The ReAl Computer Architecture

ReAl = Resource-Algebra

An Introduction

Release: 1.1 (September 11, 2006)

Prof. Dr. Wolfgang Matthes
Peukinger Weg 34
59423 Unna/Germany
http://www.realcomputerarchitecture.com

Abstract
This introduction describes basic principles and elementary examples of a hardware-software interface which can put into effect a practically unlimited number of processing resources and which allows for completely describing and exploiting the inherent parallelism of the application problems. Appropriate hardware consists of processing resources, platform resources, and storage means. The particular processing resource is not a complete processor but some kind of a comparatively simple operation unit. The proposed architectural principles may lead to:

- Instruction set architectures which can cope with a transfinite number of hardware resources.
- Processing circuits containing resources of intermediate granularity and appropriately optimized interconnects.

Contents

1. Objectives

ReAl is a universal computer architecture (in other words, the ReAl principles of operation describe interfaces between hardware and software). The primary objectives can be summarized as follows:

- To exploit the inherent parallelism as hardware permits.
- To provide for free interchangeability between hardware and software.
- To facilitate hardware-software co-design.
- To ensure independence from a particular state of semiconductor technology.
- To develop instruction set architectures suitable to control even extremely large numbers of processing resources, thus providing worthwhile implementation targets for future technologies.

The ReAl architectural principles can be used not only for instruction set design, but for various other purposes, too. Examples may be a machine-independent intermediate language (similar to the well-known Java bytecode (Table 1)) or a compiling method to detect inherent parallelism. In this introduction, however, we will concentrate on instruction set architectures. We will begin with the first principles, followed by brief descriptions of the basic machine-independent instruction set architecture and appropriate hardware structures. Finally, we will discuss the ReAl approach with regard to related work and current state of the art. As usual in computer architecture, comprehensive descriptions of instruction sets and principles of operation tend to be voluminous (see the reference manuals of current processor architectures). Hence we can give only a short outline and a few basic examples. These examples will deal with arithmetic calculations, data addressing, and conditional branching. For more details we must refer to the primary design documentation ([8]).

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 (i.e., 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 1 Java Virtual Machine (JVM) vs. ReAl. (Note: As the goal is not code compactness but to describe precisely the inherent parallelism and essential intricacies of program operation, ReAl may be better compared to Postscript than to Java.)

2. Computer Architecture as an Algebra of Resources

The proposed architecture is based on the principle of using an arbitrary number of arbitrary processing means (resources) for solving the application problems.

The principal hypothesis: There will always be enough . . .

- Hardware does not matter.
- Memory Capacity does not matter.
- Hardware requirements for compilation do not matter.

Our architectural definition is based on a set or pool of resources which can execute certain operations with data of certain types. This constitutes basically an algebraic structure. Hence the name *ReAl* = *Re*source-*Al*gebra. The basic model of a resource is a hardware unit (like an adder or a universal ALU) performing certain information processing operations. Similar paradigms have been used occasionally over more than one decade for performance analysis and abstract modeling of architectural principles ([1], [2]). Here the basic idea will be applied to universal processor architecture and instruction set design.

Our basic paradigm:
If we want to do something, we will fetch an appropriate piece of hardware out of a magazine (like a hammer to drive in a nail or a wrench to fasten a nut) and use it to perform the information processing task to be executed. If we want to add two numbers together, we take an adder, if we want to compare two values, we take a comparator and so on. A piece of hardware which has done its duty will be returned to the magazine. We will take as many tools as we need, for example 50 hammers if 50 nails are to be driven in, or 50 adders if 50 pairs of numbers are to be added together.

The architecture is based on the following principles:

- There will be always enough resources. Above all this is a theoretical assumption (hypothesis of a transfinite resource pool). Based on this assumption it is possible to request an arbitrary number of resources (like a few hundreds of multiplication units) in order to exploit the inherent parallelism up to the utmost level. In practice however, each pool of resources is limited in size. Hence the programs are to be adapted to the limits of a given pool of resources. This can be done during compile time or during runtime (emulation, virtualization). Virtual resources can be provided similarly to pages in a conventional virtual memory.
- With respect to an application problem, the universal computer is only a makeshift solution. The true optimum solution would be a dedicated hardware whose machine cycles are spent exclusively to compute the desired final results. In such a machine, neither clock cycles and memory bandwidth nor power would be wasted for fetching instructions, loading and storing intermediate values, for function calls and the like. We want to build true universal machines whose characteristics come as close to this ideal as possible.
- The basic paradigm of a resource is a piece of hardware with input registers, combinational circuitry and output registers (register-transfer model).
- The instructions (operators) describe only the basic processing steps, but not the concrete operations to be performed (like addition or multiplication).

In order to implement a certain programming intention, appropriate resources will be selected out of the resource pool. These resources will be fed with parameters. Then the processing operations will be initiated. Results will be stored in memory or written to I/O devices; intermediate results will be forwarded to other resources. Further steps of parameter passing, initiation and assignment will be executed until the processing task has been completed. Resources which are no longer needed will be returned to the resource pool. These processing steps are controlled by stored instructions. So-called platform resources are provided to fetch the instructions from memory. Additional instructions are provided to establish connections between resources (to concatenate resources) and to disconnect such concatenations. Once a concatenation has been established, the steps of parameter passing, initiation of operations and assignment of results will be performed automatically; there is no need to control each single processing step by separate instructions.

If the principles of the proposed information processing method are applied to the last extreme, machine program generation will be transformed into hardware design. Starting with the source code, a special hardware will be designed which is able to execute the application problem in question. This virtual

hardware can be created, modified and dismantled during runtime. If a resource is not available as a true hardware unit, its operation can be performed by means of other resources by applying the very method (recursion) or by conventional programs (emulation).

The steps of parameter passing, operation initiation and so on can be applied to hardware as well as to software. Program and hardware resources are invoked the same way. Each program or subroutine corresponds to the model of a hardware with input and output registers (register-transfer model).

3. The ReAl Programming Model

3.1. Resources

Elementary resources consist of input registers, combinational circuitry and output registers (Fig. 1). The operation units and ALUs of conventional processors may be seen as appropriate examples. A simple example resource (Fig. 1a) takes two operands (A. B) and calculates a single result (X):

$$X := A \ \mathbf{OP} \ B$$

Conventional computers support only a few basic data types (integers, floating-point numbers, characters and so on). An operation unit processes only operands of a certain type (like integers or floating-point numbers).

Such restrictions do not apply to our resources. Instead, a resource can calculate an arbitrary number of results from an arbitrary number of operands (Fig. 1b). Furthermore, there are no restrictions to data types. Arbitrarily complex data types are allowed (like bit and character strings, interval numbers, complex numbers, arrays, and heterogenous structures (records)). Resources can be implemented with hardware as well as with software. The operand and result registers correspond to storage cells for the operands and results, the combinatorial circuitry corresponds to programs which perform the required operations. In order to emulate the operations by software, often additional working (scratch) storage will be necessary (Fig. 1c).

The ReAl architecture exploits an arbitrary number of arbitrary resources. Fig. 2 shows how such a resource is used. In the example, the following processing steps are executed:

1. Parameters will be fetched from memory and passed to the resource.
2. The operations will be executed (hardware will be activated or a corresponding software routine will be invoked).
3. The result will be assigned (moved) to the corresponding variable in memory.

