2. The ReAl Computational Model

2.1 Operators and Parameters

The ReAl architecture supports the exploitation of an arbitrary number of arbitrary resources. Fig. 2.1 illustrates how a single resource is be used according to the ReAl computational model. In the example, the following steps are executed:

1. Parameters (operands) are fetched from memory and passed to the resource.
2. The operation is executed (by activating the hardware or by invoking an appropriate software routine (emulation)).
3. The result is stored in the memory (in other words, assigned to the corresponding variable).

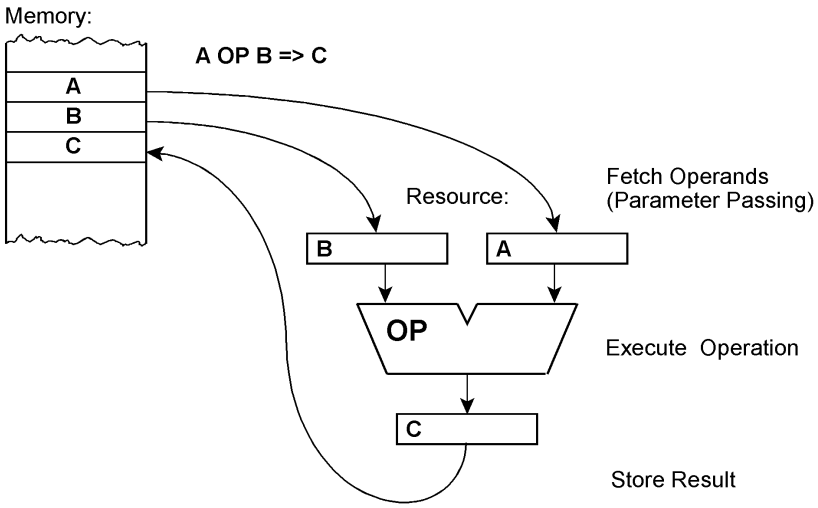

Fig. 2.1 How a single resource is used – the basic computational model.

When only one resource is considered, especially one that has only a few parameters and executes only well-known basic operations, there is virtually no difference to conventional computer architectures (fig. 2.2). It is merely the question whether the consecutive processing steps are controlled implicitly by the control unit (instruction sequencer) or explicitly by appropriate encoded commands. In this respect the ReAl architecture bears some resemblance to the principles of vertical microprogram control. The peculiar ReAl benefits however will come into effect when more than one resource is exploited and when the ReAl principles of operation are applied to the program as a whole.

Concatenation

Resources can be combined to complex arrangements. Such a configuration corresponds to the data flow diagram of the respective processing operations (fig. 2.3). The ReAl computational model implies provisions for connecting resources according to data flow diagrams and for disconnecting again these connections. In the following, such connections will be referred to as concatenations.

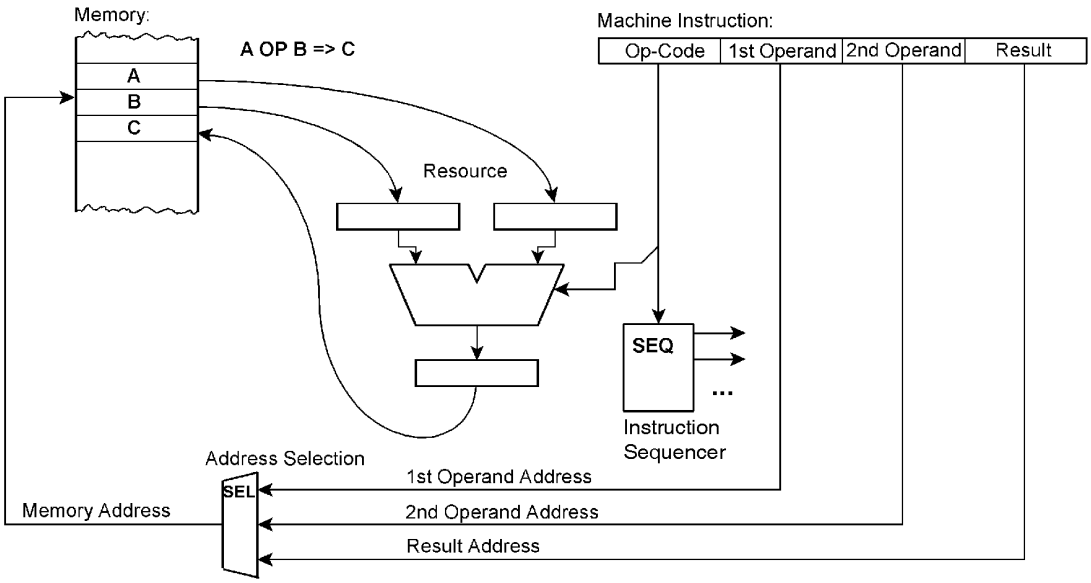

Fig. 2.2 For comparison: instruction execution in a conventional processor.

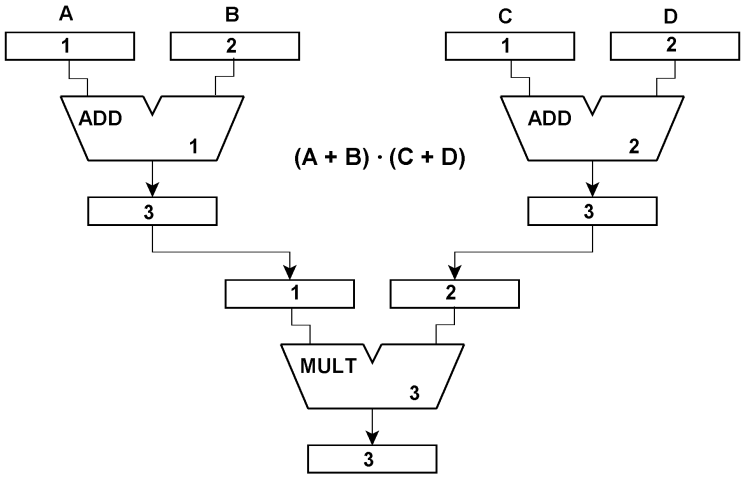

Fig. 2.3 A basic example of resources combined according to a data flow diagram: three resources have been concatenated to calculate (A + B) • (C + D).

The basic ReAl Operators

In a ReAl machine, the processing steps are controlled by stored instructions. The abstract (machine-independent) instructions are referred to as operators. There are at least eight basic types:

1. Select resources: s-operator.
2. Establish concatenations between resources: c-operator.
3. Feed resources with operands from memory (parameter passing): p-operator.
4. Initiate the information processing operations: y-operator (yield).
5. Move data between resources: l-operator (link).
6. Assign results (to variables in memory) : a-operator.
7. Disconnect concatenations: d-operator.
8. Return resources to the resource pool: r-operator.

Some basic variants:

- The concatenation is not supported at all. The operator types 2 and 7 are obsolete. The parameter transport is to be done exclusively with p-operators, l-operators and a-operators, the operation initiation exclusively with y-operators.
- Not all resources support an unlimited concatenation. When this is the case, arbitrary data flow schemes cannot be supported. In some cases, the concatenation provisions are not usable.
- The input concatenation is not supported. The concatenation can only be used for moving parameters between resources, but not for initiation of operations. Hence all operations are to be initiated by y-operators.
- The input concatenation is supported. In this case, it is possible to automatically initiate the respective operations without a y-operator. Such a resource begins – if set up appropriately – with the execution of the operation when all operands are valid, no matter in which way they are supplied (p-operator, l-operator or concatenation).

Parameter passing
The data with which the resources work are generally referred to as “parameters”. Input parameters are also referred to as operands; output parameters are referred to as results. Thre are three types of parameters:

- Inputs (operands; type IN).
- Outputs (results; type OUT).
- Combined inputs and outputs (type INOUT).

Parameters are passed in general by value. If this is not easily possible, additional resources (for example, addresing respources) must be provided in order to fetch and move the values.

Additional operators
In practice, additional operations are to be initiated and additional information is to be passed to, for example, for supporting compilers, for system administration and the like. In order to be able to provide such supporting functions in a way consistent with the basic principles of operation, some supplemental operators are introduced:

1. Hints: h-operator.
2. Meta-language support: m-operator.
3. Administrative and auxiliary functions: u-operator (utility).

2.2 Denotation of ReAl Program Operations

ReAl programs are essentially sequences of operators. ReAl program operations can be represented as follows:

- Colloquial (for example, in a cookbook-like fashion ("take a multiplier and concatenate it to a comparator" and so on). This requires no special conventions but is copious and not always free of ambiguities.

- ReAl text code. Text codes are concise formalized denotations based on simple character strings. They are used when a machine-independent and human-readable representation is required. Typical examples are documentation, debugging, program development and compiling.
- ReAl byte code. Byte codes are variable-length binary encodings. A ReAl program in byte code is essentially a string of bytes. Such codes are used when a machine-independent, but compact representation is required. Typical examples are compiling and emulation.
- ReAl machine codes. Machine codes can be laid out according to fixed-length or variable-length formats. Simple fixed-length formats resemble the well-known RISC instructions (an opcode followed by some parameter fields containing ordinal numbers or addresses). Alternate formats may contain many fields to control many resources in parallel, resembling VLIW instructions or horizontal microinstructions.

In the following descriptions a simple text code will be employed[1)]. The basic syntax:

- Designators (of resources, resource types and parameters) may be ordinal numbers or symbolic names. The primary designation is by consecutive ordinal numbers (1st, 2nd etc. resource, 1st, 2nd etc. parameter and so on). To those ordinal numbers, symbolic names can be assigned.

- Parameters are enumerated consecutively: first the inputs, then the inputs and outputs, then the outputs (fig. 2.4). Each parameter is designated by its ordinal number.

- Resources are enumerated consecutively. Each resource is designated by its ordinal number. The ordinal numbers are assigned according to the sequence of s-operators. The assigned ordinal numbers remain valid even when intermediately resources with lower numbers have been returned (r-operator).

- Assignment of a symbolic name to an ordinal number (more details see below):

  ordinal number **:** *symbolic name*.

