6. Resource Management

6.1 Resource Management via Tables

At compile time as well as at run time the resources in a ReAl system require some administrative activities. For this purpose, detailed information is required in regard to:

- The resource types available in the system.
- The properties of the individual resource types.
- The status of the resources (how many are provided, how many of them are available and so on).
- The resources assigned to the individual running program (task, thread, process or the like).

It is obvious to arrange this information in table structures. Similar problems are to be solved in virtual memory and file system management[1)], during compiling and so on. The layout of table structures and the corresponding access methods are general knowledge in the field of computer science. Therefore, a brief description should be sufficient. Table entries can be accessed in different ways:

- Via the proper name of the entry (given as a character string).
- Via the appropriate ordinal number.
- Via the corresponding address.

The following description refers to access by means of the ordinal numbers of the entries. This type of access is only slightly slower than direct addressing (in order to determine the address based on the ordinal number it is sufficient to carry out some simple calculations). Additionally, it will pose no particular problems to provide access by name, too (for example, by some kind of hashing algorithm – such functions are provided in any assembler or compiler).

There are three types of tables: the resource type table (one in the system); the resource pool table to be provided as necessary (one for each resource type); and the process resource table (one in each running program (process, task, thread).

Resource type table
The resource type table contains the descriptive and the administrative information in regard to the individual resource types. It has one entry for each resource type. Such an entry contains:

- A general type identifier.
- Parameter data (parameters are operands and results).
- Administrative data.

The parameter data comprise:

- The number of parameters.
- A description of each individual parameter.

1): A typical example: finding a suitable free resource when executing a s-operator.

The administrative data comprise:

- The number of all resources of the particular type.
- The number of resources currently available.
- The state of each individual resource. In the simplest case, one bit for each resource is sufficient in order to differentiate between the states “available” and “unavailable." Sometimes it is advantageous to also provide a reference to the program in which the resource is used.

Each parameter is described by the following data:

- Type identifier.
- Kind of parameter (operand, result or both).
- Information in regard to the concatenation control,
- The length in bits.

Fig. 6.1 provides an overview of the contents of a resource type table.

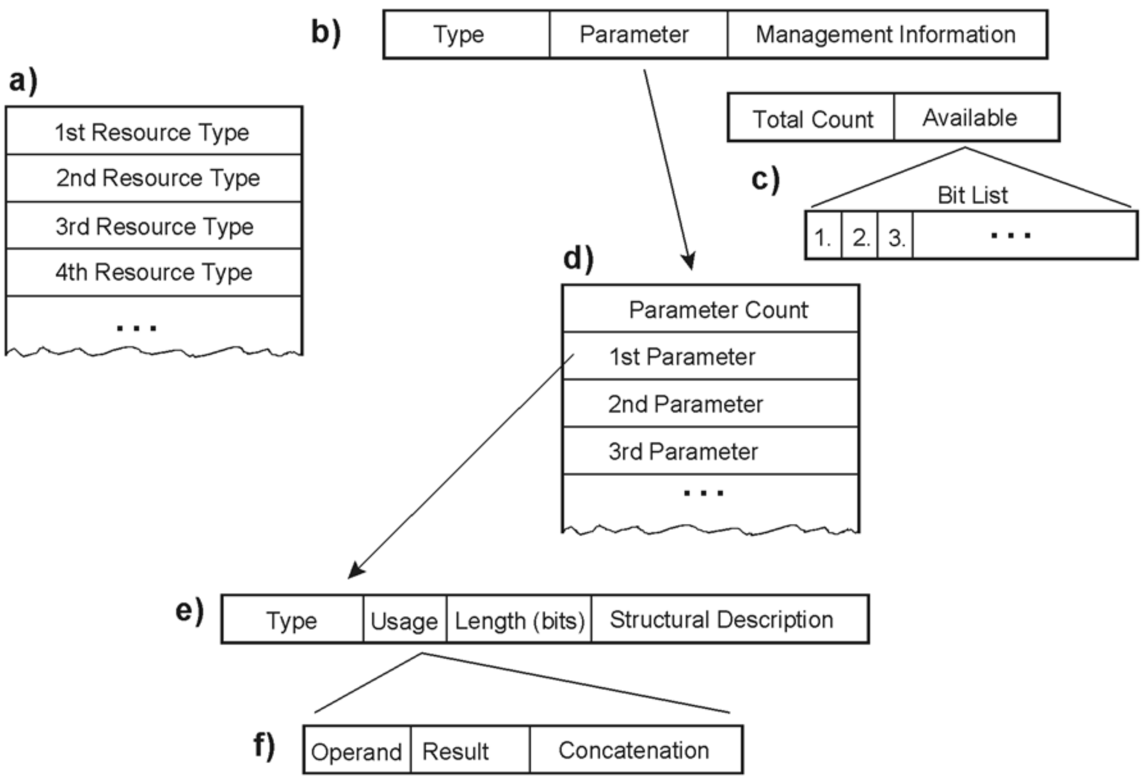

Fig. 6.1 The contents of a resource type table.

a) The resource type table as a whole. Each resource type has one entry.
b) The contents of an entry: type identifier, description of the parameters, administrative data.
c) Elementary administrative information concerns the number of available resources of this type and provides information in regard to which of these resources is available. For example, this is recorded in a bit string (one bit per resource).
d) The description of the parameters as a whole. Each parameter has an entry.

e) The contents of an entry: type identifier (elementary types are, for example, integers and floating point numbers), kind of parameter, length, structural description (as needed). The length is provided generally in bits, independent of the parameter structure. This facilitates decisions and set-up activities1) (all parameters are in the end bit strings that must be transported and stored). More complex parameters have additionally a structure description.
f) The kind of the parameter is described as follows: one bit each for the basic use (operand or result or both) as well as concatenation information (characterizes whether a concatenation is possible at all and provides the type of concatenation).

Table structures
Since the resources are different in their structure, there are table entries of different length. In computer science, there are many solutions to problems of how to arrange such tables in practically manageable data structures. Therefore, a brief description of an example will be sufficient.

The tables are comprised of a header that contains for each table entry a fixedly formatted information. The table entries can be directly addressed. The header is followed by a variable part in which the remaining information is stored. Each entry in the table header contains a pointer that points to the corresponding area in the variable part. At the beginning of each area a backward link (reference) to the table header is provided in order to support administration of the variable part (figs. 6.2 and 6.3).

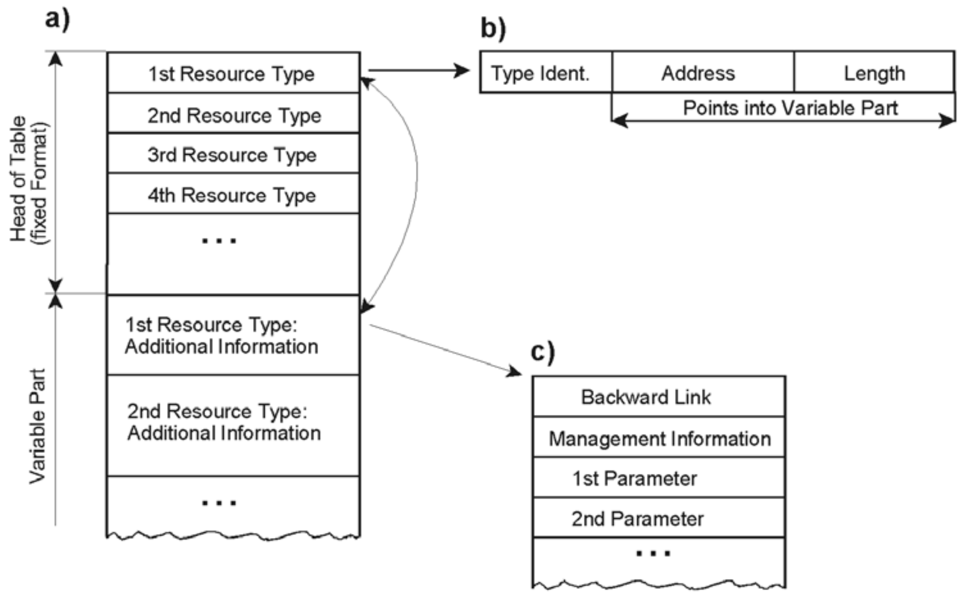

Fig. 6.2 Resource type table format example.

Fig. 6.2 illustrates an of a resource type table:

a) shows the table structure as a whole.
b) shows an entry in the table header. In the example, it contains only a descriptor that describes the assigned area in the variable part (address pointer, length information). Other implementations can contain additional information, for example, in regard to the number and availability of the resources.

1): For example, allocating memory or buffer space or moving the parameters via bus systems.

c) illustrates how one area of the variable part looks like; it contains the additional information to the resource type (compare fig. 6.2).

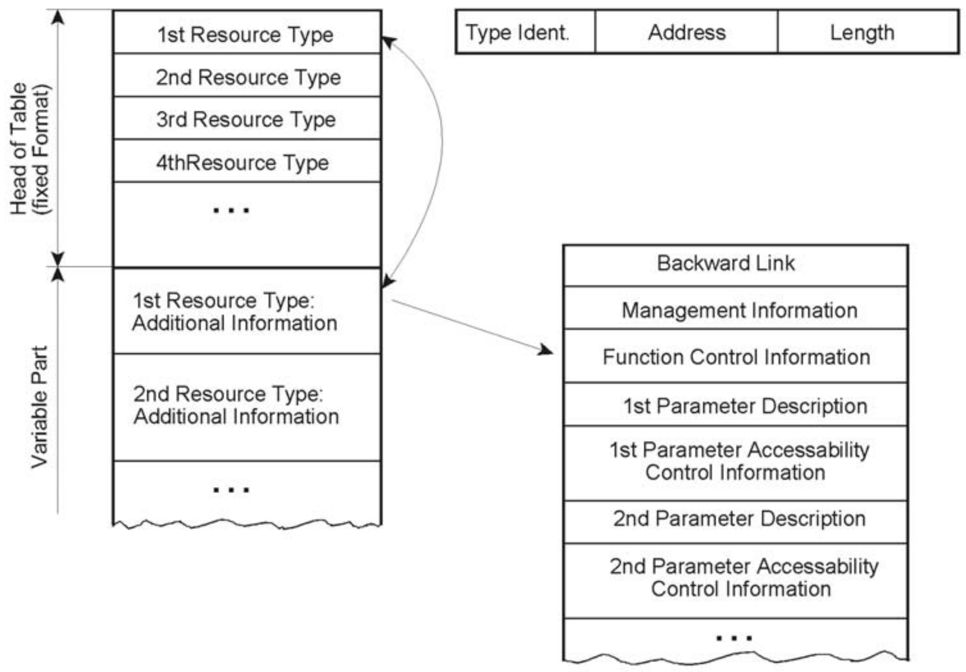
Head of Table (fixed Format)
Variable Part
1st Resource Type
2nd Resource Type
3rd Resource Type
4thResource Type
. . .
1st Resource Type: Additional Information
2nd Resource Type: Additional Information
. . .
Type Ident.
Address
Length
Backward Link
Management Information
Function Control Information
1st Parameter Description
1st Parameter Accessability Control Information
2nd Parameter Description
2nd Parameter Accessability Control Information
. . .

Fig. 6.3 Resource type table, supplemented by function and accessibility control information.