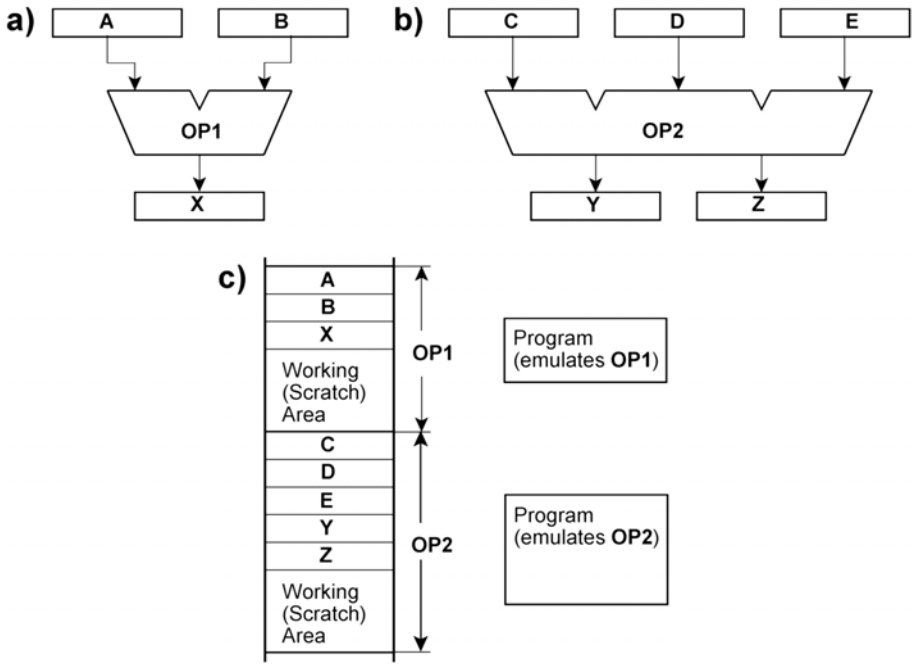

Fig. 1 Example processing resources. a), b) Hardware, c) Software

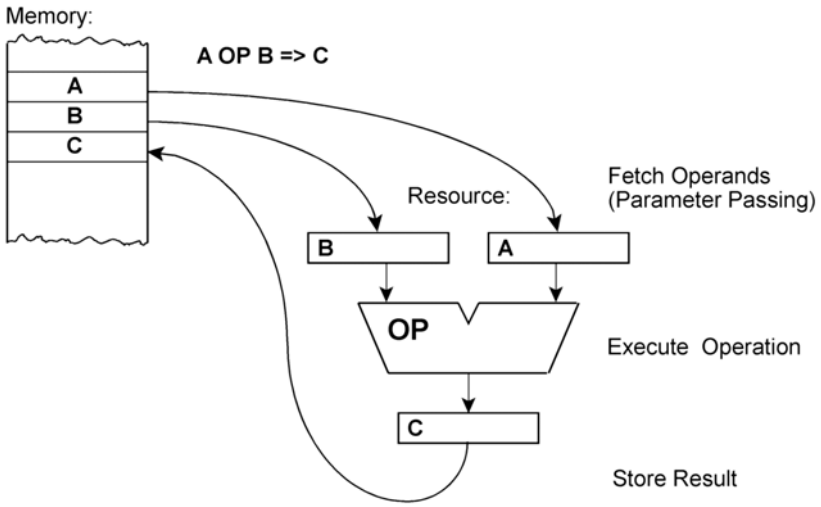

Fig. 2 Example processing steps

3.2. Operators

The processing steps are controlled by stored instructions. The abstract (machine-independent) instructions are called 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 (parameter passing): p-operator.
4. Initiate the information processing operations: y-operator (yield).
5. Move data between resources: l-operator (link).
6. Assign results: a-operator.
7. Disconnect concatenations: d-operator.
8. Return resources to the resource pool: r-operator.

Operators and Instructions
Operators describe the basic steps of information processing. Machine-independent ReAl programs are sequences of operators. To be stored and executed, operators have to be encoded. There are some flavors: textcodes, bytecodes, and fixed-format machine codes.

Textcodes are used if a machine-independent and human-readable representation is required. Typical examples are documentation, debugging, program development, and compiling.

To be executed on a certain machine, operators are to be mapped onto appropriate machine instructions. There are three basic variants:

1. One machine instruction corresponds to one operator.
2. One operator requires more than one machine instruction. An instruction encodes only an elementary processing step (for example, loading of a parameter address).
3. One instruction contains more than one operator.

Machine instructions can be laid out according to fixed-length or variable-length formats (bytecode). 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 used. The basic syntax: *operator_name (argument_list); operator_ name = s | c | p | y | l | a | d | r*. A parameter (operand or result) of a certain resource will be represented as follows: *resource . parameter*. Designators (of resources, and parameters) may be symbolic names or ordinal numbers. Parameter passing is denoted by =>. The assignment symbol is :=, the comment symbol is -- (refer to Ada and VHDL). For sake of clarity, the text code examples in the following descriptions contain only the minimum number of arguments. The extension to longer lists of arguments (more than one resource type, resource number or source-destination pair) is self-evident (refer to the right column of Listing 1).

1. Select resources: s-operator

s (resource type)
The s-operator is used to request resources of certain types out of the resource pool. The requested resources are enumerated sequentially. Other operators will reference the selected resources by those ordinal numbers. The effect of a s-operator depends on the kind of the underlying hardware:

- An appropriate function unit will be reserved and initialized. If necessary, appropriate storage areas will be reserved and initialized, too. (To initialize a resource means to set literals and initial values, to adjust the data width, to load control information (like microprograms) and so on.)
- An appropriate hardware structure will be generated (for example, by programming of cells within a programmable integrated circuit).

2. Establish concatenations between resources: c-operator

c (source resource **.** *result => destination resource* **.** *parameter)*
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. 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.

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

p (variable => resource **.** *parameter)*
The p-operator moves the specified variables (from memory or an I/O address space) into the specified operand parameter positions of the specified resources.

In resources which support concatenation, p-operators may initiate the execution of operations. The execution will begin if all operands are valid.

4. Initiate the information processing operations: y-operator

y (resource)
A y-operator initiates the execution of operations in the specified resources. The particular operation depends on the type of resource (if only one kind of operation can be executed) or on parameters (function codes) which are to be supplied previously (for example, by means of s- or p-operators).

An alternate method of operation initiation – without a y-operator – can be applied if the resource supports concatenation. The execution of an operation will be initiated if all required operands are valid. Valid operands could be delivered 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 (see [8])).

5. Move data between resources: l-operator

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

In resources which support concatenation, p-operators may initiate the execution of operations. The execution will begin if all operands are valid.

6. Assign results: a-operator

a (resource . result => result variable)
The a-operator moves the contents of the specified result parameter positions of the specified resources to the specified variables (in memory or in an I/O address space).

7. Disconnect concatenations: d-operator

d (source resource . result => destination resource . parameter)
The d-operator disconnects existing concatenations. In some implementations, the operator will change the corresponding physical connections (for example, within a switch fabric or an FPGA). Disconnected resources can be used separately or can be concatenated again.

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

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

3.3. A basic example

The following simple example (Fig. 3 and 4, Listings 1 and 2) illustrates how an application problem could be formulated by means of the described operators.

The application problem is to compute X := (A + B) • (C + D).

Fig. 3 shows a configuration of three resources; two adders (ADD) and one multiplier (MULT). Fig. 4 shows these resources concatenated according to the dataflow graph of the application problem.

The ordinal numbers of the resources: first adder = 1, second adder = 2, multiplier = 3. The ordinal numbers of the parameters (for each of the resources): inputs (operands) = 1 and 2, result = 3.

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) r (2) y (3) a (3. 3 => X) r (3)	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)

Listing 1 Computing X := (A + B) • (C + D). Left: Program written in a step-by-step notation. Right: Some code formats allow for longer argument lists, so more activities can be initiated by one operator.

```
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)
a (3.3 => X)
r (1, 2, 3)
```

Listing 2 This program concatenates the resources to a dataflow graph and initiates the calculation.

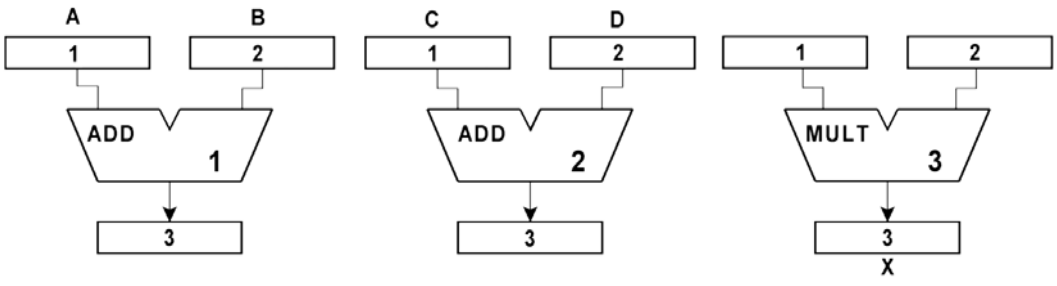

Fig. 3 Processing resources to compute X := (A + B) • (C + D).