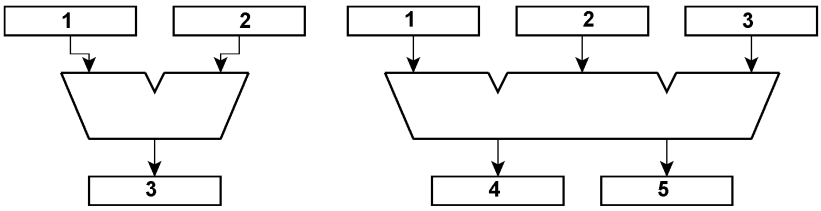

Fig. 2.4 Resources with enumerated parameters.

- Spaces can be inserted or omitted as needed. A line feed does not matter.

- The assignment symbol is := (the colon designates the destination side).

---

1): This text code is essentially intended for concisely describing architectural principles and as an intermediate (internal) formalized language during program development and compiling. Therefore, user-friendliness is not that important. The text code is not intended to be a new programming language for application programmers.

- Parameter passing is denoted by => (the arrow symbolizes the direction).
- The comment symbol is -- (refer to Ada and VHDL).
- Each operator is designated by a particular mnemonic character:

  operator_ name = s | c | p | y | l | a | d | r | h | m.

- An operator within a program: *operator_name (argument_list).*
- A parameter (operand or result) of a particular resource will be represented as follows:

  resource . parameter.

- Designation of variants of the operators: by additional abbrevaitions that are separated by an underscore (for example, s_a; p_imm or u_rs2).

Assignment of symbolic names:

- Resource types: Symbolic names are usually predefined in the resource type table or in the reference manual of the particular ReAl machine. Those names can be overloaded by appropriate u-operators:

  Assign resource type name: *u_tn*.

  u_tn (ordinal number : symbolic name).

  Rename resource type: *u_trn*.

  u_trn (previous symbolic name : new symbolic name).

- Parameters: Symbolic names are usually predefined in the resource type table or in the reference manual of the particular ReAl machine. Those names can be overloaded by appropriate u-operators:

  Assign parameter name: *u_pn.*

  u_pn (ordinal number : symbolic name).

  Rename parameter: *u_rrn.*

  u_prn (previous symbolic name : new symbolic name).

- Resources: Symbolic names can be assigned during invocation of the desired resource type (s-operator) or by appropriate u-operators:

  Assignment in the parameter string of the s-operator: *resource type : symbolic name.*

  Assign resource name: *u_rn.*

  u_rn (ordinal number : symbolic name).

  Rename resource: *u_rrn.*

  u_rrn (previous symbolic name : new symbolic name).

2.3 Basic Operators

1. Select resources: s-operator

s (list of resource type identifiers)

s (1st resource type, 2nd resource type an so on)

The s-operator is used to request resources of certain types out of the resource pool.

The effect of a s-operator depends on the requested resources as well as of the kind of the underlying hardware:

- An appropriate hardware resource (in other words, a function unit (like an ALU)) is reserved, initialized and assigned.
- Appropriate memory areas are reserved, initialized, and assigned. Optionally, the respectively required control information is loaded (programs, microprograms, net lists, Boolean equations and the like).
- The requested resource is built from other resources (recursion).
- An appropriate hardware structure is generated, for example, by programming cells and connections on a programmable integrated circuit.

Initializing a resource means to set up literals and initial values, to adjust the data width, to load control information (like microprograms) and so on. Assigning a resource means to incorporate it into the administration of the selected resources so that it can be accessed by subsequent operators under ordinal number or by addressing.

Resource type identifiers
For conventional (generic) resources, the resource type identifier is the appropriate type designator (as given, for example, in the particular reference manual). Special resources are designated by their particular symbolic name. Generally, Internet addresses and the like can also be used as identifiers.

Enumeration of resources
The requested resources are enumerated consecutively. Subsequent operators then refer to the selected resources by those ordinal numbers (or by the assigned symbolic names).

2. Establish concatenations between resources: c-operator

c (list of concatenations)

c (1st source resource **.** *1st result => 1st destination resources* **.** *1st parameter, 2nd source resource* **.** *2nd result => 2nd destination resource* **.** *2nd parameter and so on)*

Concatenation means to connect an output (result parameter) of the source resource with an input (operand parameter) of the destination resource.

The c-operator loads concatenation control information (like address pointers) into the resources[1)]. In some implementations the operator will initiate the setup or programming of appropriate physical connections (for example, within a switch fabric or an FPGA). If concatenation is exploited to the extreme, the concatenated resources constitute a structure which corresponds to the dataflow graph of the application problem.

Input concatenation
When supported appropriately, it is possible to concatenate inputs with one another (input concatenation). Such a concatenation corresponds to connecting corresponding inputs in parallel. Application: for supplying simultaneously parameters to several resources.

3. Feed resources with operands (parameter passing): p-operator

p (source-to-destination list). Source are variables in system memory or the like, destinations are parameters of resources.

p (1st variable => resource **.** *parameter, 2nd variable => resource* **.** *parameter and so on)*

The p-operator moves the specified variables (for example, from system memory or an I/O address space) into the specified operand parameter positions of the specified resources. The variables are designated by names, ordinal numbers or addresses.

In resources that support concatenation appropriately, p-operators can also initiate the execution of operations (processing begins when all operands are valid).

4. Initiate the information processing operations: y-operator

y (list of resources)

y (1st resource, 2nd resource and so on)

A y-operator initiates the execution of operations in the specified resources. What the respective resources will do, depends on directly from the type of resource (when it can perform only a single function) or on parameters (function codes) that have to be set beforehand (for example, by means of s-operators or p-operators).

---

1): The resources are to be concatenated before the corresponding operations are initiated (y-operator).

An alternative method of operation initiation – without a y-operator – can be applied when the resource supports concatenation. The execution of an operation will be initiated if all required operands are valid. Valid operands can be supplied by means of p-operators or l-operators or by concatenation.

Contrary to conventional instruction sets, the selection of the operation to be executed (s-operator) has been separated from the initiation of the operation (y-operator or concatenation). Hence the resources know in advance for which purpose the operands are destined. The initiator code (within y-operators) is typically shorter than a conventional operation code. This avoids instruction traffic during execution and can be an advantage if more than one function is to be initiated (appropriately formatted y-operators can initiate more operations simultaneously than conventional instructions of same length).

5. Move data between resources: l-operator

l (source-to-destination list). Sources and destionations are parameters of resources.

l (1st source resource . 1st result => 1st destination resource . 1st parameter, 2nd source resource . 2nd result => 2nd destination resource . 2nd parameter and so on)

The l-operator moves parameters between the specified resources (from the output (result parameter) of the source resource to the input (operand parameter) of the destination resource, respectively).

In resources that support concatenation appropriately, l-operators can also initiate the execution of operations (processing begins when all operands are valid).

6. Assign results: a-operator

a (source-to-destination list). Sources are parameters of resources, destinations are variables in system memory or the like.

a (1st resource . 1st result => 1st result variable, 2nd resource . 2nd result =>2nd result variable and so on)

The a-operator moves the contents of the specified result parameter positions of the specified resources to the specified variables (for example, in system memory or in an I/O address space). The variables are designated by names, ordinal numbers or addresses.

7. Disconnect concatenations: d-operator

d (list of concatenations)

d (1st source resource . 1st result => 1st destination resource . 1st parameter, 2nd source resource . 2nd result => 2nd destination resource . 2nd parameter and so on)

The d-operator disconnects existing concatenations. In some implementations (for example, in FPGAs) it can cause the corresponding physical connections to be changed or cut off (by reprogramming). Disconnected resources can be used separately or can be concatenated again.

8. Return resources to the resource pool: r-operator

r (resource list)

r (1st resource, 2nd resource and so on)

The specified resources are returned to the resource pool. They are therefore available to be used for other processing tasks.

2.4 Basic Examples

The following example illustrates how a programming intention can be implemented.

- The programming intention: to compute X := (A + B) • (C + D).
- Available resource types: ADD, MULT.

Fig. 2.5 illustrates the corresponding resource configuration comprising two adders (ADD) and a multiplier (MULT). The ordinal numbers of the resources: first adder = 1, second adder = 2, multiplier = 3. The ordinal numbers of the parameters of a resource: inputs (operands) = 1 and 2, result = 3.

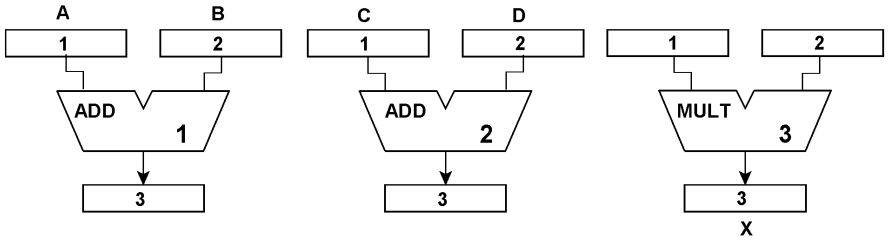

Fig. 2.5 A resource configuration to compute X := (A + B) • (C + D).

The notation at full length (each step individually):

s (ADD)
s (ADD)
s (MULT)
p (A => 1.1)
p (B => 1.2)
p (C => 2.1)
p (D => 2.2)
y (1)
y (2)
l (1.3) => 3.1)
l (2.3 => 3.2)
r (1,2)
y (3)
a (3.3 => X)
r (3)

An abbreviated notation:

```
s (ADD, ADD, MULT)
p (A => 1.1, B => 1.2, C => 2.1, D => 2.2)
y (1, 2)
l (1.3 => 3.1, 2.3 => 3.2)
r (1, 2)
y (3)
a (3.3 => X)
r (3)
```

The resources connected according to the data flow (concatenation; refer to fig. 2.3):

```
s (ADD, ADD, MULT)
c (1.3 => 3.1, 2.3 => 3.2)
p (A => 1.1, B => 1.2, C => 2.1, D => 2.2)
y (1, 2, 3)                    -- begin of processing in the concatenated resources
a (3.3 => X)
r (1, 2, 3)
```

The y-operator is not needed when the resources support an appropriate advanced concatenation mode.

