.

Information structure	Memory structure	capacity/width	Notes
Context descriptor	CRT Context Reference Table Memory	64 x 32 bits	1
Access descriptor	ART Access Reference Table (in RDM)	16k x 128 bits (2 descriptors in 1 bucket)	2
Region descriptor	RRT Region Reference Table Memory	>4k x 64 bits	3
Object descriptor	ORT Object Reference Table Memory	>16M x 128 bits	4

Notes:

1 An instruction can select between 64 Access Reference Tables which represent the context of the current program. The CRT memory is an auxiliary memory within the RDM hardware (it is somewhat similar to the display registers in the B 6700 [ORG73]).

2 The current Access Reference Tables are stored in the RDM proper. A 128-bit-bucket contains two access descriptors.

3 The object space of the system consists of max. 64k regions with 4G objects each. The RRT memory is a set-associative region descriptor cache.

4 The last recently used object descriptors are held in the dedicated ORT memory. Like the RRT memory, it is organized according to a set-associative principle.

Table 5: Overview: information structures of object-oriented access organization according to Figures 7,8.

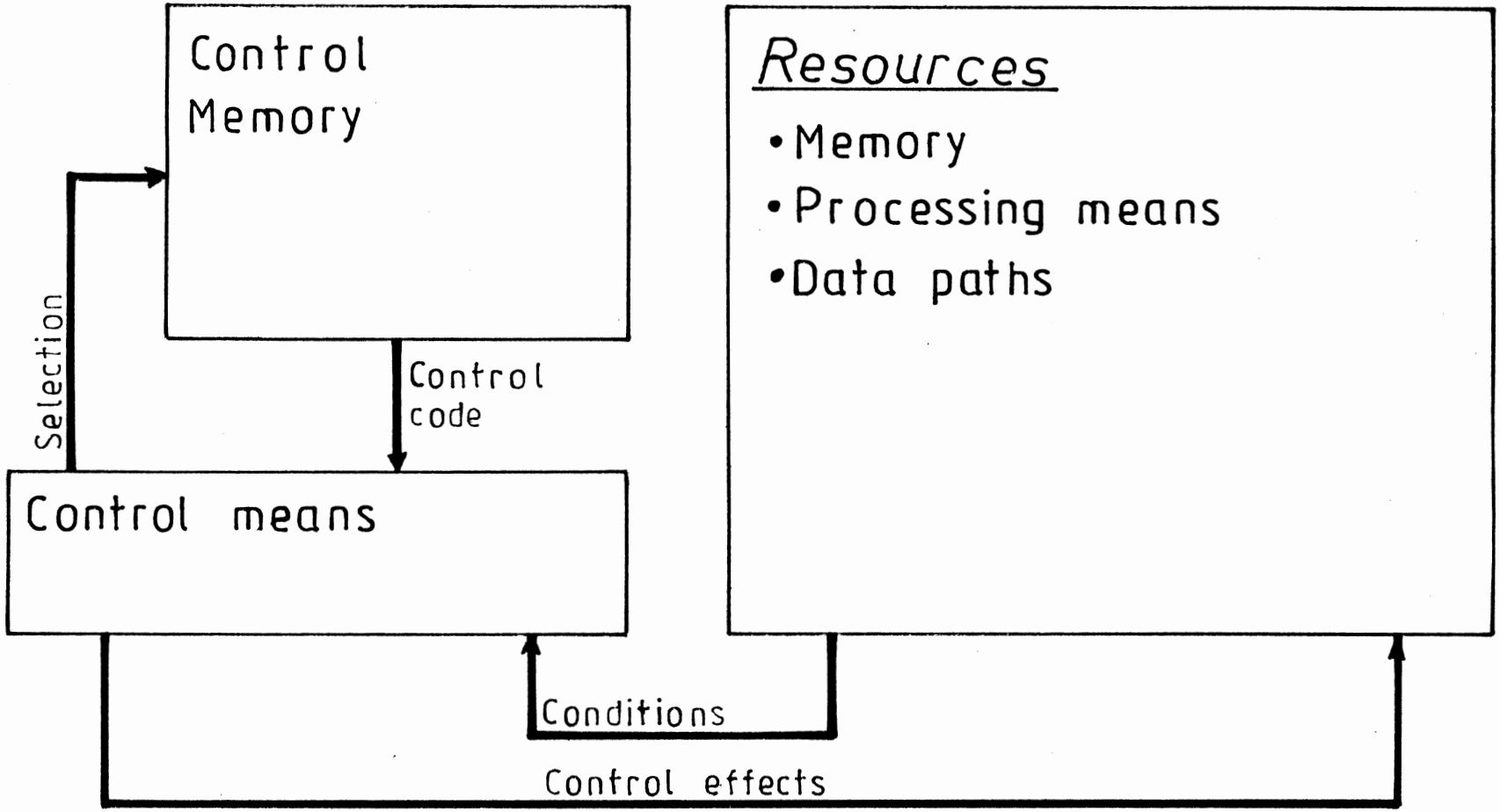


Figure 1: Computer structure as a collection of resources.