The parameter entries can contain additional information for supporting the resource utilization and administration. In the following, with the aid of fig. 6.3 two examples (function control information and accessibility control information) will be explained briefly.

Function control information
Function control information serves to set up general-purpose resources that are able to provide several functions to the respectively required function. They are two principles of function control:

1. The general-purpose resource is requested by means of s-operator and, by entering corresponding control information (for example, by p-operators or u-operators), is set up to the respective function. For example, a universal ALU is requested and by means of p-operator set up to an operand width of 16 bits and to addition as the operation to be performed (when initiated via a y-operator).

b) The individual functional variants are administered as separate resources. There are, for example, 8-bit adders, 16-bit adders and the like as well as specific resource types. When, for example, 16-bit adders are requested via s-operators, a universal ALU is selected from the resource pool and set up automatically (as an additional function of the s-operator) as a 16-bit adder.

In the second case, it must be differentiated between logical and physical resources. Both types are listed in the resource type table. When the s-operator requests a logical resource (for example, a 32-bit adder), it will find an entry that points to the respective physical resource (for example, a general-purpose 64-bit ALU). This reference is contained within the function control information. Here it is also indicated in which way the physical resource must be set up (for example, by function codes that are to be loaded into certain registers).

Accessibility control information
Accessibility control information concerns special operations that are required in order to transfer parameters into the resources or to fetch them from the resources. The respective operators (p, a, l) must initiate, depending on resource and the parameter, different control activities (for example, certain bus systems or point-to-point connections must be requested, registers must be addressed and so on). This holds true analogously for the concatenation operations. Appropriate information can be stored in the resource type table. This information can be:

- Access control words, microinstructions or microinstruction sequences that control the information flow in the hardware.
- Sequences of elementary machine instructions (for example, transport routines).
- Pointers pointing to to corresponding transport routines.
- Address information (for example, bus addresses of hardware registers).

If needed, for each type of access operation (p-operator, a-operator, l-operator, concatenation) special information can be provided.

Ressource pool table
It can be that each individual resource is accessible in its own fashion (for example, at a special address), so that not all resources of one type can be treated generally in the same way. To cope with such peculiarities, for each resource type an additional resource pool table (fig. 6.4) can be provided that contains the corresponding information in regard to each individual resource according to the following principles:

- The general parameter descriptions (type, length and so on) are provided in the resource type table.
- The status and accessibility information is contained in the resource pool table. It can be optionally supplemented by additional administrative information (providing data in regard to the frequency of use, the number of accesses and the like).

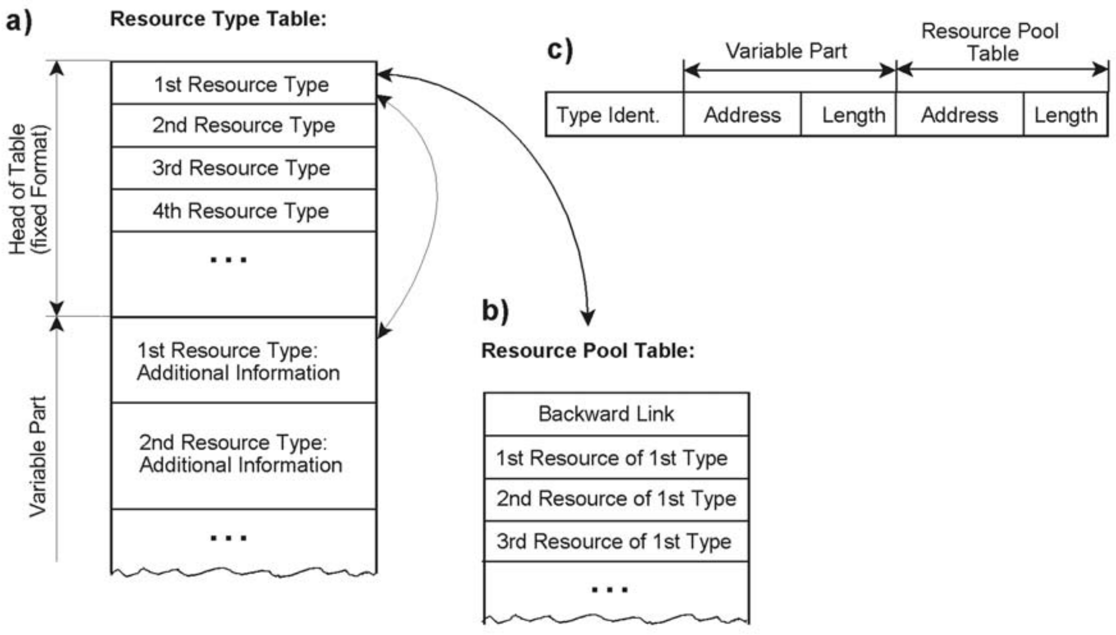

Fig. 6.4 Resource type table together with a resource pool table.

Fig. 6.4 shows a resource pool table in connection with the resource type table:

a) resource type table similar to fig. 6.2.
b) a resource pool table is provided for each different resource type and contains specific information in regard to the individual resources.
c) the entries in the header of the resource type table contain two descriptors, one for the area in the variable part of the resource type table and one for the corresponding resource pool table.

Emulating or building resources
Some resources are combined of other resources (recursion), some are not at all present as hardware. Their function is instead emulated by software or by a microprogram. In addition, resources can be generated on corresponding programmable integrated circuits as needed. The required information can be stored, for example, according to fig. 6.4, in the resource type table. Fig. 6.5 shows an entry in the variable area of a resource type table according to figs. 6.1 and 6.2. The entry is extended by an area that contains the operator code for building the resource from simple resources (recursion), a machine program or microprogram (emulation), or a corresponding circuit description (netlists, Boolean equations, FPGA programming data or the like). The administration information of the corresponding resource types contain a descriptor describing this area (start address, length). The operator codes, machine programs etc. stored therein are typically templates with space holders that are filled as needed with resource numbers or addresses (machine independent or logical coding). Example: a resource is composed of four other resources. The stored operator code addresses these resources via the consecutive numbers (ordinal numbers) 1, 2, 3, 4. Now, such a resource is to be built actually. As components, the resources No. 11, 19, 28, and 53 are available. The operators must now address resource 11 instead of resource 1 (replacement of the logical resource numbers by the physical resource numbers).

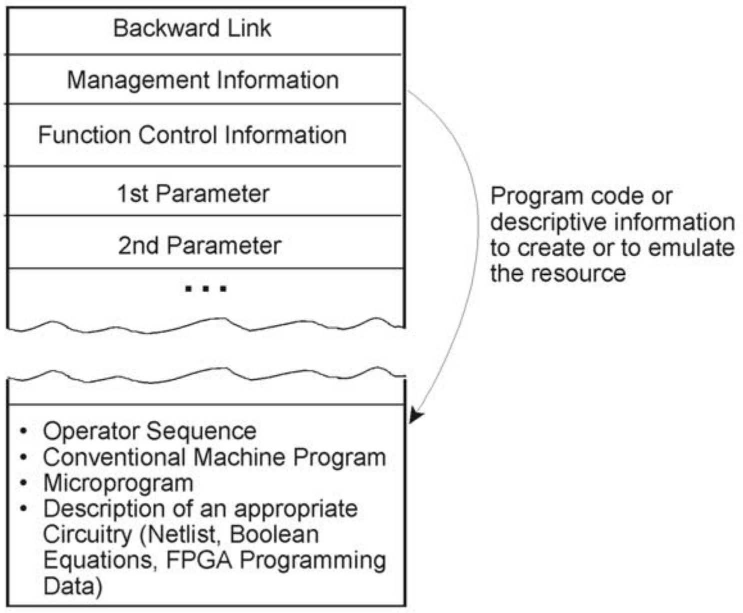
Backward Link
Management Information
Function Control Information
1st Parameter
2nd Parameter
. . .
Program code or descriptive information to create or to emulate the resource
• Operator Sequence
• Conventional Machine Program
• Microprogram
• Description of an appropriate Circuitry (Netlist, Boolean Equations, FPGA Programming Data)

Fig. 6.5 Details of a resource type table entry extended to support emulating the resource or building it from more elementary resources.

Process resource table

The process resource table (fig. 6.6) describes the resources that are requested by the respective running program (process, task or the like). Each of these resources has an entry. This entry contains:

- The resource type (backward link).
- The consecutive number (ordinal number) of the resource of the corresponding type.
- Accessibility control information, for example, memory or hardware addresses, microinstructions or pointers to transport routines. Such information is taken, as the function of the s-operator, from the resource type table or the resource pool table and is optionally modified (for example, logical addresses are replaced by physical addresses).

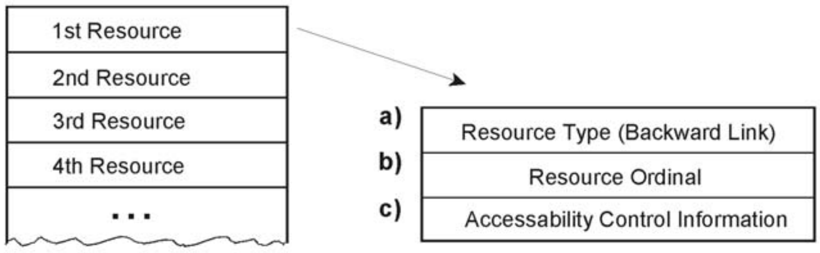

Fig. 6.6 Example of a process resource table

For a software solution (emulation), the accessibility control information is typically a pointer into the resource emulation area. This is the memory area that contains the parameters as well as optionally required working (scratch) areas. Often a single pointer is sufficient because the parameters are addressed consecutively. However, sometimes a separate pointer for each parameter is required.

How the tables are used in the course of a ReAl program

The s-operator accesses the resource type table and finds therein an available resource. A consecutive number (ordinal number) is assigned to this resource. Optionally, the required control information is set up (function codes, operand width and so on). The information required for the additional operators is transferred into the process resource table. The consecutive number of this entry is used as an ordinal number by which all other operators reference this resource.

An example:

1. A 16-bit adder is required. The appropriate statement written in text code: s (ADD_16). In the resource pool, this resource type has the consecutive number (ordinal number) 25. Therefore, the s-operator in symbolic machine code is s(25).
2. The s-operator finds that the resource No. 6 of this type is available.
3. The resource is marked as being used (busy). Optionally, the resource is initialized by entering function code settings for the required processing function.
4. The resource is entered in the next free position of the respective process resource table (resource type 25, resource No. 6). The 11th position of the process resource table be free. Accordingly, the ordinal No. 11 is assigned to the resource.
5. All p-operators, y-operators and so on relate to resource No. 11 and access with this value the process resource table in order to obtain physical addresses and other accessibility control information.

In practice, these operations typically are performed at compile time and (partially) at loading. Executable machine programs contain physical resource addresses and other accessibility control information; they must not access tables.

Address translation in the case of software-based emulation
The process resource table is used only temporarily during compiling. The information nedded to emulate the resource is stored within the resource emulation area1). All accesses at run time refer to addresses in the resource emulation area. The resource emulation area can be arranged optionally in hardware registers.

There are variants of the s-operator by which very specific resources can be addressed, for example, the adder No. 22 or the special processor MAX by IP address 123.45.67.89.