2.5 Some additional operators

1. Select a resource and assign a particular ordinal number: s_a-operator

s_a (list of pairs resource type identifier => resource ordinal)

s_a (1st resource type => 1st resource ordinal, 2nd resource type => 2nd resource ordinal and so on)

This variant of the s-operators assigns particular ordinal numbers to the requested resources. Depending on the implementation, similar operators can be provided to assign addresses in a resource address space instead of the ordinal numbers.

2. Load immediate values (literals) into resources: p_imm-operator

p_imm (list of pairs literal => destination). The destinations are parameters of resources.

p (1st literal => resource . parameter, 2nd literal => resource . parameter and so on)

The p_imm-operator moves immediate values (literals) into the specified operand parameter positions of the specified resources.

In resources that support concatenation appropriately, p_imm-operators can also initiate the execution of operations (processing begins when all operands are valid).

3. Initiate selected functions: y_f-operator

y_f (list of pairs resource . function code)

y_f (1st resource . 1st function code, 2nd resource . 2nd function code and so on)

A y_f-operator initiates the specified functions in the specified resources.

This variant contradicts the principle of encoding in the operators only the most basic processing steps but not concrete machine operations. It is some kind of stopgap minimalist solution (suitable, for example, for small FPGAs, microcontrollers and the like).

4. Provide hints to support speculative activities: h-operator
Hints (h-operators) can cause variables or program pieces to be loaded speculatively into cache memories so that, when required, they are already available[1)]. Additional h-operators can be provided in order to indicate future demand in regard to certain resource types or certain configurations of resources. Such hints can be used, for example, to select such resources out of the resource pool which are, in regard to a subsequent concatenation, conveniently located on the integrated circuit.

5. Provide information for compiling and other activities of program generation: m-operator
Meta-language operators (m-operators) concern the setup of resource configurations, the conditional execution of ReAl programs and the like. Such operators are similar to the pre-processor and compiler directives of conventional programming languages. However, they can become active not only at compile time but also at run time. A typical application: as a function of which resource types are available, one of several alternative branches of a ReAl program is selected in order to execute a certain programming task. Conventional conditional branches depend on processing results, operand values and so on. Meta-language caused branching depends, for example, on the type and number of available resources. The m-operators can access and change the contents of the table structures of resource administration.

6. Auxiliary and aministrative functions: u-operator
All those functions that are required during program execution but cannot be encoded with operators s, c, p, y, l, a, d, r, h, m are encoded with u-operators (in other words, u-operators are some kind of stopgap provision).

Implementation of h-, m- and u-operators
The functions that are encoded with h-, m-, and u-operators can be provided by means that are outside of the resource pool. This can be, for example, a conventional general-purpose computer that administers and controls the pool of processing resources. Appropriate functional units are generally referred to as *platforms*.

Alternatively, it is possible to specifically provide for many of these functions additional resources or to configure, based on already present resources, corresponding resource arrangements ad hoc, for example, resources that fill speculatively cache memories, reserve other resources, or administer

---

1): The principle of filling caches speculatively is well known and implemented in some high-performance processors. Therefore, it must not be described in detail.

resource tables. Basically, the platform outside of the resource pool can be restricted to the most elementary functions of instruction fetch, initialization and the like. All other functions can be implemented by basic ReAl operators employing a sufficiently equipped resource pool. Therefore, a more precise description of the h- and m-operators is not required here. Some u-operators will be explained in more detail when necessary.

2.6 Operators and Instructions

Operators describe the basic steps of information processing. Machine-independent ReAl programs are sequences of operators. To be stored and executed on ReAl machines, operators have to be encoded. There are several alternatives:

- One machine instruction corresponds to one operator.
- One operator requires more than one machine instruction. An instruction encodes only an elementary processing step (for example, loading of a parameter address). The functions of the ReAl operators are emulated with sequences of corresponding instructions (resembling a conventional vertical microprogram control).
- One instruction contains more than one operator (resembling the conventional VLIW instructions).
- Control words that contain individual control bits as well as literal fields, address fields, and control fields. Such control words serve primarily for supplying with parameters, activating and so on a large number of resources at once (resembling a conventional horizontal microprogram control).
- Instructions that are similar to the machine instructions of conventional architectures.
- The operators are converted into sequences of conventional machine instructions or corresponding function calls (for example, by means of compiling).

2.7 ReAl and Conventional Programming Languages

Fundamentally, ReAl programs can be likened to manufacturing or machining instructions[1)] – ReAl programming means just to plan ahead. Which manufacturing steps are to be executed? Which tools and machines are necessary? Which part has to be supplied to which machine in the course of time? No engineer would begin designing cars, ships and so on writing down instructions of this kind. Analogously, a programmer will not use a ReAl text code for jotting down his programming ideas.

Instead, ReAl programs will be generated automatically from source programs written in higher-level languages. Machine-independent ReAl codes can be seen as intermediate languages, similar to the well-known Java byte code (table 2.1). However, the goal is not code compactness but to describe precisely the inherent parallelism and essential intricacies of program operation. In this respect, ReAl may be better compared to Postscript than to Java.

---

1): Something like "To manufacture this gearbox, we will need three lathes, five milling machines and so on. Part No. 33 will be machined on lathe No. 2 and then finished on grinding machine No. 6."

Some variants related to compiling and program execution:

- Programs written in a conventional programming language will be compiled into machine-independent ReAl code.
- Programs written in a new ("ReAl-aware") programming language will be compiled into machine-independent ReAl code.
- The machine-independent ReAl code will be executed by an appropriate emulator.
- The machine-independent ReAl code will be compiled into the machine code of a conventional processor (in other words, the ReAl code is used merely to detect the inherent parallelism).
- The machine-independent ReAl code will be compiled into ReAl machine code.

Java, JVM	ReAl
• Code compactness (bytecode) • Developed for small programs (applets) • Executable on thin machines • Programs to be downloaded via internet • JVM is a conventional stack machine, hence its operations are inherently sequential • JVM bytecode describes one operation at on time, hence inherent parallelism is to be detected during runtime	• To make best possible use of hardware • Developed for large and computing-intensive programs (graphics, equation solving, simulation, data bases, neural networks, AI) • There will always be enough hardware. Memory capacity and code size are irrelevant • Executable on machines which can be built with future IC technology (dozens or even hundreds of operation units on one integrated circuit) • ReAl code describes completely the inherent parallelism of program operation • Creation of virtual special processors which correspond to the dataflow graph of the application problem • Inherent parallelism will be detected not during runtime but in statu nascendi (in other words, by examination of the programming intentions) • A sufficiently standardized ReAl instruction set is a unified machine language, which can describe hardware as well as software

Table 2.1 Java Virtual Machine (JVM) vs. ReAl

2.8 Processing Resources

Processing resources comprise memory means for the parameters (operands and results) and appropriate processing circuitry. They differ primarily in:

- The number of operands and results.
- The type of data structures of the operands and results.
- The supported operations.
- The processing width (the number of bits to be processed in parallel).
- The method of parameter passing (for example, by value or by reference).
- The concatenation support (principles of operation and supported parameters).
- The relation to the processing state (whether the resources are stateless or parameters are included in the processing state or program context, respectively).
- The auxiliary functions (range checking, measuring of the frequency of utilization and so on).

2.8.1 Basic Resources

Fig. 2.6 illustrates a simple resource that executes only one particular type of information processing operation (operation fixed, processing width (number of bits) fixed). The memory means for operands and results are typically implemented as registers. The parameters (operands and results) are passed to by value. The most basic resources support no concatenation. Operators that can be applied are: p, y, a, l.

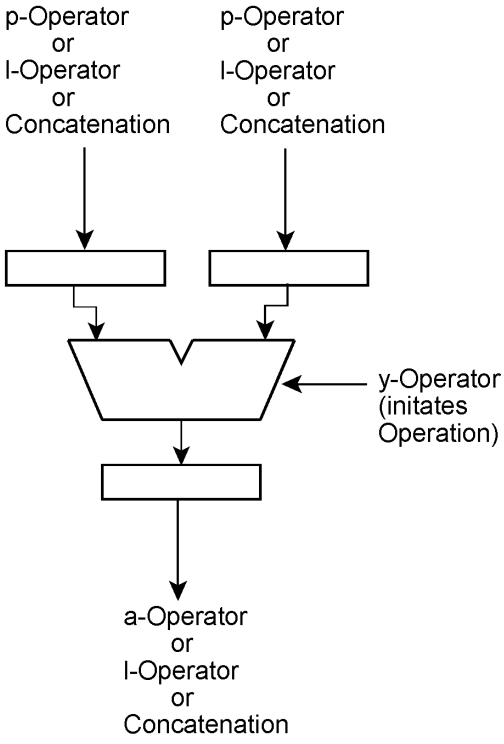

Fig. 2.6 A basic processing resource.

When the parameter passing is to be supported by reference, the corresponding address registers and access paths to the memory are to be provided. There are two principal alternatives to implement parameter passing by reference:

- The circuitry for addressing and data transport is incorporated into the processing resources (fig. 2.7).
- The means for addressing and data transport are provided as separate addressing resources (fig. 2.8).

Appropriate circuitry comprises addressing provisions and means for memory access control. Addressing provisions can be implemented according to well-known principles, for example:

- As address registers.
- As address registers with counting operation (increment, decrement).
- As address calculation units (base + displacement or the like).
- As iterator circuitry (for example, for consecutively accessing variables in typical FOR loops).

How the memory access control means are implemented depends on the respective memory interface (bus, switching hubs or the like). Some variants have also means for data buffering (like buffer registers or FIFOs).

2.8.2 Data addressing

The processing resources need access to the data stored in memory. Operands are to be fetched and results are to be stored. Principally, there are two ways of implementing these functions:

- Dedicated addressing resources are concatenated to processing resources (fig. 2.7).
- Appropriate addressing and memory access capabilities are provided within the processing resources (fig. 2.8).