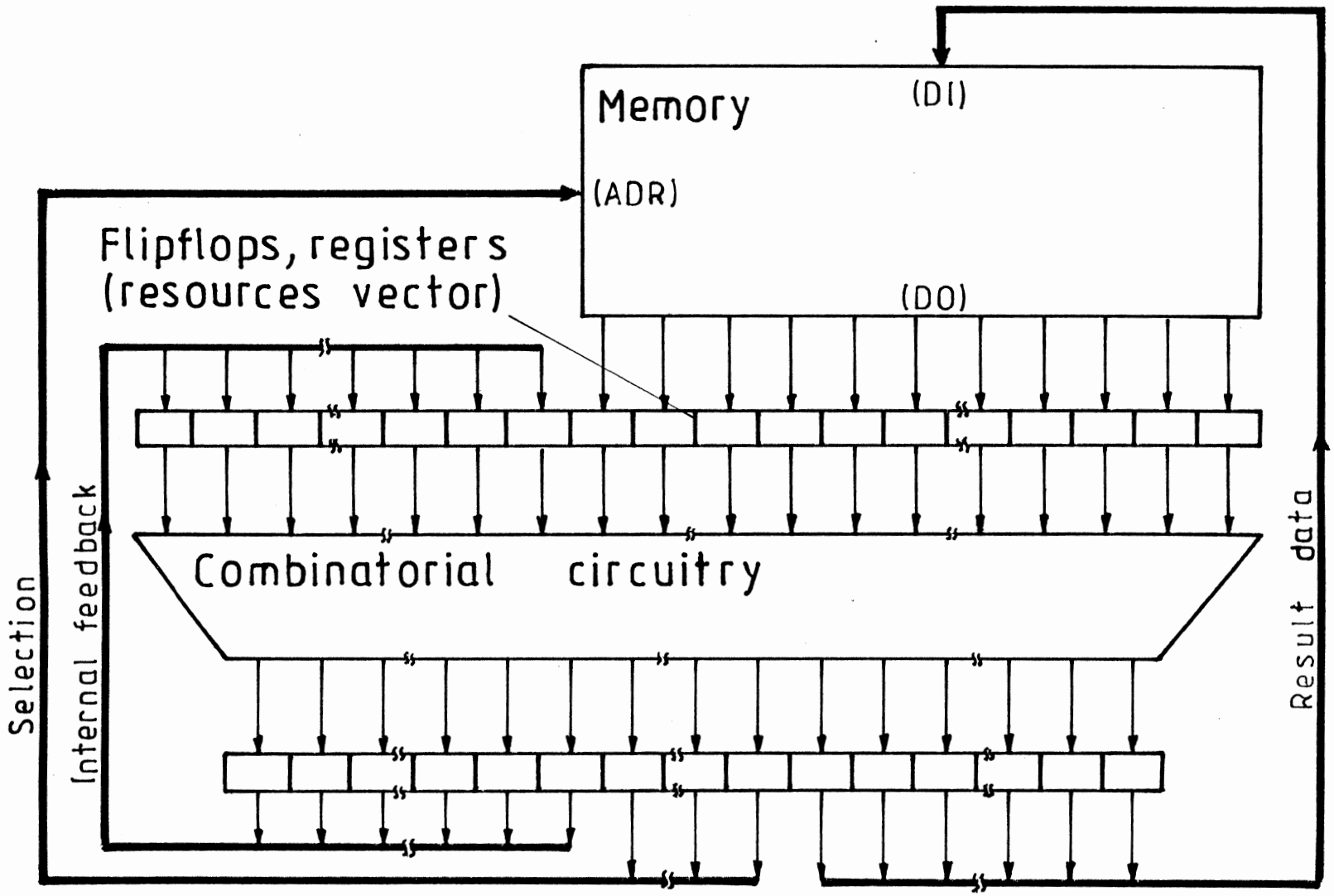
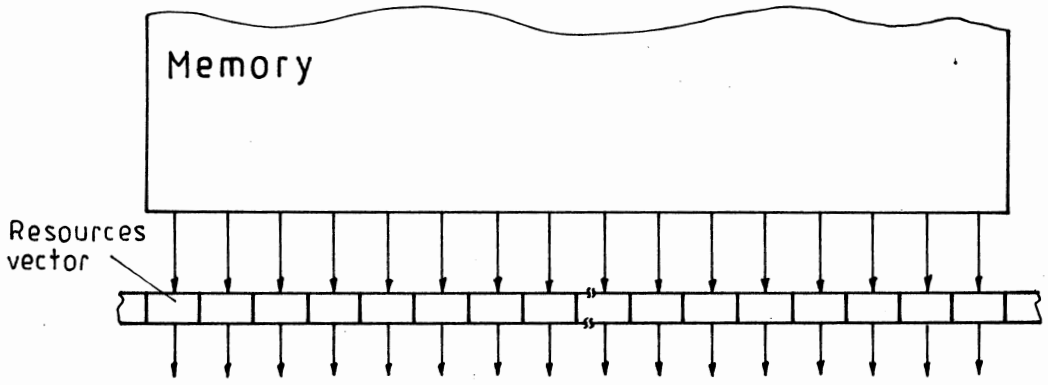


