1. Introduction

1.1 Overview

The primary objective of ReAl architecture design is to utilize the inherent parallelism in information processing operations to the highest possible degree, in other words, within the limits that result from the application problem to be programmed, on the one hand, and the respectively available hardware, on the other hand, and to provide interfaces between hardware and software that ensure arbitrary interchangeability of hardware and software.

The ReAl principles of operation are intended to provide methods for utilizing information processing means so that any number of them can be addressed freely by a program or can be freely configured to hardware structures that carry out the desired information processing operations.

The architecture 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[1)]. Hence the name ReAl = *Re*source-*Al*gebra.

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

- Hardware does not matter.
- Memory Capacity does not matter.
- Hardware requirements for machine program generation do not matter.

The 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.

ReAl principles at a glance:
The ReAl architecture is based on the fact that any program requires always a hardware in order to be executed; essentially, the program is transformed into information transports, combinational operations, and state transitions, in other words, into the flow of information through a register transfer structure. In order to implement a certain programming intention, suitable resources will be selected out of a 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 carried out 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

---

1): Similar paradigms have been used occasionally over more than one decade for performance analysis and abstract modeling of architectural principles ([6], [9]). Here this basic approach will be applied to instruction set design.

(operators). So-called platform resources are provided to fetch the instructions from memory. Additional instructions establish and disconnect, respectively, connections (concatenations) between resources. 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.

The ReAl design rationale in ten points:

1. Starting with the programming intention, an appropriate register transfer structure is configured ad hoc, basically from elementary processing devices that are referred to as resources. A resource in this respect is to be understood, for example, as a conventional arithmetic logic unit (ALU) but also as complex special circuitry. The general model of a resource is a hardware unit that performs certain information processing operations (from given data at the inputs new data at the outputs will be computed).
2. There are sufficient resources available at any time. This is initially a theoretical assumption (hypothesis of (nearly) unlimited (transfinite) resource pool). Based on this principle, it is possible to request any number of resources (for example, several hundred multiplication units) and to utilize the inherent parallelism to the fullest. Machine programs are typically generated (for example, by means of compilers) as if any number of resources were available. 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)).
3. Resources can be implemented by means of software or hardware; programs and hardware resources are handled the same way.
4. The basic model of a resource is always a piece of hardware with input registers, combinational circuitry and output registers (register-transfer model).
5. A processing operation (program sequence) will be implemented by the utilization of resources during the course of time (resources are fetched from the resource pool as needed and returned when not longer in use).
6. With respect to an application problem, the universal computer is considered to be 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.
7. The instructions (operators) describe only the basic processing steps, but not the concrete operations to be performed (like addition or multiplication).
8. More complex resources can be configured recursively from elementary resources.
9. Instructions (operators) that control the processing steps in ReAl machines can be generated, transported or modified by applying the ReAl principles of operation recursively. For example, special resource configurations could be built ad hoc for purposes of machine program generation (like compiling).
10. It does not matter where the resources are located and how they are implemented. It is even possible to request and utilize resources via the Internet (for example, special processors).

Essentially, ReAl programming means to turn program development into some kind of hardware design – a ReAl program can be thought of as an assembly instruction for building a special hardware configuration[1)] that can carry out the respective processing task, initially as a thought experiment independent of the actual practical feasibility. This virtual hardware can be configured, modified, and released dynamically during run time. It is decided case by case, which configuration is actually to be implemented in hardware and which is not. If a resource is not directly available as hardware, it could be built from more elementary resources or its function could be emulated with other resources based on the ReAl principles of operation (recursion) or with conventional machine programs.

The ReAl architecture can be implemented:

- With conventional general-purpose computers.
- With modified general-purpose processors (modified instruction decoder, modified register file, different microprograms etc.).
- With special processors designed from scratch according to the ReAl principles of operation.
- With programmable integrated circuits (FPGAs).

The individual resource can be:

- A circuitry with fixed function.
- A circuitry with selectable functions.
- A program-controlled circuitry (controlled by conventional machine instructions or by microinstructions).
- An appropriate memory area, supplemented by program control operators (emulation).
- An appropriate memory area, supplemented by a description of a circuitry that can perform the respective information processing operations (for example, in the form of netlists or Boolean equations). This description can be used to simulate the circuitry. Alternatively, the circuitry in question can be generated on the fly by programming of appropriate FPGAs.

Instructions and operators
The ReAl architecture terminology makes a difference between instructions and operators.

Operators are the basic (in a general sense, logical) control structures according to the ReAl principles of operation (we speak of s-operators, y-operators and so on (details in chapter 2)).

Instructions are the control structures of the hardware. There are different possibilities for implementing instructions:

- They are machine-specific encoded ReAl operators (1:1-correspondence).
- They are formatted similar to conventional machine instructions or microinstructions. Typically, one ReAl operator corresponds to a sequence of such instructions.
- The operator functions are emulated with or compiled into sequences of conventional machine instructions or microinstructions.

---

1): Preferably a resource configuration that corresponds to the data flow diagram of the respective application problem.