6.2 How many Resources?

In order to be able to determine the format of the table structures, descriptive information and so on, it is necessary to know how many resources, parameters etc. are to be taken into account. There are basically two types of numbers with regard to the size of the resource pool:

1. Finite. Such numbers have a fixed upper bound. A particular upper bound is valid for a certain implementation. Example: a processor with 16 processing units. Accordingly, no more than 16 processing resources can be assigned at the same time.
2. Transfinite. The number of resources is limited only by the limits of the addressing capability of the descriptive data structures.

Resources emulated by software
The number of such resources can be essentially transfinite. It is limited only by the size of available memory (resource emulation area); s-operators and r-operators control only the allocation of the emulation area. Details of the resource management methods are a matter of system optimization. The principal management task is closely related to the well-known problem in computer science of managing free memory space including the so-called garbage collection, that is, to make available again for utilization memory areas released piece by piece or to provide by appropriate data transport a single contiguous free memory area.

An example of a management strategy:
Garbage collection is initially not used. With each s-operator only available memory space is assigned and the function of the r-operators is limited to simply registering the released memory area. A garbage collection takes place only at characteristic points in the program operation, for example, when a certain program branch actually terminates and is not executed again. The memory management could be, for example, supported by corresponding h-operators and u-operators.

1): The resource emulation area can be set up, for example, by emulating the s-operators at compile time. This way the necessary information can be fetched from the resource type (refer to fig. 6.5).

Resources implemented as hardware
The number of hardware resources is essentially finite. Each resource must be managed individually (at least it must be stated whether it is available or not).

Some estimates related to cardinalities (like address length, numbers of resources and so on):

1. *ReAl processors similar to conventional superscalar machines:*

- 64 to 256 resource types (= different types of operation units for executing machine instructions[1)]).
- Not more than 4 to 8 parameters. Simple operations take two operands and generate a result plus some kind of condition code, flag bits or the like. Some operations that are usually considered to be elementary require a few more parameters[2)].
- 16 to 256 active resources. Cconventional superscalar machines have typically 4 to 16 operation units. Future large-size circuits can contain, for example, 4 to 16 conventional processors whose resource configuration corresponds to 4 to 16 processing units each, respectively.

2. *Massively parallel processing based on conventional operations[3)]:*

- 64 to 256 resource types (= different types of operation units for executing machine instructions[4)]).
- Not more thane 4 to 8 parameters (see above).
- Number of active resources: transfinite in theory, large in practice (for example, 1k to 64k).

3. *Emulation (by software):*

- Number of resource types: transfinite.
- Parameters: resources of different sizes, for example, 4, 8, 16, 32, 64, 512, 4k, and transfinite. Experience has shown that more than 4k parameters practically do not occur; most functions have fewer than 64 parameters ([12]).
- Number of active resources: transfinite.

The meaning of the word "transfinite":
In a ReAl machine, a transfinite number is limited only by the length of its binary representation which can vary from implementation to implementation. It must be assumed that an address space or a range of values defined by a particular address length or data width will be completely used up[5)].

1): Or appropriate universal operation units which can be set up to execute the required operations, respectively. For comparison, refer to the operations provided in the typical conventional instruction sets ([15]).

2): For example, the multiplication and division of binary numbers.

3): In contrast to cellular machines and the like.

4): Or appropriate universal operation units which can be set up to execute the required operations, respectively.

5): For example, a length of 16 bits corresponds to 2^{16}, a length of 32 bits to 2^{32} values and so on. Some implementations may have appropriate limitations, however (for example, to 40 address bits of 64 address bits).

6.3 Enumeration of resources

To the resources, consecutive numbers (ordinal numbers) are assigned when they are selected out of the resource pool (s-operator). All further operators then refer to the assigned numbers. Typical problems of the assignment:

- The conversion of these numbers into addresses or into other accessibility control information of the hardware, for example, in access control words.
- The treatment of resources that have been released (returned to the resource pool) in the meantime (r-operator).

Methods of resource enumeration:

1. The resources are consecutively enumerated when selected from the resource pool (s-operators).
2. The enumeration can be controlled by u-operators (for example, by setting an initial value).
3. The ordinal number or address is included in special variants of the s-operator (s_a operators): s_a (resource type => resource number or address).

Enumeration and the releasing of resources (r-operator):
To change the enumeration of the still selected resources as a consequence of each r-operator would lead to a considerable administrative overhead. The alternative: the consecutive enumeration in the s-operators is continued independent of whether in the meantime resources have been released or not. Released resources can be reassigned without difficulty; they simply obtain the higher consecutive numbers. Enumeration begins anew only when the actual configuration of resources has been completely returned to the resource pool (for example, at the end of the program).

6.4 Requesting and Releasing Resources

There are obviously no advantages to request resources individually, to use them and return them to the resource pool immediately[1]. In order to utilize the inherent parallelism to the maximum, it would be best to request all resources required for a certain program at once and to operate them in parallel as much as possible (the program begins with an s-operator that requests all required resources and ends with an r-operator that releases all resources). However, this is not always possible (limited number of hardware resources, limited memory capacity). Therefore, the resource utilization is to be organized essentially piece by piece. Obvious spots where a complex program can be divided into easily manageable blocks are:

- Individually compiled program modules including the functions called therein.
- Regular program constructs (conditional statements, loops and the like).
- Program blocks (that which in conventional programming languages is between BEGIN and END or between curly brackets) including the functions called therein.
- Base blocks (linear sequences of data transports and operation)[2].

---

1): In fact, this is the principal way conventional processor architectures work. In contrast to ReAl architectures, those steps are executed implicitly within the hardware, without need for programmed control. As the conventional instructions do not contain appropriate control information, they are obviously shorter (but the lengthy ReAl operator sequences will pay off . . .).

2): A base block ends with an branch or with a subroutine call. Base blocks in conventional machine programs comprise typically fewer than 10 instructions.

For the respective program block all resources are requested, utilized and released again. An obvious assignment algorithm starts with the base blocks. If resources are still available in the resource pool after the current base block has been translated into an appropriate configuration of resources, the subsequently called function can be taken into consideration, for example.

6.5 Resource Addressing

It is a well-established principle to operate only at compile time with ordinal numbers but at run time with addresses. In byte codes and machine codes resources and their parameters are to be addressed. Two variants of address space layout are available (fig. 6.7):

- Split resource address space: independent addresses for the resources and for the parameters within the resource,
- Flat or unified resource address space: a single address that points to a certain parameter within a certain resource.

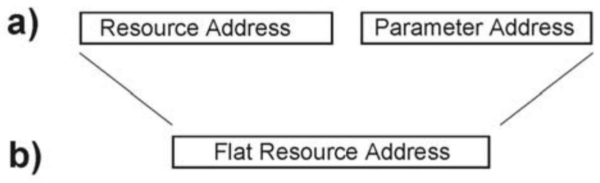

Fig. 6.7 Addressing of resources. a) split, b) flat or unified addres space.

The split resource address space (fig. 6.7a)
There are two types of addresses: one selects the respective resource (resource address) and the other the parameter within the resource (parameter address).

Advantages:

1. Most resources have only a few parameters (typically, 3 to 8). When the resource address is correspondingly buffered (for example in a buffer register), only the parameter addresses must be included in many operators.
2. Shorter address fields in y-operators and r-operators (only the resource address has to be included, but no parameter address).
3. Simplified address decoding in the interior of the resource (compared to conventional solutions in conventional microcontrollers, peripheral integrated circuits and the like).
4. Addressing of parameters in the interior of the resources is independent from addressing the resources as a whole.

Disadvantages:

1. More complex machine code (because two address types must be supported).
2. Buffer registers or other buffering means are required.
3. The parameter address must be long enough to support the resources having the most parameters. This means wasted address bits for all those resources that have only a few parameters. When different parameter address lengths are provided to solve this problem, the machine code becomes even more complex.

The parameter address in the split resource address space – two alternatives:

1. Independent addressing of operands and results. Depending on the respective operator, the parameter address selects either an operand or a result. In p-operators operands are addressed; in a-operators results are addressed. In l-operators, c-operators, and d-operators, the first parameter address selects a result and the second one an operand, respectively. Function code registers and the like are to be addressed only at the time the resource is set up to its currently selected function (s-operator). The y-operator typically must address only the resource; a parameter address is not required.
2. Unified addressing of operands and results. The parameter address covers all parameters.

Independent addressing typically saves one bit in each address field. Example: there are 4 operands and 3 results. Independent addressing requires 2 bits, unified addressing 3 bits. However, this leads sometimes to a more complex hardware, as not only the address must be supported, but also the type of the address1). Special functions would have to be optionally added to the machine instruction set (for example, special variants of the c-operators and the d-operators would have to be provided for supporting the input concatenation).

A further approach to reduce code size can be based on that the resources typically have more operand parameters than result parameters. Accordingly, for the results a shorter address can be provided than for the operands.

The flat or unified resource address space (fig. 6.7b)
There is a single address space in which a certain address is assigned to each parameter of each resource (consecutive addressing).

Advantages:

1. More straightforward machine code.
2. The unified linear address space is a proven architecture principle.
3. A sufficient address length enables supporting arbitrary resources with arbitrary numbers of parameters in a simple way.

Disadvantages:

1. The address fields are longer. So will be the machine instructions.
2. For each individual parameter the address must be decoded in full length. Hence address decoding hardware will be more complex2.
3. Higher administration overhead when resources are selected and released at run time (dynamic resource management).

1): For example, an appropriate bus system will need fewer address lines, but additional signal lines are to be provided to carry the type information.

2): When, in order to simplify the decoder hardware, the address decoding is restricted to address *ranges* (like on the PCI bus), unsatisfactory utilization of the address space can result (which can lead to the problem of having to extend the addresses even more).

Which magnitudes are to be expected?

Depending on the system configuration, technology and area of application, different sizes of resource address spaces will result. Table 6.1 shows a few examples.

resources	**parameters**	**address length**[1)]	**remarks**
4	4 or 8[2)]	4 or 5 bits	superscalar machines of conventional size
32	4 or 8[2)]	7 or 8 bits	expanded superscalar machines, emulation by software to run on microcontrollers
64	4, 8 or 16[2), 3)]	8 to 10 bits	expanded superscalar machines, FPGAs, emulation by software to run on microcontrollers
256	8 or 16[2), 3)]	11 or 12 bits	FPGAs, emulation by software to run on microcontrollers
1024	8 or 16[2), 3)]	13 or 14 bits	FPGAs (very large), parallel processing systems, software solution to run on microcontrollers
64k	16	20 bits	run time environment on high-performance computers; compiler target (for example, for conventional C -programs)
1024k	64	26 bits	large-scale software-based systems (for program development and the like)

1) The number of bits that are required for address encoding (ld number of resources (resource address) + ld number of parameters (parameter address)).
2) Simple arithmetic logic units (calculatincg A **op** B => C) do not have more than three operands (A, B, function code) and two results (C, flags). When the function is initialized at the time of selecting the resource (s-operator), only two operands (A, B) must be addressed. It is then possible to only use one bit for split addressing or two bits for unified addressing, respectively. When the function code is provided as a parameter, in the case of unified addressing three bits are required and in the case of split addressing 2 bits for the operands and 1 bit for the results are required. For addressing a more powerful arithmetic logic unit, typically three bits are sufficient. For a corresponding configuration (function code is no parameter, some of the memory access operations are eliminated) it is possible to use only two bits, respectively, for split addressing (operands A, B, C, D; results X, Z; flags).
3) By means of an additional address bit up to 16 parameters can be addressed. This is sufficient for many special processing units and for resources that correspond to typical functions in C programs (most of those functions have fewer than 16 parameters).