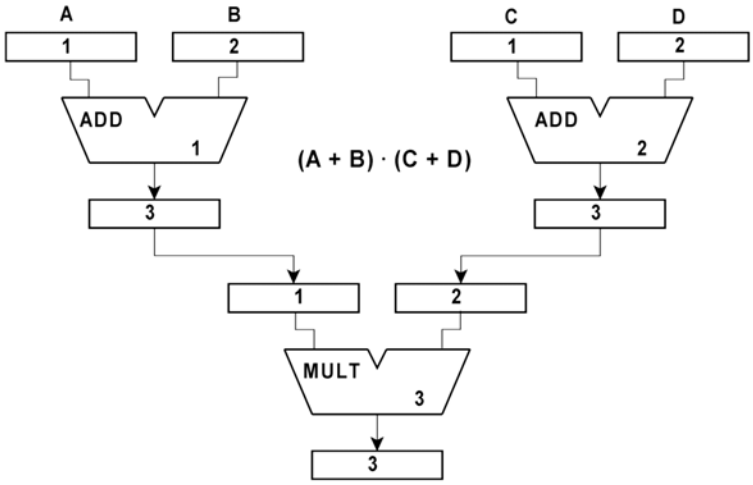

Fig. 4 Processing resources concatenated to compute X := (A + B) • (C + D). The registers 3 of the resources 1, 2 and the registers 1, 2 of resource 3 could be coalesced depending on the particular implementation of concatenation.

4. ReAl Machines

4.1. Processing Resources

A processing resource in a ReAl machine is a functional unit – more than a logic block of an FPGA ([11]) and less than a complete processor. An arithmetic-logic unit (ALU) with some addressing, control, and storage means may serve as a typical example. Compared to the operation units in contemporary high-performance processors, the resources in ReAl machines are less complicated (for example, typical resources will have no pipelining provisions).

Resources should be able to work autonomously. Our principal goal is to avoid information transport and operation cycles which are not necessary. Therefore, instruction fetch cycles as well as load and store cycles (concerning intermediate values, local variables and the like) have to be avoided whenever possible. In an ideal machine, only data related to the solution of the application problem would be fetched and stored. In a ReAl machine, instructions will set up configurations of resources which corresponds to parts of the dataflow graph of the application problem and leave the execution of operations and the transport of intermediate data to the processing resources. Each resource knows what it has to do (set up by y-operators), which parameters are to be processed and where the results are to be delivered (set up by p- and c-operators).

Some potential candidates for partial dataflow graphs:

- Basic blocks (linear sequences of instructions between jumps or subroutine calls).
- Innermost loops.
- Conditional statements.
- Subroutines (for example, C-like functions).

A particular resource can be:

- A circuitry with fixed functions.
- A circuitry with selectable functions.
- A program-controlled circuitry (controlled by conventional machine instructions or by microinstructions).
- A storage area and an appropriate program (emulation).
- A storage area and a description of an appropriate circuitry (like net lists, Boolean equations or HDL code). This hardware description can be simulated by software or used to synthesize the circuitry and implement it on a programmable IC (FPGA).

The proposed architecture can be implemented with:

- Conventional processors.
- Modified processors (new instruction decoder, modified register set, appropriate microprograms).
- Optimized processing hardware designed from scratch.
- Programmable integrated circuits (FPGAs).

The advantages of the proposed architecture will be fully effective if the hardware is designed thoroughly with the particular principles of operation in mind. Appropriate systems comprise:

- Platform resources.
- Processing resources.
- Storage and I/O means.

I/O devices are attached the conventional way (for example, via some of the well-known bus and network interfaces). The programming interface comprises simply some kind of address space, complemented by an appropriate interrupt mechanism, if required. These principles are well known. Hence more detailed descriptions of I/O provisions may be omitted.

Basic configurations are similar to conventional computer systems (Fig. 5). Processing resources, storage means and the platform are attached to a common system bus. In contrast to conventional computers, however, the processing resources are not restricted to a single arithmetic-logic unit or to a few operation units. In order to increase performance, more then one bus can be provided (Fig. 6, 7).

Fig. 7 illustrates how more than one resource can be controlled in parallel. Instead of conventional instructions with only one operation code, so-called access control words are provided in which the effects of many operators are encoded together. Principally, access control words can be made wide enough so that one access control word can initiate all transport and processing operations which can take place at one time in all hardware resources. In the example (Fig. 7) the access control word comprises one bit position for each resource and operation. The meaning of those bits:

1.Op: Load a parameter from the operand bus into the first operand register of the particular resource.
2.Op: Load a parameter from the operand bus into the second operand register of the particular resource.
Y: Initiate a processing operation.
Res: Gate the result register to the result bus.

The access control word contains an additional field Comm.Ctl. In this field, the following operations are encoded:

- Gate memory data to the operand bus.
- Store result.
- Loop back result to the operand bus.
- Select next access control word (including branching and subroutine call).

Obviously, Fig. 7 shows an extremely simplified example, as bus structures allow only for one data transfer in one time. Fig. 8 illustrates an alternate solution: fast point-to-point connections via switch fabric or switching hubs. Such point-to-point interfaces are readily available. Switched interconnections allow for numerous data transfers occurring simultaneously. Appropriate access control words have address fields instead of the single bits shown in Fig. 7.

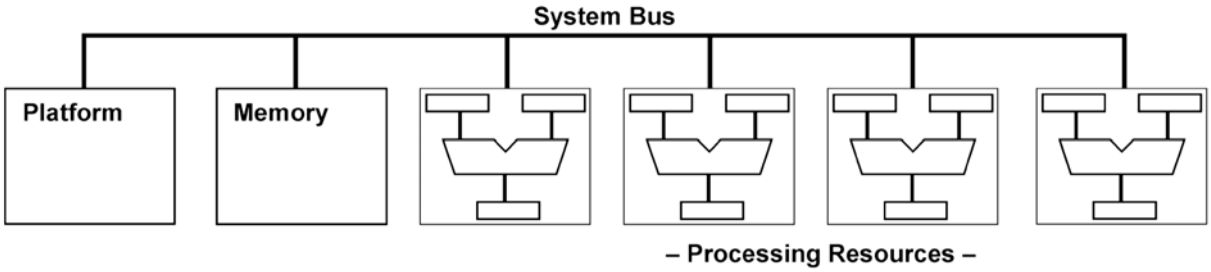

Fig. 5 A ReAl machine with multiple buses.