Some ReAl advantages:

- The inherent parallelism in the programs to be executed can be exploited according to the hardware being actually available.
- Essentially, an arbitrary number of resources can be addressed (no limitation of the number of resources, as is the case, for example, in so-called VLIW architectures).
- Hardware means for conflict detection, instruction retry, retirement and the like are not required.
- The allocation of resources can be program-controlled in detail; it is possible to configure for each processing task ad hoc a type of virtual special machine and to release it again if no longer needed.
- Once such structures are configured, the overhead during run time is significantly reduced in comparison to conventional machines (no build-up and release of stack frames, no storing and rereading of intermediate values).
- Memory means and operation units can be connected directly with one another (in comparison to the register files of the conventional high-performance processors, fewer access paths are required and the address decoding is simplified).
- The operation units can be embedded in memory arrays (resource cells, active memory arrays) so that very short access paths are provided. This simplification of the hardware provides for increasing the clock frequency, saving on pipeline stages (shortening of the latency of operation execution), and arranging on a given silicon real estate a larger number of operation units (or more powerful operation units).
- The perspective possibilities that are provided by the semiconductor technology (for example, a few hundred million transistors on an integrated circuit) can be utilized to a large degree. Since the inherent parallelism is detected directly from the programmer's intentions (in statu nascendi), it is possible to optionally utilize even hundreds of processing units at the same time in order to accelerate the execution of the individual programs.
- Depending on the cost and performance goals and depending on the state of technology, hardware and software can be interchanged with one another (for example, a subroutine can be exchanged for a special processing unit and vice versa). Corresponding programs are therefore invariant with regard to technological development; they can utilize any progress of circuit integration without problems.
- Systems can be realized on programmable integrated circuits that represent basically arbitrary combinations of hardware and software.
- Auxiliary functions, for example, debugging, system administration, data encryption and the like, that require conventionally additional software routines (loss of speed) or special hardware (cost) can be organically embedded into the resources (the additional cost is minimal because, as a result of the direct connections, more possibilities for circuit optimization are present and the system efficiency is not affected). Moreover, additional resources can be taken from the general resource pool in order to configure corresponding devices as needed (when the respective function, for example, for debugging, is no longer required, the resources in question are again available for general use).
- In contrast to conventional operating instructions, the selection of the operations (s-operator) is separate from the initiation of the operation (y-operator). Each resource knows thus from the beginning for which purpose the transferred parameters are to be used. This can be utilized optionally for optimizing the hardware. The initiation encoding (y-operator) is typically shorter than the selection encoding (s-operator). This is advantageous (reduction of code size) when the same operations are to be initiated again and again or when many operations are to be

initiated at once. For a given instruction length, more operations can be initiated at once in comparison to conventional principles. For example, one of the best-known architectures for high-performance processors has instruction words of a length of 128 bits that contain three instructions. Accordingly, up to three operations can be initiated at once. A modification of this format according to the ReAl principles could provide, for example, an operation code of 8 bits. In y-operators, the remaining 120 bits are therefore available for initiating functions. Depending on the configuration of the instruction format, the following can be initiated, for example:

- When each resource has assigned 1 bit: up to 120 operations.
- When the resource address is 6 bits: up to 20 operations.
- When the resource address is 12 bits: up to 10 operations.

1.2 Typical Areas of Application

The ReAl principles of operation can be exploited as follows:

1. For theoretical considerations.
2. For intermediate languages and the like in conventional programming.
3. For building systems based on programmable logic circuits.
4. For developing advanced processor and system architectures.

1. Theory

Conventional programs are essentially represented as character strings. Based on the analysis of the program text alone, the behavior of the program can be predicted only insufficiently; instead, the program must be executed in order to recognize its behavior. On the other hand, on the basis of the ReAl principles it is possible to convert the programming intention into a virtual hardware configuration whose operations can be dissolved down to the individual Boolean equations. To such virtual hardware, tools and methods of graph theory, automata theory, Boolean algebra and so on could be applied, facilitating, for example, correctness proofs and investigations of complexity problems.

2. Programming[1)]

Programming in high level languages

It is well established to convert the source code first into a virtual machine code. Such virtual machines are usually designed as stack machines. Stack machines operate however inherently sequentially (one operation at a time). In contrast, virtual machines designed according to the ReAl principle of operation have primarily the following advantages:

- All potential parallelism inherent in the application program could be detected (at least theoretically), including opportunities for utilization of SIMD and VLIW instructions.
- Fewer housekeeping operations will be required, for example, when calling functions and when moving parameters (in other words, less overhead).

---

1): Here we will illustrate how ReAl principles could be used for intermediate (descriptive or interpretative) languages and the like (to be applied in the context of conventional architectures), not the programming of genuine ReAl machines.

A typical development process:

- The programming intention is written down in a suitable programming language.
- In a first pass, a ReAl compiler generates an intermediate code for a virtual ReAl machine.
- Then the program could be run by means of emulating the ReAl machine operations.
- Alternatively, in a second pass the ReAl machine code could be compiled into the native machine code of the target architecture.

Program development based on graphic design tools
The application programs are developed with design tools that support the expression of the design intentions by graphic means, for example, based on block diagrams, flowcharts, and state machines. Such design systems generate typically an intermediate code in a conventional programming language (C, C++ etc.) that is subsequently converted into a program for the corresponding target machine by means of a conventional compiler. However, conventional general-purpose programming languages are not especially suitable for many application problems in question. Programs generated according to the ReAl principles describe basically hardware structures. Therefore, such programs can be derived obviously from block diagrams, schematics, state diagrams and so on. Instead of the intermediate code (for example, in C) ReAl code is provided that selects an appropriate configuration of resources[1)].

Application areas with high requirements in regard to functional safety
When based on ReAl principles, the processor (or microcontroller) interprets a virtual circuit structure that, in contrast to a conventional program (which must be run in order to verify its correct behavior), is accessible to examination and verification also in the static state. Accordingly, microprocessors, microcontrollers and the like can also be used in cases where in the past, for safety reasons, the use of programmable devices has been excluded.

The conventional way: A hardware solution is developed, tested with regard to compliance to the respective regulations, and finally built.

The alternative: The hardware configuration is described by means of ReAl operators or instructions. This description is emulated by the processor or controller. A correctly written emulator can never crash, no matter which error is present in the system to be interpreted. Therefore, the software based on the ReAl principles has the same functional safety as a true hardware implementation.