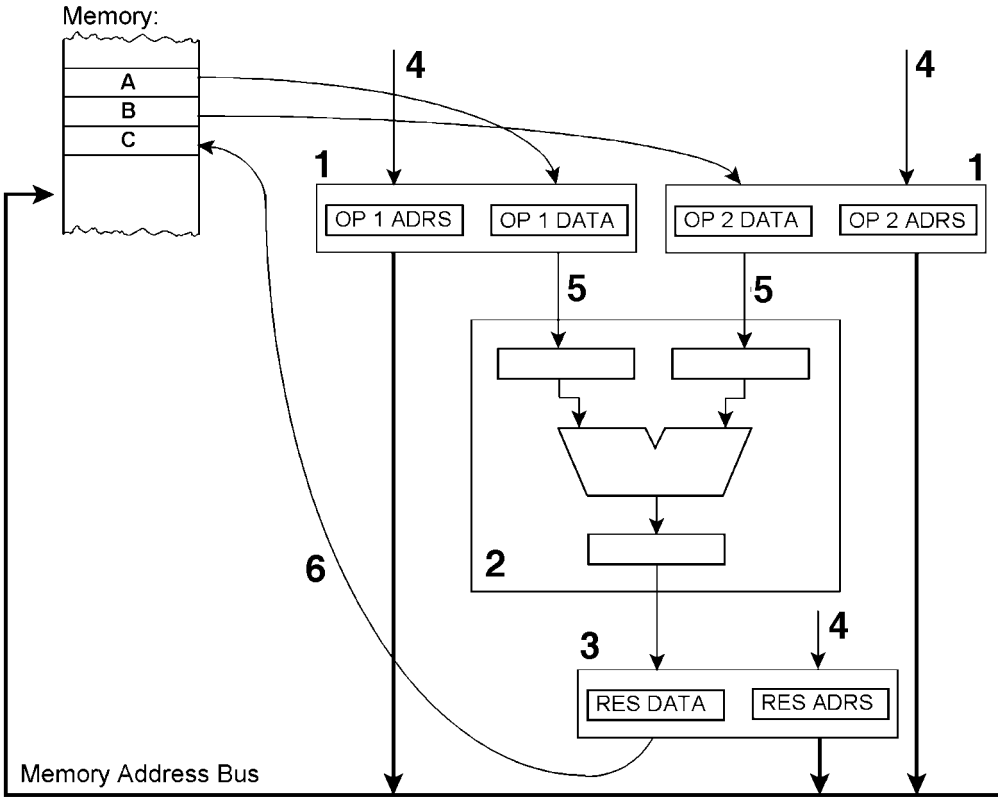

Fig. 2.7 Addressing resources concatenated to a processing resource. 1 - operand addressing resources; 2 - processing resource; 3 - result addressing resource; 4 - address parameters delivered by p-operators, l-operators or preceding concatenations; 5 - adressing resources deliver operands fetched from memory; 6 - adressing resource stores result into memory.

Basic addressing resources have a memory address and a data word as parameters. The address parameter is held in an address register, the data parameter in a data register. The address is an operand. In case of read operations, the data parameter is a result (to be delivered to the processing resource). In case of write operations, the data parameter is an operand (to be passed from the processing resource). Usually, addressing and processing resources are not hard wired together. Instead, the resources will be concatenated on the fly (c-operators). The simple concatenated configuration, depicted in fig. 2.7, operates as follows:

- Operand and result addresses are loaded into the address registers of the addressing resources 1 (p-operators or l-operators or concatenation).
- Y-operators or the arriving of address parameters (caused by concatenation) activate the operand addressing resources 1. They initiate read accesses to memory.
- Once the data registers (OP 1 DATA, OP 2 DATA) within the operand addressing resources 1 have received the memory data, the concatenation to the operand registers within the processing resource becomes effective. The operand values are passed to the processing resource 2. As soon as the last operand value has been received, the processing resource 2 initiates its processing operations.
- When processing resource 2 has completed its operations, the concatenation of its result register to the data register of result addressing resource 3 (RES DATA) becomes effective.
- The delivery of the result causes the result addressing resource 3 to initiate a write access in order to transfer the data register contents into the memory.

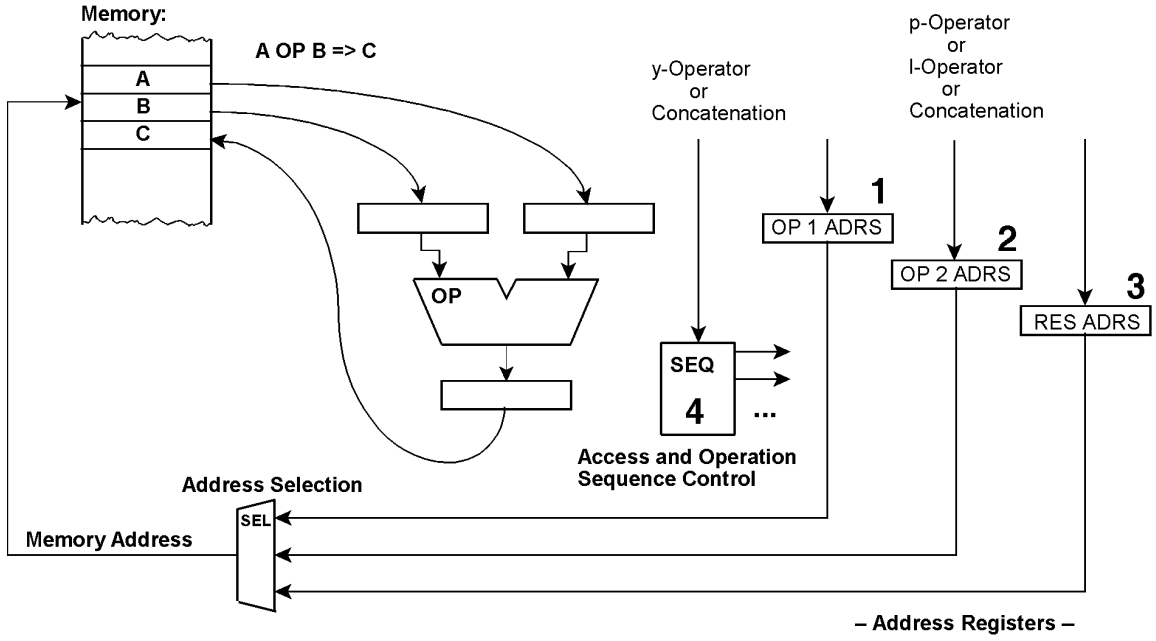

Fig. 2.8 A basic processing resource supporting parameter passing by reference. 1, 2 - operand address registers; 3 - result adress register; 4 - sequencer.

According to fig. 2.8, the addressing and memory access provisions can be built into the processing resources. Such a resource operates as follows:

- The operand addresses are written into the operand address registers 1 and 2, the result address into result address register 3 (by means of p-operators or l-operators or concatenation).
- Y-operators or the arriving of address parameters (caused by concatenation) causing the sequencer 4 to be activated, thus initiating the processing operations.
- To fetch the operands, the sequencer 4 launches two read access cycles, using the operand address registers 1 and 2, respectively.
- The operands read are entered into the operand registers of the resource.
- The result is computed.
- The sequencer 4 initiates a write access to the memory, using result address register 3.

Addressing means can be extended beyond simple address registers. Here are a few examples:

- Address registers with increment/decrement capabilities.
- Address calculation provisions (for example, according to the principle base + displacement).
- Iterator hardware which supports the address calculation within the loop body as well as loop control.

The concatenation of addressing and processing resources leads to deeper pipelines (there are more register stages to be passed), but provides for nearly unlimited flexibility (for example, the body of an innermost loop may be implemented by resources concatenated according to the data flow, supplemented by address iterator resources to fetch the operands and to store away the results). It is even possible to use the same kind of resource for processing as well as for addressing (such a resource may be an appropriately enhanced universal arithmetic-logic unit).

Basic addressing circuitry
Based on figs. 2.9 to 2.14, typical provisions for address calculation will be explained in more detail. Such address calculation units may constitute the cores of dedicated addressing resources[1)] or may be provided within processing resources.

Fig. 2.9 shows an addressing circuitry that acts as an address adder and forms a memory address according to the principle base + displacement. It depends on the configuration of the entire system whether this address addition is carried out in the processing or addressing resources or centrally on the platform (in the latter case the resources receive address parameters that have been computed beforehand by the platform). In a few applications it is even possible to operate with absolute addresses (calculated at compilation time)[2)].

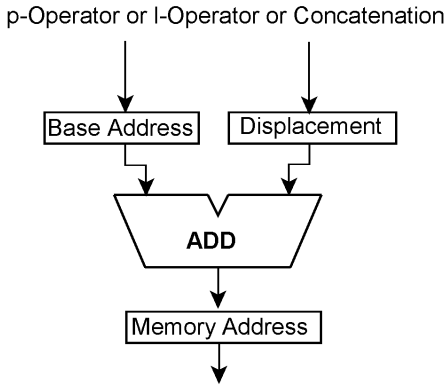

Fig. 2.9 Address calculation according to the principle base + displacement.

It is advantageous to support consecutive (sequential) addressing. For this purpose, the address registers (compare figs. 2.7 and 2.8) can be laid out as address counters. After each access the address is incremented by one (or according to the respective access width) so that data arranged consecutively in memory can be accessed sequentially.

---

1): The address calculation core is to be supplemented by control, concatenation and interface circuitry.

2): Example: Embedded systems with ROM-resident programs.

In the address calculation unit shown in fig. 2.10, the address increment) is an additional parameter. The memory address is calculated based on a base address, a displacement, and a distance value (stride) D. The displacement will be modified after each access.

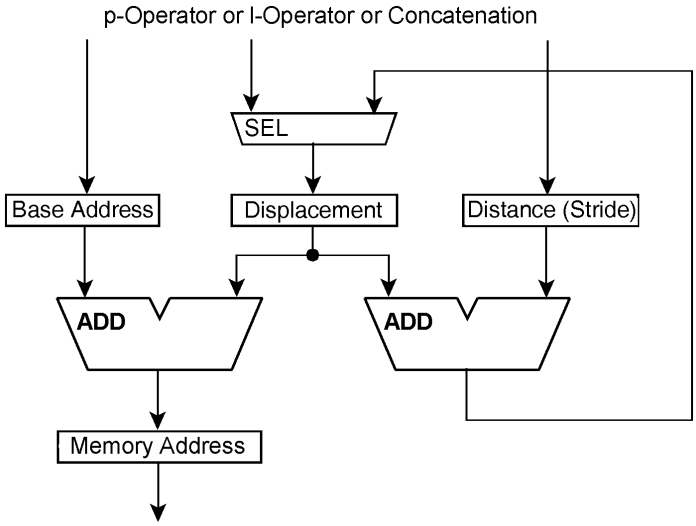

Fig. 2.10 Address calculation including an additional address increment parameter.