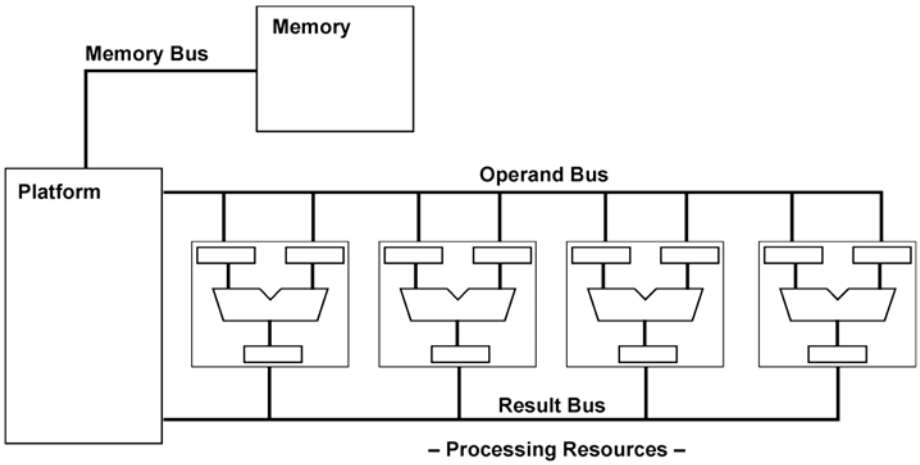

Fig. 6 A machine structure example with multiple buses

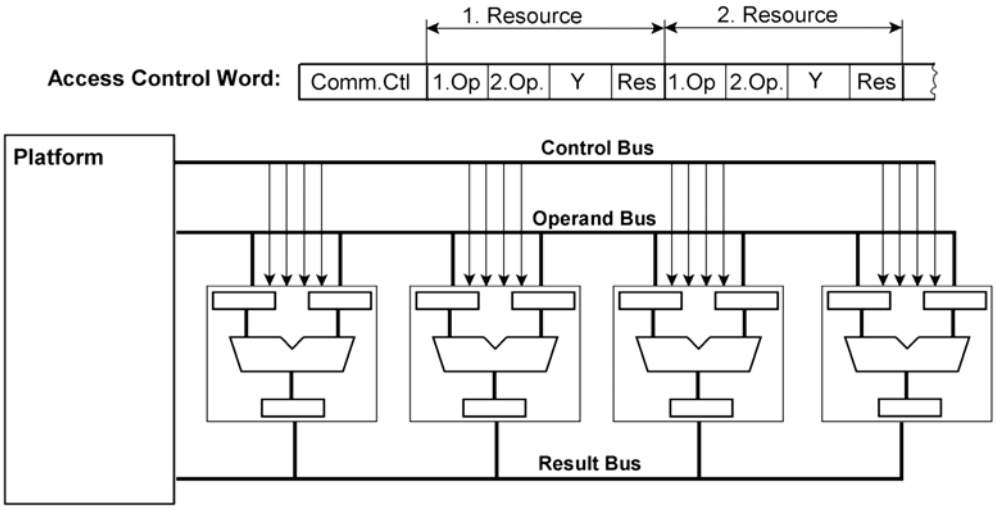

Fig. 7 Processing resources controlled in parallel

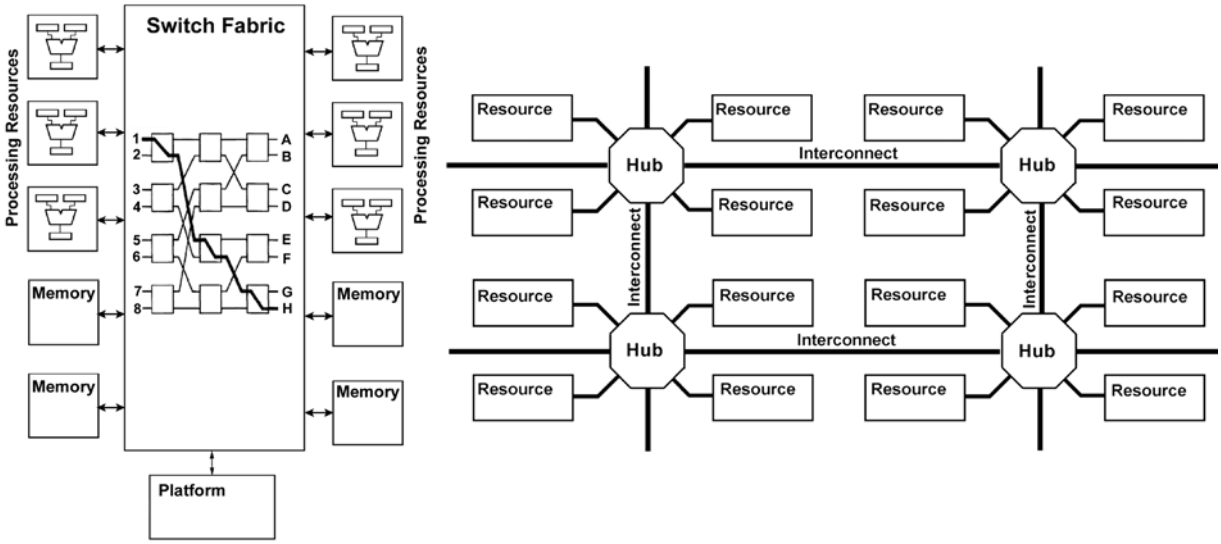

Fig. 8 ReAl machines based on switched interconnections.

High-Performance Resource configurations

The interaction of resources concatenated arbitrarily seems to require complicated and expensive interconnection networks. Furthermore, one may suspect that the communications overhead may cost more machine cycles than the conventional instruction fetches and the like we try to avoid. We assume that these problems can be circumvented by restricting the hardware topology to a few essential configurations and to coalesce processing resources and memory.

Processing resources and memory

The history of computer architecture has seen numerous attempts to bring memory and processing hardware together more closely. A few of them have been proven quite successful (above all, caches and register files). For ReAl machines, some variants are under investigation (more details can be found in the design documentation ([8])):

1. Resources with a certain amount of memory (for example, with a memory capacity large enough to hold the local variables of typical C-functions or some vectors of floating point data).
2. Resources built into the system memory (for example, each memory bank has its own processing resources).
3. Processing resources attached to cache lines.
4. Processing resources connected to general purpose registers (for example, a register file of 32 to 256 registers is divided into blocks of four registers, each with its own processing resource).

Fig. 9 shows a processing resource (a so-called resource cell) comprising processing hardware and a certain amount of memory. Fig. 10 illustrates an arrangement of such resource cells on an integrated circuit. The cells are connected via bus systems. Each bus can supply parameters to one row of cells and pick up the results of the cells in the row above.

The inverted binary tree – an effective topology
We follow a conjecture stated in [6] and assume that only two configurations need to be supported in hardware (all other topologies could be emulated (virtual connections)):

1. Independent resources operating in parallel.
2. Inverted binary trees.

The evaluation of nested expressions (including function calls) can be mapped well onto inverted tree structures. Ths is also true for operations which compute a single result from multiple data (like SAXPY or SAD (an example of the latter will be discussed below)). Architectures comprising inverted tree structures of operation units have been suggested in [6], [10] and [11].

In Fig. 11 the resource cells between two bus systems constitute inverted binary trees. Within such a tree, the data paths between the cells are simple point-to-point connections; they are short and need no programming provisions. In order to make best use of silicon real estate, the tree structures are arranged alternately.

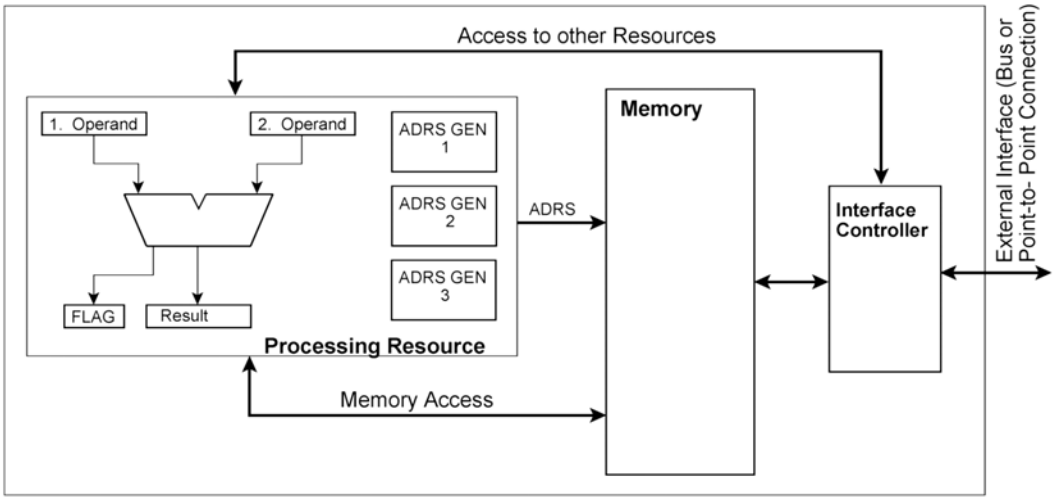

Fig. 9 An example resource cell.