Debugging
The fact that in the end a virtual hardware structure is present can also be used for program debugging. All methods (and tricks) that have been found useful for troubleshooting in hardware can be applied (dividing the entire "circuit" into blocks that can be tested individually, setting up test configurations with test data generation and test result analysis, injection of test patterns into suspected circuitry and the like). As in the case of troubleshooting in hardware where optionally signal generators, logic analyzers etc. are used, in such a system corresponding testing aids can be combined as needed from the already present resource pool.

---

1): To support such application areas, the resource pool could be optimized appropriately (for example, to support Boolean equations and automata tables).

Program migration
A program based on ReAl principles can describe the programming intention in all essential details – if needed, down to the individual Boolean equation. Therefore, it is to be expected that such programs can be converted without problems into machine code of future systems.

Meta-language
All programs, no matter in which language they are formulated, are in the end control instructions for information processing operations, transports and state transitions in register transfer structures. A sufficiently equipped resource pool (with regard to data types, operations and so on) is suitable therefore, in combination with the ReAl operators, as a general-purpose compiler target, or (from a theoretical viewpoint) as a general-purpose meta-language in which all expressions of the different programming languages can be reproduced.

3. Systems based on programmable logic circuits

The ReAl principles allow for describing complex designs independent of their implementation. An instruction set that is based on the ReAl principles of operation is a unified machine language that can describe hardware as well as software. Hence it will be possible to exchange hardware and software with one another.

Conventional programmable integrated circuits (FPGAs) contain basically only two types of programmable devices:

- General-purpose function blocks, macrocells etc. that carry out only comparatively simple combinational operations and can store only a few bits in flip-flops (up to four flip-flops per cell are typical). This can be referred to as "fine granularity."
- Hardwired function blocks or even complete processors (which are typically optimized down to the transistor level). This corresponds to “coarse granularity."

It takes a lot of silicon area to implement a particular design with function blocks of fine granularity ("soft" implementation). Comparing hard (optimized down to the transistor) and soft implementations of the same design, the soft implementation typically requires more than ten times as many transistors than the hard one. Also the speed is correspondingly reduced (ratio of clock frequency typically 4:1 to more than 10:1).

Hard implementations however are expensive (development cost) and not as flexible. It is not easily possible to connect them arbitrarily to other circuits. Instead, these circuits have to be adapted to the particular interfaces (for example, bus systems) of the "hard" function block. This requires typically additional circuitry (and development effort).

According to the ReAl principles of operation, programmable integrated circuits can be provided that have a medium granularity – the "fat" hard processor is essentially dissolved into its components that are made available as individual modules. Also, the connecting structures can be optimized with regard to typical information transports. Based on the available resources (operation units, addressing units and the like) general-purpose computers or special circuitry can be configured as needed and the configurations can be changed dynamically while in operation.

4. Processors and system architectures

The ReAl principles of operation allow for the decomposition of the processor structures into the individual functional units and the seamless transition between hardware and software. If the resource pool has been standardized appropriately, the fixed (monolithic) proprietary processor architectures, operating systems, and application programs can be replaced by resources of arbitrary origin (distributed system architecture). System functions as well as application functions are provided by resources that, as needed, are implemented as hardware or software[1)].

This interchangeability is facilitated especially by the principle, that the operators or instructions describe only the selection, activation etc. of the resources while the actual functions are hidden in the interior of the respective resource.

Prior attempts to implement such concepts are based on higher formal languages or virtual machines that can be emulated comparatively easily. When utilizing higher formal languages, the migration from one platform to another or the change between hardware and software always requires new compilation. For this purpose, an appropriate compiler is required. Because the internal interfaces (for example, the parameter transfer) are not uniformly standardized, there are always compatibility problems. When a virtual machine is used, such difficulties can be avoided to a large degree. Conventional virtual machines however have been developed primarily under the premise of effective compilation of software (example: P-code (Pascal), Forth machines, JVM (Java Virtual Machine)). They are therefore hardly suitable as general-purpose interfaces for complex high-performance hardware (parallel processing, application-specific processing units etc.). By exploiting the ReAl principles of operation, the inherent parallelism can be detected directly based on the programming intention. In this way, it is possible to utilize simultaneously even hundreds of operation units. Memory and processing circuitry can be connected directly with one another. In the extreme, the individual (hardware) resource is a memory array with a built-in operation unit (resource cell).

1.3 ReAl Architecture and the State of the Art

1.3.1 Overview

The operation of computers relies on the combination of circuitry (hardware) and stored programs (software). Decisive for this interaction is the interface between hardware and software. This interface (in other words, the architecture) can be characterized by three sets: the set of elementary data structures, the set of machine instructions and the set of the general principles of operation. These architectural features have been developed based on experience. The first computers have been invented as automated calculating machines. Therefore, it was self-evident to implement the basic arithmetic operations. An instruction typically initiates such an arithmetic operation, an auxiliary activity (data transport, input, output and so on), or an activity of program sequence control (branching, subroutine call and the like). The individual architectures differ primarily in the auxiliary functions (like operand addressing). Programs for conventional computer architectures are sequential by their nature; the programming model is based on instructions being executed one after another[2)].

---

1): In order to make the desire for of unlimited interchangeability come true (beyond the promises of Ada, Java and the like), resource descriptions and universal instruction codes (for example, byte codes) have to be standardized comprehensively.

2): There are numerous sources concerning computer architecture. As the space is limited, we can give only a few introductory hints [15].