Table 6.1 Typical resource addresses (some examples).

Which kind of resource addressing should be chosen?

- There are only resources with comparatively few parameters: split address space.
- There are only resources of the same type: split address space or activation by access control words (1-out-of-n encoding instead of (binary addressing)
- There are different resource types including resources with many parameters: flat address space.

6.6 Addressing of Variables

In addition to the resources and their parameters, the program variables must be addressed. The variables are typically stored in the system memory, in memory means of the platform and the like. They are loaded as operands into the resources or overwritten with results of the resources. Such transports are carried out by the platform (for example, by means of p-operators and a-operators), but can also be carried out by corresponding resources.

Variable addressing by the platform
Usually it is sufficient to provide for supporting the well-known access principles of the run time systems of conventional higher programming languages:

- Address calculation according to the principle base + displacement.
- The base address registers comprise at least a frame or base pointer (FP/BP), a stack pointer (SP), and an additional (auxiliary) pointer register.
- The stack frame organization of typical (C-type) run time environments is supported, including the entry into and exit from subroutines (functions ENTER and LEAVE).

6.7 Addressing Resources Implemented in Hardware

Each parameter corresponds typically to a register. Addresses are basically ordinal numbers (selection of the 1st, 2nd, 3rd register and so on). Often, it is sufficient to assign particular fixed addresses to the individual registers. More advanced resources can be equipped with special configuration registers through which the respective addresses can be set. Simple address decoders can be, for example, AND gates that are connected (directly or inverted, respectively) to the respective address lines or comparators that are connected to the address lines and to address setting means which provide the corresponding address values. An alternative to this is the central address decoding in the platform. All address decoders are located in the platform; the load control inputs of the memory means at the input side and the output enable inputs of the memory means at the output side of the resources are connected to the address decoder of the platform1).

Parameter addressing in a hardware resource
Fig. 6.8 illustrates the parameter addressing in a hardware resource. Each parameter register has an address comparator 1 with address setting means 2. This can be a hardwired address or an address register that can be loaded by configuration access cycles. The outputs of the address comparator 1 are connected to load control inputs of the operand register or to output enable inputs of the result register. The destination of the parameter that is to be overwritten is laid onto the operand address bus, the source of the parameter to be read is laid onto the result address bus. When one of the address comparators1 recognizes that the supplied address corresponds to the set address 2, the corresponding access function is carried out (loading of an operand register from the operand bus, driving result data onto the result bus).

1): Refer to the addressing within I/O circuitry and on bus systems (including those with plug-and-play support).

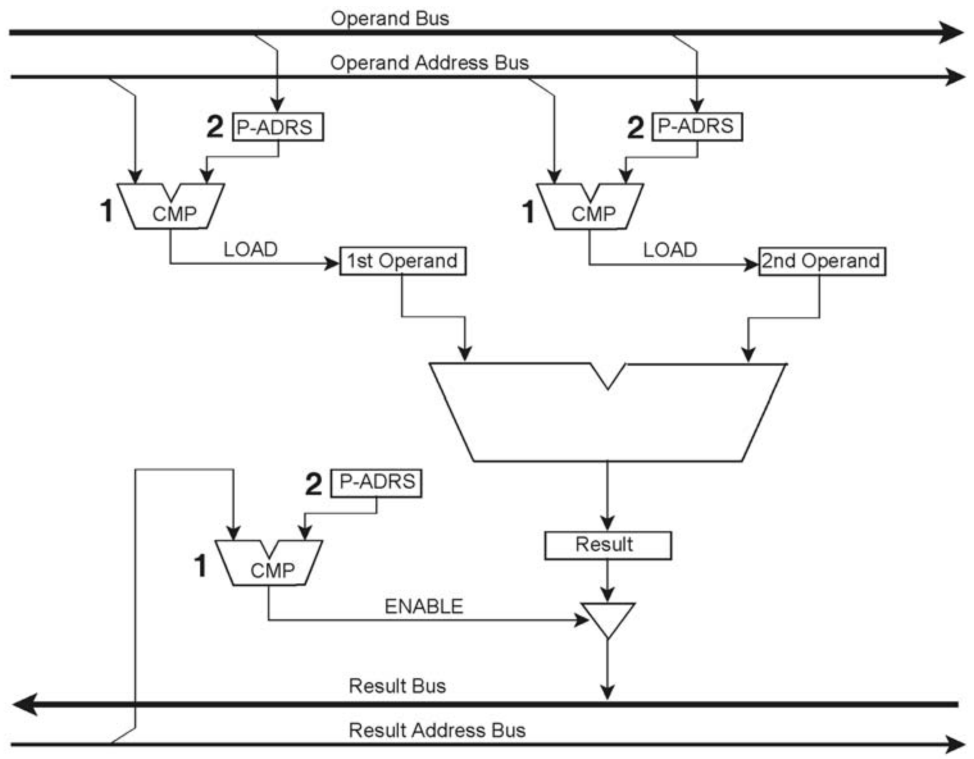

Fig. 6.8 Parameter addressing in a hardware resource. 1 - parameter address comparator; 2 - parameter address setting.

Moving parameters between resources

Fig. 6.9 shows the parameter transport between two resources with the aid of an l-operator. The resources are configured according to fig. 6.8. The result of the resource B becomes the first operand of the resource A (l-operator). In detail:

a) The source address (SOURCE) is laid onto the result address bus. The corresponding address comparator 1 in resource B is activated. As a result of this, the content of the result register is driven ontothe result bus.
b) The destination address (DEST) is passed to the operand address bus. The corresponding address comparator 1 in resource A is activated. As a result of this, the data on the operand bus are loaded into the respective operand register.

The principle can be applied analogously to serial interfaces that are connected, for example, to switching hubs[1].

1): The conversion of conventional bus protocols into protocols of a bit-serial high-speed transfer poses no principal problems (refer, for example, to the serial bus system PCI Express).

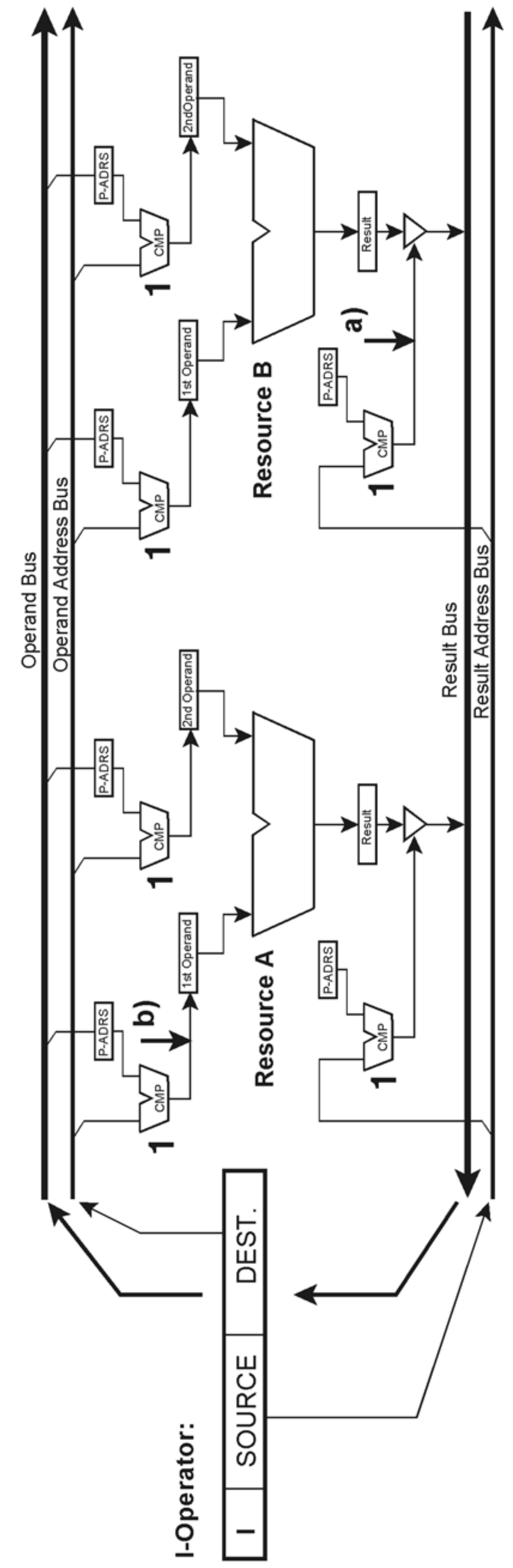

Fig. 6.9 Parameter transport between two resources. 1 - address comparator.

Address decoding supporting a split address space

Fig. 6.9 illustrates the parameter addressing in a resource addressed via a split address space. The arrangement comprised of address comparator 1 and address setting 2 is provided only once within the entire resource. The address comparator 1 is connected to the resource address, the address decoder 3 to the parameter address[1]. The outputs of the address decoder 3 are connected to the load inputs and output enable input of the parameter registers. If the resource address matches the address settings 2, the address comparator 1 enables the address decoder 3.

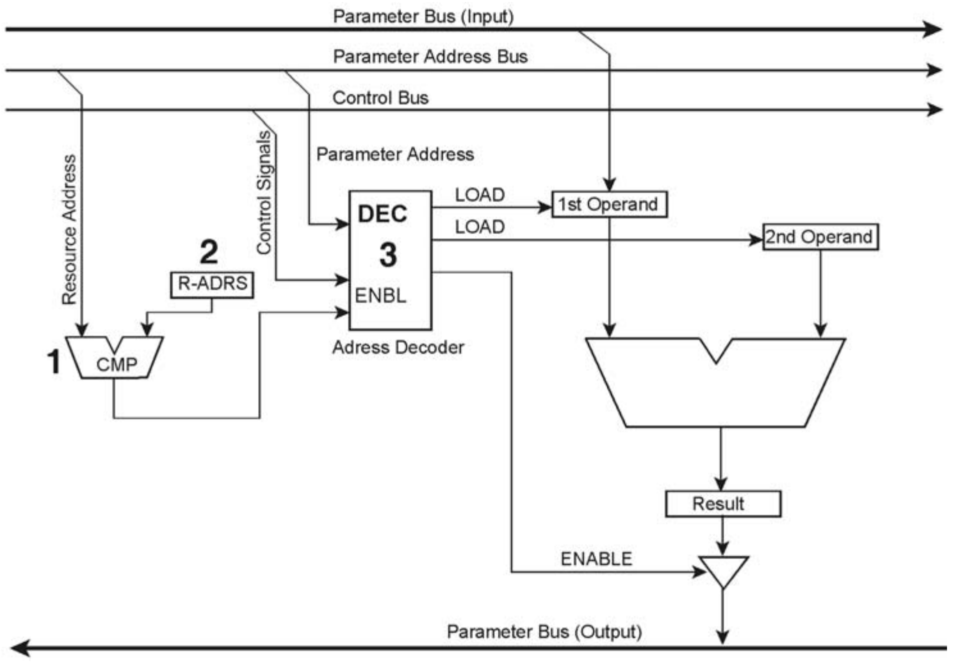

Fig. 6.10 Parameter addressing via a split address space. 1 - resource address comparator; 2 - resource address setting; 3 - parameter address decoder.

