(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
27 September 2007 (27.09.2007)

PCT

(10) International Publication Number
WO 2007/106959 A2

(51) International Patent Classification:
G06F 15/80 (2006.01)

(21) International Application Number:
PCT/BE2007/000027

(22) International Filing Date: 19 March 2007 (19.03.2007)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
0605349.0 17 March 2006 (17.03.2006) GB

(71) Applicants *(for all designated States except US)*: **INTERUNIVERSITAIR MICROELEKTRONICA CENTRUM VZW** [BE/BE]; Kapeldreef 75, B-3001 Leuven (BE). **FREESCALE SEMICONDUCTORS INC.** [US/US]; 6501 William Cannon Dr., Austin, TX 78735 (US).

(72) Inventors; and
(75) Inventors/Applicants *(for US only)*: **KANSTEIN, Andreas** [DE/BE]; Gouvernementsweg 168, B-1950 Kraainem (BE). **BEREKOVIC, Mladen** [DE/DE]; Laubeichenfeld 32, D-30966 Hemmingen (DE).

(74) Agents: **BIRD, William, E.** et al.; Bird Goën & Co, Klein Dalenstraat 42A, B-3020 Winksele (BE).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BH, BR, BW, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IS, JP, KE, KG, KM, KN, KP, KR, KZ, LA, LC, LK, LR, LS, LT, LU, LY, MA, MD, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PG, PH, PL, PT, RO, RS, RU, SC, SD, SE, SG, SK, SL, SM, SV, SY, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.

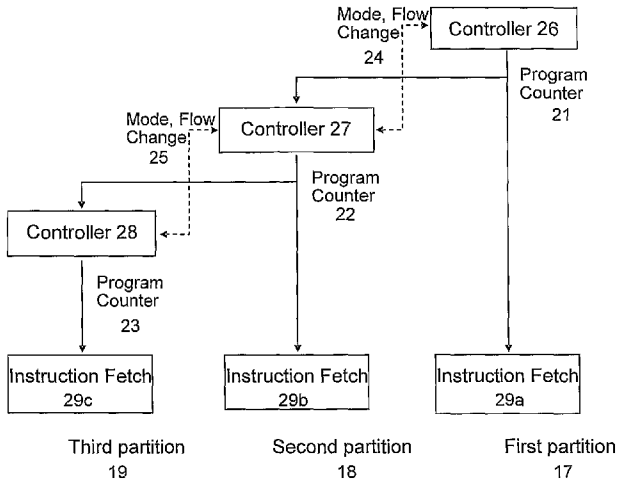
(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IS, IT, LT, LU, LV, MC, MT, NL, PL, PT, RO, SE, SI, SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Declaration under Rule 4.17:
— *of inventorship (Rule 4.17(iv))*

Published:
— *without international search report and to be republished upon receipt of that report*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) **Title:** RECONFIGURABLE MULTI-PROCESSING COARSE-GRAIN ARRAY



(57) **Abstract:** A signal processing device adapted for simultaneous processing of at least two process threads in a multi-processing manner is described. It comprises a plurality of functional units capable of executing word- or subword- level operations on data, a means for interconnecting said plurality of functional units, said means for interconnecting supporting a plurality of interconnect arrangements that can be dynamically switched, at least one of said interconnect arrangements interconnecting said plurality of functional units into at least two non- overlapping processing units each with a pre-determined topology, the signal processing device furthermore comprising at least two control modules, each control module being assigned to one of said processing units. The present invention also provides a method for executing an application on such a signal processing device, a method for compilation of application source code in order to obtain compiled code being executable on such a signal processing device, and to optimisation methods for applications to be executed on such a signal processing device.

WO 2007/106959 A2

Reconfigurable multi-processing coarse-grain array

Technical field of the invention

The present invention relates to signal processing devices adapted for simultaneously processing at least two threads in a multi-processing or multi-threading manner, to methods for executing an application on such a signal processing device, to methods for compilation of application source code in order to obtain compiled code being executable on such a signal processing device, to methods for adjusting applications to be executed on such a signal processing device, to a computer program product for executing any of the methods for executing an application on such a signal processing device, to machine readable data storage devices storing such computer program product and to transmission of such computer program products over local or wide area telecommunications networks.

Background of the invention

Nowadays, a typical embedded system requires high performance to perform tasks such as video encoding/decoding at run-time. It should consume little energy so as to be able to work hours or even days using a lightweight battery. It should be flexible enough to integrate multiple applications and standards in one single device. It has to be designed and verified in a short time-to-market despite substantially increased complexity. The designers are struggling to meet these challenges, which call for innovations of both architectures and design methodology.

Coarse-grained reconfigurable architectures (CGRAs) are emerging as potential candidates to meet the above challenges. Many designs have been proposed in recent years. These architectures often comprise tens to hundreds of functional units (FUs), which are capable of executing word-level operations instead of bit-level ones found in common FPGAs. This coarse granularity greatly reduces the delay, area, power and configuration time compared with FPGAs. On the other hand, compared with traditional "coarse-grained" programmable processors, their massive computational resources enable them to achieve high parallelism and efficiency. However, existing CGRAs

have not yet been widely adopted mainly because of programming difficulty for such a complex architecture.

Summary of the invention

In a first aspect, the present invention relates to a signal processing device adapted for simultaneous processing of at least two process threads in a multi-processing manner. The signal processing device comprises a plurality of functional units capable of executing word- or subword-level operations on data, and routing resources for interconnecting said plurality of functional units, said routing resources supporting a plurality of interconnect arrangements that can be dynamically switched, at least one of said interconnect arrangements interconnecting said plurality of functional units into at least two non-overlapping processing units each with a pre-determined topology, each of said processing units being configured to process a respective one of said process threads. Another of said interconnect arrangements can interconnect said plurality of functional units into a single processing unit. The signal processing device furthermore comprises at least two control modules, each control module being assigned to one of said processing units for control thereof. With word- or subword-level operations is meant non-bit level operations.

It is an aspect of the invention that said functional units can be grouped in predetermined/static groupings including at least one functional unit, each of said groupings defining a processing unit.

The control modules may include instruction fetch units and control units. Said control modules are adapted for controlling the word- or sub word-level (non-bit level) operations within their assigned processing unit.

In embodiments of the present invention, said control module may perform operations (increment, change) on a program counter. Preferably it also supports some debugging.

In embodiments of the present invention, a plurality of data storages may be provided, wherein the routing resources interconnect the plurality of functional units and the plurality of data storages. The data storages may be registers. The data storages may be shared between said functional units. In

embodiments of the present invention, one data storage may be provided for each processing unit.

A signal processing device according to embodiments of the present invention may include a data storage in which an application code is stored, said application code defining a process comprising the at least two process threads and being executable by said processing units. The routing resources may then be adapted for dynamically switching between interconnect arrangements at pre-determined points in the application code.

In a signal processing device according to embodiments of the present invention, the routing resources may be adapted for dynamically switching interconnect arrangements depending on data content of a running application. Such data content may for example be a parameter file describing to which processing unit functions of a thread are to be mapped, or data, e.g. one or more bits, in a data storage of one of the functional units. The routing resources may comprise multiplexing and/or demultiplexing circuits. The signal processing device may have a clock, wherein the multiplexing and/or demultiplexing circuits are adapted to be configured with appropriate settings for dynamically switching interconnect arrangements, wherein the settings may change every clock cycle.

A signal processing device according to embodiments of the present invention may furthermore comprise at least one global storage shared between a plurality of functional units.

A signal processing device according to embodiments of the present invention may include at least two different types of functional units.

In a signal processing device according to embodiments of the present invention, at least another of said interconnect arrangements may interconnect said plurality of functional units into a single processing unit under control of a single control module.

In a signal processing device according to embodiments of the present invention, at least one of the at least two control modules may be part of a global control unit for use in an interconnect arrangement with a single processing unit. In at least one interconnect arrangement with a single processing unit, at least one of the control modules may drive control signals

of all the functional units by having at least one other control module to follow it.

A signal processing device according to embodiments of the present invention may be adapted for re-using, in an interconnect arrangement with a single processing unit, at least part of the control modules assigned to the processing units in an interconnect arrangement with a plurality of non-overlapping processing units in the control module used.

In a second aspect, the invention relates to methods for executing at least one application on a signal processing device as disclosed above. An application is typically executed on a signal processing device as a single process thread, meaning under control of a single control module.

It is an aspect of the invention to provide a method for executing at least one application wherein switching between a single thread approach and a multi thread approach is applied, wherein a portion of the application is split in parts, and each part is executed as a separate process thread, on one of the predefined processing units. The method according to embodiments of the present invention thus comprises dynamically switching the signal processing device into a device with at least two non-overlapping processing units, and splitting a portion of the application in at least two process threads, each process thread being executed simultaneously as a separate process thread on one of the processing units, each processing unit being controlled by a separate control module.

This single or multi-threading switching method of executing an application is supported by the configuration capabilities of the signal processing device, in particular the capability to operate it in unified mode, wherein all functional units of the device operate in one thread of control and in split mode, wherein all functional units within a single processing unit operate in one thread of control, and the processing units themselves simultaneously operate in different threads of control.

Or thus, the signal processing device comprises of a plurality of partitions, each capable of running independently a process thread.

A consequence is that within each processing unit the instruction flow can change, for instance due to a branching in the code, independently of the instruction flow in another processing unit.

The use of a signal processing device with a plurality of functional units enables instruction level parallelism while the organization of these functional units in groupings defining processing units enables thread-level parallelism. Because the groupings can be changed dynamically, more flexibility can be obtained than with a multi-core approach.

The capability is realized by providing said two or more control modules, each of said control modules being capable of executing a single thread of control.

In accordance with embodiments of the present invention, switching the signal processing device into a device with at least two processing units may be determined by a first instruction in application code determining the application. The first instruction may contain a starting address of the instructions of each of the separate process threads. The starting address may be an indicator of where instructions are to be found. It can be a direct reference to a location or a pointer to a location, the location for example being in a register or in a data storage.

A method according to embodiments of the present invention may furthermore comprise dynamically switching back the signal processing device into a device with a single processing unit, synchronising the separate control modules and joining the at least two threads of the application into a single process thread, the single process thread being executed as a process thread on the single processing unit under control of the synchronised control modules. Switching back the signal processing device into a device with a single processing unit may be determined by a second instruction in the application code determining the application. The second instruction may contain a starting address of the instructions to be executed as the single process thread.