Figure 2: In-detail view: the resources vector.

a) Immediate assignment



b) Encoded selection

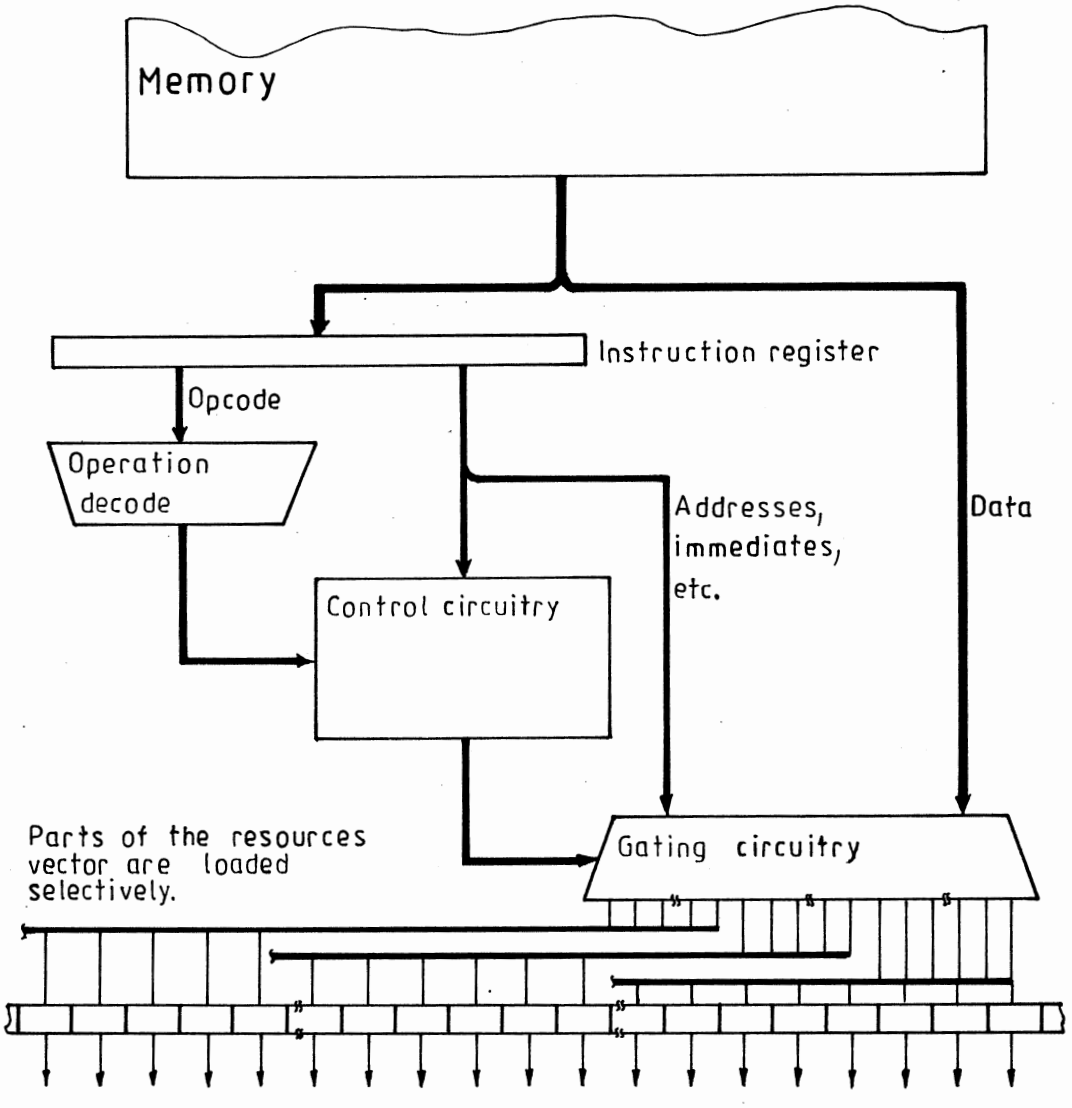


Figure 3: Alternatives: feeding the resources vector out of the memory.

