US 20100122105A1

(19) **United States**

(12) **Patent Application Publication**

(10) **Pub. No.: US 2010/0122105 A1**

(43) **Pub. Date: May 13, 2010**

Arslan et al.

(54) **RECONFIGURABLE INSTRUCTION CELL ARRAY**

(75) Inventors: **Tughrul Arslan**, Edinburgh (GB); **Mark John Millward**, Bristol (GB); **Sami Khawam**, Edinburgh (GB); **Ioannis Nousias**, Edinburgh (GB); **Ying Yi**, Edinburgh (GB)

Correspondence Address:
EDWARDS ANGELL PALMER & DODGE LLP
P.O. BOX 55874
BOSTON, MA 02205 (US)

(73) Assignee: **The University Court of the University of Edinburgh**, Edinburgh (GB)

(21) Appl. No.: **11/919,270**

(22) PCT Filed: **Apr. 28, 2006**

(86) PCT No.: **PCT/GB2006/001556**

§ 371 (c)(1),
(2), (4) Date: **Jan. 26, 2010**

(30) **Foreign Application Priority Data**

Apr. 28, 2005 (GB) 0508589.9
Jun. 6, 2006 (GB) 0604428.3

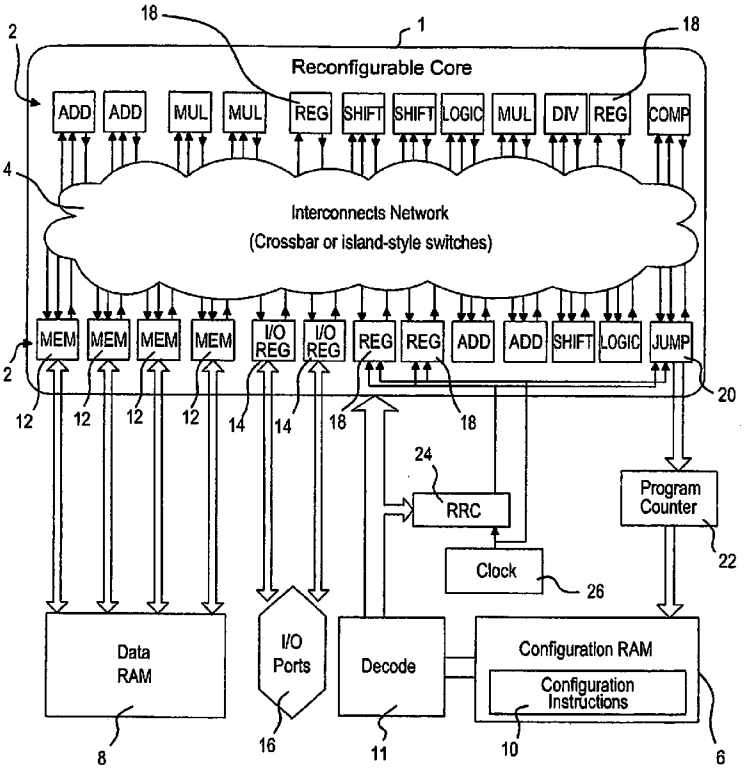
Publication Classification

(51) **Int. Cl.**
G06F 9/30 (2006.01)
G06F 1/04 (2006.01)

(52) **U.S. Cl.** .. **713/500**; 712/208; 712/220; 712/E09.016; 712/E09.028

(57) **ABSTRACT**

A reconfigurable processor architecture, compiler and method of program instruction execution provides reduced cost, short design time, low power consumption and high performance. The processor executes program instructions having datapaths of both dependent and independent program instructions. Simultaneous multithreading is also Interconnects Network supported. The processor has a reconfigurable core (**1**) with an interconnection network (**4**) and a heterogeneous array of instruction cells (**2**) each connected to the interconnection network (**4**). A decoding module (**11**) receives configuration instruction (**10**), each instruction encoding the mapping of one of the datapaths to a circuit of the instruction cells (**2**). The decoding module (**11**) decodes each configuration instruction (**10**) and configures the interconnection network (**4**) and instruction cells in order to map the datapath to the circuit of the instruction cells and execute the program instructions. A clock module (**24**) is reconfigurable each clock cycle by the configuration instruction (**10**). The compiler generates configuration instructions (**10**) for the processor by identifying the datapaths of both dependent and independent program instructions then mapping them as circuits of the instruction cells (**2**) using operation chaining.