In accordance with embodiments of the present invention, the single control module may re-use at least one of the separate control modules when executing the application as a single process thread.

In accordance with embodiments of the present invention, in an interconnect arrangement with a single processing unit, one of the separate control modules may drive control signals of all the functional units by having the other control modules to follow it.

It is clear that this single or multi-threading switching method is easily extendable toward a method wherein switching between a first organization of partitions, wherein some of the partitions are executed together as a single thread and others as another thread and a second different organization of partitions.

This generalised switching method may be rephrased as a dynamically, run-time reconfiguring of the device, supported by a static, predetermined organization of the device.

A method is furthermore provided for dynamically reconfiguring a signal processing device in a process for executing at least one application on said signal processing device, said signal processing device comprising a plurality of functional units, capable of executing word- or sub-word level (non-bit level) operations on data, said functional units being grouped into one or more non-overlapping processing units, and routing resources for interconnecting said functional units, said application being arranged as a plurality of threads of which at least a part thereof are at least partly simultaneously executable, said method comprising:

configuring said computing signal processing device by providing a first assignment of one or more threads to said processing units;

after said configuring, simultaneously executing said one or more threads, wherein each of said executed threads being executed on one or more of said processing units in accordance with said first assignment;

ending said execution;

configuring said signal processing device by providing a second, different, assignment of one or more threads to said processing units;

after said configuring, simultaneously executing said one or more threads, each of said executed threads being executed on one or more of said processing units in accordance with said second assignment.

To avoid much overhead, the control modules being used for single thread processing may be re-using at least a part (or even complete) the control modules, available for each processing unit.

In an embodiment thereof, this re-use may be realized by synchronizing the control modules of the partitions, meaning using the same inputs by each of said elements and distributing their respective outputs to their assigned partition.

It is to be noted that the functional units may be flexibly connected, for instance by providing multiplexing and/or de-multiplexing circuits in between them. The dynamical reconfiguring may be realized by providing the appropriate settings to said multiplexing and/or de-multiplexing circuits. Said settings can change from cycle-to-cycle.

The invention also relates to a method for compilation of application source code in order to obtain compiled code being executable on a signal processing device as described, in particular to instruction(s) to be included at source code level to partition the code and also to instruction(s) to be included automatically into the compiled code, for switching being execution modes (e.g. unified and split mode).

In this aspect, the present invention provides a method for compilation of application source code in order to obtain compiled code being executable on a signal processing device as described. The method comprises inputting application source code and generating compiled code from the application source code. Generating the compiled code comprises including, in the compiled code, a first instruction for configuring the signal processing device for simultaneous execution of multiple process threads and for starting the simultaneous execution of the process threads, and including a second instruction to end the simultaneous execution of the multiple process threads such that when the last of the multiple process threads decodes this instruction, the signal processing device is configured to continue execution in unified mode. Hence the configuring may be done by the code itself (dynamically).

According to a further aspect of the present invention an architectural description of said signal processing device is provided, including a description

of the grouping of the functional units. Indeed such signal processing device is typically generated as an instance of a generic template. For the described invention the generic template should include the possibility to group functional units so as to form one or more processing units, and to provide control modules per group.

In embodiments of the present invention, the method may furthermore comprise providing an architectural description of the signal processing device, the architectural description including descriptions of pre-determined interconnect arrangements of functional units forming processing units. Providing the architectural description may include providing a separate control module per processing unit.

Still a further aspect of the invention is to provide a compilation method, comprising inputting application source code and the above-described architectural description, and generating compiled code, including a first instruction (e.g. denoted fork), configuring the signal processing device for execution of multiple threads and starting the execution of the threads, and a second instruction (e.g. denoted join) to end the execution of the multiple threads. In particular said second instruction is such that when the last of the threads decode this instruction, the signal processing device is configured to continue in unified mode.

In embodiments of the present invention, the first instruction may contain the start address of instructions of each of the multiple process threads. In embodiments of the present invention, the second instruction may contain the start address of instructions to be executed in unified mode after the execution of the multiple process threads.

In embodiments of the present invention, generating the compiled code may comprise partitioning the application source code, thus generating code partitions, labelling in which mode and on which processing unit the code partitions are to be executed, separately compiling each of the code partitions, and linking the compiled code partitions into a single executable code file.

Yet another aspect of the present invention is to provide a compilation method, comprising a step of inputting application source code, and the above-described architectural description, a step of partitioning the code and labeling

how (unified/split mode) and where (which processing element) the code will be executed, separate compilation of each of the code partitions and linking the compiled code into a single executable.

The invention further relates to adjustment environments wherein, for applications, exploration of various partitioning is performed, said adjustment environment also being capable of changing the instance of an architectural description of said signal processing device for exploring various configurations of said signal processing device.

Specific embodiments of the invention are set out in the accompanying claims. Features from the dependent claims may be combined with features of the independent claims and with features of other dependent claims as appropriate and not merely as explicitly set out in the claims.

The above and other characteristics, features and advantages of the present invention will become apparent from the following detailed description, taken in conjunction with the accompanying drawings, which illustrate, by way of example only, the principles of the invention. This description is given for the sake of example only, without limiting the scope of the invention. The reference figures quoted below refer to the attached drawings.

Brief description of the drawings

Fig. 1 illustrates an example of an embodiment of a coarse grain array for use with embodiments of the present invention.

Fig. 2 illustrates a concept for the reusability and scalability of control modules and instruction fetch units in accordance with embodiments of the present invention.

Fig. 3 illustrates a detailed datapath of a functional unit in accordance with embodiments of the present invention.

Fig. 4 illustrates scalable partitioning-based threading in accordance with embodiments of the present invention.

Fig. 5 illustrates a hierarchical multi-threading controller in accordance with embodiments of the present invention.

Fig. 6 illustrates source code reorganization in accordance with embodiments of the present invention.

Fig. 7 illustrates a multi-threading compilation tool chain in accordance with embodiments of the present invention.

Fig. 8 illustrates, as an example, threading on an MPEG2 decoder.

Fig. 9 illustrates an experimental dual-threading compilation flow.

Fig. 10 illustrates dual-threading memory management in accordance with embodiments of the present invention.

Fig. 11 illustrates a shadow register file set-up according to embodiments of the present invention.

Description of illustrative embodiments

The present invention will be described with respect to particular embodiments and with reference to certain drawings but the invention is not limited thereto. The drawings described are only schematic and are non-limiting. In the drawings, the size of some of the elements may be exaggerated and not drawn on scale for illustrative purposes. The dimensions and the relative dimensions do not correspond to actual reductions to practice of the invention.

Furthermore, the terms first, second, third and the like in the description and in the claims, are used for distinguishing between similar elements and not necessarily for describing a sequential or chronological order. It is to be understood that the terms so used are interchangeable under appropriate circumstances and that the embodiments of the invention described herein are capable of operation in other sequences than described or illustrated herein.

Moreover, the terms top, bottom, over, under and the like in the description and the claims are used for descriptive purposes and not necessarily for describing relative positions. It is to be understood that the terms so used are interchangeable under appropriate circumstances and that the embodiments of the invention described herein are capable of operation in other orientations than described or illustrated herein.

It is to be noticed that the term "comprising", used in the claims, should not be interpreted as being restricted to the means listed thereafter; it does not exclude other elements or steps. It is thus to be interpreted as specifying the presence of the stated features, integers, steps or components as referred to,

but does not preclude the presence or addition of one or more other features, integers, steps or components, or groups thereof. Thus, the scope of the expression "a device comprising means A and B" should not be limited to devices consisting only of components A and B.

Similarly, it is to be noticed that the term "coupled", also used in the claims, should not be interpreted as being restricted to direct connections only. Thus, the scope of the expression "a device A coupled to a device B" should not be limited to devices or systems wherein an output of device A is directly connected to an input of device B. It means that there exists a path between an output of A and an input of B which may be a path including other devices or means.

A particular coarse-grained reconfigurable architecture (CGRA), ADRES (architecture for dynamically reconfigurable embedded systems), is known and manufactured by Interuniversitair Microelektronicacentrum vzw, Leuven, Belgium. ADRES addresses issues of existing CGRAs. The present invention will be described with reference to the ADRES architecture. This is, however, not intended to be limiting; the present invention may also be used for other suitable coarse grain array architectures.

The ADRES archirecture is a datapath-coupled coarse-grained reconfigurable matrix. The ADRES architecture is a power-efficient flexible architecture template that combines a very long instruction word (VLIW) digital signal processor (DSP) with a 2-D coarse-grained heterogeneous reconfigurable array (CGA), which is extended from the VLIW's datapath. VLIW architectures execute multiple instructions per cycle, packed into a single large "instruction word" or "packet", and use simple, regular instruction sets. The VLIW DSP efficiently executes control-flow code by exploiting instruction-level parallelism (ILP). The array, containing many functional units, accelerates data-flow loops by exploiting high degrees of loop-level parallelism (LLP). The architecture template allows designers to specify the interconnection, the type and the number of functional units.

The ADRES template thus tightly couples a very-long instruction word (VLIW) processor 11 and a coarse-grained array 12 by providing two functional modes on the same physical resources. It brings advantages such as high

performance, low communication overhead and easiness of programming. An application written in a programming language such as e.g. C can be quickly mapped onto an ADRES instance. ADRES is a template instead of a concrete architecture. Architectural exploration becomes possible to discover better architectures or design domain-specific architectures.

The ADRES array is a flexible template instead of a concrete instance. An architecture description language is developed to specify different ADRES instances. A script-based technique allows a designer to easily generate different instances by specifying different values for the communication topology, supported operation set, resource allocation and timing of the target architecture. Together with a retargetable simulator and compiler, this tool-chain allows for architecture exploration and development of application domain specific processors. As ADRES instances are defined using a template, the VLIW width, the array size, the interconnect topology, etc. can vary depending on the use case.

The ADRES template includes many basic components, including computational, storage and routing resources. The computational resources are functional units (FUs) 13 that are capable of executing a set of word-level operations selected by a control signal. Data storages such as register files (RFs) 14 and memory blocks 15 can be used to store intermediate data. The routing resources 16 include wires, multiplexers and busses. An ADRES instance thus comprises functional units 13, registers 15 and register files 14, and routing resources 16 such as busses and multiplexers to connect the functional units 14 and the register files 14.