The following address calculations take place:

1. Memory address := base + displacement.
2. Displacement := displacement ± distance (autoincrement/autodecrement depending on the sign of the distance value).

Access example: a two-dimensional array (matrix) of floating point numbers that are stored row by row.

a) Access to sequential elements of a line: with distance (stride) = 1 (word addressing) or = length of the floating point number in bytes (byte addressing).
b) Access to sequential elements of a column: with distance (stride) = column number (word addressing) or column number • length of floating point number in bytes (byte addressing); the respective value is to be set once before accessing.

Fig. 2.11 illustrates a modification of the address calculation unit according to fig. 2.10 that supports accessing consecutive data structures of equal length based on index values (= ordinal number 0, 1, 2, 3 etc.). The current index value will be modified by adding or subtracting the distance value (stride) after each access. The multiplier calculates the displacement address of the n-th element (n = 0, 1, 2,...) of a one-dimensional array. In most performance-critical applications the element length will be a power of two with an exponent being not very large (for example, the length is 2, 4, 8, or 16 bytes). When only such access operations are to be supported, the multiplier can be a simple shifting circuitry (multiplication by 2^n corresponds to left-shifting by n bits).

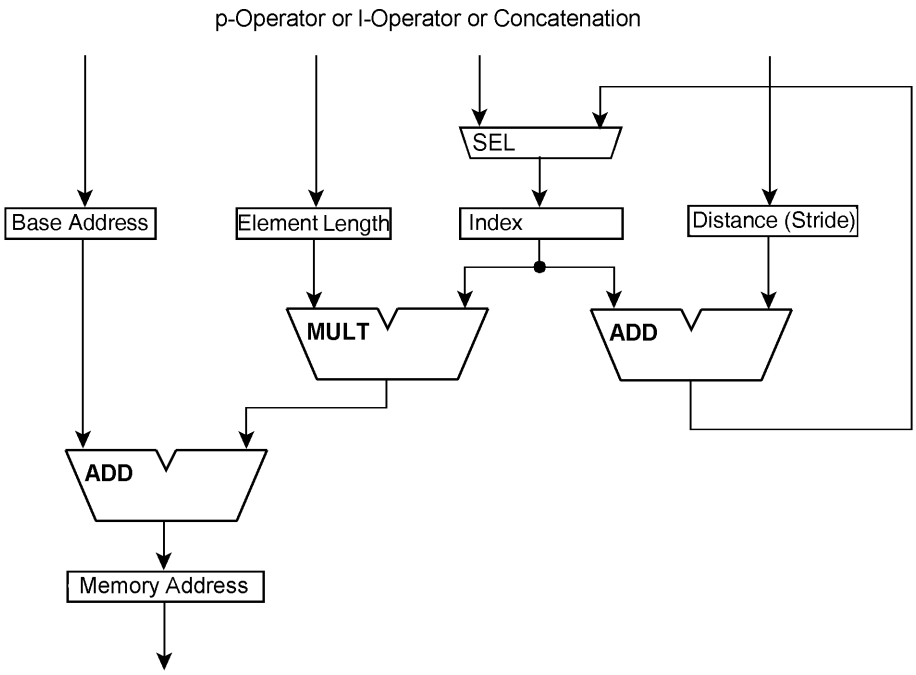

Fig. 2.11 Address calculation unit supporting index (ordinal number) to address conversion.

Address calculations:

1. Memory address := base + (index • element length)
2. Index := index ± distance (autoincrement/autodecrement depending on the sign of the distance value).

Access examples: a two-dimensional array (matrix) of floating point numbers that are stored row by row. Byte addressing applies. The floating point numbers have a length of 8 bytes. Accordingly, the element length ist is to be set to 8.

a) Access to consecutive elements of a row: with distance (stride) = 1.
b) Access to consecutive elements aof a column: with distance (stride) = the number of elements in a row (this value is to be set once before access).

2.8.3 Supporting loops

Many sequential accesses are executed in program loops. When performing program loops, there is the problem of recognizing when to exit the loop (loop termination). In general, the loop condition is queried by a conditional branch. The usual loop constructs of conventional programming languages can be supported also by dedicated resources.

Fig. 2.12 illustrates a resource configuration (in the following referred to as iterator) for supporting typical FOR loops. Operands: initial value A, step width B, end value C. Results: the actual value of the control variable X (in the following: loop value) as well as the ending condition (termination condition). The initial value register A and an adder ADD are connected by means of a selector upstream to the loop value register X; the adder ADD is connected to the step width register B and

the loop value register X is returned to the adder ADD. Moreover, a comparator CMP is connected downstream to loop value register X. The comparator has arranged upstream thereof an end value register C. The comparator output provides the ending condition. The loop value register X can be used as a source of a memory address or the current control variable.

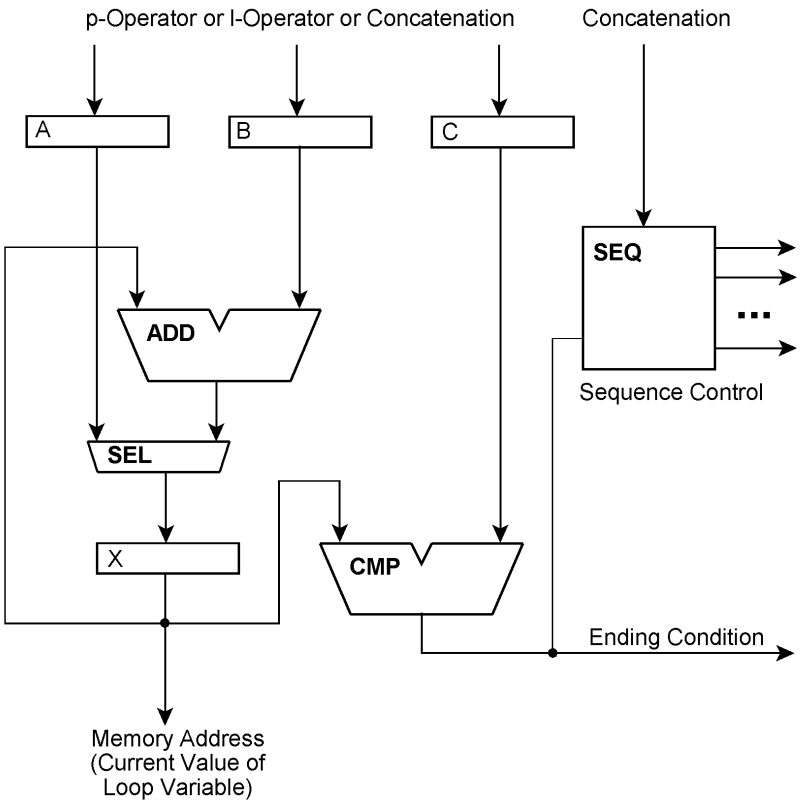

Fig. 2.12 An iterator resource to support typical FOR loops.

The described arrangement supports loops of the type FOR X := A TO C STEP B. It can be used, for example, as follows:

1. concatenations are generated (c-operators), if necessary,
2. initial address, step width and end values are entered (p-operators, l-operators or concatenation),
3. the program loop is activated (by y-operator or as an effect of an input concatenation); the sequence control is activated; in the first step the contents of the initial value register A is transported through the selector into the loop value register X; in the following steps, the contents of the step width register B is added to the contents of the loop value register X;
4. the contents of the loop value register X is compared to the contents of the end value register C; the loop pass is repeated cyclically as long as C < X; when C = X, the last loop pass is completed and the ending condition becomes effective.

The arrangement according to Fig. 2.12 can be operated during a single pass or continuous passes:

a) Single pass: each y-operator or input concatenation (for example, from the sequence control to the iterator) initiates a loop pass. This causes the operators of the loop body to be carried out.

loop start:
y (operator)
-- loop body --
y (branch) -- continuation or return to loop start.

b) Continuous passes: a y-operator or the input concatenation initiates passing of the entire loop. Upon continuation of the loop the output concatenation of the loop value register X becomes effective (conditional concatenation), and at the termination of the loop the ending condition becomes effective (for example, it can be concatenated to the platform). By means of the output concatenation of the loop value register X the iterator resource triggers the subsequent processing resources that carry out the functions of the loop body.

Multiple loop passes can be replaced by corresponding multiple parallel executions of the operations of the loop body (loop unrolling). When the number of passes (1) is known from the beginning (at compile time) and (2) is not too large, this does not present any particular problem for a system configured according to the invention.

Example:
FOR n = 1 TO 20
-- loop body --
NEXT N

is processed in parallel (unrolled) in that the resource configuration required for implementing the loop body is requested 20 times and supplied correspondingly with parameters.

When not enough resources are available, parallel processing is possible only in stages.

Example: in the expression
FOR n = 1 TO 20
-- loop body --
NEXT n

it is possible, for example, to support for parallel processing only four loop body functions (because the number of utilizable processing resources is limited correspondingly). For this purpose, the loop must be reconfigured:

FOR n = 1 TO 20 STEP 5
1st loop body | 2nd loop body | 3rd loop body | 4th loop body
NEXT n

The resource configuration of the loop body is requested four times; the loop is executed in five passes.

When the number of loop passes is not known at the compiling time (example: FOR n = 1 TO x), this simple type of unrolling is not possible. One solution is that a certain number of processing resources are made available for parallel processing in general and that their utilization is controlled by correspondingly designed iterator resources and memory access resources.

The number of parallel-supported processing resources is referred to in the following as degree of parallelization P (P = 1: 1 resource, P = 2: 2 resources, etc.). The utilization is controlled usually by the compiler. It assigns, for example, to a certain loop four processing resources (P = 4) and corrects the step width accordingly:

FOR n = 1 TO x is changed to FOR n = 1 TO x STEP P.