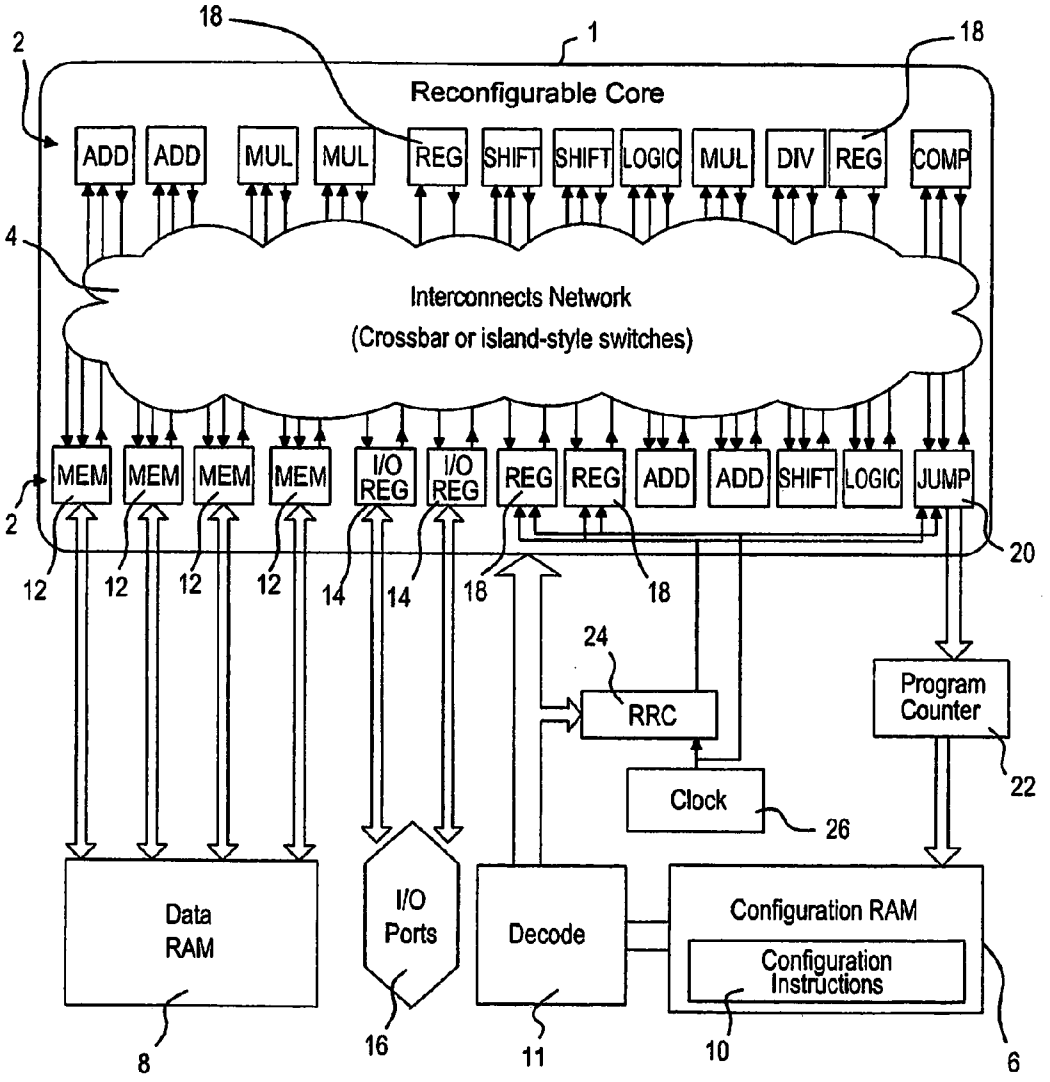

Reconfigurable Core
ADD
ADD
MUL
MUL
REG
SHIFT
SHIFT
LOGIC
MUL
DIV
REG
COMP
Interconnects Network
(Crossbar or island-style switches)
MEM
MEM
MEM
MEM
I/O REG
I/O REG
REG
REG
ADD
ADD
SHIFT
LOGIC
JUMP
RRC
Program Counter
Clock
Data RAM
I/O Ports
Decode
Configuration RAM
Configuration Instructions
1
2
4
6
8
10
11
12
14
16
18
20
22
24
26