Basically, computational resources (FUs) 13 and storage resources (e.g. RFs) are connected in a certain topology by the routing resources 16 to form an instance of an ADRES array. The whole ADRES array has two functional modes: the VLIW processor 11 and the reconfigurable array 12, as indicated by the dashed lines in Fig. 1. These two functional modes 11, 12 can share physical resources because their executions will never overlap thanks to a processor/co-processor model. The processor operates either in VLIW mode or in CGA mode. The global data register files RF' 15 are used in both modes

and serve as a data interface between both modes, enabling an integrated compilation flow.

The VLIW processor 11 includes several FUs 13 and at least one multi-port register file RF' 15, as in typical VLIW architectures, but in this case the VLIW processor 11 is also used as the first row of the reconfigurable array 12. Some FUs 13 of this first row are connected to the memory hierarchy 10, depending on the number of available ports. Data accesses to the memory of the unified architecture are done through load/store operations available on these FUs.

When compiling, with a compiler, applications for an ADRES architecture, loops are modulo-scheduled for the CGA 12 and the remaining code is compiled for the VLIW 11. By seamlessly switching the architecture between the VLIW mode and the CGA mode at run-time, statically partitioned and scheduled applications can be run on the ADRES instance with a high number of instructions-per-clock (IPC).

To remove the control flow inside loops, the FUs 13 support predicated operations. The results of the FUs 13 can be written to data storages such as the distributed RFs 14, i.e. RFs 14 dedicated to a particular functional unit 13, which RFs 14 are small and have fewer ports than the shared data storage such as register files RF' 15, which is at least one global data storage shared between a plurality of functional units 13, or the results of the FUs 13 can be routed to other FUs 13. To guarantee timing, the outputs of FUs 13 may be buffered by an output register. Multiplexers 32 are part of the routing resources 16 for interconnecting FUs 13 into at least two non-overlapping processing units. They are used to route data from different sources. The configuration RAM 31 (see Fig. 1 and Fig. 3) stores a few configurations locally, which can be loaded on a cycle-by-cycle basis. The configurations can also be loaded from the memory hierarchy 10 at the cost of extra delay if the local configuration RAM 31 is not big enough. Like instructions in microprocessors, the configurations control the behaviour of the basic components by selecting operations and controlling multiplexers. An example of a detailed datapath as described above is illustrated in Fig. 3

An embodiment of the invention extends a highly parallel data processing architecture, e.g. the ADRES, or a coarse-grain reconfigurable array, to a multi-threading/processing device. As set out above, an ADRES instance comprises functional units 13, data storages such as registers and register files 14, and connecting resources 16 such as busses and multiplexers to connect the functional units 13 and the register files 14. ADRES supports an MIMD (Multiple Instruction Multiple Data) programming model by, every cycle if needed, independently configuring every element of the array. In addition, functional units 13 may support SIMD (Single Instruction Multiple Data) processing to utilize the width of the data path. A special programming approach is used to extract very high instruction level parallelism (ILP) from suitable portions of the code. ADRES also implements a traditional VLIW (Very Long Instruction Word) mode in which less functional units are executing. This may be used for code where less instruction-level parallelism (ILP) is obtainable, and where a traditional programming model is sufficient.

For the embodiment of the invention the ADRES array is being sub-divided into partitions, to enable thread-level parallelism. Every partition or combination of partitions can execute in VLIW mode and in array mode. This multi-threading could also be achieved by instantiating multiple ADRES instances, but the novel partitioning approach allows to also run a thread on two or more joined partitions. This is essentially providing another dimension of reconfigurability.

The topology, the connectivity, and the features of the functional units 13 and register files 14 of an ADRES instance are defined at design time with an architecture template. For the multi-threaded ADRES the template is extended to include information about the partitioning and also to guarantee that each partition or combination of partitions is itself a valid ADRES instance. The architecture template defines all aspects of a specific ADRES instance for the compiler.

Figure 1 shows an example for a possible ADRES template with three partitions 17, 18, 19. Thereby, for example one, two or three threads may be executed in parallel, using various combinations of partitions to execute a thread. For example, a single thread could execute on the whole 8x8 array

(indicated as first partition 17), or on the 1x2 sub-partition (indicated as third partition 19), or on the 4x4 sub-partition (indicated as second partition 18), with the rest (non-used part) of the array in a low-power mode. In the following, the partitions of this example are indicated first partition 17, second partition 18 and third partition 19 respectively. The partitions 17, 18, 19 have been chosen to be of different size, to better adapt the architecture to the degree of parallelism available in a thread. This can be combined with using heterogeneous functional units, and even with heterogeneous data path widths, to optimize a partition for some specific functionality.

The embodiment of the invention leverages and extends the programming model of ADRES. The compiler generates code for the VLIW mode and for the array mode e.g. based on the data in a parameter file and based on some constructs in the code, like a function name prefix and intrinsics, that is, special instructions. Similarly, the split mode operation, i.e. the mode of operation when a plurality of threads are run in parallel on a plurality of non-overlapping processing units, each processing unit being under the control of a control module assigned to said processing unit, may be indicated e.g. by settings in a parameter file and by using special instructions for splitting and joining partitions. The entries in the parameter file describe to which partition the first and subsequent functions of a thread shall be mapped, so that the compiler knows which architecture template to use.

An enhanced compiler may automatically identify parallel threads in the code and explore the available partitions for a mapping which is improved, or even optimized, for performance and power consumption.

Alternatively, it may be the task of a programmer to define threads in the code by inserting instructions for splitting the array, or a partition, into sub-partitions. The arguments of the instruction will at least contain a reference, e.g. pointers, to the first function in each parallel thread. The mechanism of the split instruction thus is to change the state of the array or partition, and to trigger the threads through something that resembles a subroutine call. Besides saving the return address, the instruction also sets up the partitions' registers 15 for split operation, that is, it is initializing the stack pointers for all new threads. The data register file of the array should be implemented as a

clustered register file so that clusters are not shared between any potential partitions of the array.

Every partition 17, 18, 19 has its own set of control signals: program counters 21, 22, 23 for VLIW and array mode, and mode and other status flags 24, 25. Also, every partition 17, 18, 19 has its own control module 26, 27, 28 to drive these signals. When running in unified mode, i.e. in an interconnect arrangement with a single processing unit, one control module 26 drives the control signals of all partitions within that interconnect arrangement by having the other control modules 27, 28 follow it, as shown in Figure 2. The control modules 26, 27, 28 may be multiple instantiations of the same module. For synchronized execution the program counter 21, 22, 23 may be pushed from one control module to the next. Other implementations for synchronizing the control modules 26, 27, 28 are possible. An aspect is that control modules can be reused from existing implementations and only need very few enhancements. When the split is executed, the individual control modules 26, 27, 28 start executing the first instruction in each thread, as provided by the function pointers in the split instruction. Each control module 26, 27, 28 in the split mode drives its respective signals. From a programmer's viewpoint, the partitions now operate like individual ADRES instances and can independently switch between VLIW mode and CGA mode.

However, the programmer may work with a data memory that is shared between all threads. Again, an enhanced compiler can support the programmer, in this case with the task of memory allocation for multiple threads. For synchronization of and sharing data between threads the enhanced architecture provides special instructions for efficiently implementing semaphores and other multi-processing primitives.

A shared architectural element is the instruction memory. This is not directly affecting the programming model, it only requires that, when linking the code partitions, the linker or linking module is aware of the partitions for packing the code of parallel threads. Every partition has an independent instruction fetch unit 29a, 29b, 29c, connected directly to its respective control module 26, 27, 28. When in unified mode, the control modules 26, 27, 28 are synchronized via the program counter 21, 22, 23. In this case, the instruction

units 29a, 29b, 29c fetch portions of the set of instructions for the unified partition, and therefore the execution follows the same flow. When in split mode, each controller 26, 27, 28 follows its own thread's flow of execution, and each instruction unit 29a, 29b, 29c fetches the set of instructions for the respective sub-partition. The linkage between control modules 26, 27, 28 and instruction fetch units 29a, 29b, 29c is shown in Figure 2.

To join threads, special join instructions are inserted in the code which will end the current thread. The respective partition can automatically be put into low-power mode. When the last thread initiated from a split instruction ends, the execution will continue on the now combined partitions, with the next instruction following the split. For recovery routines special mechanisms are provided to permit a thread to monitor and, when necessary, abort another thread.

In summary, this embodiment of the invention extends an already highly parallel, reconfigurable architecture with another dimension of parallelism and reconfigurability. It leverages the existing architecture and tools, especially the compiler, while retaining a simple programming model. The multi-threading extension allows users of the coarse-grain array to exploit function-level parallelism, as well as complex data-level parallelism, which allows efficiently implementing the variability in applied algorithms found in emerging applications. The programmability and simplicity of the invention is the key differentiating factor.

As an example, an MPEG2 decoder is used for a demonstration of a multithreaded architecture in accordance with embodiments of the present invention. Most MPEG2 decoder kernels can be scheduled on a CGA with the number of instructions per clock (IPC) ranging from 8 to 43. It has been observed, however, that some modulo-scheduled kernels' IPC do not scale very well when the size of the CGA increases. Some of the most aggressive architectures have the potential to execute 64 instructions per clock cycle, but few applications can utilize this level of parallelism, resulting in a much lower average IPC. This is caused by two reasons: (1) The inherent ILP of the kernels is low and cannot be increased efficiently even with loop unrolling, or the code is too complex to be scheduled efficiently on so many units due to

resource constraints, for example the number of memory ports. (2) The CGA is idle when executing sequential code in VLIW mode. The more sequential code is executed, the lower the achieved application's average IPC, and in turn, the lower the CGA utilization. In conclusion, even though the ADRES architecture is highly scalable, the challenge is faced of getting more parallelism out of many applications, which fits better to be executed on smaller ADRES arrays. This is commonly known as Amdahl's law, as described by G.M. Amdahl in "Validity of the single processor approach to achieve large-scale comput ing capabilities", Proc. AFIPS Spring Joint Computer Conf. 30, 1967 Page(s):483-485.

If properly reorganized and transformed at programming time, multiple kernels in a same application can be efficiently parallelized by an application designer. Low-LLP kernels can be statically identified through profiling, the optimal choice of ADRES array size for each kernel can be estimated, and a large ADRES array can be partitioned into several small-scaled ADRES sub-arrays that fit each kernel, which is parallelized into threads if possible. When an application is executed, a large ADRES array can be split into several smaller sub-arrays for executing several low-LLP kernels in parallel. Similarly, when a high-LLP kernel is executed, sub-arrays can be unified into a large ADRES array. Such a multi-threaded ADRES (MT-ADRES) is highly flexible, and can increase the over utilization of large-scaled ADRES arrays when the LLP of application is hard to explore.