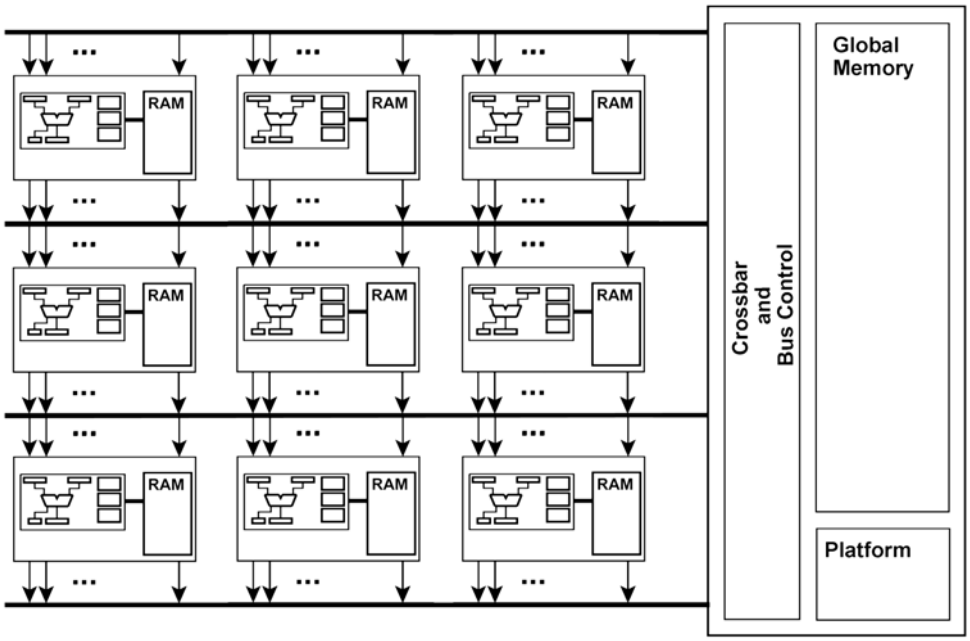

Fig. 10 An integrated circuit with resources placed between bus systems.

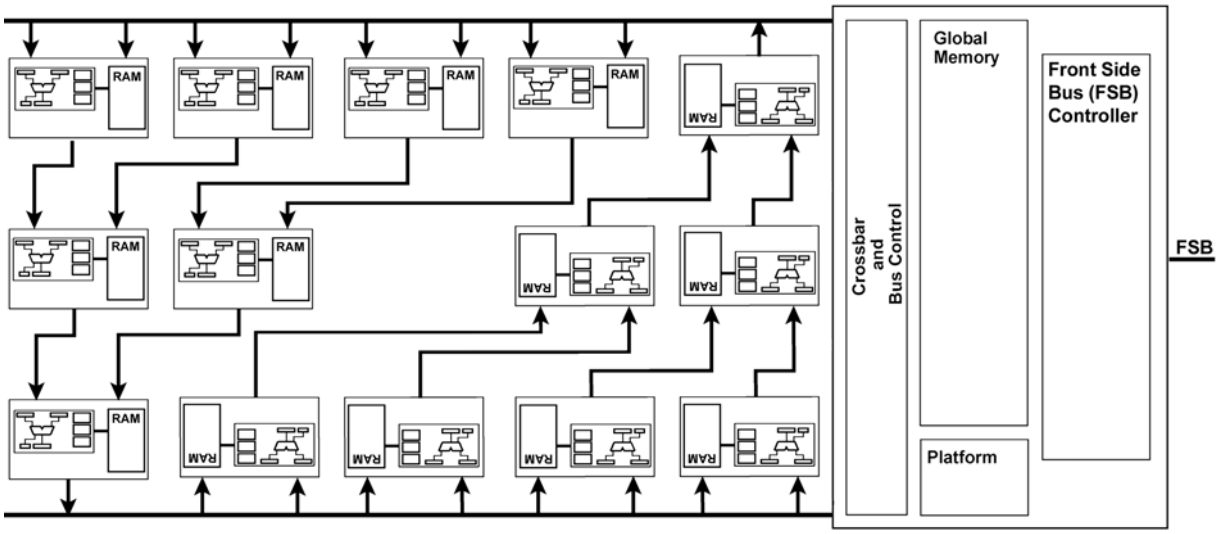

Fig. 11 An FPGA circuit with resource cells connected to inverted binary tree dataflow graphs.

ReAl architecture and FPGA hardware

Fig. 8 to 11 may be used to illustrate configurations of resources on programmable integrated circuits (FPGAs). The usual FPGAs contain (besides interconnections and I/Os) basically nothing but two kinds of building blocks:

- Universal function blocks, macrocells and the like. Usually, one to four flipflops are provided in each block or cell, and few product terms can be implemented.
- Hard IP Cores, for example, complete RISC processors or DSPs.

The ReAl architectural principles may lead to programmable integrated circuits containing resource cells of intermediate granularity and appropriately optimized interconnects; the “fat“ hard processor will virtually be broken up into its functional units which can be used to build universal or special processors on the fly under program control.

4.2. 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:

- Special addressing resources are concatenated to processing resources (Fig. 12).
- Appropriate addressing and memory access capabilities are provided within the processing resources (Fig. 13; see also the address generators in Fig. 9).

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. 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 to 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. 12, will operate as follows:

- Operand and result addresses will be loaded into the address registers of the addressing resources 1 and 3 (p-operators or l-operators or concatenation).
- The addressing resources 1 will be activated (by means of y-operators or as a result of operand concatenation). They will initiate read accesses to memory.
- Once the data registers within the addressing resources 1 has received the memory data, the concatenation to the operand registers of the processing resource will become effective. The operand values will be passed to the processing resource 2. As soon as the last operand value has been received, the processing resource 2 will initiate its processing operations.
- When the processing resource 2 has completed its operations, the concatenation of its result register to the data register of addressing resource 3 will become effective.
- The delivery of the result will cause the addressing resource 3 to initiate a write access to memory.

According to Fig. 13, the addressing and memory access provisions can be built into the processing resources. Such a resource will operate as follows:

- Operand and result addresses will be loaded into the corresponding address registers (p-operators or l-operators or concatenation).
- The processing operations will be initiated (by means of y-operators or as a result of operand concatenation).
- The access control sequencer will initiate the read accesses.
- The operands read out of memory will be loaded into the operand registers.
- The result will be computed.
- The access control sequencer will initiate a write access to store the result.

Addressing means can be enhanced 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.

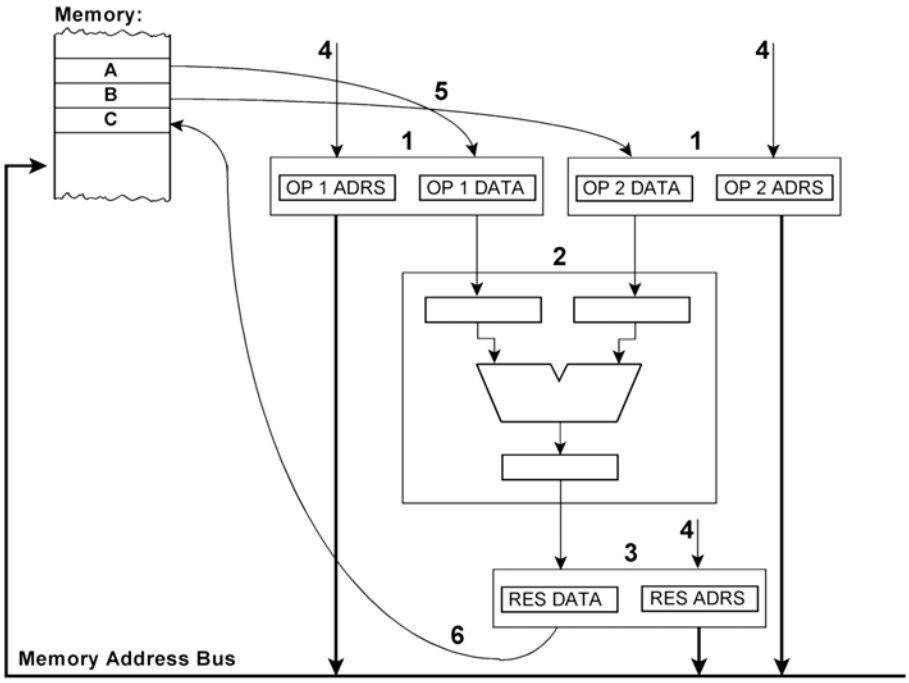

Fig. 12 Processing resources concatenated to addressing resources. 1 - operand addressing resources; 2 - processing resource; 3 - result addressing resource; 4 - address parameters delivered by p-operators, l-operators or preceding concatenations; 5 - adressing resources fetch operands from memory; 6 - adressing resource stores result into memory