Fig. 1

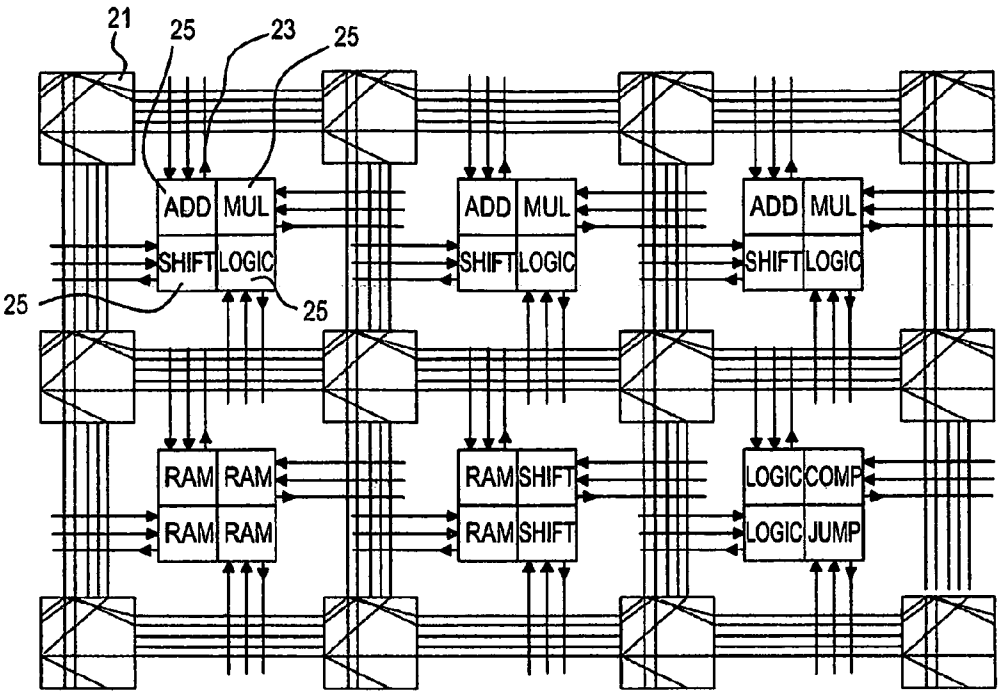

21
25
23
25
ADD
MUL
SHIFT
LOGIC
25
25
RAM
RAM
RAM
RAM
RAM
SHIFT
RAM
SHIFT
LOGIC
COMP
LOGIC
JUMP

Fig. 2a

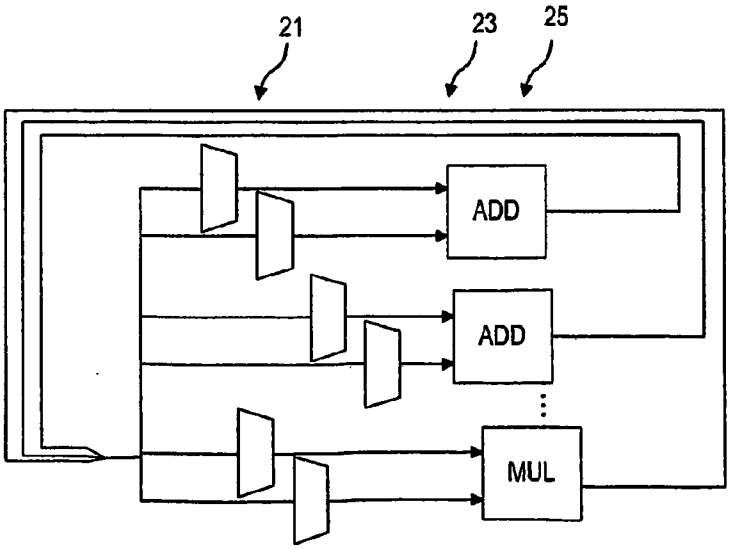

21
23
25
ADD
ADD
MUL

Fig. 2b

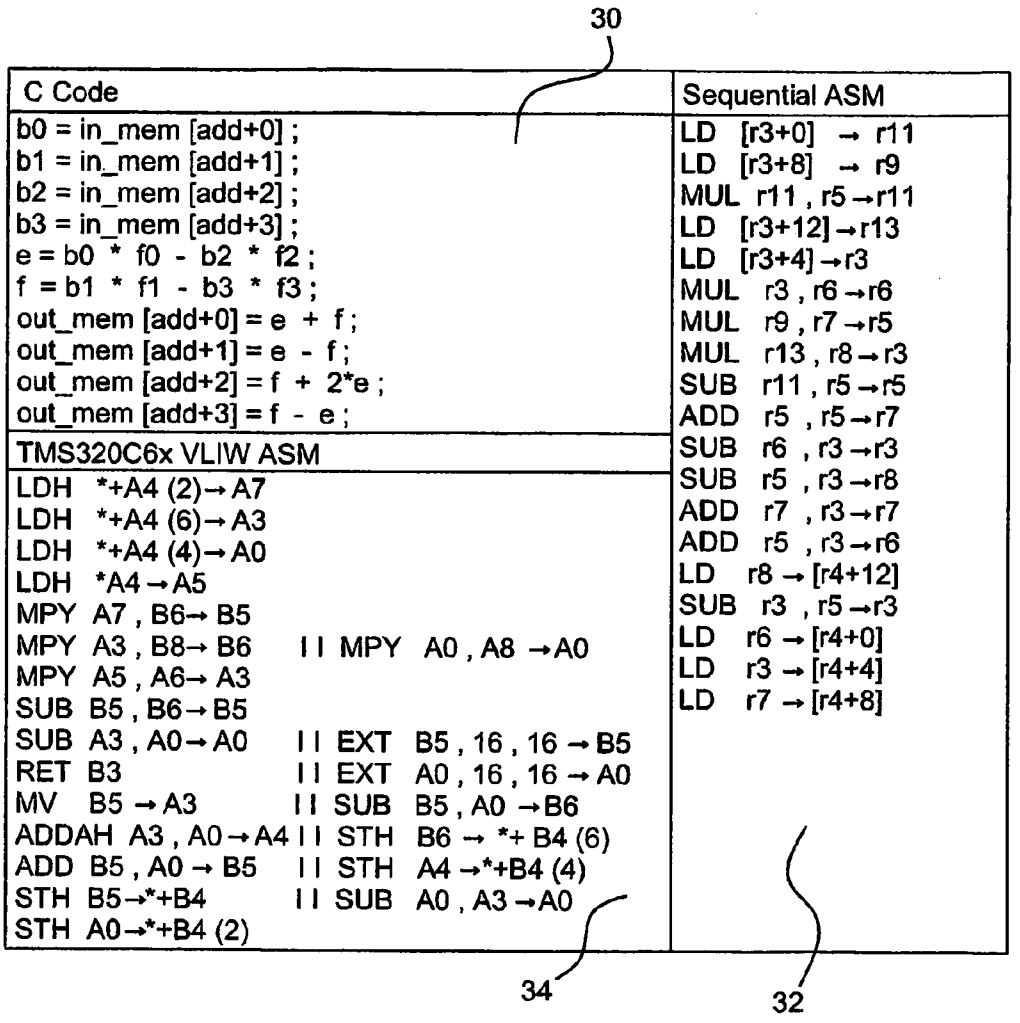

30
C Code
b0 = in_mem [add+0] ;
b1 = in_mem [add+1] ;
b2 = in_mem [add+2] ;
b3 = in_mem [add+3] ;
e = b0 * f0 - b2 * f2 ;
f = b1 * f1 - b3 * f3 ;
out_mem [add+0] = e + f ;
out_mem [add+1] = e - f ;
out_mem [add+2] = f + 2*e ;
out_mem [add+3] = f - e ;
TMS320C6x VLIW ASM
LDH *+A4 (2)→A7
LDH *+A4 (6)→A3
LDH *+A4 (4)→A0
LDH *A4→A5
MPY A7 , B6→ B5
MPY A3 , B8→ B6 || MPY A0 , A8 →A0
MPY A5 , A6→ A3
SUB B5 , B6→B5
SUB A3 , A0→A0 || EXT B5 , 16 , 16 →B5
RET B3 || EXT A0 , 16 , 16 →A0
MV B5 →A3 || SUB B5 , A0 →B6
ADDAH A3 , A0→A4 || STH B6 → *+ B4 (6)
ADD B5 , A0 → B5 || STH A4 →*+B4 (4)
STH B5→*+B4 || SUB A0 , A3 →A0
STH A0→*+B4 (2)
Sequential ASM
LD [r3+0] → r11
LD [r3+8] → r9
MUL r11 , r5→r11
LD [r3+12]→r13
LD [r3+4]→r3
MUL r3 , r6→r6
MUL r9 , r7→r5
MUL r13 , r8→r3
SUB r11 , r5→r5
ADD r5 , r5→r7
SUB r6 , r3→r3
SUB r5 , r3→r8
ADD r7 , r3→r7
ADD r5 , r3→r6
LD r8 → [r4+12]
SUB r3 , r5→r3
LD r6 → [r4+0]
LD r3 → [r4+4]
LD r7 → [r4+8]
34
32