For sake of simplicity, fig 6.10 shows common address and control lines for input and output (parameter address bus, control bus). The data paths can be combined in a bidirectional data bus[2]. In fig. 6.10 the decoding of a unified parameter address is illustrated. For this purpose, a single address decoder 3 is provided that serves the operand registers as well as the result registers. For decoding split parameter addresses two address decoders are required, one for the operand registers and one for the result registers. The enable inputs of these address decoders must be connected additionally to the respective access control signal.

Addressing and initiation by means of access control words

Fig. 6.11 shows an alternative configuration. The resources are not activated by a binary address but by access control words. This can include the operation initiation (y-operator, concatenation) as well as parameter transfer. An access control word acts at the same time on several resources. In one such control word the functions of several operators can be combined. If this principle is applied to the last extreme, a control word can initiate all the transport and processing operations that can be performed actually at the same time in all resources.

1): For the structure of a split resource address refer to fig. 6.7.

2): Refer to the typical microprocessor bus systems.

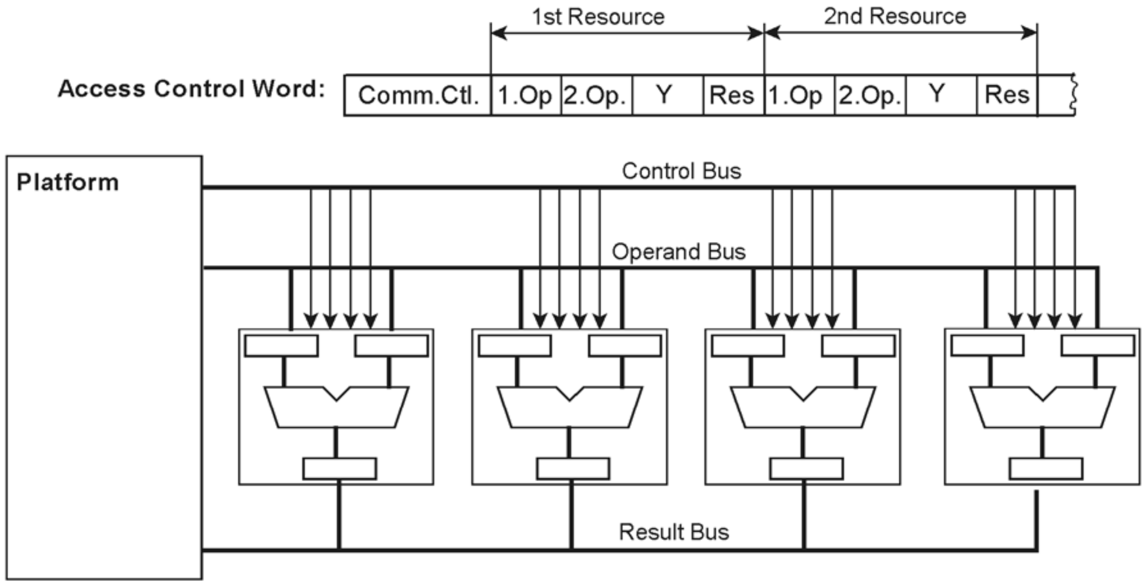

Fig. 6.11 Resource addressing and initiation by means of access control words.

In the example of fig. 6.11 the access control word has one bit position for each resource and operation. The functions of the individual bits:

- 1.Op: load parameter of operand bus into the first operand register of the resource.
- 2.Op: losd parameter of operand bus into the second operand register of the resource.
- Y: initiate operation,
- Res.: drive result onto the result bus.

These bits have a common control field (Comm.Ctl.). Here the following functions are encoded:

- Drive a data word for the system memory onto the operand bus.
- Store the result.
- Drive a result from the result bus onto the operand bus (similar to fig. 6.9).
- Select the next control word (for example, by consecutive addressing, branching or subroutine call).

In Fig. 6.11 one of the simplest system structures (one bus system with two data paths) is illustrated. At one time only one operand and one result can be transported. The operand can be input at the same time in any number of resources. More advanced systems can have several bus structures or switched point-to-point connections. Then the control words contain address fields instead of the individual bits.

6.8 Addressing Resources Implemented in Software

Each parameter corresponds typically to a memory position (for example, a word as defined in the particular processor architecture). The well-known principles of memory addressing and memory management can be applied easily to resource addressing. Fig. 6.12 illustrates how parameters in the memory can be addressed.

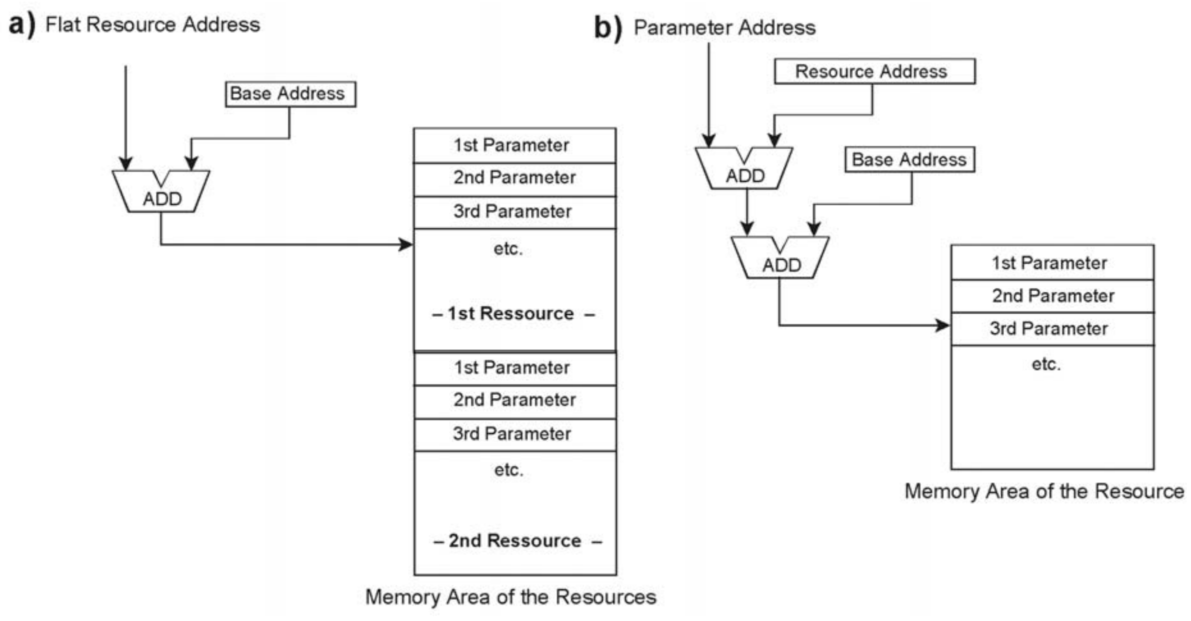

Fig. 6.12 Parameter addressing in memory. a) flat, b) split address space.

It is a matter of course to place the resources on consecutive memory positions. Obviously, addressing within a flat resource address space can be done according to the straightforward principle base + displacement (fig. 6.12a).

To support a split resource address space requires a somewhat more complex scheme of address calculation(fig. 6.12b). An address calculation that goes beyond the principle base + displacement is however not supported at all by conventional processors or only unsatisfactorily, resulting in lower processing speed1).

It is possible in principle to incorporate memory areas that are allocated to resources into a virtual memory organization. Resources that are currently not utilized can be swapped out to the secondary storage. In this way, for the emulation of the resources a memory address space in the magnitude of the entire architecture-based address capacity is available. ReAl operating systems can specifically provide virtual address spaces for the resource emulation (for this purpose, it is only required that a proper set of address translation tables (page tables) is managed for each address space).

Moreover, it is possible to store complete resource allocations as files and to load them for the purpose of execution. The resource structure must therefore be built only once (with s-operators and c-operators). For each following invocation a simple loading procedure is sufficient. Such pre-manufactured structures can be generated by the program developer and can be delivered completed within the corresponding software so that at the user site the corresponding s-operators and c-operators must not be executed.

1): It is well known that typical superscalar processors accelerate or execute in parallel only simple instructions, whereas the more complex instructions are executed the conventional way (under microprogram control).

6.9 Stateless Resources and Resources with States

The resources differ from one another furthermore in that they either are stateless or have a state. The term “state” is to be understood in the context of the general programming model. A program is at any given time in a given processing state. In the case of interruptions, task switching and so on, this state is to be saved. When the program operation will be resumed again, the saved state is to be restored. Designing a ReAl system poses the problem to decide whether or not to include the resources into the processing state. There are two alternatives:

1. Resources with states

The information stored in the resources belong to the processing state or the program context. When a resource is included into the processing state, this has the following consequences:

- The information stored in the resource is to be saved in the case of interruptions, task switching and so on, and, at a later time, to be restored, requiring corresponding access paths and the like and increases latency of the context switching.
- The memory means in the resources (for example, registers) are memories according to the programming model, hence it is possible to keep variables, intermediate results and the like in the resources alone (there is no need for saving them into the system memory, for example).
- Results can be fed back to inputs of the same resource (INOUT parameter; fig. 6.13).
- Concatenation can be used without restrictions.
- Interruptions, task switching and so on can be carried out anytime, in other words, without having to consider the internal processing state of the resources; all processing operations that take longer (for example, than a few microseconds) can be interrupted anytime.

If concatenation is applied to the extreme, there are practically no local variables that must be especially saved in the corresponding memory areas (for example, stack frames). Also, the corresponding transport instructions for storing and fetching again of intermediate results (a-operators and p-operators) are no longer required.

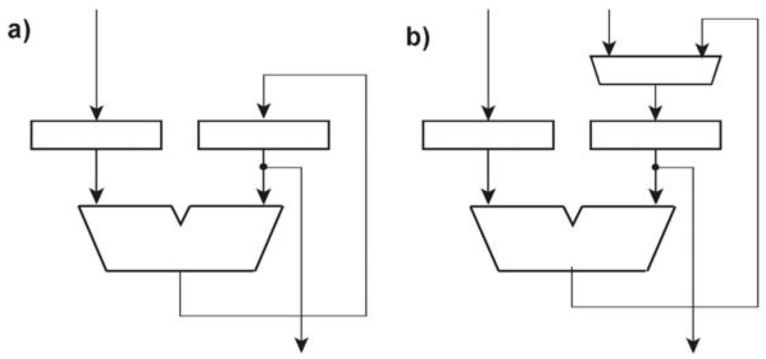

Fig. 6.13 Resources with states. a) a result is used as an operand in the next operation (local feedback); b) a parameter can be an operand as well as a result (INOUT type).

2. Stateless resources

These resources are not included into the processing state. A resource is referred to as stateless when it does not store its parameters beyond the respective actual processing operation; in other words, it is essentially acting as a combinational circuit. This has the following consequences:

- The information stored in the resource is not saved when interruptions, task switching and the like occur. Hence corresponding access paths are not required and the latency of the context switching is comparatively small.
- All variables, intermediate results and so on are to be kept within the main memory (system memory).
- There are no parameters that at the same time are inputs and outputs (INOUT). No input can be read back, no output can be overwritten (by entering operands).
- Concatenation can be used only to a limited extent (for example, for fetching operands and for storing results).
- Before initiating a y-operator all inputs must be always entered anew. This concerns also those values that are unchanged since the execution of the previous y-operator.
- Interruptions or task switching can take place only after the results of the processing operations initiated in the resources have been transferred into the main memory. Also, all operations that have been initiated by concatenation must have been terminated. No processing operation is interruptible in itself.