Hereinafter, a demonstrative dual-threading experiment is presented on an MPEG2 decoder implemented on top of a single-threaded architecture, as well as its matching compilation tools. Through this experiment, it has been proven that multithreading is feasible for the ADRES architecture.

A scalable partitioning-based threading approach is proposed for a coarse-grained reconfigurable architecture such as ADRES. The rich resource on the ADRES architecture allows partitioning a large coarse-grained reconfigurable array into two or more sub-arrays, each of which can be viewed as a down-scaled coarse-grained reconfigurable architecture and can be partitioned further down hierarchically, as shown in Fig. 4. With the partitioning technique in accordance with embodiments of the present invention it is

possible to dynamically share HW resources between threads without the cost of the control logic of dynamic out-of-order execution, as used in general-purpose processors.

Each thread has its own resource requirement. A thread that has high ILP requires more computation resources, thus executing it on a larger partition results in a more efficient use of the ADRES array and vice versa. A globally optimal application design demands that the programmer knows the IPC of each part of the application, so that he can find an efficient array partition for each thread.

The easiest way to find out how many resources are required by each part of a certain application is to profile the code. A programmer starts from a single-threaded application and profiles it on a large single-threaded reconfigurable coarse-grain array. From the profiling results, kernels with low IPC and which are less dependent to the other kernels are identified as the high-priority candidates for threading. Depending on the resource demand and dependency of the threads, the programmer statically plans on how and when the reconfigurable coarse-grain array should split into partitions during application execution. When the threads are well-organized, the full array can be efficiently utilized.

Architecture Design Aspects

The FU array on the ADRES is heterogeneous, meaning that a plurality of different FUs 13 are present in the array. There exist dedicated memory units, special arithmetic units and control/branch units on the array that constrain the partitioning. When partitioning the array, it has to be guaranteed that the program being executed on certain partitions can be scheduled. This requires that any instruction invoked in a thread is to be supported by at least one of the functional units in the array partition. The well-formed partitions usually have at least one VLIW FU that can perform branch operations, one FU that can perform memory operations, several arithmetic units if needed, and several FUs that can handle general operations.

On the ADRES architecture, the VLIW register file (RF') 15 is a resource that cannot be partitioned easily. The ADRES architecture may employ a clustered register file. If the RF bank is prohibited to be shared among several

threads, the RF cluster can be partitioned with the VLIW/CGA, and the thread compilation can be greatly simplified. In case a single register file is used, the register allocation scheme must be revised to support the constrained register allocation.

The ADRES architecture may have ultra-wide memory bandwidth. Multi-bank memory adapted to the architecture to reduce bank conflicts has proven to cope nicely with a static data-allocation scheme. On ADRES, the memory and the algorithm core may be interfaced with a crossbar with queues. Such a memory interface offers a scratchpad style of memory presentation to all the load/store units, thus the multi-bank memory can be used as a shared synchronization memory.

Besides the shared memory, other dedicated synchronization primitives like register-based semaphores or pipes can also be adapted to the ADRES template. These primitives can be connected between pairs of functional units that belong to different thread partitions. Synchronization instruction can be added to certain functional units as intrinsics.

In the single-threading ADRES architecture, the program counter and the dynamic reconfiguration counter may be controlled by a finite-state-machine (FSM) type control unit. When implementing the multithreading ADRES, an extendable control mechanism may be used to match the hierarchically partitioned array.

As shown in Fig. 5, the FSM type controller may be duplicated and the controllers may be organized in a hierarchical manner. In this multi-threading controller, each partition is still controlled by an FSM controller 50, but the control path may be extended with two units called merger 51 and bypasser 52. The merger 51 and bypasser 52 form a hierarchical master-slave control that is easy to manage during program execution. The merger path is used to communicate change-of-flow information to the master controller of a partition, while the bypasser propagates the current PC or configuration memory address from the master to all slaves within a partition.

The principle of having such a control mechanism is as follows. Suppose an ADRES architecture that can be split into two halves for dual threading, while each half has its own controller. In order to reuse the

controllers as much as possible, each controller controls a partition of the ADRES when the program is running in dual threaded mode, but it is also preferred that one of the controllers takes full control of the whole ADRES when the program is running in the single-threaded mode. By assigning one of the controllers to control the whole ADRES, a master is created. When the ADRES is running in the single-thread mode, the master controller also receives a signal from the slave partition and merges it with the master partition's signal for creating global control signal. At the same time, the slave partition should bypass any signal generated from the local controller and follow the global control signal generated from the master partition. When the ADRES is running in the dual-threaded mode, the master and slave controllers completely ignore the control signals coming from the other partition and only respond to the local signals. This strategy can be easily extended to cope with further partitioning.

Multithreading Methodology

Before a threaded application can be compiled, the application should be reorganized. As shown in Fig. 6, the application may be split into several thread files 61, 62, 63, 64, each of which describes a thread that is to be executed on a specific partition, e.g. C-files, assuming the application is programmed in C. The data shared among threads are defined in a global file that is included in all the thread-files, and protected with a synchronization mechanism. Such reorganization takes modest effort, but makes it easier for a programmer to experiment on different thread/partition combinations to find an efficient, e.g. optimal, resource budget. In the embodiment illustrated in Fig. 6, task 1 is first executed in unified mode. After execution of task 1, the ADRES architecture is split into three parallel processing units for executing task 2, task 3 and task 4 in parallel. After having executed tasks 2, 3 and 5, the ADRES architecture is again brought in unified mode, for executing task 4.

The multithreading architecture description, e.g. the ADRES architecture description, is extended with the partition descriptions, as shown in Fig. 7. Similar to the area-constrained placement and routing on a commercial FPGA, when a thread is scheduled on an ADRES partition, the instruction placement and routing is constrained by the partition description.

The generated assembly code of each thread goes though the assembling process separately, and gets linked in the final compilation step.

The simulator 70 reads the architecture description 71 and generates an architecture simulation model before the application simulation starts. As shown in Fig. 5, each partition has its own controller 50, thus the generation of the controller's simulation model depends on the partition description as well. Furthermore, the control signal distribution is also partition-dependent, thus requires the partition description to be consulted during simulation model generation.

Some other minor practical issues need to be addressed in the multithreading methodology according to embodiments of the present invention. The most costly problem is that different partitions of the ADRES are conceptually different ADRES instances, thus a function compiled for a specific partition cannot be executed on any other partitions. When a function is called by more than one thread, multiple partition-specific binaries of this function have to be stored in the instruction memory for different callers. Secondly, multiple stacks need to be allocated in the data memory.

Each time the ADRES splits into smaller partitions due to the threading, a new stack should be created to store the temporary data. Currently, the best solution to decide where the new stack should be created is based on the profiling, and the thread stacks are allocated at compile time. And finally, each time the new thread is created, a new set of special purpose registers needs to be initialized. Several clock cycles are needed to properly initial the stack points, the return register, etc. immediately after the thread starts running.

Experiment

In order to understand which features are desirable for supporting the multi-threaded methodology according to embodiments of the present invention and to prove its feasibility, an experiment has been carried out based on an MPEG2 decoder, a well-understood benchmark. An objective is to go through the whole process of generating the threaded application executable, partitioning the instruction/data memory for threads, upgrading the cycle-true architecture simulation model and successfully simulating the execution of MPEG2 decoder with a simulator according to embodiments of the present

invention. By going through the whole process, ample knowledge can be acquired on how to automate the compilation for threads and simulation/RTL model generation of MT-ADRES.

The proof-of-concept experiment achieves dual-threading on the MPEG2 decoder. The MPEG2 decoder can be parallelized on several granularities, as described by E. Iwata et al. "Exploiting Coarse-Grain Parallelism in the MPEG-2 Algorithm", Stanford University Computer Systems Lab Technical Report CSL-TR-98-771, September 1998, thus it is a suitable application to experiment on. The Inverse Discrete Cosine Transform (IDCT) and Motion Compensation (MC) have been chosen as two parallel threads, and reorganized the MPEG2 decoder as shown in Fig. 8. The decoder starts its execution on an 8x4 array 80, executes the Variable Length Decoding (VLD) and Inverse Quantization (IQ), and switches to the threading mode (split mode). When the thread execution starts, the 8x4 array 80 splits into two 4x4 ADRES arrays 81, 82 and continues on executing the threads. When both threads are finished, the two 4x4 arrays 81, 82 unify and continue on executing the add block function in unified mode on the 8x4 array 80. The MPEG2 program has been reorganised as described in Fig. 8, and added "split" instructions 83 (fork instruction) and "unify" instructions 84 (join instructions) as intrinsics. These instructions 83, 84 currently do nothing by themselves, and are only used to mark where the thread mode should change in the MPEG2's binary code. These marks are used by the split-control unit at run time for enabling/disabling the thread-mode program execution.

The dual-threading compilation flow in accordance with embodiments of the present invention is shown in Fig. 9. The lack of partition-based scheduling forces us to use two architectures as the input to the scheduling. The 8x4 architecture 90 is carefully designed so that the left and the right halves are exactly the same. This architecture is the execution platform of the whole MPEG2 binary. A 4x4 architecture 91 is also needed, which is a helping architecture that is compatible to either half of the 8x4 array. This architecture is used as a half-array partition description of the 8x4 architecture 90. With these two architectures 90, 91 in place, the single-threaded file 92 , e.g. C-file, is compiled, as well as the threads on the 8x4 architecture and the 4x4

architecture, respectively. The later linking by linker 95 stitches the binaries from different parts of the program seamlessly.

The memory partitioning of the threaded MPEG2 is shown in Fig. 10. The instruction fetching (IF), data fetching (DF) and the configuration-word fetching (CW) has been duplicated for dual-threading. The fetching unit pairs are step-locked during single-threaded program execution. When the architecture goes into the dual-threading mode, the fetching unit pairs split up into two sets, each of which is controlled by the controller in a thread partition.

During the linking, the instruction memory 101 and data memory 102 are divided into partitions. Both the instruction memory 101 and configuration memory 103 are divided into three partitions. These three partition pairs store the instructions and configurations of single-threaded binaries, IDCT binaries and MC binaries, as shown on Fig. 10. The data memory 102 is divided into four partitions. The largest data memory partition is the shared global static data memory 105. Both single-threaded and dual-threaded programs store their data into the same memory partition 105. The rest of the data memory 102 is divided into three stacks. The IDCT thread's stack 106 grows directly above the single-threaded program's stack 107, since they use the same physical controller and stack pointer. The base stack address of the MC thread is offset to a free memory location at linking time. When the program execution goes into dual-threading mode, the MC stack pointer is properly initialized at the cost of several clock cycles.