Fig. 3

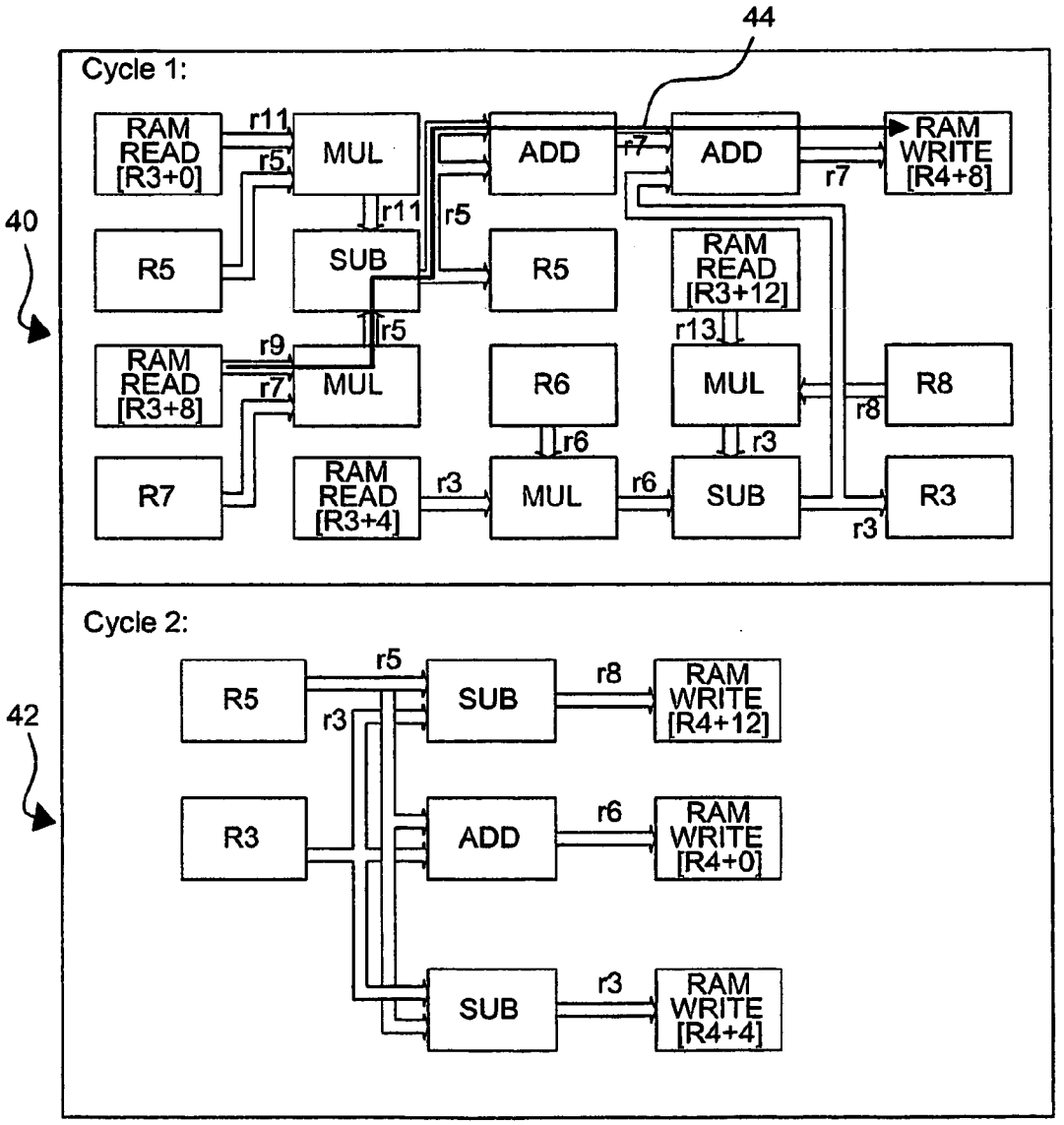

44
Cycle 1:
RAM READ [R3+0]
r11
r5
MUL
ADD
r7
ADD
r7
RAM WRITE [R4+8]
40
R5
r11
r5
SUB
R5
RAM READ [R3+12]
r13
RAM READ [R3+8]
r9
r7
MUL
r5
R6
MUL
r8
R8
r6
r3
R7
RAM READ [R3+4]
r3
MUL
r6
SUB
R3
r3
Cycle 2:
42
R5
r5
r3
SUB
r8
RAM WRITE [R4+12]
R3
ADD
r6
RAM WRITE [R4+0]
SUB
r3
RAM WRITE [R4+4]