Example:
FOR n = 1 TO 14 is changed with P = 4 to FOR n = 1 TO 14 STEP 4.

Values for n (each processing resource takes care of one of these values):
in the first pass: 1, 2, 3, 4
in the second pass: 5, 6, 7, 8
in the third pass: 9, 10, 11, 12
in the fourth pass: 13, 14

It is apparent that in the last pass not all four processing resources are busy. In order to recognize the last pass, the actual value A is subtracted from the end value E: remainder value R = E – A. When the remainder value R is smaller than the degree of parallelization (R < P), the loop end has been reached. When R = 0, exit from the loop takes place. When R > 0, the last pass is carried out in which some processing resources are not busy.

Fig. 2.13 shows an iterator resource that is modified relative to Fig. 2.12. In addition to the operand registers A, B, C illustrated in Fig. 2.12, an additional operand register P is provided. Its content indicates the degree of parallelization. Loop value register X and end value register C are connected to a subtractor that has arranged downstream thereof an arithmetic comparator whose second input is arranged downstream of the register P of the degree of parallelization.

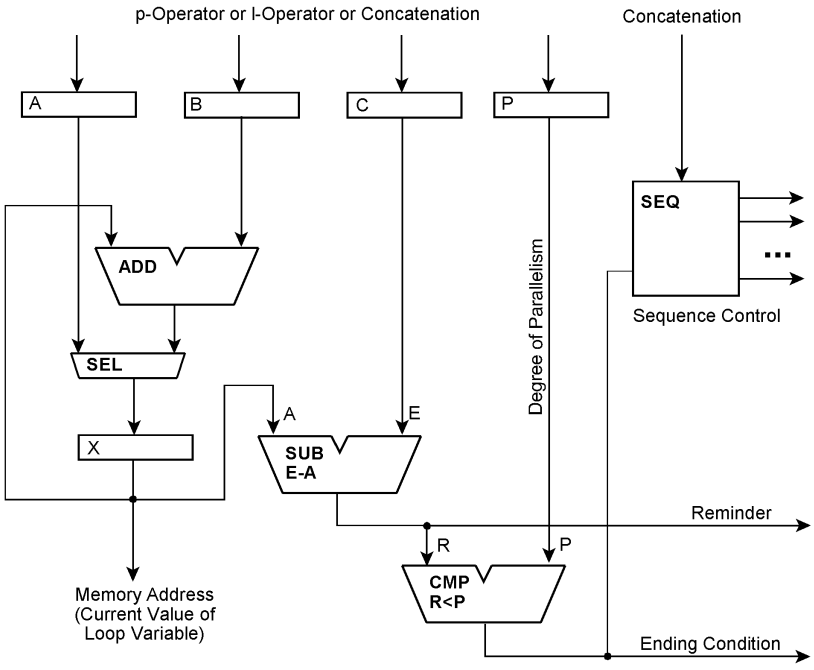

Fig. 2.13 An iterator resource to support loop unrolling.

The following calculations take place:

1. the subtractor determines the actual remainder value = end value – loop value,
2. the arithmetic comparator recognizes the ending condition remainder value < degree of parallelization; if this condition is met, it is necessary to deactivate in the last pass some of the processing resources.

Because the activation and supply of the parallel-operating processing resources must be coordinated, it is expedient to arrange all respective processing resources downstream of a single memory access resource of a corresponding design. For example, a memory access resource has concatenated downstream thereof four identical processing resources. The memory access resource, like the entire memory subsystem, must be able to support the memory bandwidth that is required to supply the parallel-operating processing resources with data and to transport the results.

Example: when four resources with access width of 64 bits are connected, the memory accesses are carried out with an access width of 256 bits (or, for example, with 128 bits and twice the data rate). Data buffers that collect accesses with different width and different addresses and convert them into accesses with greater widths are contained in modern high-performance processors and in bus control circuits (bridges, hubs).

Fig. 2.14 illustrates a memory access resource that contains an iterator 1 according to Fig. 2.13. The iterator 1 provides the memory address. The memory interface comprises also the memory data buffer 2. The memory data buffer 2 is designed for a memory bandwidth which results from the memory bandwidth of the individual processing resource multiplied by the degree of parallelization. In the example, this bandwidth is ensured by a quadrupled access width (for the same data rate; for example, 256 bits for four processing resources with 64 bit data path). For each processing resource a data buffer 3 and a concatenation control 4 are assigned to the memory data buffer 2.

A concatenation process is triggered when data are available (reading) or to be retrieved (writing). A concatenation control 4 becomes effective only when its enable input (for example, E1) is active. The enable inputs E1 to E4 are excited by a remainder value decoder 5 that is connected downstream of the remainder value output of the iterator 1. The remainder value decoder 5 is a combinational circuit that generates the enable signals E1 to E4 for the concatenation controls 58 (Table 1). If the loop pass is not the last, all four concatenation controls 4 become active. In the last loop pass, the activation is based on the remainder value.

Table 1

Remainder value	E1	E2	E3	E4
1	1	0	0	0
2	1	1	0	0
3	1	1	1	0
> 3	1	1	1	1

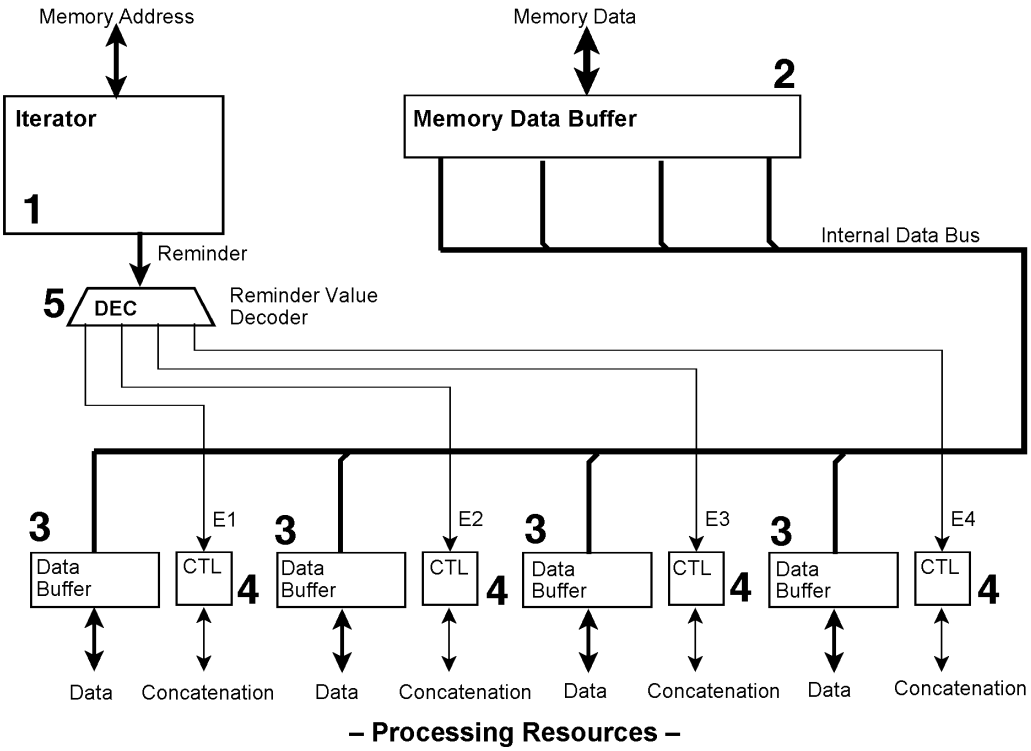

Fig. 2.14 Memory access resource with an iterator to support unrolled loops.

2.8.4 Processing resources

In the following, with the aid of Figs. 2.15 to 2.23, further details of the configurations of typical processing resources will be explained in more detail. In this context, it will be explained how functions are selected and the processing width (number of bits) is controlled. There are processing resources with fixed functions and processing resources that can perform one of several information processing operations. For function selection, additional input parameters are provided. This enables also to select functions depending on previous processing results. In this regard, it is expedient to have a function code without effect (no operation, NOP). When such a function has been entered, the resource will change nothing. The processing state (program context) remains unaltered. Output concatenations are not initiated. In this way, it is possible to circumvent the resource depending on previous results or certain processing states (conditional operation).

Fig. 2.15 illustrates a processing resource that has a function code register FC as an additional input parameter. This feature can be used to set up a certain resource configuration or to change resource operation on the fly. The function code can be treated like every other parameter, i.e., can be loaded by p-operators, l-operators or concatenation. An additional possibility is to let the parameter set by s-operators. This embodiment allows to provide reconfigurable and multi-purpose hardware resources, which are configured to the particular function requested. E.g, the resources are general-puropse arithmetic-logic units (ALUs), and s-operators will request three adders for 16-bit binary numbers, one AND of 8 bits, and five identity comparators of 12 bits. Than those s-operators will set the FC prameters of the requested resources appropriately, this way morphing general-purpose resources into single-purpose functional units.

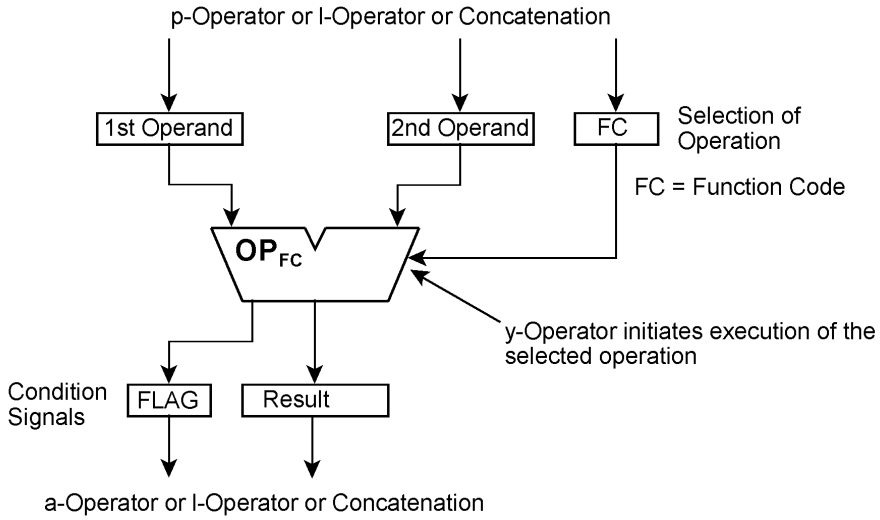

Fig. 2.15 A processing resource with selectable operations.