In an alternative embodiment, the clustered register file can be clustered among the array partitions so that each thread has its own register file(s). However, due to the lack of a partitioning-based register allocation algorithm at the current stage, the partitioning approach is not very feasible. We experiment on the ADRES architecture with a single global register file and go for the duplication based approach to temporary accommodate the register file issue. As shown in Fig. 11, a shadow register file 110 may be added into the architecture. When a single-threaded program is being executed, the shadow register file 110 is step-locked with the primary register file 15. When the program initiates the dual-thread execution, the MC thread gets access to the shadow register file 110 and continues the execution on the array partition 112

and shadow register file 15. When the program resumes to the single threaded execution, the shadow register file 110 becomes hidden again. The MPEG2 program is slightly modified so that all the data being shared between threads and all the live-in and live-out variables are passed through the global data memory.

The scalable control concept in Fig. 5 has been verified in the simulation model in accordance with embodiments of the present invention. It has been shown that this scheme can be extended to a certain scale, and the control unit simulation model generation can be automated.

During the program linking, it is identified where the "split" and "unify" instructions are stored in the instruction memory. These instructions' physical addresses mark the beginning and the ending point of the dual-threading mode. During the simulation model generation, these instructions' addresses are stored in a set of special-purpose registers in a split-control unit. After the program starts executing, the program counter's (PC) values are checked by the split-control unit in each clock cycle. When the program counter reaches the split point, he split-control unit sends control signals to the merger and bypasser to enable the threading mode. After the program goes into the threaded mode, the split-controller waits for both threads to join in by reaching the PC value where the "unify" instructions are stored. The first thread that joins in will be halted till the other thread finishes. When the second thread eventually joins in, the split-control switches the ADRES array back to single-threaded mode, and the architecture resumes to the 8x4 array mode. The overhead of performing split and unify operations mainly comes from executing several bookkeeping instructions on some special-purpose registers, and such overhead is negligible.

When an application gets more complicated and has multiple splitting/unifying point, the current approach will become more difficult to manage, thus architectures according to embodiments of the present invention may only rely on the instruction decoding to detect the "split" and "unify" instructions. The split-control unit may be removed, and part of its function may be moved into each partition's local controller.

The simulation result shows that the threaded MPEG2 produces the correct image frame at a slightly faster rate. Table 1 shows the clock count of the first 5 image frames decoded on the same 8x4 ADRES instance with and without threading.

frame number	single-thread cc count	dual-thread cc count	single-thread decoding time	dual-thread decoding time	speed-up
1	1874009	1802277			
2	2453279	2293927	579270	491650	15.1%
3	3113150	2874078	659871	580151	12.1%
4	3702269	3374421	589119	500343	15.1%
5	4278995	3861978	576726	487557	15.5%

Table 1. Clock cycle count of single and dual threaded MPEG2 on the same architecture

The cc count column shows the clock count of the overall execution time when an image frame is decoded, while the decoding time column shows the clock count between two frames are decoded. The dual-threaded MPEG2 is about 12-15% faster than the single-thread MPEG2 for the following reasons.

Both IDCT and MC algorithm have high loop-level parallelism, thus can optimally utilize the single-threaded 8X4 architecture. When scheduled on the x4 architecture as threads, the IPCs of both algorithms are reduced by half due to the halved array size, thus the overall IPCs of the non-threaded and the threaded MPEG2 are nearly the same. As mentioned earlier, when the ADRES' size is increased to certain extent, the scheduling algorithm has difficulty exploring parallelism in the applications and using the ADRES array optimally. It is clear that doubling/quadrupling the size of the ADRES array or choosing low-parallelism algorithm for threading will result in more speed-up.

As observed, the marginal performance gain is mostly achieved from he ease of modulo-scheduling on the smaller architecture. When an application is scheduled on a larger CGA, many redundant instructions are added into the kernel for routing purpose. Now the IDCT and MC kernels are scheduled on a half-CGA partition instead of the whole ADRES, even if the overall IPC of the application is not improved much, the amount of redundant instructions added during scheduling for placement and routing purpose has been greatly reduced.

By carrying out the dual-threading experiment on MPEG2 decoding algorithm, ample knowledge on the MT-ADRES architecture has been gained. The simulation results show that the MPEG2 has gain 12-15% of speed up. The results so far demonstrate that the threading approach is adequate for the ADRES architecture, is practically feasible, and can be scaled to a certain extend. So far, the only extra hardware cost added onto ADRES is a second control unit, the size of which can be neglected for an ADRES larger than 3X3.

It is to be understood that although preferred embodiments, specific constructions and configurations have been discussed herein for devices according to the present invention, various changes or modifications in form and detail may be made without departing from the scope and spirit of this invention.

CLAIMS

1.- Signal processing device adapted for simultaneous processing of at least two process threads in a multi-processing manner, comprising
a plurality of functional units capable of executing word- or subword- level operations on data,
routing resources for interconnecting said plurality of functional units, said routing resources supporting a plurality of interconnect arrangements that can be dynamically switched, at least one of said interconnect arrangements interconnecting said plurality of functional units into at least two non-overlapping processing units each with a pre-determined topology, each of said processing units being configured to process a respective one of said process threads,
the signal processing device furthermore comprising at least two control modules, each control module being assigned to one of said processing units for control thereof.

2.- Signal processing device according to claim 1, furthermore comprising a plurality of data storages, wherein the routing resources interconnect the plurality of functional units and the plurality of data storages.

3.- Signal processing device according to any of the previous claims, including a data storage in which an application code is stored, said application code defining a process comprising the at least two process threads and being executable by said processing units, and wherein the routing resources are adapted for dynamically switching between interconnect arrangements at pre-determined points in the application code.

4.- Signal processing device according to any of the previous claims, wherein the routing resources are adapted for dynamically switching interconnect arrangements depending on data content of a running application.

5.- Signal processing device according to claim 4, wherein the routing resources comprise multiplexing and/or demultiplexing circuits.

6.- Signal processing device according to claim 5, the signal processing device having a clock, wherein the multiplexing and/or demultiplexing circuits are adapted to be configured with settings for dynamically switching interconnect arrangements, wherein the settings may change every clock cycle.

7.- Signal processing device according to any of the previous claims, furthermore comprising at least one global storage shared between a plurality of functional units.

8.- Signal processing device according to any of the previous claims, including at least two different types of functional units.

9.- Signal processing device according to any of the previous claims, wherein at least another of said interconnect arrangement interconnects said plurality of functional units into a single processing unit under control of a single control module.

10.- Signal processing device according to claim 9, wherein at least one of the at least two control modules is part of a global control unit for use in an interconnect arrangement with a single processing unit.

11.- Signal processing device according to claim 10, wherein in at least one interconnect arrangement with a single processing unit, at least one of the control modules drives control signals of all the functional units by having at least one other control module to follow it.

12.- Signal processing device according to any of the previous claims, adapted for re-using at least part of the control modules assigned to the

processing units in an interconnect arrangement with a plurality of non-overlapping processing units in the control module used in an interconnect arrangement with a single processing unit.

13.- Method for executing an application on a signal processing device as in any of the previous claims, the method comprising:
executing the application on the signal processing device as a single process thread under control of a primary control module, and
dynamically switching the signal processing device into a device with at least two non-overlapping processing units, and splitting a portion of the application in at least two process threads, each process thread being executed simultaneously as a separate process thread on one of the processing units, each processing unit being controlled by a separate control module.

14.- Method according to claim 13, wherein switching the signal processing device into a device with at least two processing units is determined by a first instruction in application code determining the application.

15.- Method according to claim 14, wherein the first instruction contains a starting address of the instructions of each of the separate process threads.

16.- Method according to any of claims 13 to 15, furthermore comprising dynamically switching back the signal processing device into a device with a single processing unit, synchronising the separate control modules and joining the at least two threads of the application into a single process thread, the single process thread being executed as a process thread on the single processing unit under control of the synchronised control modules.

17.- Method according to claim 16, wherein switching back the signal processing device into a device with a single processing unit is

determined by a second instruction in application code determining the application.

18.- Method according to claim 17, wherein the second instruction contains a starting address of the instructions to be executed as the single process thread.

19.- Method according to any of claims 13 to 18, wherein the single control module re-uses at least one of the separate control modules when executing the application as a single process thread.

20.- Method according to any of claims 13 to 19, wherein, in an interconnect arrangement with a single processing unit, one of the separate control modules drives control signals of all the functional units by having the other control modules to follow it.

21.- Method for compilation of application source code in order to obtain compiled code being executable on a signal processing device as in any of claims 1 to 12, comprising
inputting application source code and
generating compiled code from the application source code,
wherein generating the compiled code comprises including, in the compiled code, a first instruction for configuring the signal processing device for simultaneous execution of multiple process threads and for starting the simultaneous execution of the process threads, and including a second instruction to end the simultaneous execution of the multiple process threads such that when the last of the multiple process threads decodes this instruction, the signal processing device is configured to continue execution in unified mode.

22.- Method according to claim 21, furthermore comprising providing an architectural description of the signal processing device, the architectural

description including descriptions of pre-determined interconnect arrangements of functional units forming processing units.

23.- Method according to claim 22, wherein providing the architectural description includes providing a separate control module per processing unit.

24.- Method according to any of claims 21 to 23, wherein the first instruction contains the start address of instructions of each of the multiple process threads.

25.- Method according to any of claims 21 to 24, wherein the second instruction contains the start address of instructions to be executed in unified mode after the execution of the multiple process threads.

26.- Method according to any of claims 21 to 25, wherein generating the compiled code comprises
partitioning the application source code, thus generating code partitions,
labelling in which mode and on which processing unit the code partitions are to be executed,
separately compiling each of the code partitions, and
linking the compiled code partitions into a single executable code file.

27.- A method for adjusting an application to be executed on a signal processing device, comprising performing exploration of various partitionings of the application,
wherein performing the exploration comprises changing an instance of an architectural description of said signal processing device for exploring various interconnect arrangements of said signal processing device.