Fig. 4

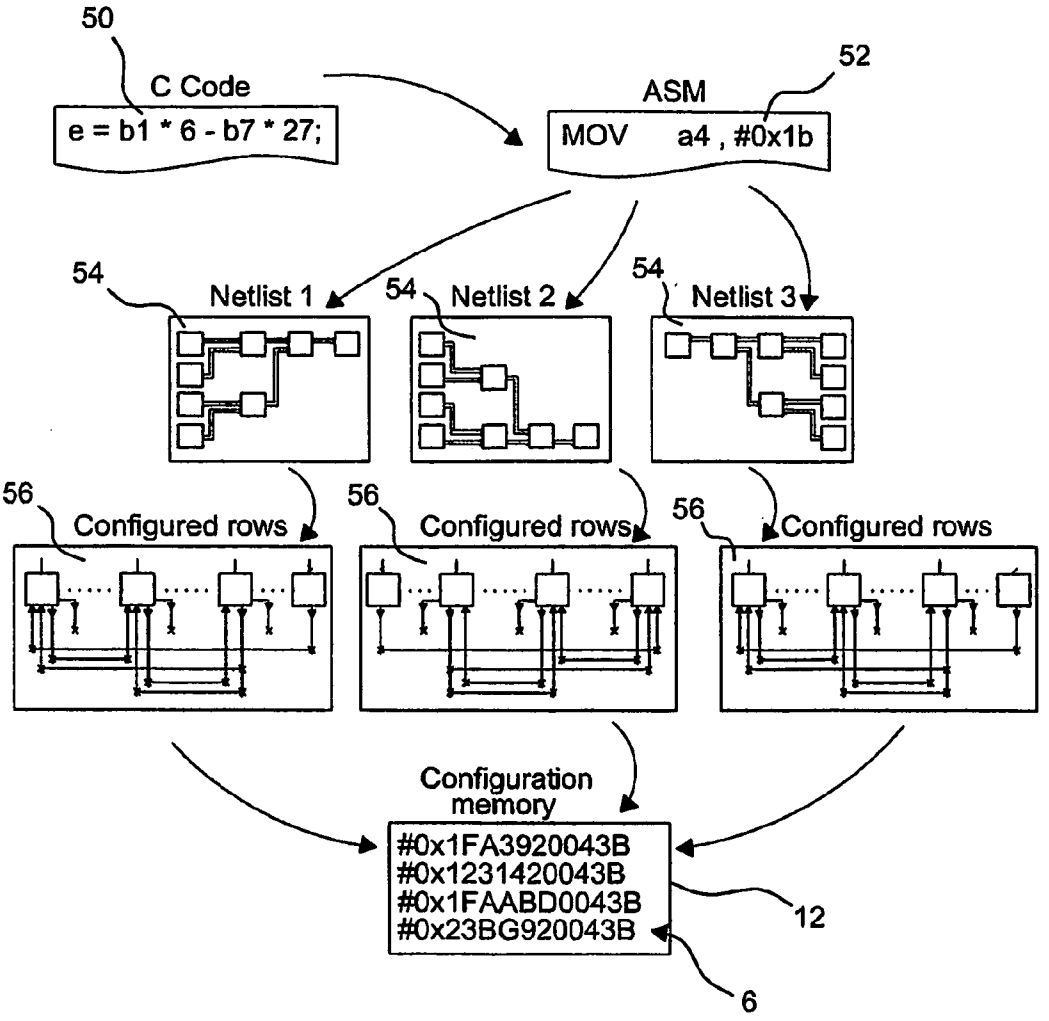

50
C Code
e = b1 * 6 - b7 * 27;
52
ASM
MOV a4 , #0x1b
54
Netlist 1
54
Netlist 2
54
Netlist 3
56
Configured rows
56
Configured rows
56
Configured rows
Configuration memory
#0x1FA3920043B
#0x1231420043B
#0x1FAABD0043B
#0x23BG920043B
12
6