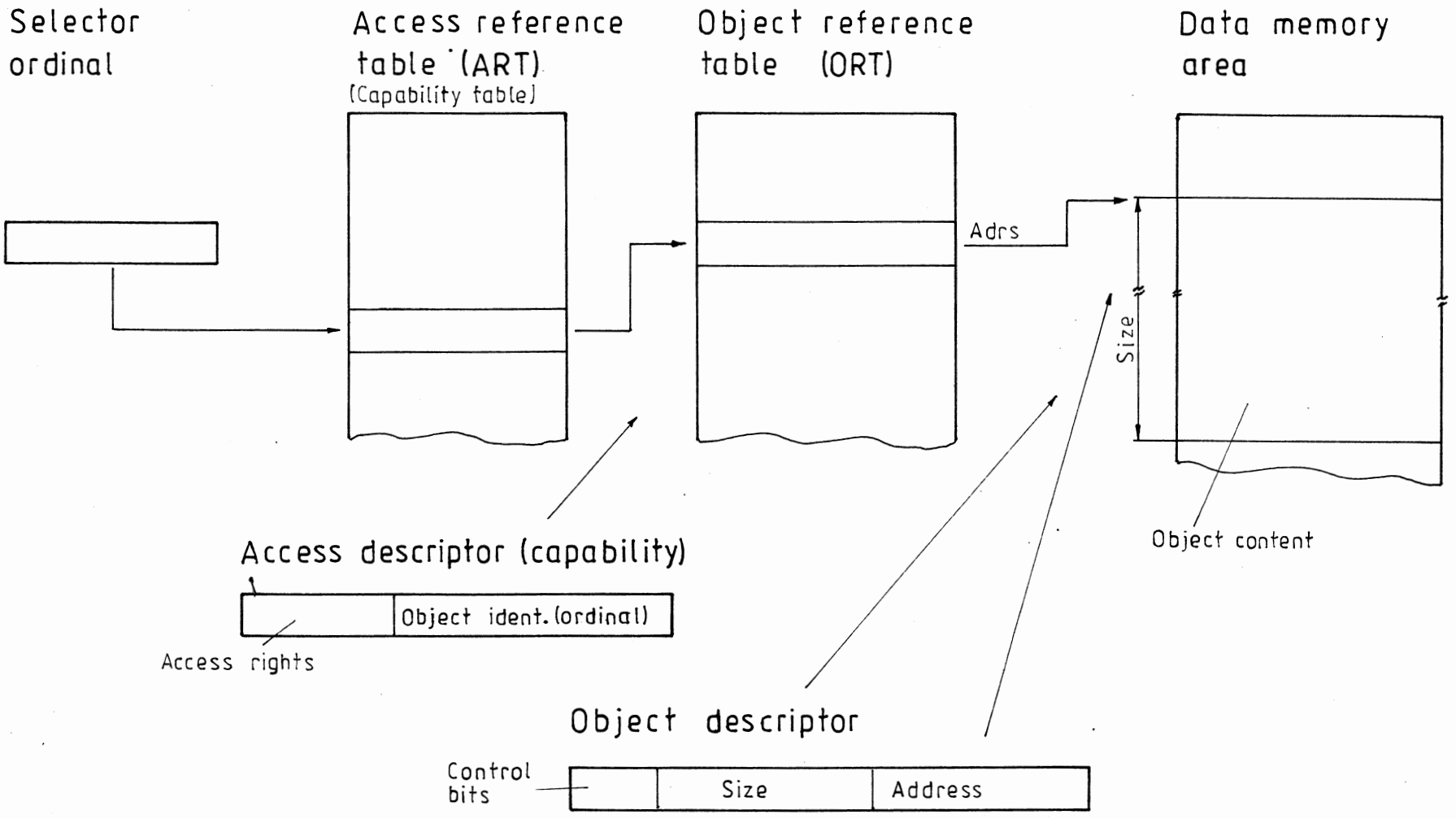


Figure 4: Object-oriented access scheme.