It is always desirable to increase the processing performance. The execution time of a particular operation however cannot be reduced arbitrarily. The limits are set mainly by the propagation delays of the hardware. In order to increase the processing performance beyond these limits, computers have been equipped with additional operation units. Those units can be operated simultaneously (parallel to one another). This leads to the problem how to make best use of such hardware configurations. In some fields of application it is apparent that a plurality of information processing operations can be performed simultaneously (parallel processing). In many cases, however, the possibility of parallel processing is not easily recognizable. Most programs are not written to take into account parallel processing, and the conventional programming languages are based on the sequential execution of instructions. Not all commands or instructions, however, must be performed sequentially. Example:

1st instruction: X := A + B
2nd instruction: Y := C + D

When two operation units are available, both instructions can be executed at the same time. The fact that instructions and instruction sequences in conventional programs occasionally can be executed simultaneously (parallel to one another) is referred to as inherent parallelism. There are different approaches to detect the inherent parallelism and take advantage of it. The decisive prerequisite is the availability of several operation units (superscalarity). Basically, there are two principles.

1. Conventional instruction set

The individual instruction causes a single operation to be executed, respectively, and initiates thus the utilization of a single operation unit. The inherent parallelism is detected during run time. For this purpose, several sequential instructions are fetched and decoded at the same time. Since parallel processing has not been considered when writing the program, conflicts may arise. Example:

1st instruction: X := A + B
2nd instruction: Y := C + X

When both instructions are executed simultaneously, the second instruction uses the prior value of X and therefore delivers a wrong result. Such conflicts are detected by special circuitry and are resolved by repeating the instruction in question.

In addition to the operation units, circuitry for recognizing the opportunities for parallel execution and for detecting and resolving of conflicts are required. Such circuitry is rather complex. Therefore, only a few instructions can be checked with regard to the possibility of parallel execution, and the number of operation units cannot be increased arbitrarily. Typically, two to four operation units are provided for each data type (binary numbers, floating point numbers and the like). A significantly greater number of operation units would require an unbearable complexity detecting the possible conflicts.

The shortcomings of the established machines are caused primarily by still relying on basically conventional general-purpose computer architectures. Hardware that attempts to recognize the inherent parallelism in conventional programs at run time can take into consideration only a few sequential instructions, respectively. Moreover, conflict situations are to be detected and optionally to be resolved by repeated execution of instructions. For this purpose, comparatively complex circuitry is required. The operation units are utilized only insufficiently because, for the purpose of conflict resolution, they must be passed optionally several times (instruction retry).

2. *Instruction sets with provisions to control parallel operations*

When the parallel operation is controlled explicitly by the instructions, these disadvantages are eliminated. There are some variants, for example, extremely long instructions with control fields for all operation units (VLIW = very long instruction word) or instructions that contain information wether subsequent instructions can be performed in parallel or not. Circuitry to detect the inherent parallelism is not required. The number of operation units supported in this way is however limited (instructions cannot become arbitrarily long) and fixed in the respective architecture (for example, to 3, 4, or 8 operation units). The actual performance depends on the compiler that must detect the inherent parallelism based on the source code and must decide how the available resources are to be used best. The migration to systems that are designed only minimally differently requires a new compilation (a system that comprises, for example, eight operation units can be utilized only insufficiently by means of machine instructions that support only three operation units).

1.3.2 Superscalar Architectures

The multiple operation units 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. Fig. 1.1 and 1.2 show block diagrams of typical superscalar processors.

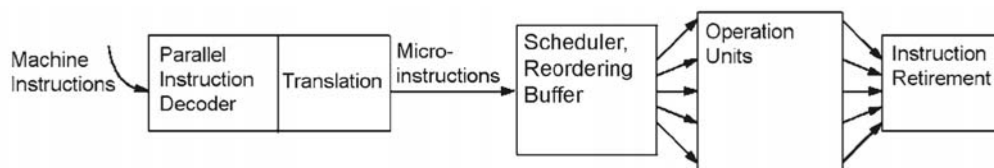
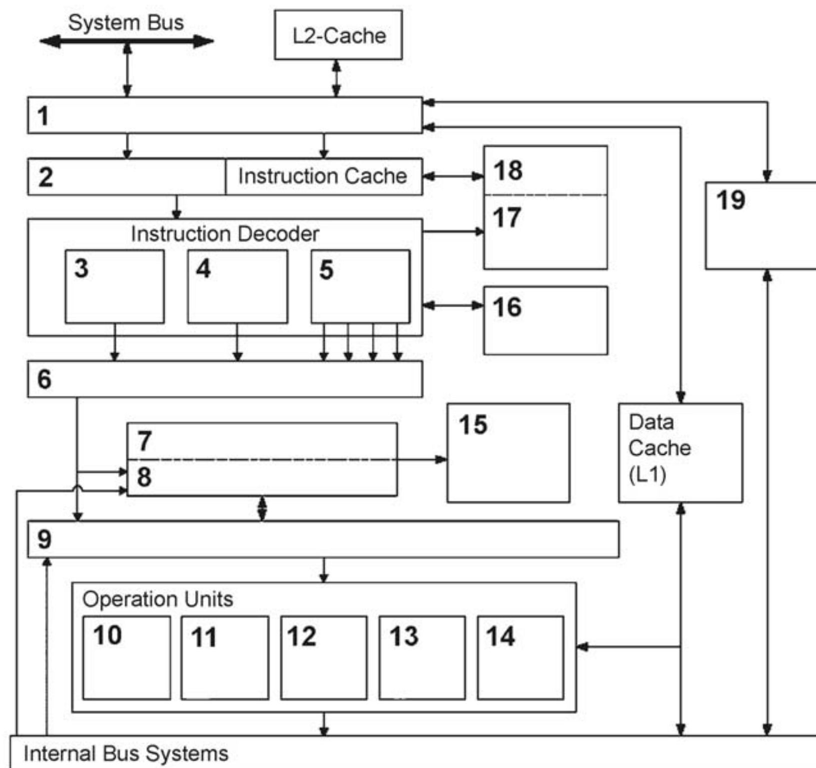


Fig. 1.1 Principal structure of a superscalar processor.