The selection of the configuration typically depends on whether primarily short latency or high processing efficiency is important. The problem in question occurs only when the resources are to be used for multiple purposes, for example, for interrupt handling or for execution of several tasks in time slices.

Resources with states require time for saving and restoring but the processing operations are interruptible at any time and during processing fewer memory accesses are needed. When the resources are stateless, saving and restoring is not necessary, but the processing operations are not interruptible and, overall, more memory access operations are required.

When minimal latency is required, it is to be investigated, which activity takes longer: saving and restoring or terminating all processing operations including the additional memory access cycles for fetching operands and for saving the results.

When maximum performance is desired, typically resources with states are to be preferred because only this configuration makes it possible to concatenate the resources without limits, to feed back results to inputs, and to utilize the internal memory means for storing data. In order to reduce latency, the resources with states can be equipped with enhanced memory means (fig. 6.14).

Fig. 6.14 shows a simple processing resource that calculates a result based on two operands. The operand and result memories however are no simple registers but addressable memory arrays that are implemented, for example, with register or RAM arrays. The memory addresses are supplied from outside, for example, from the platform. To each task (each interrupt level) a memory position is allocated. The memory positions with which the resource operates are selected by means of the ordinal number of the task (Task No.) or the interrupt level. Task switching or interruption only means delivering the corresponding ordinal number. Such memory arrays cannot be too large (nominal value: 4 to 64 memory positions) because the access time would otherwise be too long (which would require that the clock rate is lowered or pipeline stages are added). One solution is to transfer of memory contents that currently are not in use via independent access paths (in fig. 6.14: save/restore bus) into the main memory and, as needed, to retrieve them out of the memory again (restore operation). These operations can take place parallel to the current processing operations.

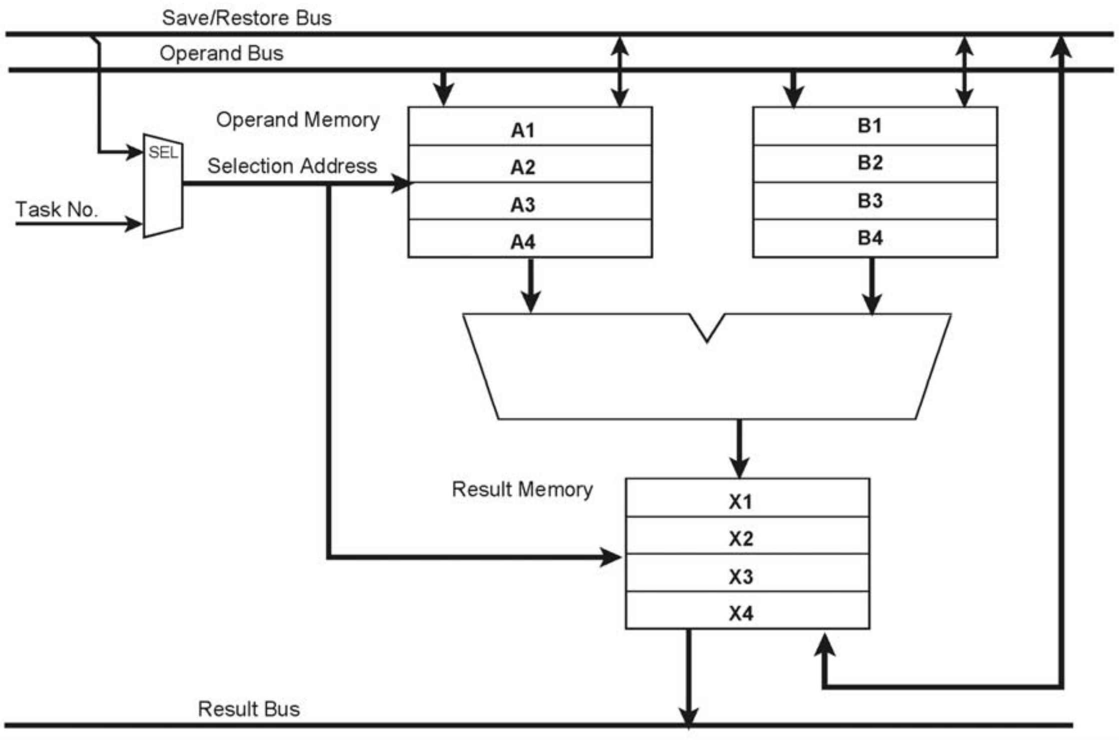

Fig. 6.14 A Processing resource with states containing addressable memory arrays instead of simple registers.

Fig. 6.15 shows how the principle illustrated in fig. 6.14 (the operand and result registers are memory arrays that can be addressed from the outside) can be used in order to implement with one processing unit more than one resource (cost reduction). For this purpose, the operand and result registers (A1, B1, X1 and so on) are supplemented by function code registers (FC1, FC2 and so on). These register arrays are addressed by resource addresses that are supplied, for example, by the platform. Each resource address selects a set of operand and result registers as well as a function code that selects in the operation units the functions of the respective resource type. Example: the first resource (A1, B1, X1, FC1) carries out additions, the second resource (A2, B2, X2, FC2) carries out AND operations.

Further modifications:

- Independent address paths for the operand and result registers. In this way, l-operators and concatenations can be accelerated.
- Combining the addressing and operation modes according to figs. 6.14 and 6.15. The memory configuration (operand registers, result registers and so on) can be used alternately in order to support the execution of different tasks or in order to provide the running task with several processing resources.

The fig. 6.16 and 6.17 illustrate a program-controlled switching between these two modes of operation. For this purpose, the address of the memory arrays shown in fig. 6.15 is a combination of a task address and a resource address. The program controls how many address bits are supplied by the task address and how many by the resource address. In this way, one has the possibility of assigning few resources to many tasks, respectively, or of assigning many resources to a few tasks, respectively.

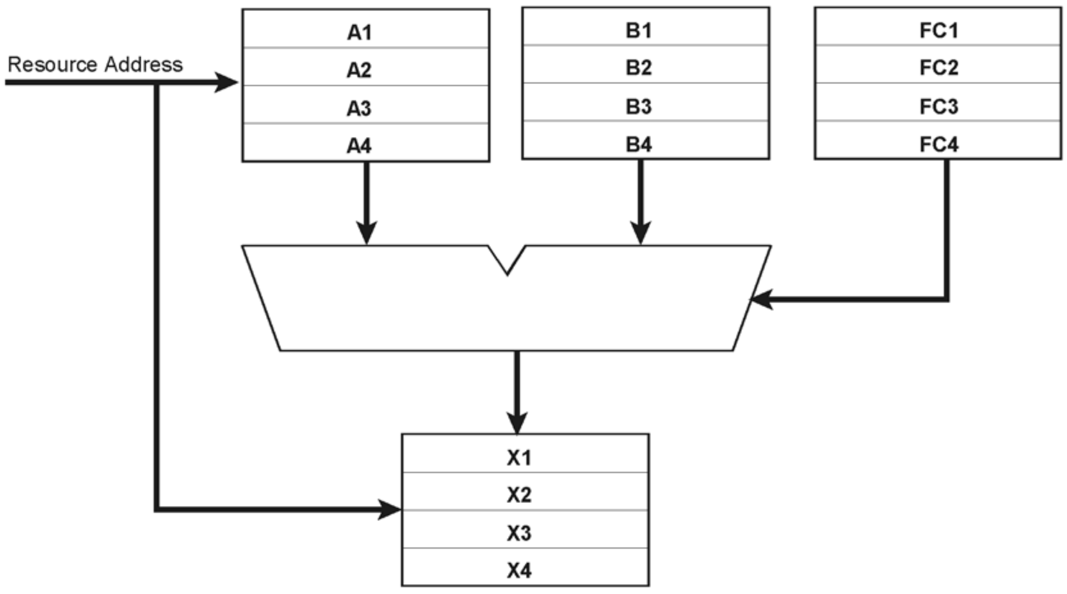

Fig. 6.15 A processing unit capable of implementing more than one resource.

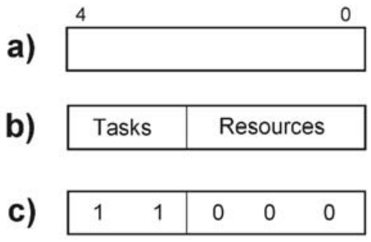

Fig. 6.16 The two addressing modes shown by an example of a 5 bit address.

a) The entire 5 bit address. It can support up to 32 parameter positions. The save address (compare fig. 6.14) that is supplied by the save/restore bus comprises all address bits so that all memory positions can be incorporated into saving and restoring.
b) Address combination for supporting 4 tasks (2 address bits) and 8 resources for each task (3 address bits).
c) A control register that controls the address combination. Each address bit position is selected individually: 0 = bit is coming from the resource address, 1 = bit is coming from the task number.

Fig. 6.17 shows a corresponding circuitry. Resource address and task number are supplied to data selectors whose selection inputs are connected to the outputs of the control register illustrated in fig. 6.16c. A further data selector enables the save address (refer to fig. 6.16a) to be gated to the combined address in order to facilitate saving and restoring of resource register contents. The combined address corresponds to the resource address of fig. 6.15.

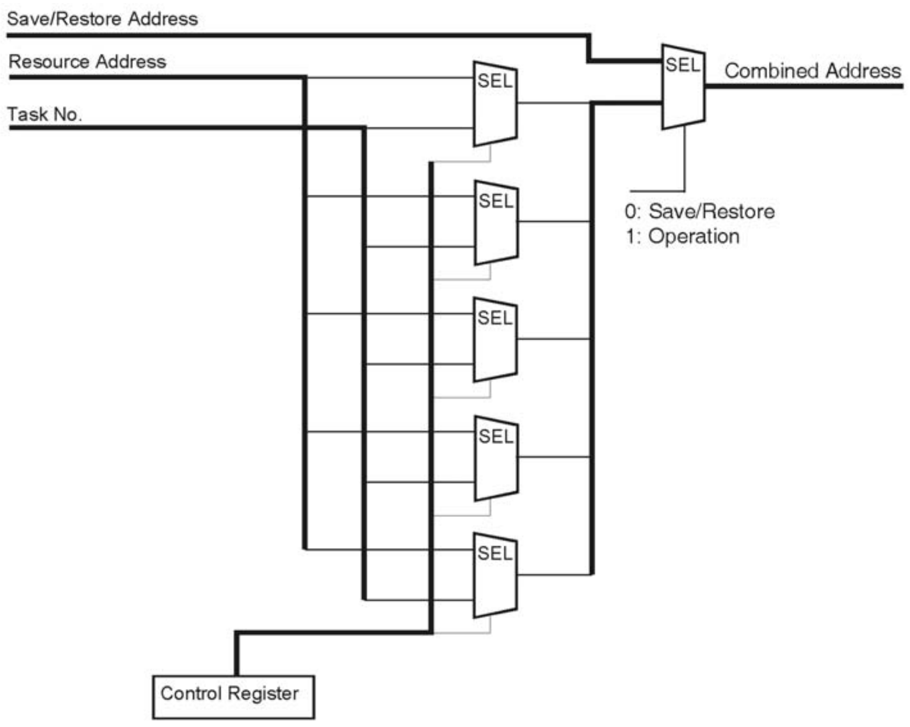

Fig. 6.17 An example resource address selection circuitry.