28.- A method according to claim 27, wherein exploring interconnect arrangements of said signal processing device includes exploring dynamically switching between an interconnect arrangement having a

single processing unit under control of a single control module and an interconnect arrangement having at least two processing units each under control of a separate control module.

29.- A computer program product for executing any of the methods as claimed in claims 13 to 20 when run on a signal processing device as claimed in any of claims 1 to 12.

30.- A machine readable data storage device storing the computer program product of claim 29.

31.- Transmission of the computer program product of claim 29 over a local or wide area telecommunications network.

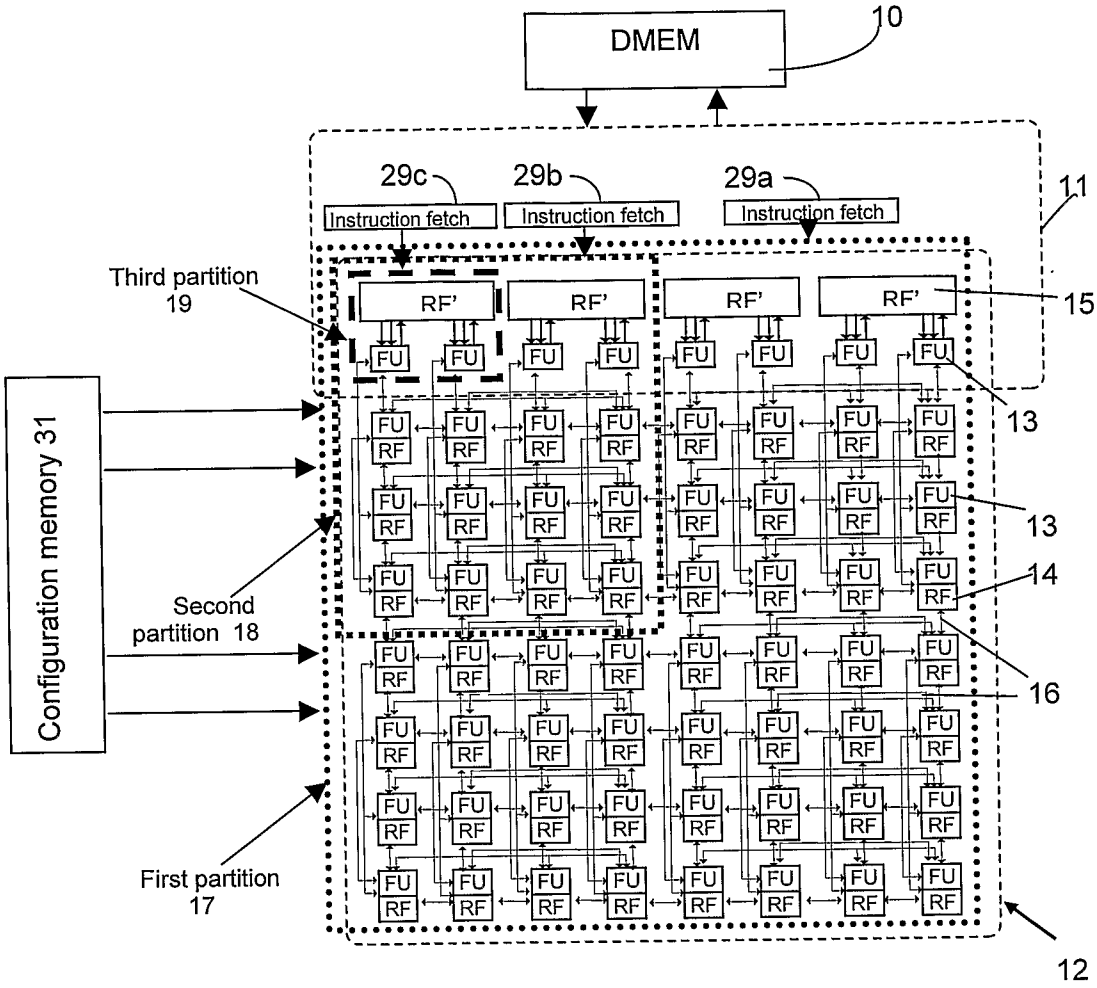

DMEM
10
29c
29b
29a
Instruction fetch
Instruction fetch
Instruction fetch
11
Third partition
19
RF'
RF'
RF'
RF'
15
FU
RF
13
13
14
16
Configuration memory 31
Second
partition 18
First partition
17
12

Fig. 1

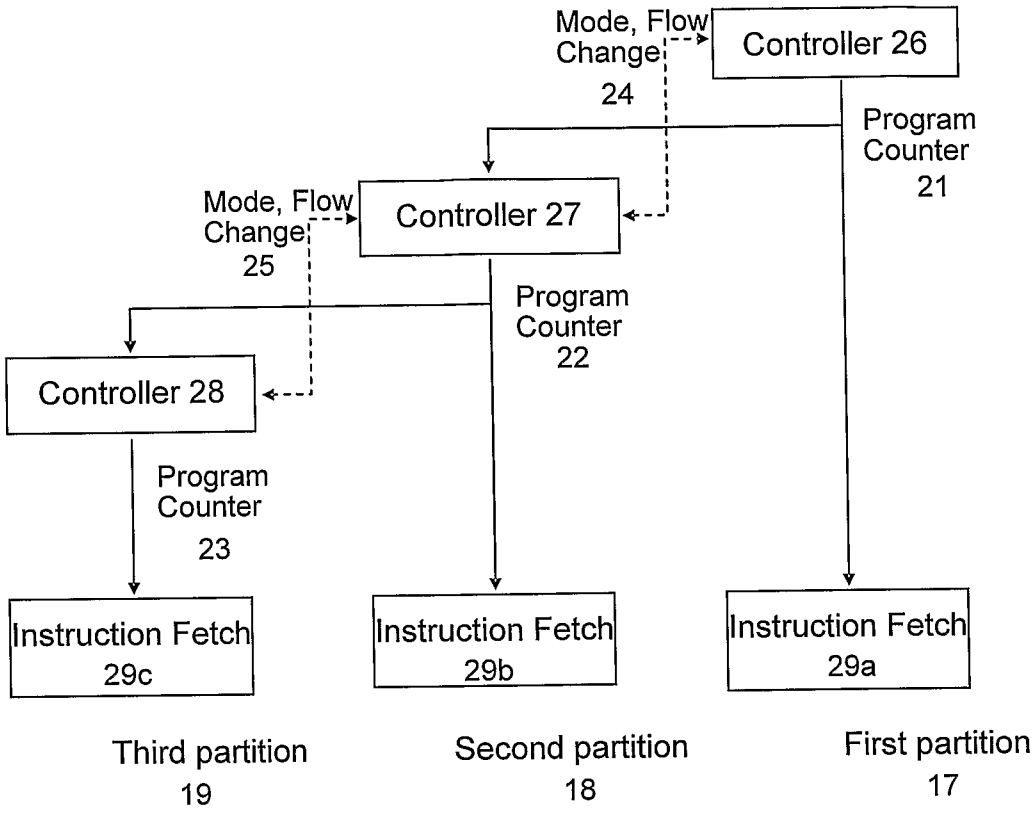

Mode, Flow Change
24
Controller 26
Program Counter
21
Mode, Flow Change
25
Controller 27
Program Counter
22
Controller 28
Program Counter
23
Instruction Fetch
29c
Instruction Fetch
29b
Instruction Fetch
29a
Third partition
19
Second partition
18
First partition
17

Fig. 2

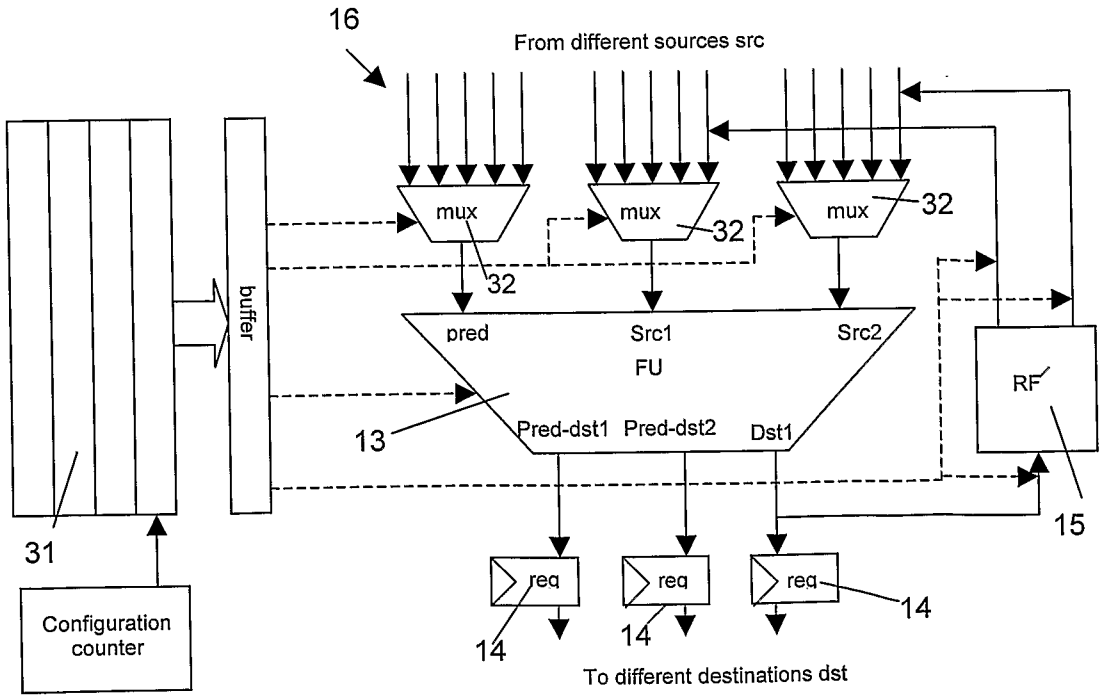

16
From different sources src
mux
mux
mux
32
32
32
buffer
pred
Src1
Src2
FU
Pred-dst1
Pred-dst2
Dst1
13
RF
15
31
Configuration counter
req
req
req
14
14
14
To different destinations dst