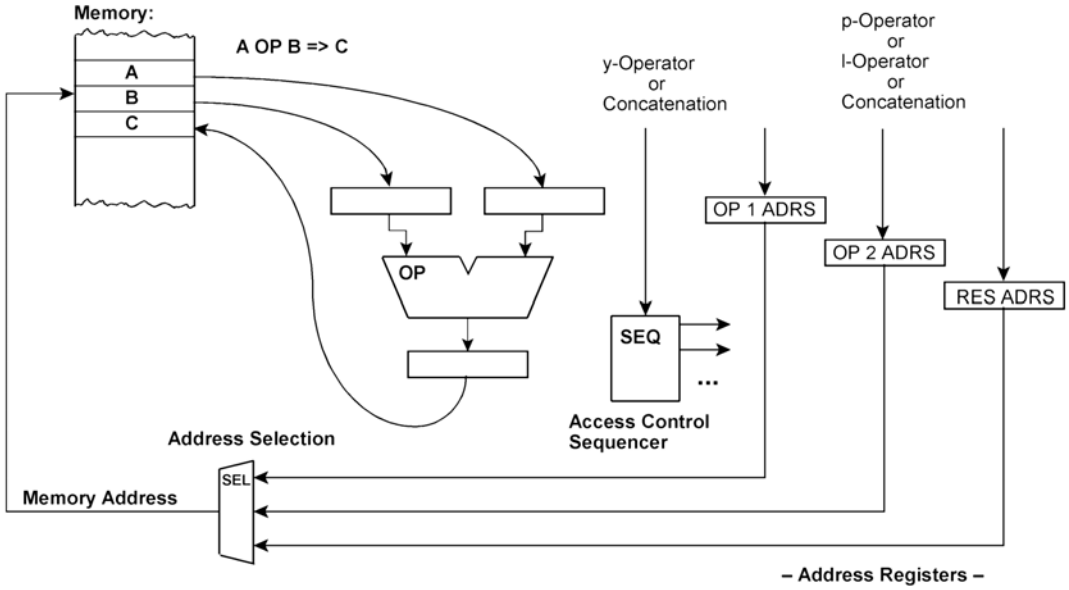

Fig. 13 Processing resource with built-in addressing capabilities

4.3. The Platform

The platform comprises the resources which are necessary to initiate and sustain the operation of the system. The most important task is the fetching and execution of machine instructions. There are some alternatives to implement platforms, for example:

- A conventional processor.
- A microprogrammed control unit.
- An arrangement of more basic resources.

The following description will concentrate on platforms built from basic resources. The most elementary platforms can only read instructions. All other functions (like fetching operands from memory) require additional resources to be provided.

Control hardware of all kind tends to be complicated. Hence we will illustrate the principles of platform implementation only by one example: conditional branching.

Fig. 14 and Listing 3 show how conditional branches are to be executed. The platform has three parameters which can be loaded by means of p-operators, l-operators, or concatenation. The branch address is to be brought into the branch address register BA, the branch condition (for example, branch on zero or branch on carry) into the branch condition register BCTL. The third parameter is the result condition on which the branch decision is to be based (for example, flag bits resulting from an algebraic operation). The resource which computes the corresponding result will pass this condition to the condition bit register BC (which can be seen as some kind of centralized flag register). The BC register can be concatenated to the processing resource so that the branch will be taken as soon as the result has been calculated.

```
p (address => BA, condition => BCTL)   -- set branch address and branch condition
....                                   -- other operators may follow
l (result => BC)                       -- current condition is passed to the branch resource
y (branch resource)                    -- branch will be taken if conditions match
```

Listing 3 Conditional branching basic operator sequence. If the current condition (passed to BC) matches the branch condition (in BCTL), the branch will be taken (branch address BA will be passed to instruction counter IC).

A basic branch operation example

According to Fig. 14, a processing resource is concatenated to the branch resource. Conditional branch operation will be shown by way of a simple example (Listings 4 and 5).

The programmer‘s intention:

```
C := A + B
if CARRY_OUT then goto OUT_OF_RANGE
```

The resources:

1. Processing resource ADD. A conventional adder (or ALU). Computes sum 3 and flag bits 4 from operands 1 and 2.
2. Branching resource INSTR_CTR. The instruction counter consists of the instruction register IA and an incrementing circuitry CT. The parameter ordinals: 1 - branch address; 2 - branch condition; 3 - condition bits to be evaluated; 4 - result = address of next instruction.

```
s (ADD, INSTR_CTR)
p (A => 1.1, B => 1.2, OUT_OF_RANGE => 2.1, CARRY_OUT => 2.2)
y (1)                                          -- add (A + B)
a (1.3 => C)                                   -- assign result to variable C
l (1.4 => 2.3)                                 -- pass flag bits to the branch resource
y (2)                                          -- activate the branch resource
```

Listing 4 A branch operation example.

```
s (ADD, INSTR_CTR)
c (1.4 => 2.3)                                 -- concatenate flag bits to BC register
p (A => 1.1, B => 1.2, OUT_OF_RANGE => 2.1, CARRY_OUT => 2.2)
y (1)                                          -- add (A + B)
a (1.3 => C)                                   -- assign result to variable C
y(2)                                           -- activate the branch resource
```

Listing 5 Branching by concatenation.

In conventional pipelined hardware, branches are difficult to implement, since taken branches interrupt the flow through the pipeline. To reduce the loss of speed, extensive countermeasures are to be implemented (branch prediction, branch target buffering). Within a ReAl machine, branch target buffering can be completely under program control. If performance has first priority, it is even feasible to request enough resources in order to perform the operations in both branch directions simultaneously.

Even in the simple configuration of Fig. 14 the branch target address can be provided ahead of the branch operation, facilitating speculative instruction fetching in branch direction. Fig. 15 illustrates the extension of this principle to more than one branch target. The branch target addresses will be loaded into the branch address buffer 1, the corresponding branch conditions into the branch control buffer 2, and the first branch target instructions into the branch target buffer 3.

The conditions signalized by the processing resources (l-operators or concatenation) will be passed to evaluation circuitry 4 attached to the branch control buffer 2. If the conditions match, the corresponding entries of branch address buffer 1 and branch target buffer 3 will be passed to instruction address register IA and instruction register IR, respectively. The configuration shown in Fig. 15 could be extended to support multiway branches. Example: an arithmetic compare operation may yield the conditions $<$, $\leq$, $=$, $>$, $\geq$. Corresponding to those five conditions, five buffer entries cold be used to hold the addresses and first instructions of the five possible branch directions.