There are resources with fixed and with changeable processing width (operand size, number of bits). Fig. 2.16 illustrates a simple resource modified in comparison to Fig. 2.15 and provided with an additional processing width register (BITS). The width set therein is valid for all parameters. Operands are processed and results are delivered according to the entered width.

The treatment of excess bit positions depends on the configuration of the resources and the platform (in regard to the memory). Typical variants:

a) filling with zeroes (zero extension),
b) filling with the content of the highest-order bit position according to the current processing width (sign extension),
c) ignoring excess bits and inserting short operands right-aligned (the remaining content remains unaltered).

For the purpose of processing, the operands are typically extended to the maximum processing width of the processing circuit (zero extension, sign extension etc.). The details of operand extension for numerical and non-numerical elementary operations are within the general knowledge.

Fig. 2.17 shows a processing resource that allows to individually set the width of different parameters. In the example, two operand size registers BITS_1, BITS_2 are provided, controlling operand extension circuits 1, 2, that are inserted into the signal paths between the operand registers and the processing circuits. They act in such a way that they supply to the actual processing circuits operands that are extended to the respective maximum processing width.

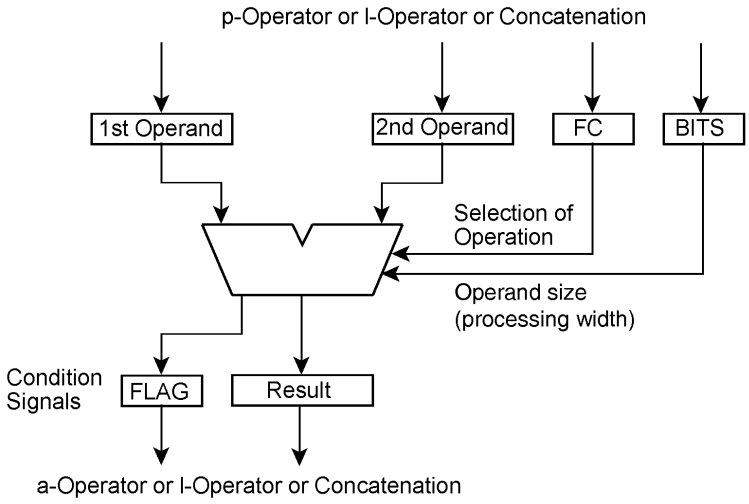

Fig. 2.16 A processing resource with selectable operations and number of bits (operand size, processing width).

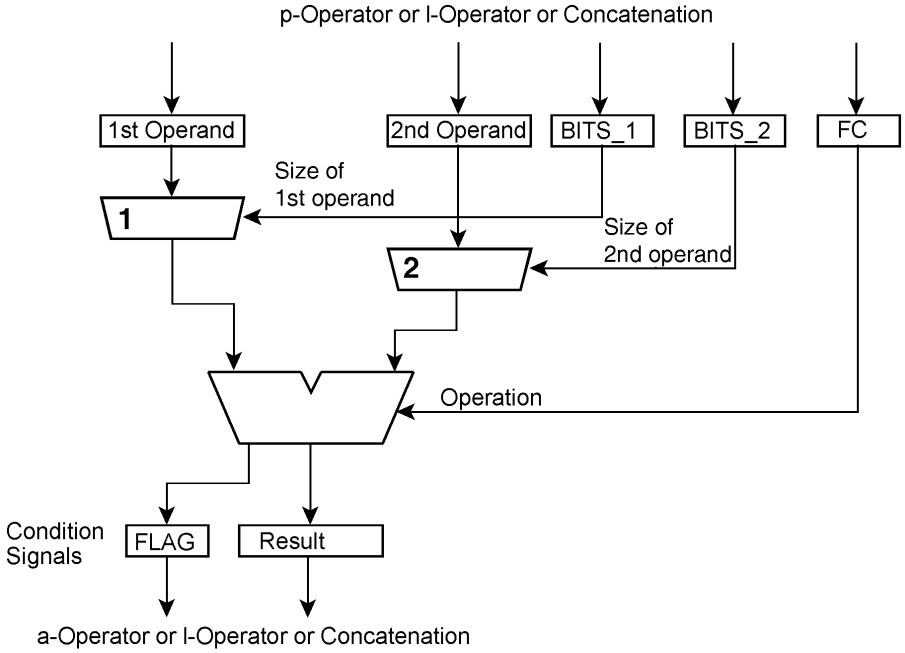

Fig. 2.17 A processing resource, which allows to individually set the width of the operand parameters.

Processing widths not only can be set but also have to be queried sometimes. The resources and the platform must know of how many valid bits the individual parameters are composed. For this purpose, the following principles can be used:

1. The actual access width is described in corresponding tables. This solution has the disadvantage that at run time a table lookup must be executed before carrying out a corresponding operand access.

2. The machine code comprises information in regard to the access width. This solution is similar to conventional instruction set architectures, providing separate instructions to move bytes, words and so on. For reasons of cost, only a few access widths can be supported. Moreover, the respective processing width must be fixed at compile time..
3. The memory means in the resources that indicate the processing width (processing width register, TAG bits) are designed to be queried. This information is transferred together with the data bits, respectively, or can be queried from other resources. For this purpose, interfaces between the resources (bus systems, point-to-point interfaces or the like) are appropriately supplemented.

Fig. 2,18 shows a processing resource whose parameter registers have been extended by TAG bits that indicate the respective access width. The operand bus and the result bus are supplemented by additional lines (TAG bus). During normal operation, the TAG bits can only be read out. Writing via the TAG bus takes place only in order to set up the resources, for example, when executing s-operators.

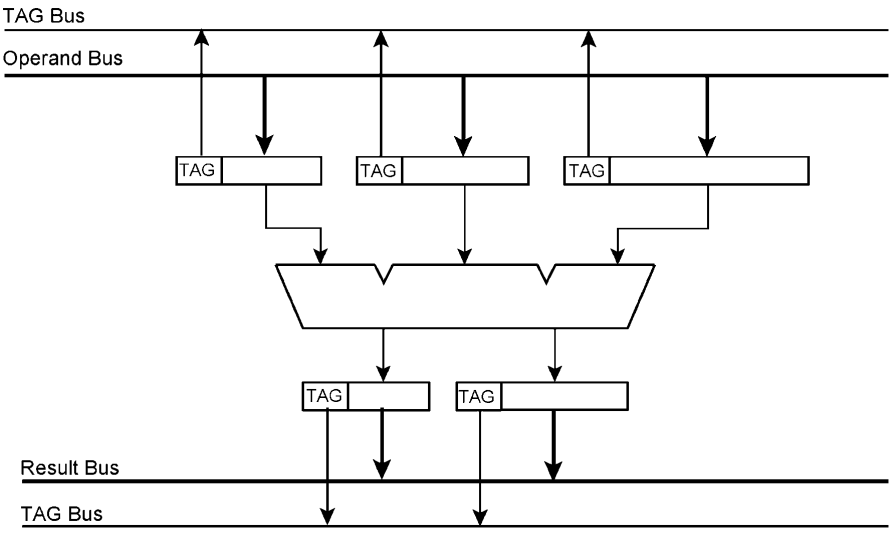

Fig. 2.18 Processing resource whose parameter registers have been extended by TAG bits, indicating the respective access width.

The resources can be designed for various forms of parameter transfer. The most basic mode is to transfer the value of the parameter (value transfer, parameter passing by value). Occasionally, it is advantageous to enhance processing resources by addressing means, so that the resource can address parameters itself (parameter passing by reference), without the need to attach special addressing resources. Fig. 2.19 illustrates a processing resource which allows to select between both methods of parameter transfer (by value or by reference). The resource is connected to a universal memory bus and to signal lines (not shown) for value transfer (for example, via an operand bus and a result bus). To some of the parameter registers 1, only values can be transferred. With regard to the operand registers 2 and the result registers 3, both transfer modes are supported. In order to address the parameters in the memory, for each of those parameters, an address generator 4 is provided. In the simplest case, this is a loadable address counter.

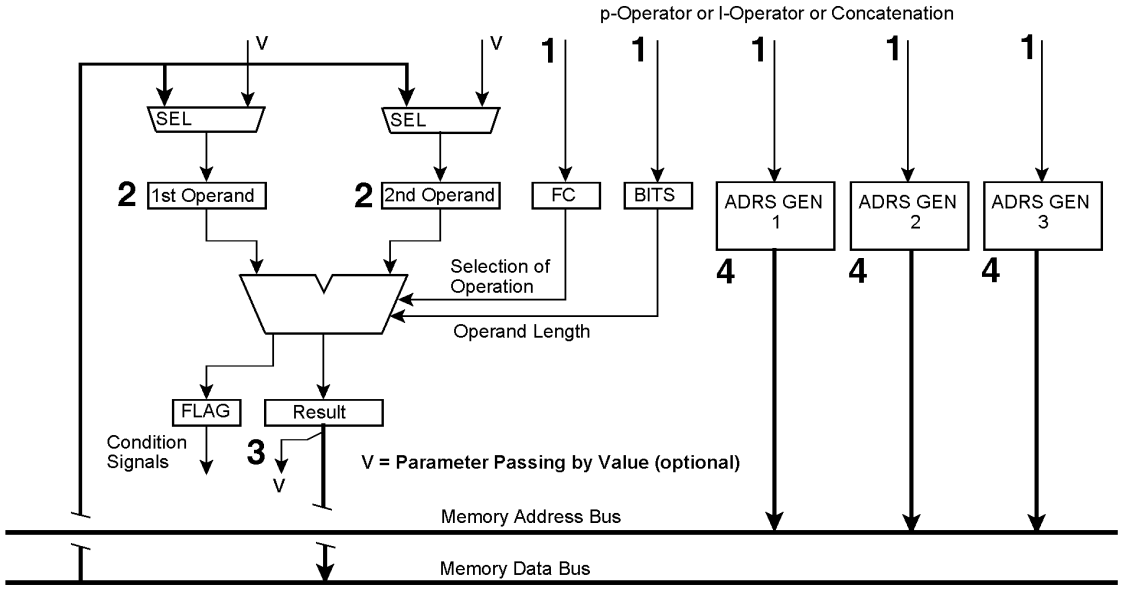

Fig. 2.19 This processing resource supports parameter transfer by value as well as by reference.