6.10 Virtualization of Hardware Resources

It is one of the basic principles of the ReAl architecture to consider the resource pool as unlimited. Because the number of resources is however always limited in practice, there is sometimes the necessity to carry out operations that require an almost unlimited resource pool with a limited number of resources. This will be described in the following in more detail. There are in principle three types of limitations:

1. The addressing capability.
2. The memory capacity.
3. The number of hardware resources (for example, operation units).

The addressing capability limits principally the size of the resource pool (the resource pool is not infinite but transfinite). The resources that are taken from such a pool will be referred to in the following as virtual resources.

The actually usable memory capacity can be expanded using well-know principles of virtual memory organization to the limit of the addressing capability.

The number of actually usable physical processing resources will be always comparatively small (magnitude, for example, 2^2 to 2^{12} compared to typical address spaces of 2^{32} to 2^{64}).

The information processing operations are sequentially carried out by means of actually present (physical) resources (serialization). For this purpose, conventional machine instructions, microinstructions or the like can be used (emulation). This corresponds to the principles of operation of conventional general-purpose computers. In another variant the physical resources carry out the information processing operations of several identical virtual resources (virtualization).

For all selected virtual resources, working areas are provided in the memory. These working areas can be included into a virtual memory organization as it is supported by modern operating systems. When an operation is to be carried out, the operands held in memory are transported into a corresponding hardware resource. The results are returned optionally into the memory. There are different variants for implementing this principle:

- The transport operations are programmed. The compiler adds optionally corresponding transport instructions (translation at compile time).
- The transport operations are part of the respective operators. For this purpose, the operation control circuits can be implemented, for example, as microprogrammed control units.
- Processing resources are embedded in cache memory arrays so that the transports are carried out according to conventional principles of cache operation. In this way, it is also ensured that unnecessary transports are avoided (when the memory area of the corresponding virtual resource is present already within the cache, a cache hit results and the processing resource can become active immediately).
- Processing resources are furnished with addressable memory arrays similar to figs. 6.14 and 6.15. Such a processing resource corresponds for example to 2 to 8 virtual resources wherein one of the virtual resources is active at a time. Entering operands and transporting results can be carried out parallel to the processing operations being performed in the respectively active resource (compare the save/restore bus in fig. 6.14).
- The processing resources have their own associative hardware (fig. 6.18).

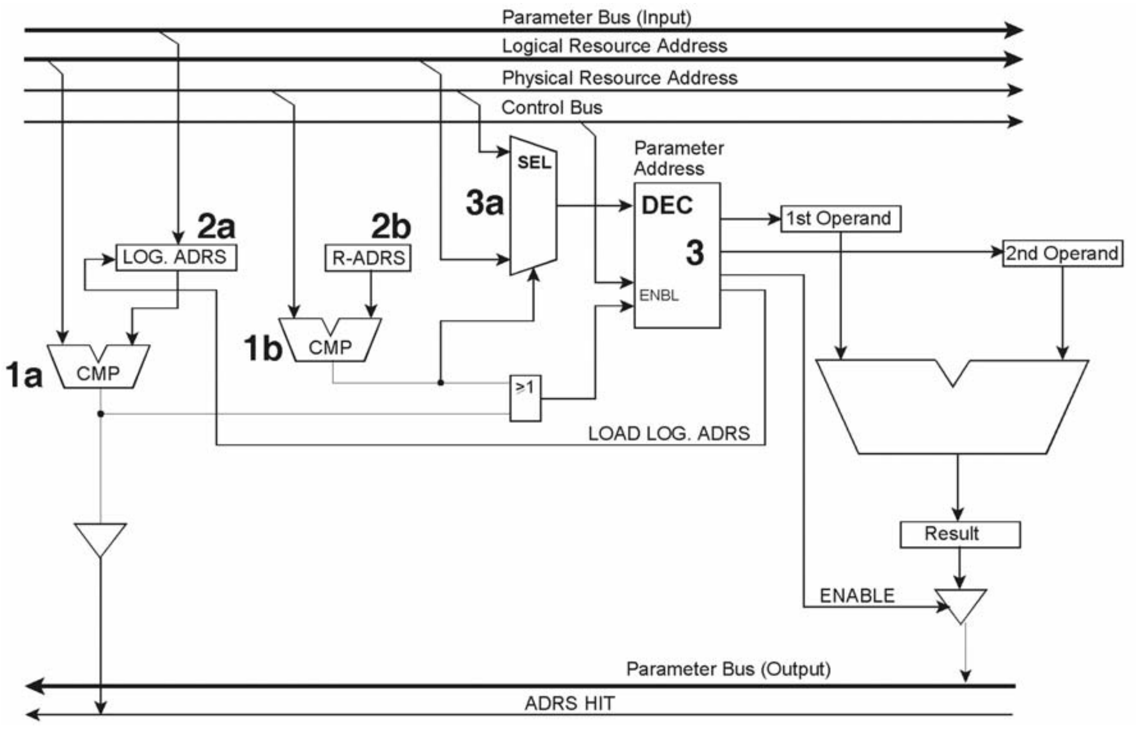

Fig. 6.18 Resource with built-in virtualization support.1 - resource address comparators (1a logical; 1b physical); 2 - resource address setting (2a logiccal, 2b physical); 3 - parameter address decoder; 3a - parameter address selector (selects logical or physical address).

The resource illustrated in fig. 6.18 is essentially a modification of the resource shown in fig. 6.10. Address comparator 1b and address setting circuitry 2b act in the way described in connection with fig. 6.10. They are used for decoding the physical resource address. In addition, an address comparator 1a and an address register 2a for the address of the respectively correlated virtual resource are provided (logical address). The output signal of the "logical" address comparator 1a is connected to a new bus line ADRS HIT. There are also physical and logical parameter addresses. To feed the right parameter address to the decoder 3, an additional selector 3a has been provided. There are two ways of accessing:

1. Physical access via address decoder 1b. The address is fixed (it is either unchangeable or it is set during the hardware configuration process after power-on). The address length is not very large (for example, 6 bits when a total of 64 resources are provided). The address selector 3a sends the physical parameter address to the parameter address decoder 3.
2. Logical access with a virtual resource address by address decoder 1a. Such address information can be long (for example, 32 to 64 bits). The address selector 3a sends the logical parameter address to the parameter address decoder 3. The respective logical address must be loaded prior to this, by means of physical accesses, into the address register 2a. For this purpose, the address register 2a is connected like an additional operand register to the parameter bus and to the parameter address decoder 3 (load control signal LOAD LOG. ADRS). When upon access with a certain logical address the address comparator 1a becomes active, it excites the bus line ADRS HIT, and the circuitry acts as the corresponding virtual resource. When upon accessing with a virtual resource address ADRS HIT remains inactive, one of the hardware resources must be assigned to the respective virtual resource. Procedures for selecting a suitable hardware resource as well as for swapping operands and results are well known (refer to conventional cache memories and virtual memories).

The respective access mode can be selected, for example, by means of different instruction or operator formats. In the extreme, all operators are provided twice (logical and physical). In an alternative configuration the access mode can be made dependent on the system state. For example, typical conventional architectures know at least two states: user state and supervisor state. Obviously, the user state corresponds to the logical and the supervisor state to the physical access mode.

6.11 Instrumentation

In computer systems architecture, instrumentation means to provide systems with additional provisions for system management, for efficiency measurement, for debugging and so on. There are different ways to provide such functions in ReAl systems:

- The resources are expanded with additional devices (figs. 6.19 to 6.21).
- Special resources for this purpose are provided (fig. 6.22).
- Corresponding arrangements are generated ad hoc by connecting appropriate resources (fig. 6.23).

Fig. 6.19 shows how a simple parameter (for example, a binary number) can be supplemented by additional information. Each of the entries shown corresponds to a register (in the hardware) or a memory position (in the resource emulation area).

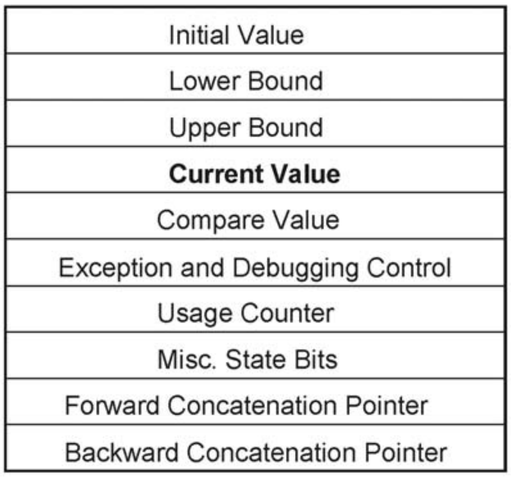
Initial Value
Lower Bound
Upper Bound
Current Value
Compare Value
Exception and Debugging Control
Usage Counter
Misc. State Bits
Forward Concatenation Pointer
Backward Concatenation Pointer

Fig. 6.19 A parameter within a resource. An example showing a rather extreme layout.

The run time systems of many programming languages support only the current value. The values above support the implementation of appropriate programming languages (example: Ada). The compare value is provided for debugging purposes. Example: stopping of program operation (in order to examine and display program states, current values of variables and the like) if the parameter is of a certain actual value. The usage counter can be used, for example, in order to determine how many times the parameter has been used in processing operations or how much time has passed between two new computations.

Fig. 6.20 illustrates a resource with built-in debugging provisions. A simple iterator is expanded by a comparator that compares the generated memory address with a set value and signalizes a stop condition if values are identical (compare match). Analogously, the resources can be provided with circuitry for monitoring value ranges, with metering counters and so on.

The stop addresses, range information, counter values an so on can also be set and transported like the usual operands and results. They are simply viewed as additional parameters. This requires however a corresponding expansion of the parameter address space and thus more address bits in the machine code.

Alternatively, special signal paths for the instrumentation information can be provided. Fig. 6.21 shows a somewhat more complex processing resource that is provided with instrumentation provisions and in addition is connected to an instrumentation bus. Since the transports of the instrumentation information is not critical to performance (such information is set or queried only from time to time) a correspondingly simple configuration is sufficient (for example, as a bit-serial bus). In addition to the already explained debugging and performance measuring provisions, the resource according to fig. 6.21 is provided with the following functions:

- Decrypting of incoming operands and encrypting of outgoing results. Encryption means are well known. Here, they are incorporated into the resource. This has the advantage that over the external bus systems (that are provided, for example, on printed circuit boards) only encrypted data are moved.

- Owner-specific identification. In the simplest case this is embodied as fixed values that can be queried. More advanced resources can be provided with authorization provisions, for example, with password protection; they can be used only when corresponding correct authorization input has been provided beforehand. The advantage is that this protection is inseparable from the processing hardware so that, for example, copying of software (that is to be protected) is of no use because the same-type resources in other machines require different authorization data.