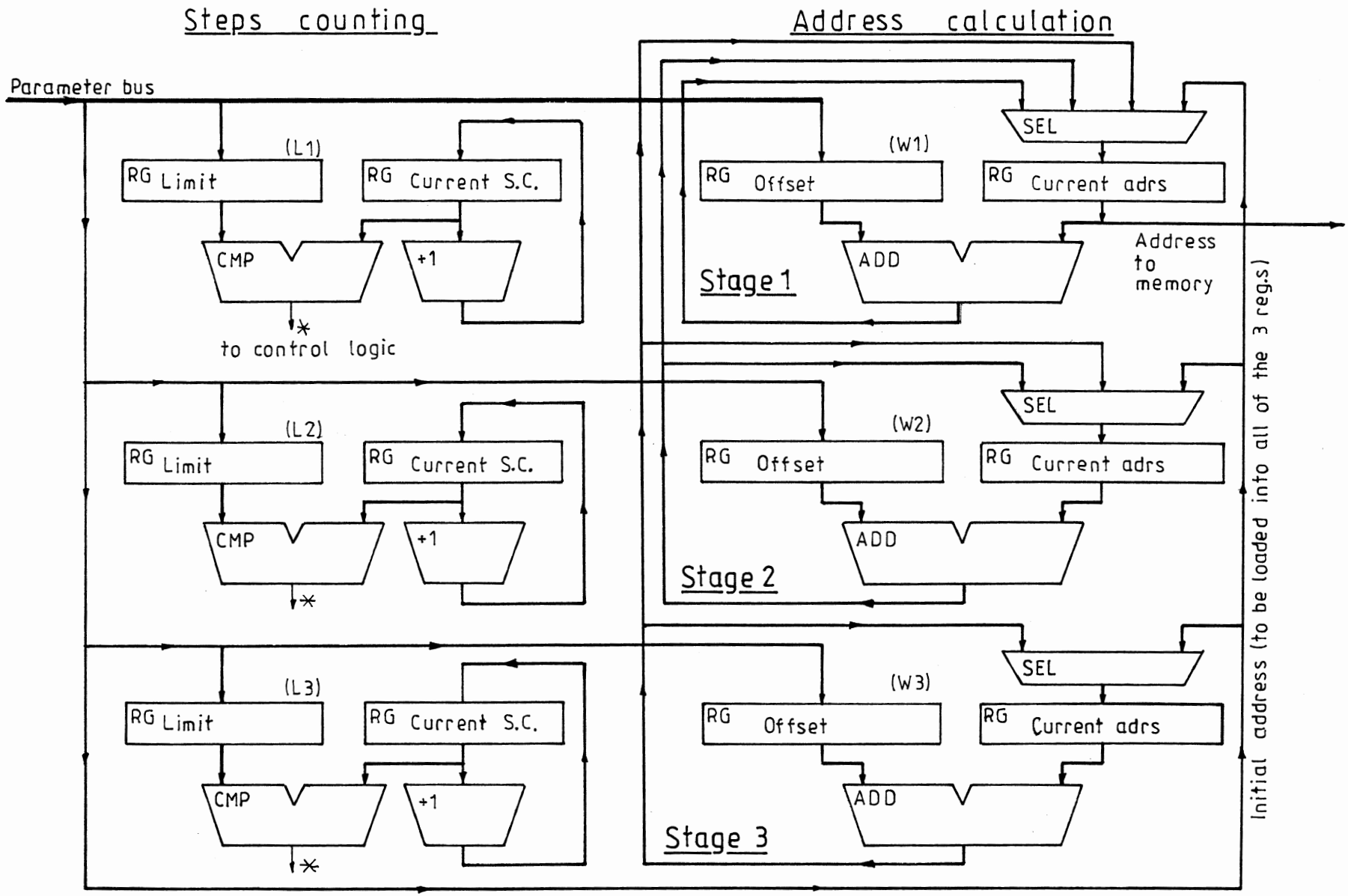


Figure 5: Iterator hardware for three nested DO-loops. (Stage 1 corresponds to innermost loop.)