The principal operation of such processors can be summarized as follows:

1. Several conventional machine instructions are read and decoded at the same time.
2. They are translated to microinstructions.
3. The microinstructions are buffered in an associative control memory (reordering buffer) and supplied to the operation units.
4. A microinstruction is executed as soon as an appropriate operation unit is available.
5. The operations are executed without considering the original instruction sequence.
6. If conflicts are detected, the corresponding microinstructions are repeated as often as needed for the conflicts to disappear.
7. Finally, the results of instructions that have been terminated without conflicts are stored in such a way that they appear to the programmer as if they had been computed by serial execution according to the original instruction sequence (instruction retirement).

This type of parallel processing is essentially a trial and error approach. Inherent parallelism can be detected only within short instruction sequences. Since in case of a conflict the execution of the instruction must be repeated, the processing performance will drop. Moreover, because of the controlling and monitoring overhead, typically only elementary instructions will be supported this way. Instructions with complex functions are often executed serially. The complexity of the control circuitry is comparatively high.



1 - system bus controller; 2 - instruction fetch unit; 3, 4 - instruction decoder for simple instructions; 5 - instruction decoder for complex instructions; 6 - register allocation unit; 7 - instruction retirement; 8 - microinstructions reordering buffer; 9 - microinstructions scheduler; 10, 11 - floating point operation units; 12, 13 - integer operation units; 14 - memory access controller; 15 - architecture registers; 16 - conventional microprogram control (controls everything that is too complex to be executed in parallel; 17- branch target buffer; 18 - architecture instruction counter; 19 - memory access buffer.

Fig. 1.2 A superscalar processor in more detail (source: Intel).

Superscalar and ReAl Architectures

What a current superscalar machine does implicitly and speculatively at a comparatively small scale (e.g., 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 1.1 illustrates characteristic features of conventional superscalar and ReAl machines in contrast.

Fig. 1.3 shows how a conventional superscalar processor could be turned into a ReAl machine. The operation units, the cache memories, the buffers as well as the bus interfaces remain. The instruction decoder is significantly more straightforward. The general-purpose register file could be extended significantly in comparison to conventional processors (for example, to 64 to 256 registers). The operation units could be directly coupled to the general-purpose registers. Since the complicated control circuitry (positions 3 to 9 in Fig. 1.2) is not needed, optionally the set of operations could be expanded or additional operation units could be provided.

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

Table 1.1 Conventional superscalar vs. ReAl machines.

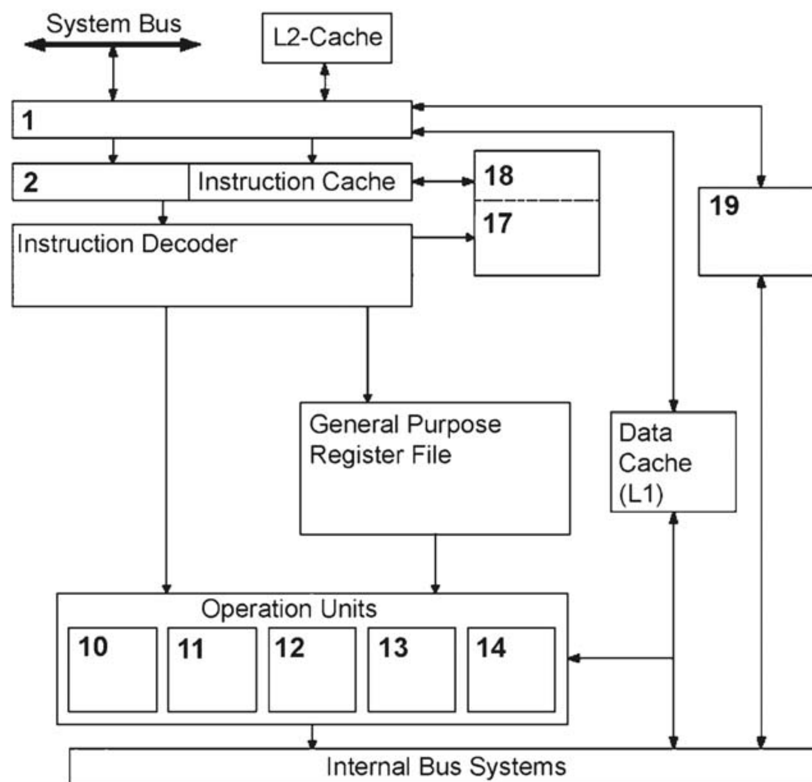


Fig. 1.3 The superscalar processor of fig. 1.2 turned into a ReAl machine. For positions 1 to 19 see fig. 1.2.

A rough estimate:
Conventional high-performance processors (for example, similar to fig. 1.2) consist of about 10 to 50 million transistors. An integrated circuit with 200 million transistors can accomodate four superscalar processor cores, each comprising approximately 50 million transistors ([14]). However, the performance capability of this arrangement can become effective only when at least four programs are to be executed at the same time; the individual program cannot be accelerated in itself. The operation units of one of the processor cores correspond roughly to eight 64-bit arithmetic/logic units (the differences between integer and floating point units etc. 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, bus systems etc. are maintained (same size, but modified structure). Some more resources could be located on the silicon area otherwise occupied by auxiliary and control circuitry (pipelining, detection of hazards and the like). Therefore, one can reasonably expect a processor IC containing approximately 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 etc. under control of ReAl operators.