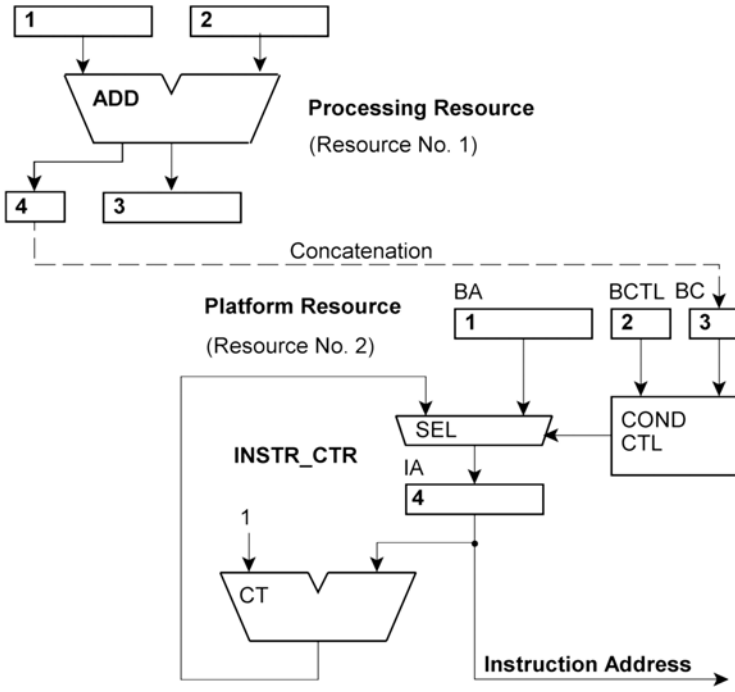

Fig. 14 Processing resource initiates branch on condition. BA - Branch address; BCTL - Branch control; BC - Branch condition; IA - Instruction address. The numbers 1...4 are the parameter ordinals of the resources.

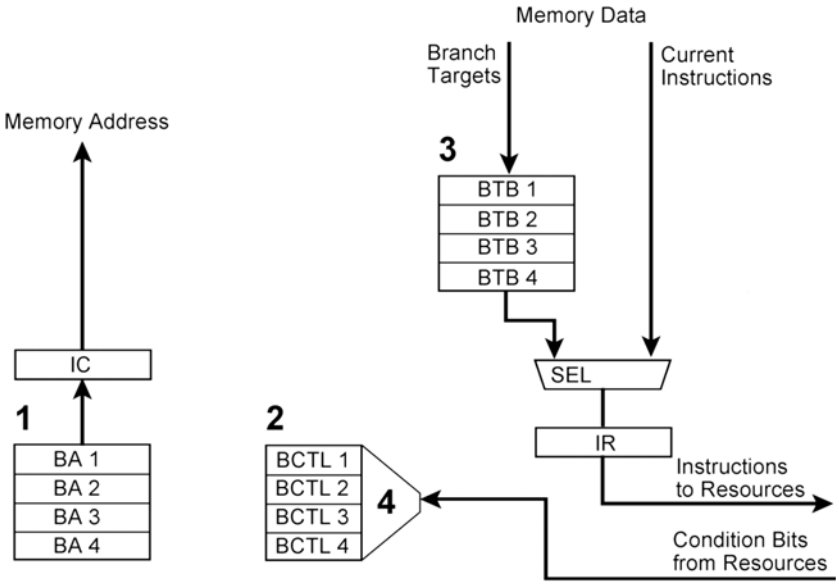

Fig. 15 Multiple branch target buffering within a platform resource. 1 - branch address buffer; 2 - branch control buffer; 3 - branch target buffer; 4 - branch condition evaluation.

5. Related Work

Increasing performance by providing more operation units or even whole processors and by exploiting every kind of parallelism have been primary research topics over decades. Therefore, it will be sufficient to mention only a few areas of research and development which are related to the ReAl proposal more closely.

Superscalar Architectures
The multiple operation units (resources) in superscalar machines are controlled by appropriately formatted instructions (explicit instruction level parallelism) or by a speculation mechanism. This mechanism tries to emulate some kind of dataflow machine, executing instructions according to the availability of the data to be processed (for state-of-the-art processors see [13]).

Multiprocessor systems
Multiprocessor systems have a long history. The current semiconductor technology is able to provide more than one processor on one integrated circuit. There are complete systems optimized thoroughly for certain kinds of application and multiple processor cores on FPGA circuits which can be supplemented by application-specific hardware (for examples, see [3] and [12]). The particular processors are typically high-performance cores (like 32-bit or 64-bit RISC Machines). These processors are powerful, but complex and inflexible in themselves. To cope with inherently sequential as well as with easily parallelizable algorithms, it has been proposed to provide a powerful (and hence complicated) processor for inherently sequential problems surrounded by a few less complex processors for problems which are suited for parallel execution ([4]).

Spatial Computing
It is obviously a buzzword for more than one research topic, covering semiconductor technology, processing units and interconnects on integrated circuits ([2]) and the like as well as large networks of processing resources of different granularity ([1]).

6. Discussion

Superscalar Architectures and ReAl
What a current superscalar machine does implicitly and speculatively at a comparatively small scale (for example, with 4 to 16 operation units), a ReAl machine could do explicitly and in a deterministic way at a scale only limited by semiconductor technology. Table 2 shows characteristic features of conventional superscalar and ReAl machines in contrast.

A rough estimate:
On an integrated circuit with 200 million transistors, it would be possible to arrange four superscalar processor cores, each having approximately 50 million transistors (refer to [8] and [13]). The operation units of such a processor correspond roughly to eight 64-bit arithmetic/logic units (the differences between integer and floating point units being neglected here). These 4 cores • 8 operation units correspond to 32 resources. The instruction fetch and execution control hardware is to be replaced by ReAl platform circuitry. Cache memories, control circuits, and bus systems are maintained (same size, but modified structure). Some more resources could be located on the silicon area otherwise occupied by additional circuitry (pipelining, detection of hazards and so on). Therefore, one can reasonably expect a processor IC containing approx. 48 to 64 high-performance processing resources. According to the requirements of the applications to be executed, this ensemble of resources could be morphed into graphic engines, database engines and so on under control of ReAl instructions.

Conventional superscalar machines	ReAl machines
• More than one operation unit (2 to 16 are typical) • Complex pipelined circuitry • Provides partial data flow operation by speculation • Complex hardware to detect inherent parallelism between instructions • Complex hardware to detect conflicts and hazards during execution • Rigid processor structures (the application problem must match, or there will be inefficiencies) • Conventional instruction set. Downwardly compatible instruction set architectures can be supported (see the processors of the personal computers)	• More than one processing resource (see below) • Each resource is a comparatively less complex, non-pipelined circuitry • Provides partial data flow operation based on a deterministic description (c-operators), • Inherent parallelism detected during compile time; no dedicated circuitry required • The ensemble of resources could be morphed to (virtual) application-specific machines • New instruction set architecture; describing parallelism and dataflow operation in detail

Table 2 Conventional superscalar vs. ReAl machines

Optimization of high-performance processors vs. ReAl

Some activities to develop optimized high-performance processors correspond to important goals of the ReAl approach (Table 3, Fig. 16).

Recommendations to improve computational throughput in conventional processors ([9])	Within ReAl machines, these recommendations will be more than fulfilled . . .
• Reduce the amount of load/store cycles (more than 30% of the instructions executed in a RISC architecture are load and store instructions)	• If resources can be set up according to the data flow (for example, of an innermost loop), no load/store cycles (concerning intermediate results, local variables and the like) will be needed at all. Even the instruction fetch cycles are avoided, as the control codes have been loaded into the resources.
• Streamline repetitive operations (perform time-critical operations on multiple data simultaneously)	• This will pose no problem if enough processing resources are available (see Fig. 16)
• Maximize utilization of pipeline resources	• Since the processing resources are not part of a rigid hardware pipeline but can be interconnected freely, some utilization problems and hazards are avoided which are typical of pipelined hardware
• Minimize branch latency	• Can be achieved by appropriate platform design. Even multiway branches can be supported (see Fig. 15)