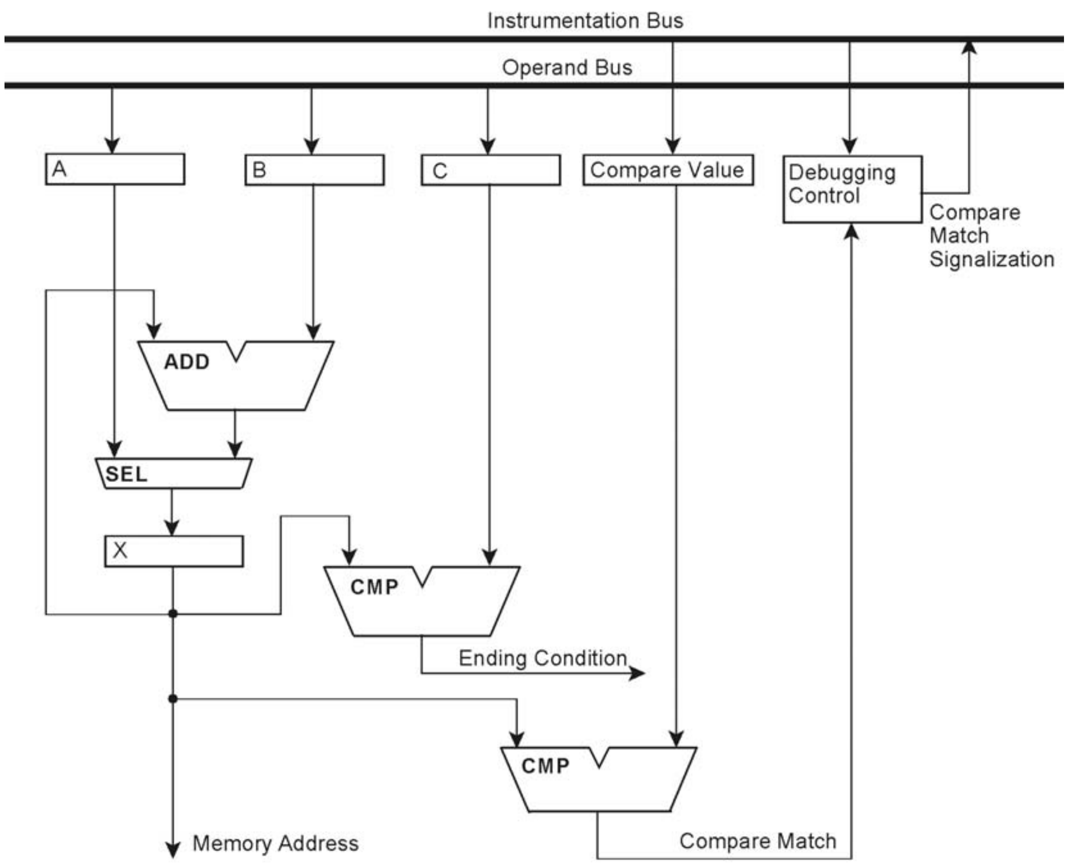

Fig. 6.20 A resource with built-in debugging provisions.

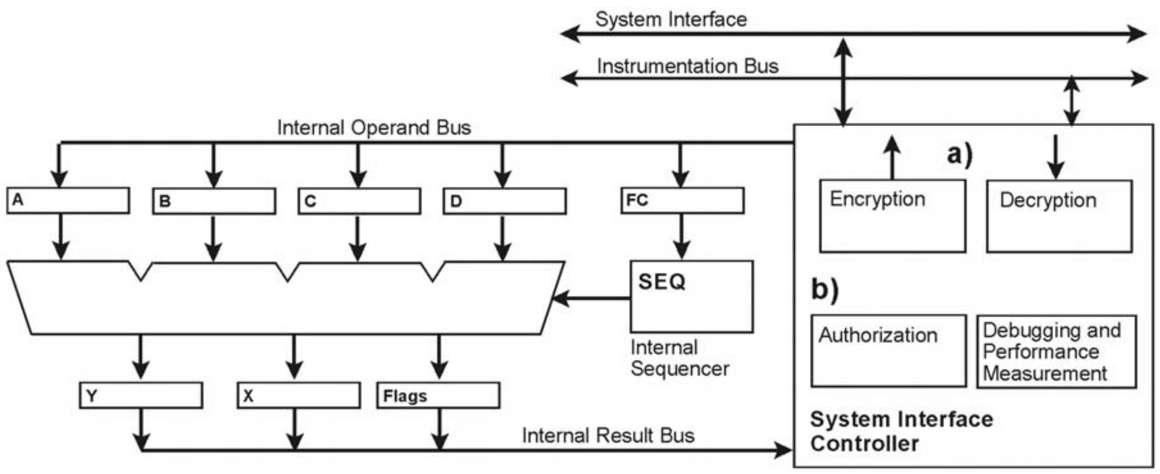

Fig. 6.21 A resource with built-in instrumentation provisions.

Conventional solutions that make available so-called trusted computer platforms are based on deriving a type of signature from the particular properties of the hardware.

Such methods do not solve the basic problem. They require that the corresponding signature is verified via the Internet. This is inconvenient, contradicts the basic principles of privacy or data protection, and impedes the free utilization of the respective computer. The incorporation of directly acting protective measures in ReAl hardware resources has the advantage that the free utilization of the computer is not impaired and that a query of hardware and configuration data is not required because the protective measures are provided directly. Free programs require no such resources (for example, according to fig. 6.21). Programs that are subject to proprietary rights do not run on machines that are not provided with corresponding resources. The encryption and identification functions are provided by the hardware, in particular in the interior of the circuits. The details of operations thus cannot be reverse-engineered by observing the data flow over external connections. The software emulation would be futile (computing times would be too long).

Fig. 6.22 illustrates how a processing resource is concatenated with a special debugging resource. In addition to the processing resources supplementing instrumentation resources are made available (for debugging, for performance measurements and so on). They are called as needed (s-operators) and concatenated to the processing resources (the actual application program does not change by doing this). In the example, a simple processing resource (an adder) is connected to a debugging resource that supports value comparison. When the result of the processing resource is identical to the set compare (CMP) value, a stop condition is signalized. The debugging resource is designed such that it can concatenate the information to be compared (here: the result of the processing resource) to the actual destination resources. If the stop condition is met, the concatenation does not become effective (conditional concatenation). Accordingly, the processing operation is stopped and, by fetching the register contents, it is possible to examine the actual processing state. If the processing operation is to be continued, the concatenation control in the debugging resource receives a corresponding signal via the instrumentation bus.

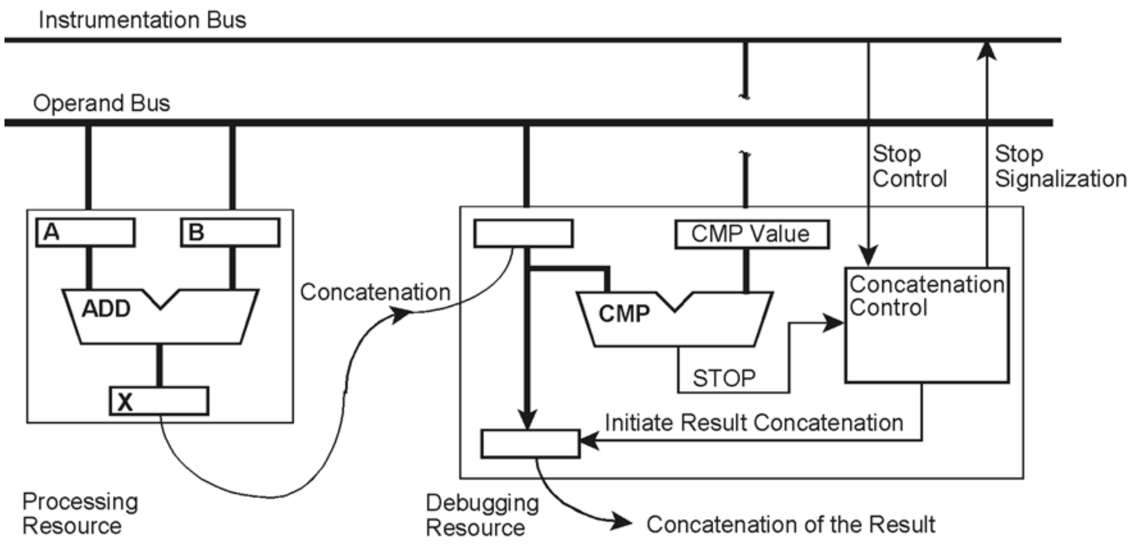

Fig. 6.22 A processing resource concatenated with a special debugging resource.

Fig. 6.23 shows how conventional processing resources can be utilized for instrumentation purposes. In this way, it is possible to combine configurations for debugging, for performance measurements and the like as needed. In the example, an adder (processing resource) is combined with a subtractor (debugging resource). The subtractor compares the result of the adder with a predetermined compare value. Moreover, it transports the incoming results to other resources. The stop condition is signalized by a concatenation to the platform (for example, it can trigger an interruption). This however does not ensure always exact stopping at the stop point. More advanced general-purpose resources, which are also suitable for instrumentation purposes, can be provided with conditional operand concatenation so that, for example, they do not transport a result when a stop condition is met so that therefore the processing operation is stopped temporarily for the purpose of examination.

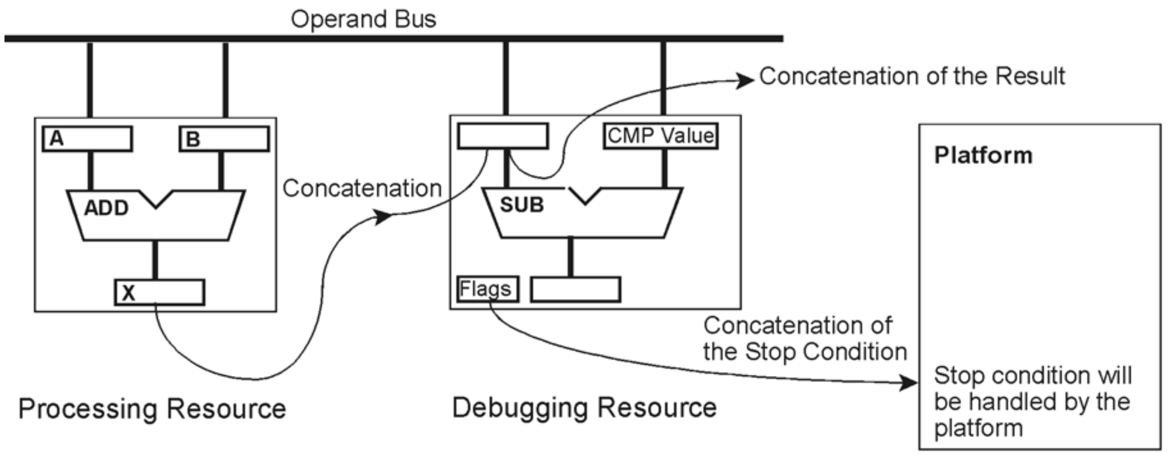
Operand Bus
Concatenation of the Result
A
B
CMP Value
Platform
ADD
Concatenation
SUB
X
Flags
Concatenation of the Stop Condition
Stop condition will be handled by the platform
Processing Resource
Debugging Resource

Fig. 6.23 Two processing resources concatenated to perform debugging tasks.