Optimization of high-performance processors vs. ReAl
Some activities to develop optimized high-performance processors correspond to important goals of the ReAl approach (Table 1.2, Fig. 13).

Recommendations to improve computational throughput in conventional processors ([10])	**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 (e.g., of an innermost loop), no load/store cycles (concerning intermediate results, local variables etc.) 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 (cf. Fig. 13)
• 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

Table 1.2 Optimization of conventional high-performance processors vs. ReAl.

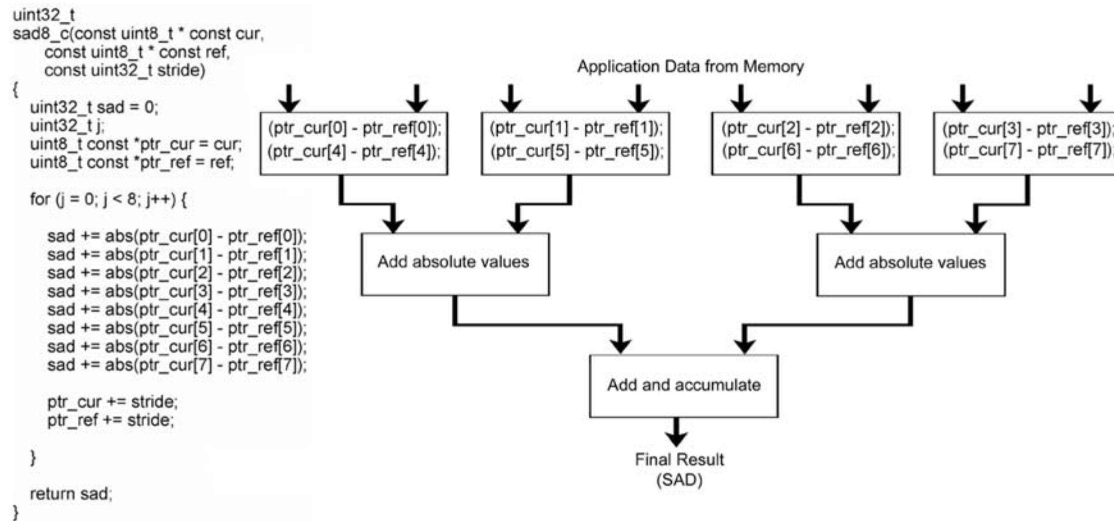


Fig. 1.4 An algorithm to calculate the sum of absolute differences (SAD). Left: The original C-code (source: xvid codec). Right: an appropriate ReAl configuration of processing resources.

The example in Fig. 1.4 has been used in [10] 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. 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 bandwith will be available for moving the application data.

1.3.3 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 [14]). The particular processors are typically high-performance cores (for example, 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]).

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 etc.
- 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 (e.g., 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.

1.3.4 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]). 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 (e.g., 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 (cf. JVM)) for programming of spatial computing systems.

1.3.5 Specialized Hardware

The desired information processing operations are not carried out with sequences of comparatively simple functions encoded in the instructions. Instead, the hardware is designed specifically in regard to the desired functions. Such devices are preferably implemented with programmable logic circuits (field programmable gate arrays FPGAs). In order to facilitate the developmental work, various functional units (up to complete processors) are made available that can be embedded into one's own designs (IP cores; IP = intellectual property) There are two kinds of such IP cores:

- Soft IP cores: They are delivered as circuit descriptions to be incorporated into the own development flow and are implemented with the means of programmable circuits (function blocks, macrocells etc.).
- Hard IP cores: They are present on the circuit in a rigid form (not programmable). Typically, such designs will be optimized down to the transistor level.

Specialized hardware systemss are comparatively expensive and the development task is complex. It is therefore obvious to search for compromise solutions and to solve the respective application problem by combining of hardware and software. Typical principles are:

- A conventional general-purpose computer (typically a microprocessor provided as a hard IP core) interacts with specialized hardware.
- Only functions that are really time-critical are supported by specialized hardware.
- If no extreme performance requirements are to be satisfied, specialized hardware is not used.

When a general-purpose processor is supplemented by special hardware, two different development processes must be mastered (hardware/software co-design). Conventionally, such problems have been solved by using two languages (programming language plus hardware description language). More recent approaches try to combine hardware and software development more closely ([5]):

- The general-purpose processor and the specialized hardware reside both on the same FPGA.
- In order to make best use of the silicon, the general-purpose computer can be modified, too (this concerns, for example, the size of cache memories and register files, the arrangement of floating point units and so on).
- Special hardware that is attached to the general-purpose processor is addressed by special instructions that are added to the instruction set of the processor.
- There is only one programming language for describing the application algorithms. The functional decomposition (between software and specialized hardware) is done automatically.
- The design of the specialized hardware is derived automatically from ordinary high-level code (for example, the C language serves as hardware design language, too; there is no need to resort to VHDL, Verilog and the like).

But such systems on silicon are still highly specialized systems. Their structure can be changed only during development time. Typically, genuine application-specific circuitry can be implemented only the soft way (whose drawbacks have been discussed above). This is also true for highly flexible processor structures.

In contrast, ReAl systems don't know a rigid division between general-purpose processors and specialized circuitry. On a ReAl programmable integrated circuit, application-specific machines can be created on the fly by exploiting all of the resources according to the particular needs of the application problem.