Fig. 3

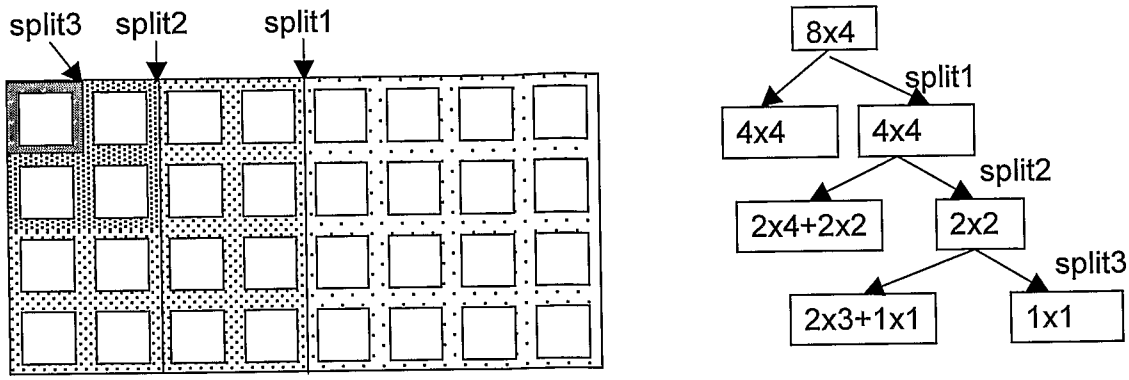

split3
split2
split1
8x4
split1
4x4
4x4
split2
2x4+2x2
2x2
split3
2x3+1x1
1x1

Fig. 4

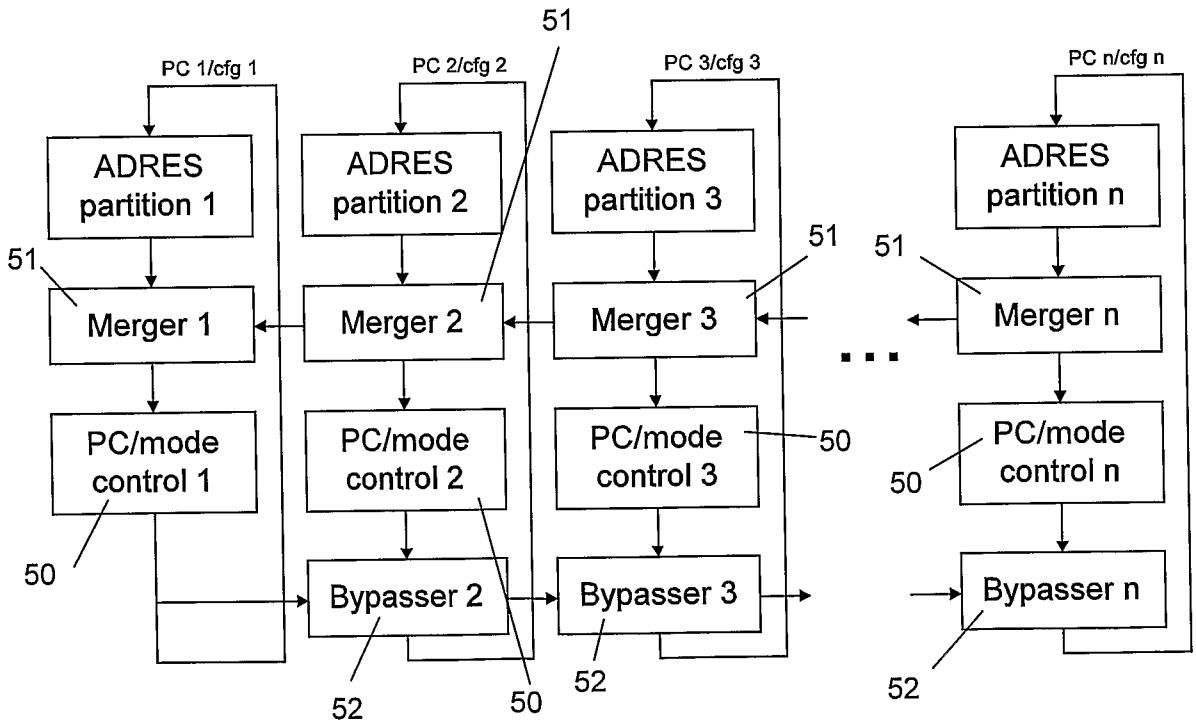

51
PC 1/cfg 1
PC 2/cfg 2
PC 3/cfg 3
PC n/cfg n
ADRES partition 1
ADRES partition 2
ADRES partition 3
ADRES partition n
51
51
51
Merger 1
Merger 2
Merger 3
Merger n
PC/mode control 1
PC/mode control 2
PC/mode control 3
PC/mode control n
50
50
50
Bypasser 2
Bypasser 3
Bypasser n
52
52
50
52
52

Fig. 5

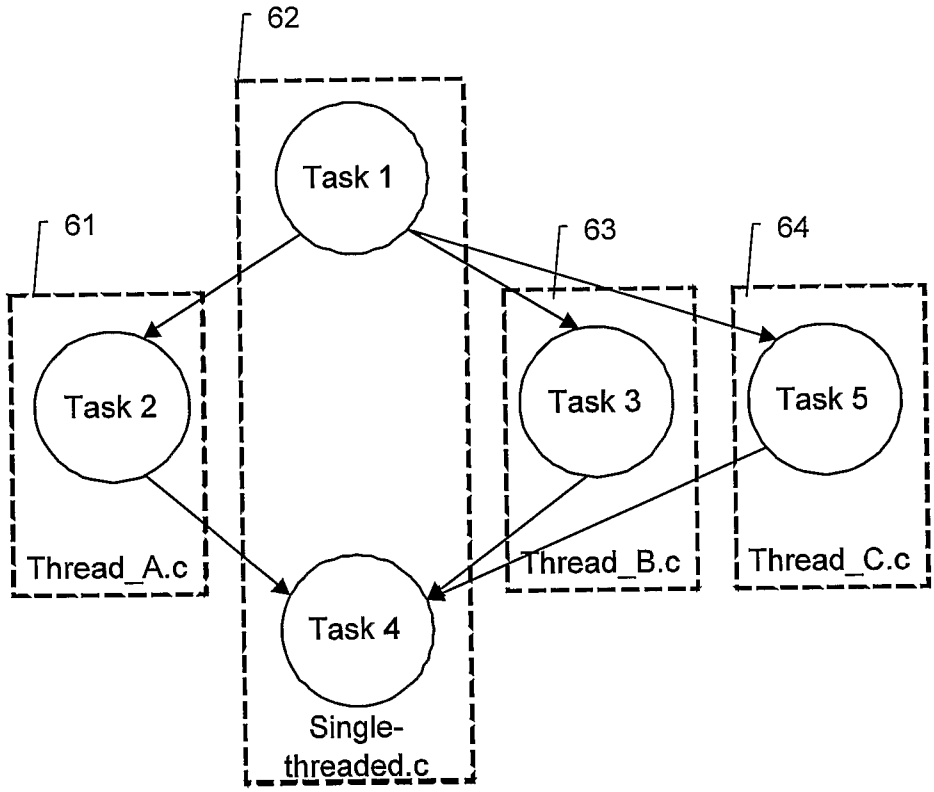

62
Task 1
61
63
64
Task 2
Task 3
Task 5
Thread_A.c
Thread_B.c
Thread_C.c
Task 4
Single-
threaded.c

Fig. 6

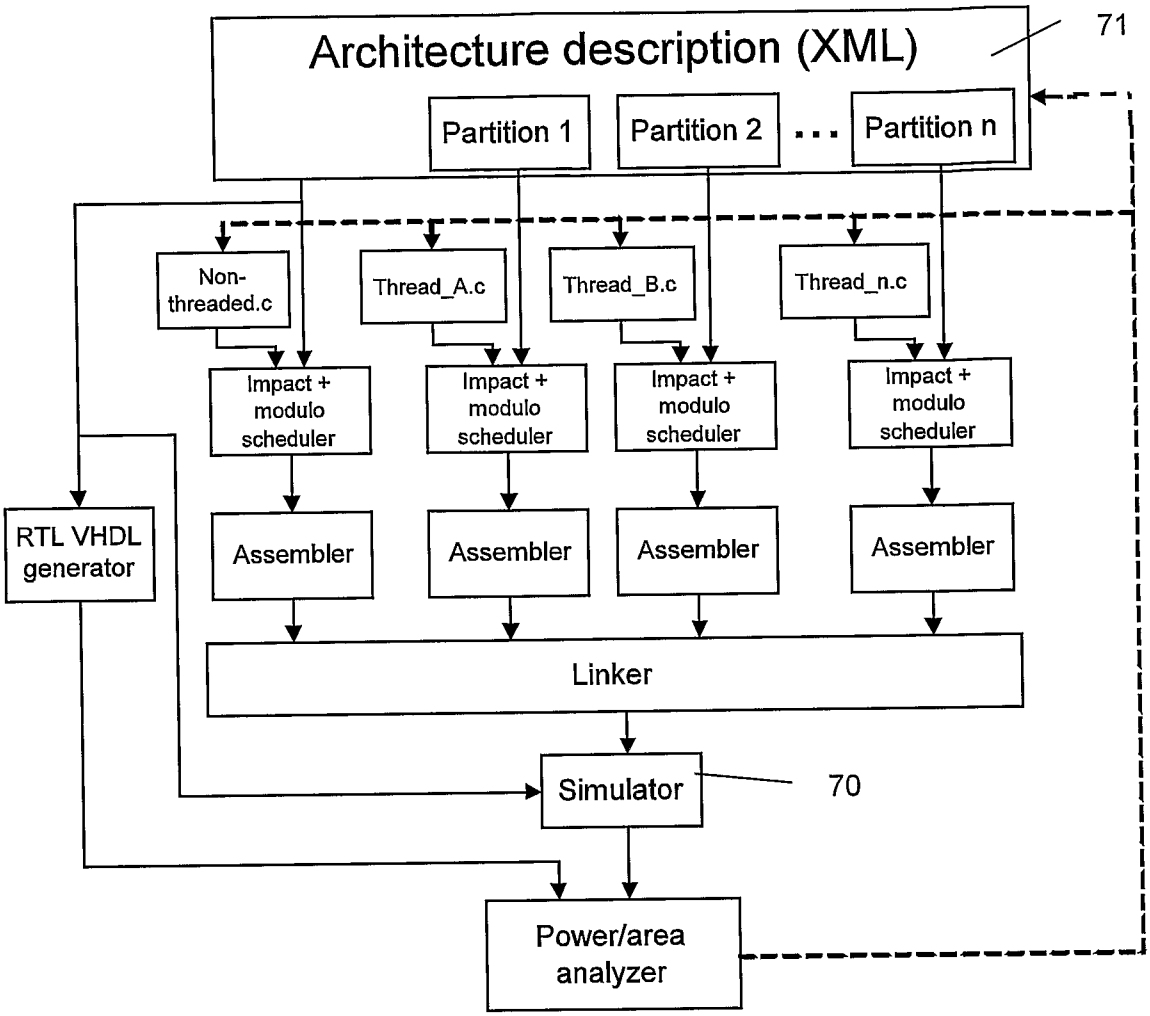

Architecture description (XML)
71
Partition 1
Partition 2
...
Partition n
Non-threaded.c
Thread_A.c
Thread_B.c
Thread_n.c
Impact + modulo scheduler
Impact + modulo scheduler
Impact + modulo scheduler
Impact + modulo scheduler
RTL VHDL generator
Assembler
Assembler
Assembler
Assembler
Linker
Simulator
70
Power/area analyzer