Table 3 Optimization of conventional high-performance processors vs. ReAl

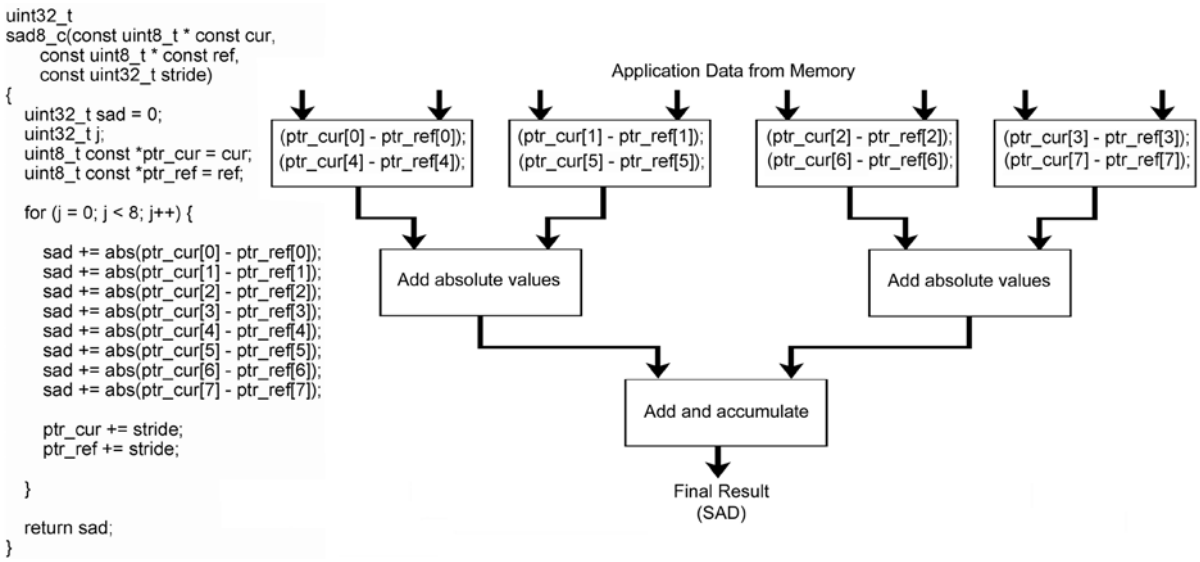

Fig. 16 An algorithm to calculate the sum of absolute differences (SAD). Left: The original C-code (source: xvid codec). Right: an appropriate configuration of processing resources (see also Fig. 11).

The example of Fig. 16 has been used in [9] to illustrate optimization problems of conventional processor architectures. Obviously, the SAD value could be calculated within a loop. But, to gain some speed, this (innermost) loop has been unrolled. 16 operand values are to be fetched and 32 operations have to be executed. Each operation requires at least one instruction, which is to be fetched, too. These operations can easily be mapped onto an inverted tree of concatenated processing resources. Some kind of pipelining will occur automatically as consequence of the concatenation mechanism (see [8] for details). Once the resource configuration has been set up, there is no need to fetch more instructions, as all control codes reside within the resources. Therefore, the memory access paths and the total memory bandwidth will be available for moving the application data.

Multiprocessor Systems vs. ReAl

Obviously, multiprocessor systems are advantageous if the application problem matches the system structure. However, there are some basic drawbacks which apply to all systems built of multiple independent processors:

- Each processor needs its own instruction fetch mechanism, instruction cache, instruction sequencer and so on.
- The synchronization between processors is difficult, requiring special hardware means (like test-and-set instructions and cache coherency provisions), and causing overhead during runtime.
- If the particular processor is too small, and if the amount of non-parallelizable code cannot be neglected, then Amdahl's Law will be effective.
- Some processors will be unused if the processor arrangement does not match the structure or the size of the application problem (for example, 16 processors but only 7 threads to be executed in parallel).

To a large extent, the ReAl approach has been stimulated by the desire to circumvent these drawbacks. The key points can be summarized as follows:

- To break down the complete processor into its functional units (in other words: to provide less complicated processing resources, but more of them).

- To provide for sufficiently efficient, optimized interconnections.
- To develop principles of operation and instruction set architectures which can cope with such hardware configurations.

Spatial Computing and ReAl

What spatial computing research has in common with the ReAl proposal is that it relies on an abundance of resources. Spatial computing ideas have emerged from semiconductor technologies and large-scale networking, whereas the starting point of the ReAl proposal has been mathematical abstraction and instruction set design, deliberately omitting semiconductor and networking problems. Obviously, there may be some convergence:

- Results of networking and semiconductor research (for example, new interconnection circuitry as described in [2] could be used to implement ReAl machines).
- ReAl principles could be used to develop instruction set architectures (including some kind of universal bytecode (like JVM)) for programming of spatial computing systems.

7. Summary

A universal machine language and hardware-software interface has been proposed which can bring a practically unlimited number of processing resources into effect and which allows for completely describing and exploiting the inherent parallelism of the application problems. During runtime, the exploitation is only limited by the available hardware.

The execution of the processing steps is controlled by appropriate instructions (operators). Connections can be established corresponding to the dataflow graph of the application problem (creation of virtual special processors). If such structures once have been set up, they will show less overhead than conventional architectures (it will not be necessary to create stack frames, to store and read again intermediate results and so on). Resources can be implemented with hardware as well as with software. Appropriate hardware consists of processing resources, platform resources and storage means.

Since inherent parallelism is not detected during runtime but immediately from the programming intentions, it will be feasible to exploit hundreds of processing units to speed up the operation of a particular application program. Hence the proposed architectural principles will allow to make use of future progress in circuit integration (some hundred millions of transistors on one integrated circuit).

The proposed principles may lead to future true supercomputers being neither vector processors nor processor farms but ensembles of resources which can be dynamically configured to support even massively data-dependent algorithms.

Call to Action:

- Implement emulators on industry standard computing platforms (PCs, microcontrollers).
- Implement appropriate hardware solutions (experimental machines based on state-of-the-art FPGAs).
- Develop new FPGA architectures with embedded medium grain resource cells.
- Modify state-of-the-art superscalar processors into ReAl hardware platforms (caches, operation units and bus systems remain, instruction decoder, register sets, instruction sequencing and microprograms are to be modified).
- Develop true supercomputer architectures (neither vector processors nor processor farms).
- Write appropriate compilers.
- Write top-notch applications which make use of the ReAl paradigm.

8. Bibliography

[1] DeHon, A.: Very Large Scale Spatial Computing. In: Proc. of Third International Conference on Unconventional Models of Computation 2002.

[2] Goldstein, S. C.; Budiu, M.: nanoFabrics: Spatial Computing using Molecular Electronics. In: Proc. of The 28th Annual International Symposium on Computer Architecture, pages 178 - 189, June 2001.

[3] Hofstee, H. P.: Cell Broadband Engine Processor: Motivation, Architecture, Programming. Keynote at MICRO 2005.

[4] Jouppi, N.: The Future Evolution of High-Performance Microprocessors. Keynote at MICRO 2005.

[5] Matthes, W.: Hardware Resources: a generalizing view on computer architectures. ACM SIGARCH Computer Architecture News, Vol. 18 , Issue 2 (June 1990), pages 7-14.

[6] Matthes, W.: How many operation units are adequate? ACM SIGARCH Computer Architecture News, Vol. 19, Issue 4 (June 1991), pages 94-108.

[7] Mueller-Wichards, D.: An Algebraic Approach to Performance Analysis. In: Parallel Computing in Science and Engineering. Springer, 1988 (LNCS 295), pages 159-185.

[8] ReAl Design Documentation. Sources:
- Homepage http://www.realcomputerarchitecture.com.
- Pending patent applications:
  1) DE 10 2005 021 749.4 “Verfahren und Vorrichtung zur programmgesteuerten Informationsverarbeitung".
  2) US 11/430,824 “Method for Information Processing“.

[9] Uthus, J.; Strom, Oy.: MCU Architectures for Compute-Intensive Embedded Applications. Atmel White Paper. Atmel Corporation, 2006. Http://www.atmel.com.

[10] Vlakos, H.; Milutinovic, V.: GaAs microprcessors and Digital systems. An Overview of R&D efforts. IEEE Micro Vol. 8 No. 1 (February 1988), pages 28-56.

[11] Wulf, Wm. A.: The WM Computer Architecture. ACM SIGARCH Computer Architecture News, Vol. 15, Issue 4 (September 1987), pages 70-84.

[12] Xilinx Corporation. Virtex series FPGAs. Http://www.xilinx.com.

[13] Intel Corporation and AMD Corporation. Their Internet sites contain comprehensive literature related to state-of-the-art superscalar processors and corresponding future development. Http://developer.intel.com, http://www.amd.com.