Conventional systems on silicon	ReAl systems on silicon
• Hardware structure to be determined during development time • Hard IP cores cannot be modified • If a hardware unit is to be laid out or modified depending on the application problem, it must be implemented the soft way • The general-purpose processor can only be modified, but not changed in its basic instruction set architecture)	• Hardware structure can be changed during run time • Typical ReAl resources are hard IP cores of intermediate complexity (considerably more than a macrocell, but much smaller than a general-purpose processor) • If necessary, general-purpose processing hardware can be created on the fly according to the requirements of the application

Table 1.3 Conventional vs. ReAl systems on silicon.

1.4 Basic Resources

In this section, the meaning of the term resource will be explained in more detail. Moreover, it will be demonstrated how such resources can be used in order to carry out elementary tasks of information processing.

Fig. 1.5 illustrates the term of register transfer level (RTL). Principally, each digital information processing system consists of memory means (flip-flops, registers, memory cells) and combinational networks. Its function is determined completely by the memory means RG and by the Boolean equations that describe the combinational networks CN. In the following, the resources can therefore be illustrated by simple register transfer level (RTL) diagrams. RTL diagrams and Boolean equations that describe the functional units of conventional computers can be found in numerous textbooks, hardware manuals and the like [16].

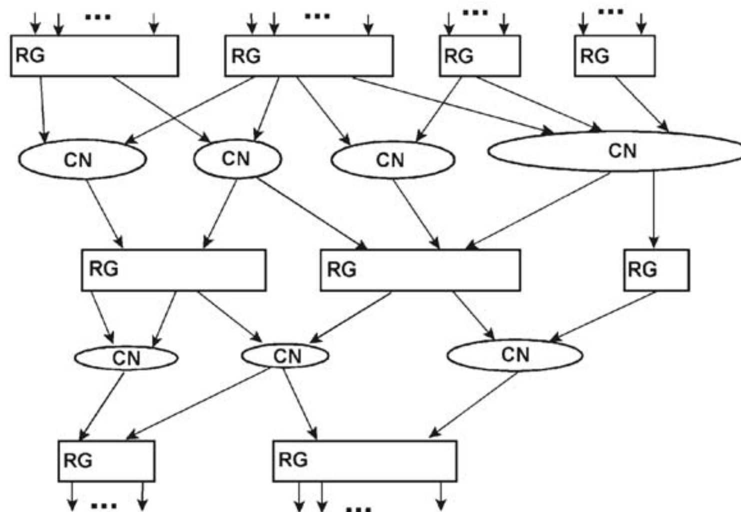


Fig. 1.5 Principal illustration of the register transfer level (RTL). RG = memory means (registers), CN = combinational networks.

Fig. 1.6 shows register transfer level diagrams of simple resources, comprising registers and combinational networks. For example, elementary resources (fig. 1.6a) generate a single result (X) from two operators (A, B):

X := A **OP** B or X := **OP** (A, b)

Most of the processing instructions of typical general-purpose computers correspond to this scheme (the differences lie primarily in the way how the operands are delivered and how the result is assigned). The well-known arithmetic logic units (ALUs) can be viewed as examples of such elementary resources.

General-purpose computers know only a few elementary data types, for example, integers, floating point numbers, characters, strings and so on. The operation unit usually processes only data of some particular types (for example, integers or floating point numbers). In case of elementary operations, the operands and the results have the same format.

ReAl resources have no such limitations. A resource can create from an arbitrary number of operands any number of results, wherein the operands and the results may belong to any data type or data format (fig. 1.6b, c). There is also no limitation to elementary data types. The data types can be as complex as desired (bit and character strings, arrays, heterogeneous structures (records) and the like).

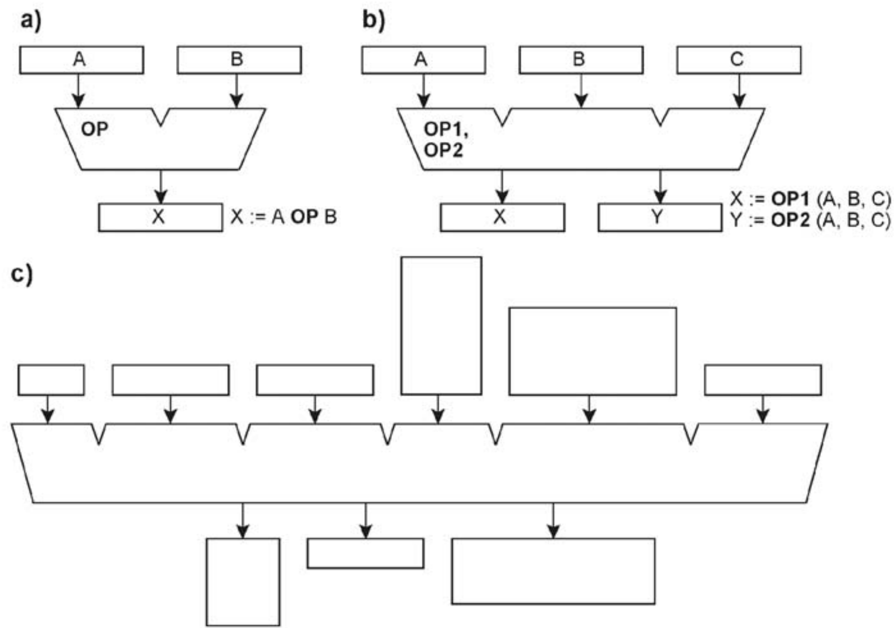


Fig. 1.6 Some basic resources.

The typical general-purpose computer executes one instruction at a time. Hence a single processing resource is sufficient. It is self-evident to increase the performance by providing several processing resources. Fig. 1.7 shows two examples. When the resources are independent (fig. 1.7a), utmost flexibility is ensured. However, there remains the problem of supplying them with operands (parameters) and to remove the results. Fig. 1.7b illustrates one solution: The resources are to be connected according to the most frequent data flow, so that the results can immediately become operands of other resources (concatenation).