The transfer mode (by value or by reference) can be encoded within the function code (register FC). A further selection method could be to make the transfer method dependent by the last transfer before initiation of operation (y-operator or concatenation). Example 1: if a parameter has been entered into the first of the address generators 4, transfer by reference will be established for the first operand. Example 2: a p-operator that enters the second operand directly causes the second of the address generators 4 to be deactivated.

Fig. 2.20 shows a processing resource that is designed for parameter transfer by reference. In addition to the operands and the result, the function code can be addressed, too. Before activating the resource, the address of the first function code must be brought into the function code address register 1 (p-operator, l-operator, concatenation). When the resource is activated (for example, by a y-operator), the sequence control 2 will initiate a first a memory access that enters the actual function code into the function code register 3. In enhanced resources of this type, the function code address register 1 is an address counter. Having executed the first function, the sequence control 3 will fetch the next function code. This way, the resource can execute operation sequences (i.e. program pieces) autonomously. Essentially, this is a program within a program. It will be terminated by appropriate function codes.

The autonomous sequencing of function codes can be further enhanced to true instruction processing. Fig. 2.21 illustrates, how conditional branching can be supported. The function code address register 1 serves as an instruction counter. To facilitate branching, contents of the function code register 3 can be entered into the instruction counter 1. The sequence control 2 is preferably some kind of microprogram control based on an elementary set of microcodes. Fig. 2.21 shows further, that parts of the result could also contribute to the next function code (instruction) address, allowing for functional branching (multiway branches) depending on certain results (this is a well-known principle of microprogram control). Such resources can execute complex functions by means of comparatively simple hardware. A typical example is a resource to compute trigonometric functions. Another realm of application is the autonomous execution of elementary subroutines or innermost loops (i.e., subroutines that contain no additional subroutine calls and loops that contain no additional loops). In

contrast to a fully-fledged processor core, such resources require considerably fewer gates and flip-flops. Applying principles of microprogramming allows to control the operations up to the machine clock cycle and to avoid the overhead, which typically occurs when conventional processors and special-purpose circuitry collaborate.

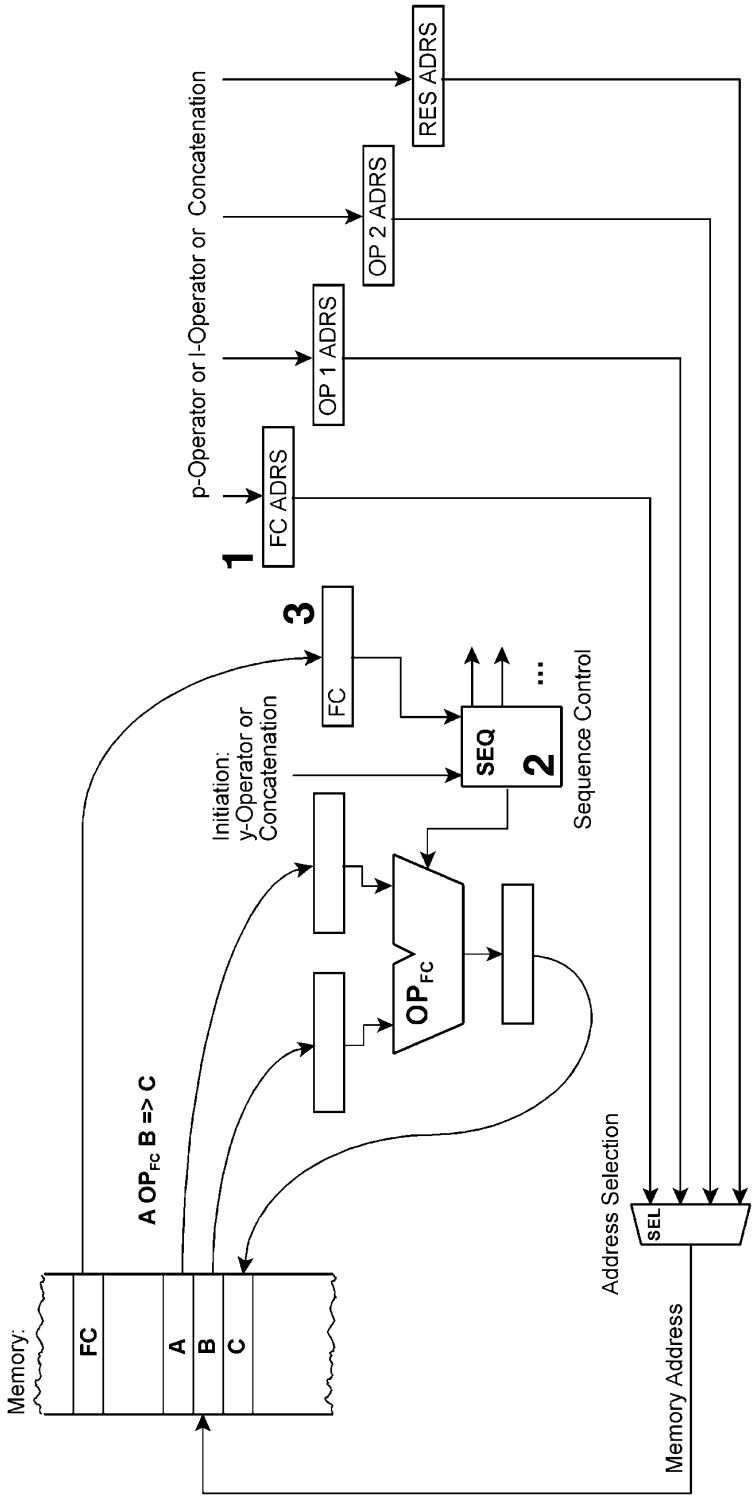

Fig. 2.20 This processing resource supports parameter transfer by reference. Even the function code can be addressed.

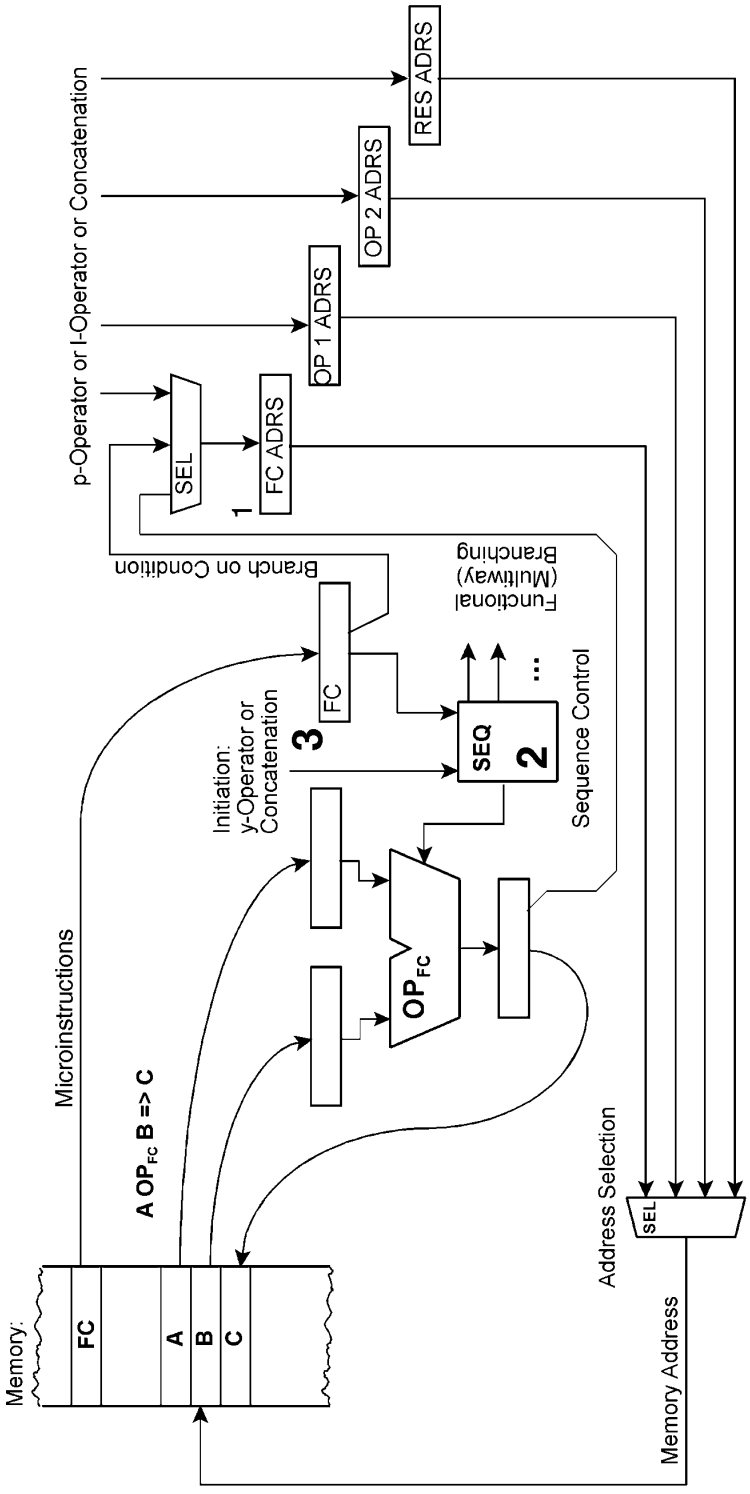

Fig. 2.21 This processing resource contains an autonomus microprogram control.