Fig. 5

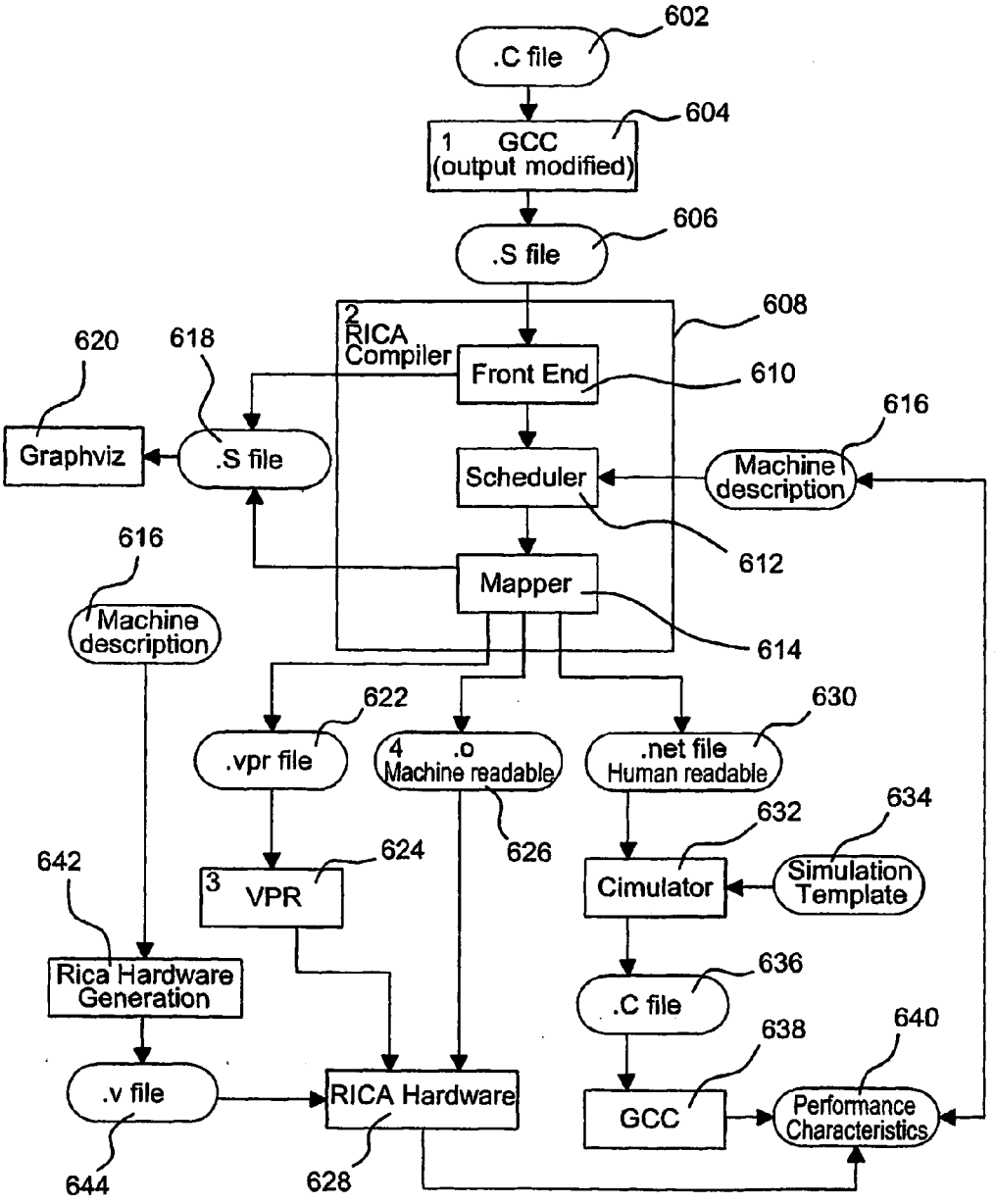

602
.C file
604
1 GCC
(output modified)
606
.S file
608
2
RICA
Compiler
Front End
610
620
618
Graphviz
.S file
Scheduler
616
Machine
description
612
616
Mapper
Machine
description
614
622
630
.vpr file
4 .o
Machine readable
.net file
Human readable
634
632