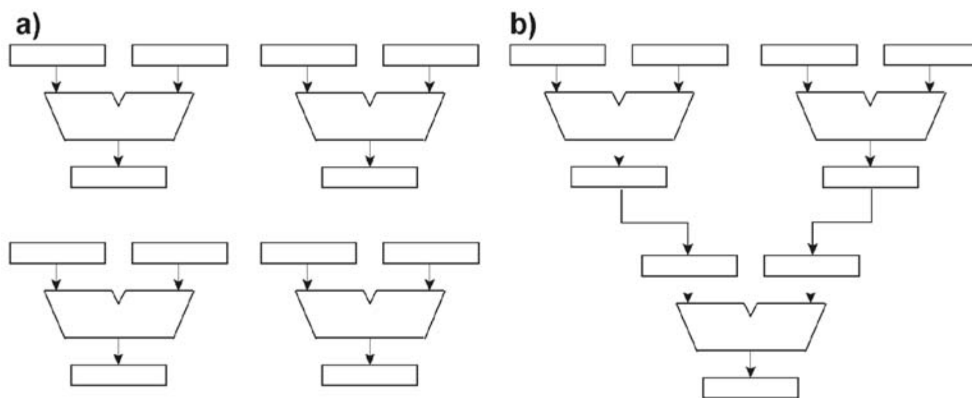


Fig. 1.7 Independent (a) and concatenated (b) resources.

Fig. 1.8 illustrates an alternative configuration. The resources are connected to a random access memory (RAM). Program execution is to be divided into sequences of three steps:

1. The operands are loaded into the resources.
2. The operands are processed within the resources (simultaneously in all of them).
3. The results are brought back into the memory.

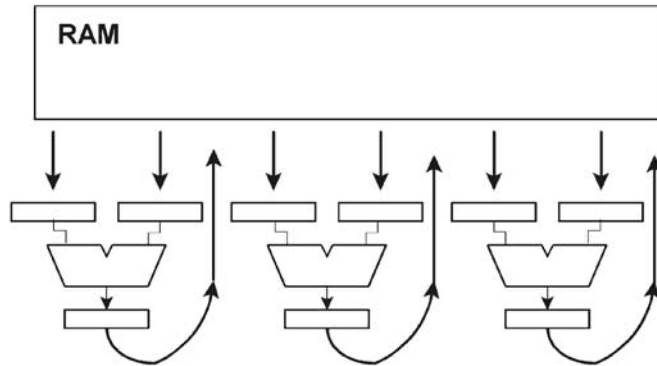


Fig. 1.8 Independent resources attached to a random access memory (RAM).

Fig. 1.9 illustrates that the resources can be implemented with hardware as well as with software. Memory areas with memory cells for the parameters and the results (fig. 1.9b) correspond to the operand and result registers of the hardware (fig. 1.9a); programs that carry out the respective information processing operations (fig. 1.9c) correspond to the combinational networks.

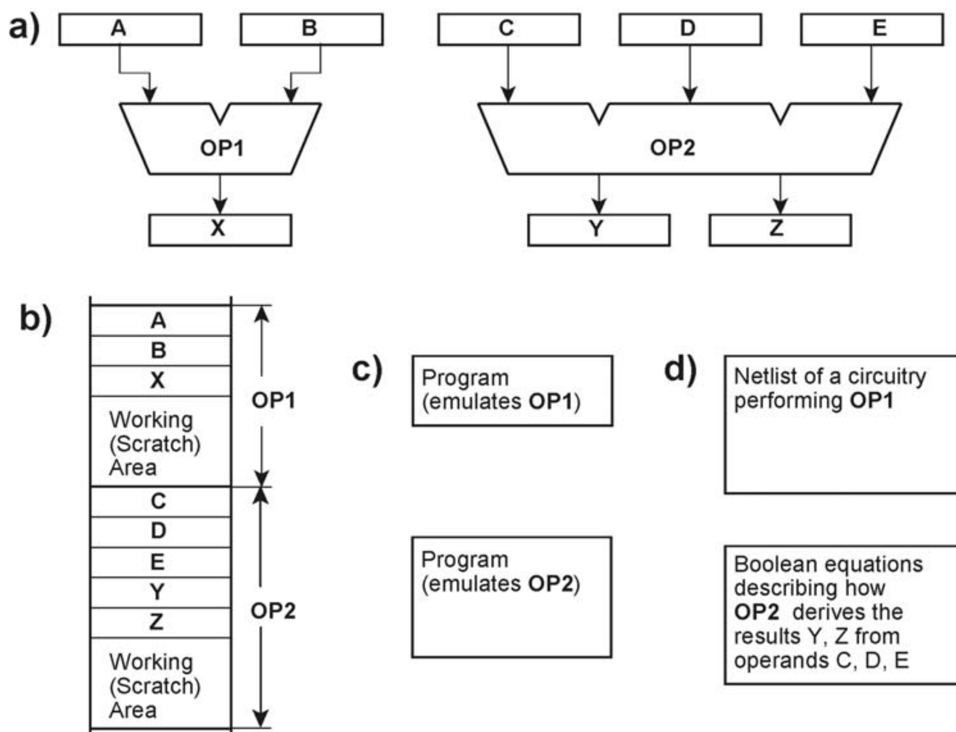


Fig. 1.9 Two resources (OP1, OP2) implemented in different ways.

Another alternative is to store the description of a circuitry that can perform the respective information processing operations (fig. 1.9d). Such descriptions can be, for example, provided as net lists or as Boolean equations. Based on this, it is possible to generate the respective hardware (for example, on a programmable integrated circuit) or to emulate its function (functional hardware simulation).