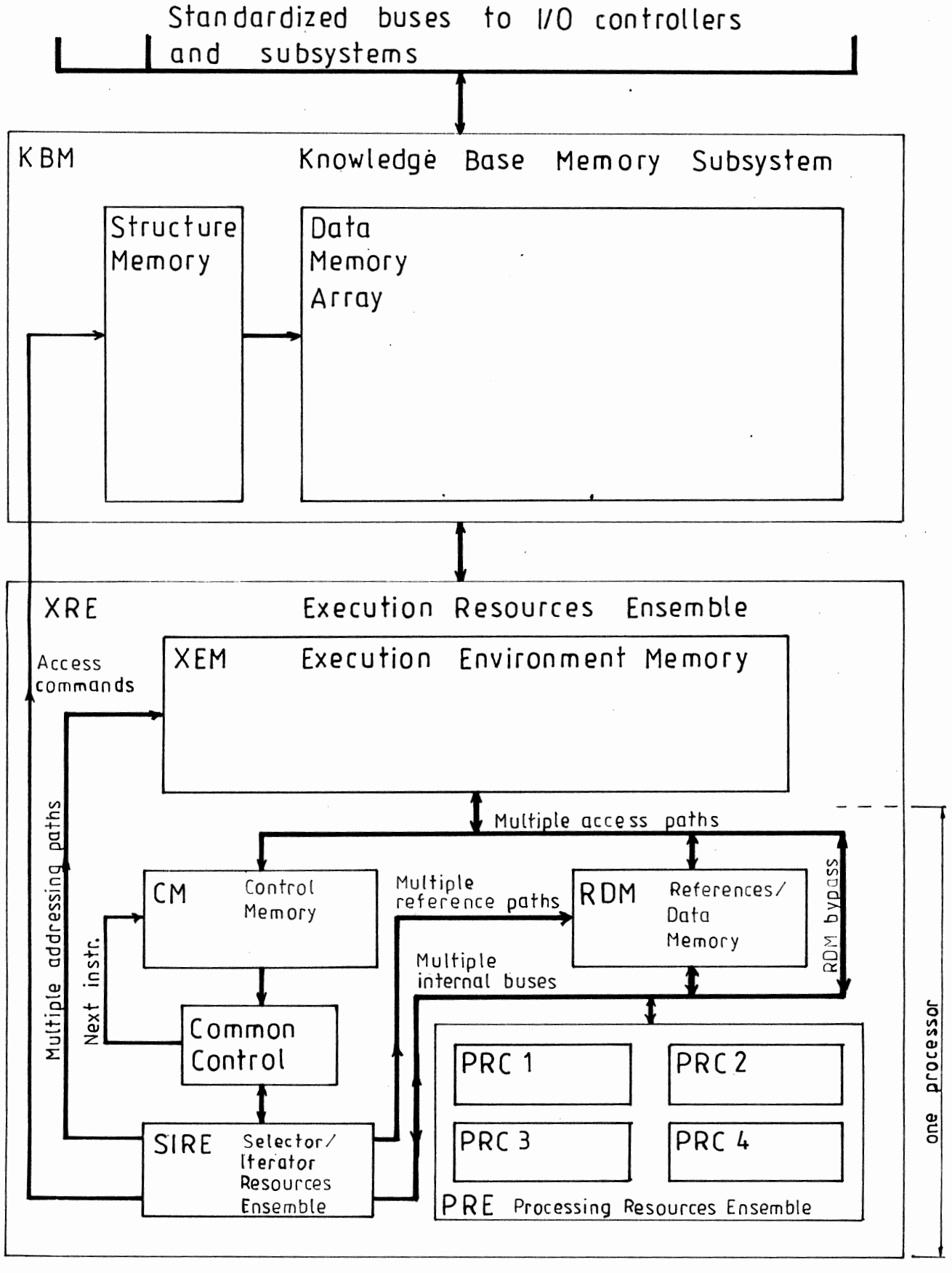


Figure 6: System overview.

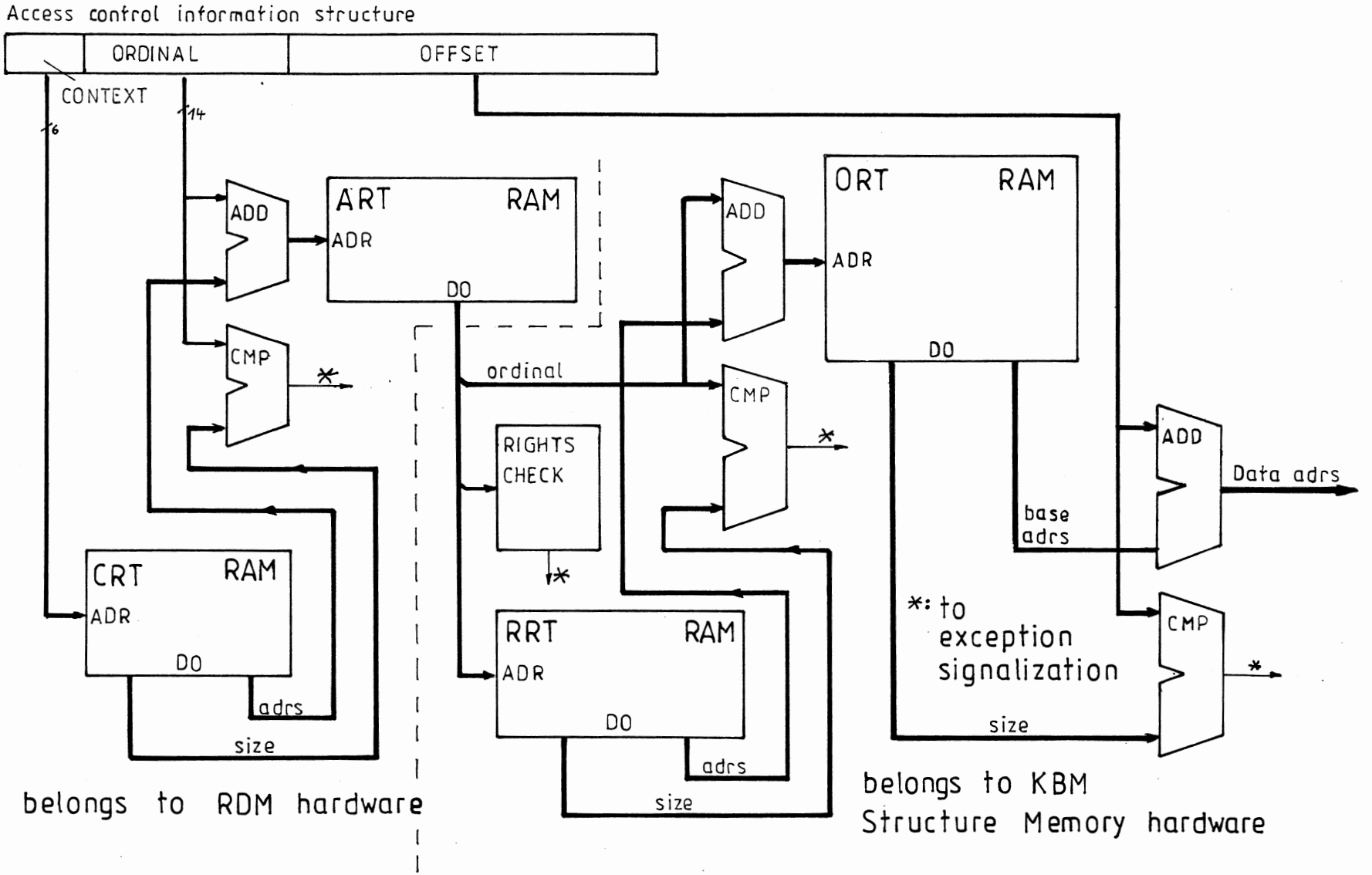
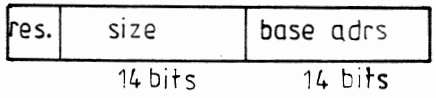
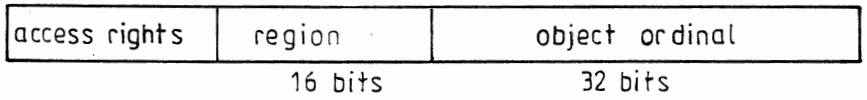