Fig. 7

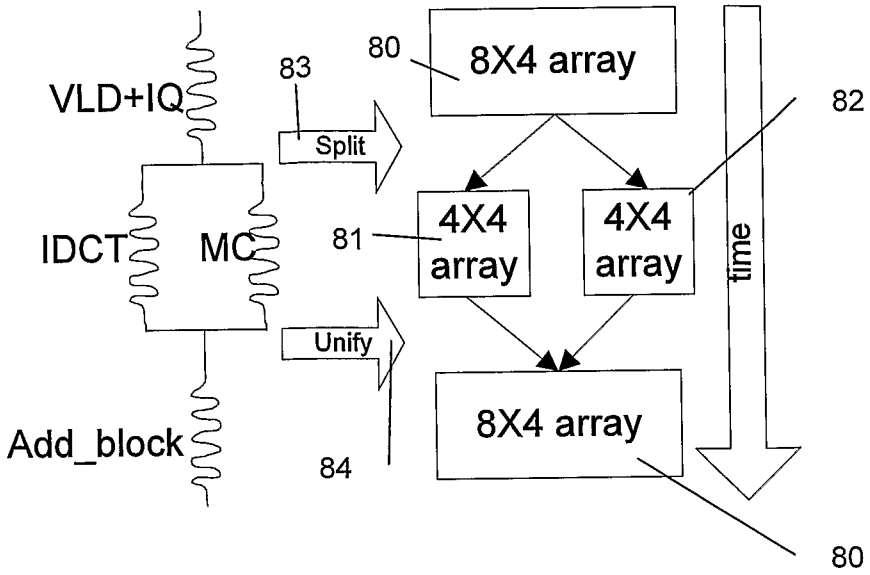

VLD+IQ
IDCT
MC
Add_block
83
Split
Unify
84
80
8X4 array
81
4X4 array
4X4 array
8X4 array
82
time
80

Fig. 8

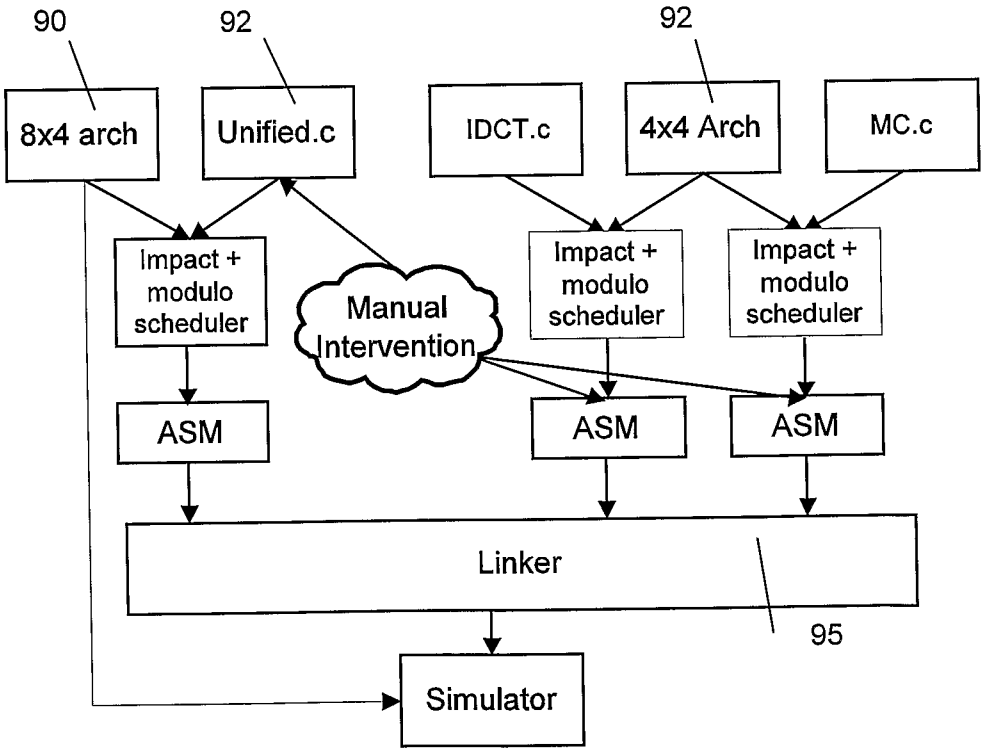

90
92
92
8x4 arch
Unified.c
IDCT.c
4x4 Arch
MC.c
Impact + modulo scheduler
Manual Intervention
Impact + modulo scheduler
Impact + modulo scheduler
ASM
ASM
ASM
Linker
95
Simulator

Fig. 9

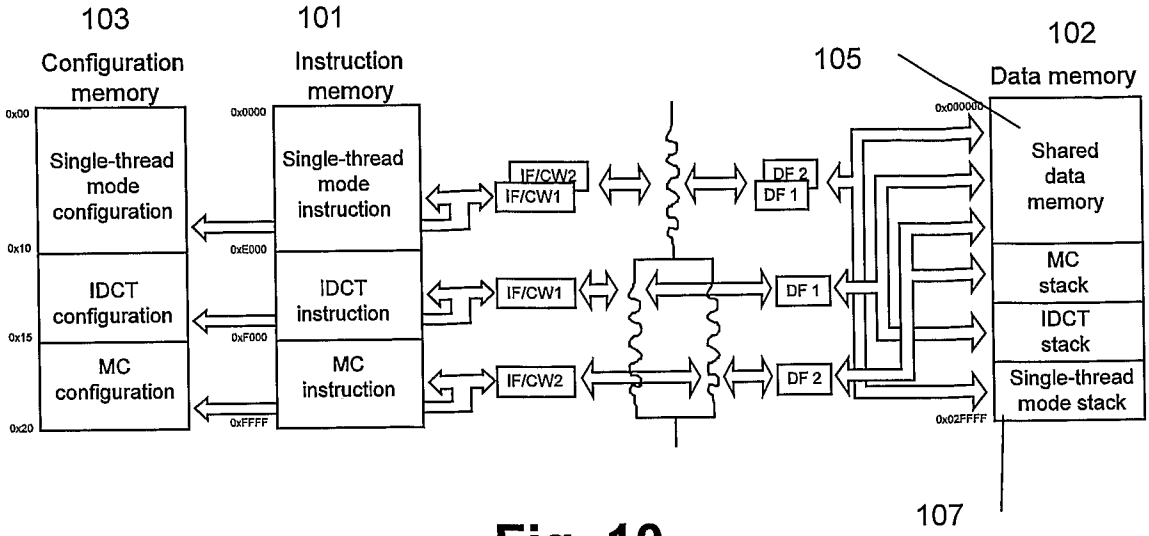

103
Configuration memory
101
Instruction memory
105
102
Data memory
0x00
Single-thread mode configuration
0x10
IDCT configuration
0x15
MC configuration
0x20
0x0000
Single-thread mode instruction
0xE000
IDCT instruction
0xF000
MC instruction
0xFFFF
IF/CW2
IF/CW1
IF/CW1
IF/CW2
DF 2
DF 1
DF 1
DF 2
0x000000
Shared data memory
MC stack
IDCT stack
Single-thread mode stack
0x02FFFF
107

Fig. 10

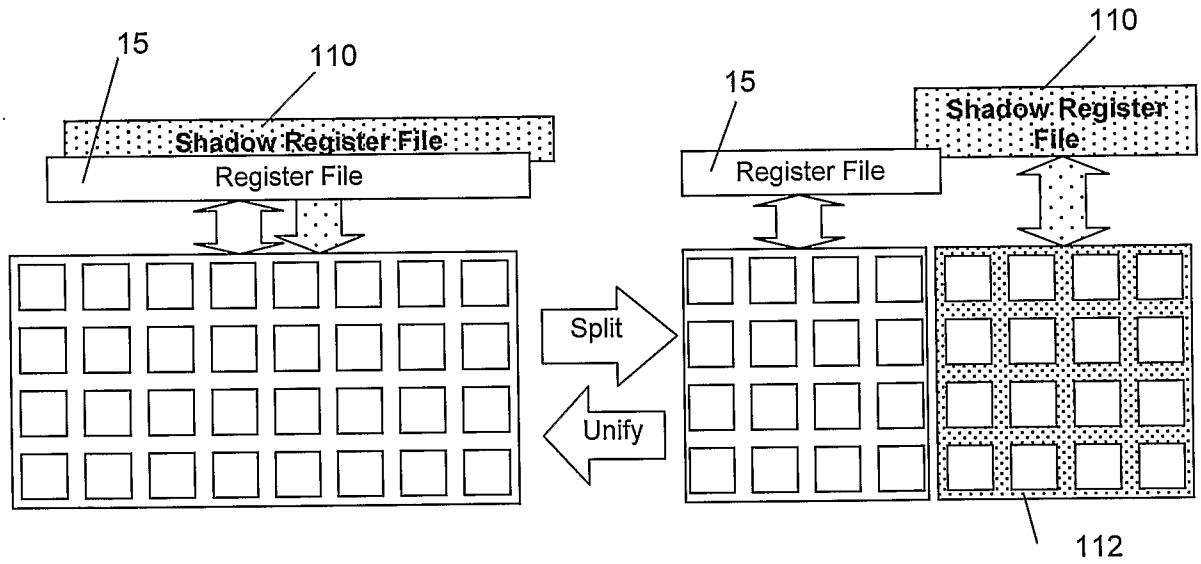

15
110
Shadow Register File
Register File
Split
Unify
15
110
Shadow Register File
Register File
112

Fig. 11