Figure 7: Hardware incarnation of object oriented data access.

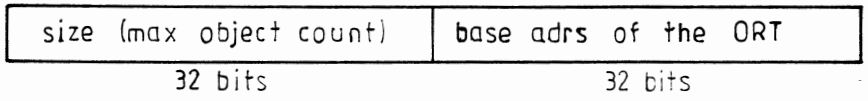
Context descriptor



Access descriptor



Region descriptor



Object descriptor

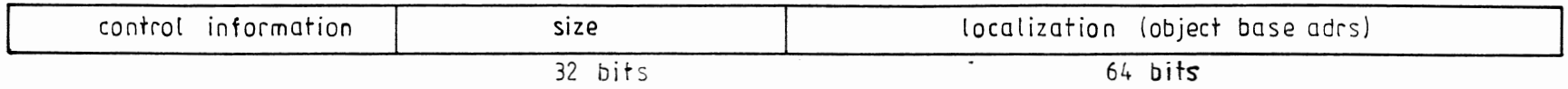


Figure 8: Information structures for object-oriented data access.

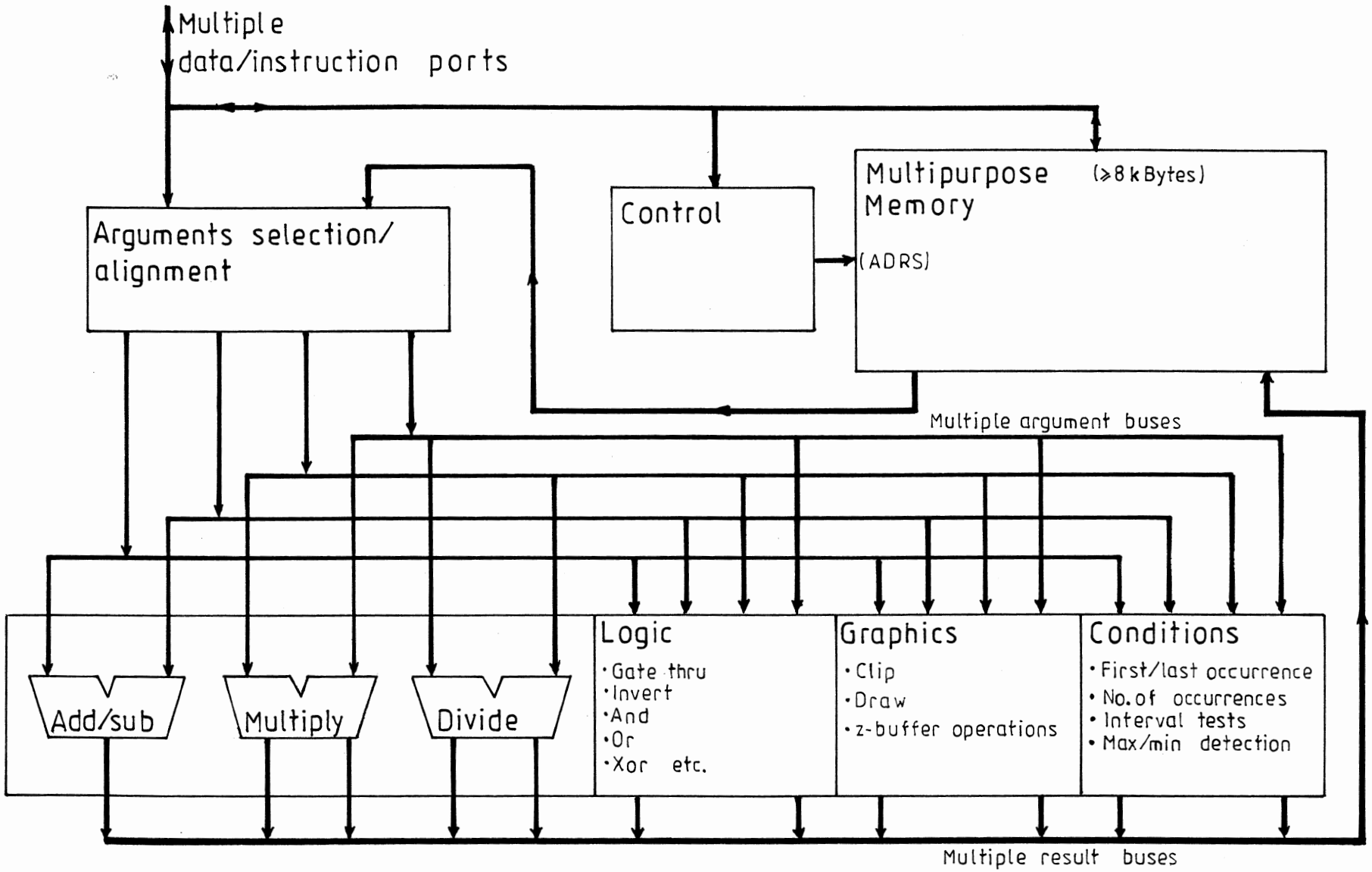


Figure 9: Processing resources collection (PRC) overview.

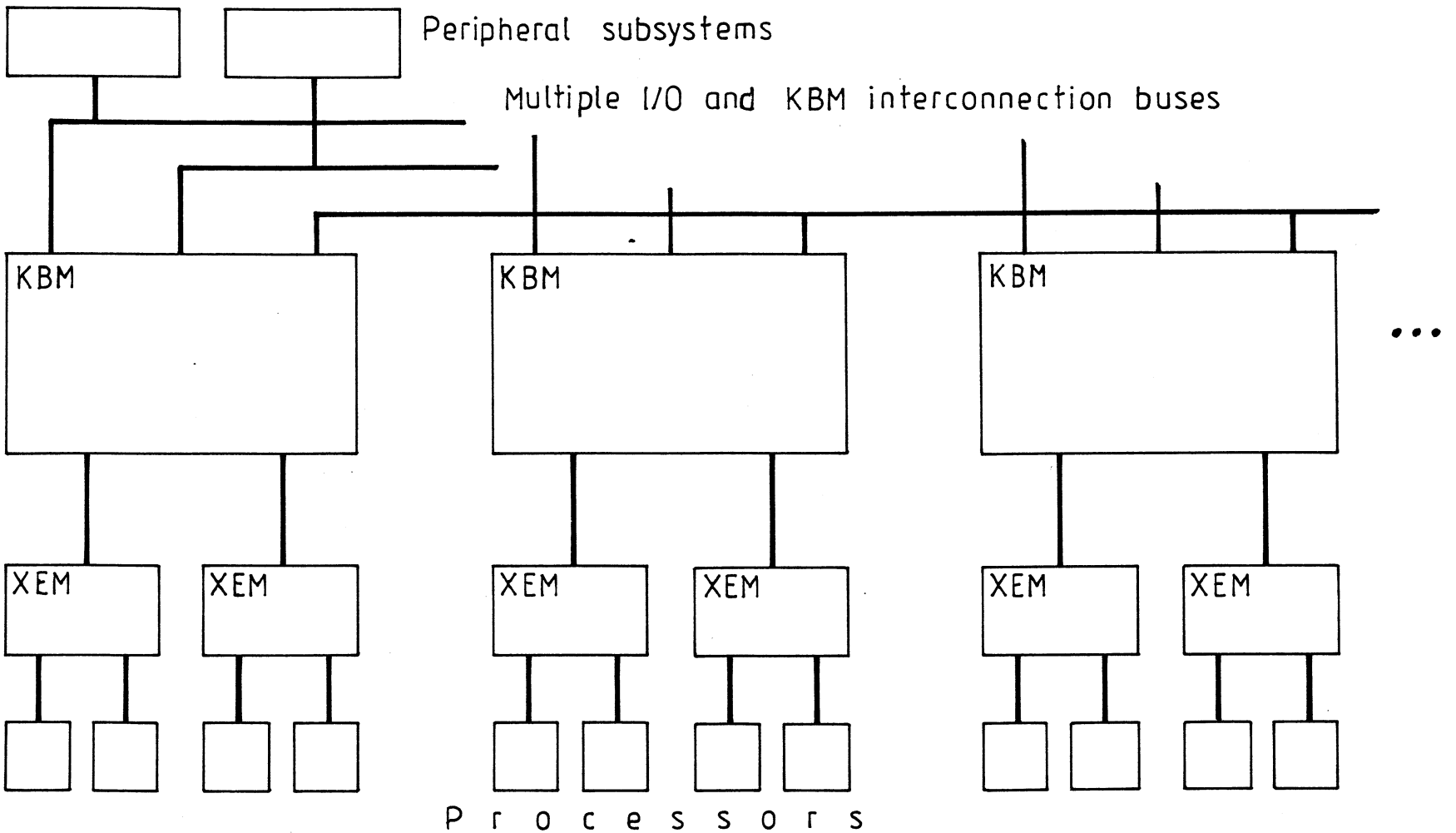


Figure 10: System expansion scheme.