642
624
626
3 VPR
Cimulator
Simulation
Template
636
Rica Hardware
Generation
.C file
638
640
.v file
RICA Hardware
GCC
Performance
Characteristics
644
628

Fig. 6

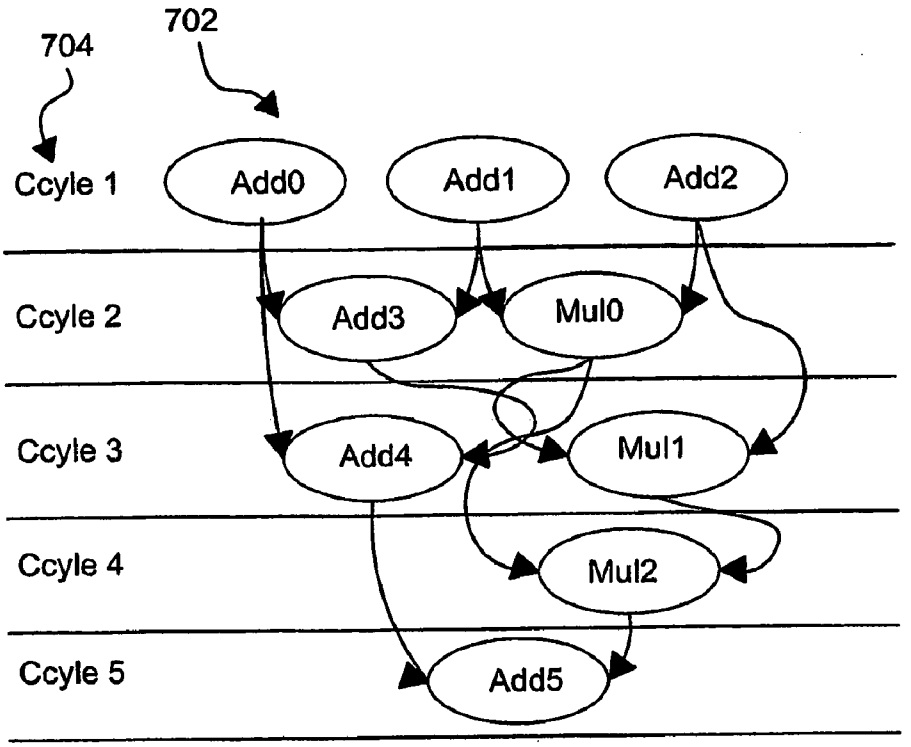

704
702
Ccyle 1
Add0
Add1
Add2
Ccyle 2
Add3
Mul0
Ccyle 3
Add4
Mul1
Ccyle 4
Mul2
Ccyle 5
Add5

Fig. 7a

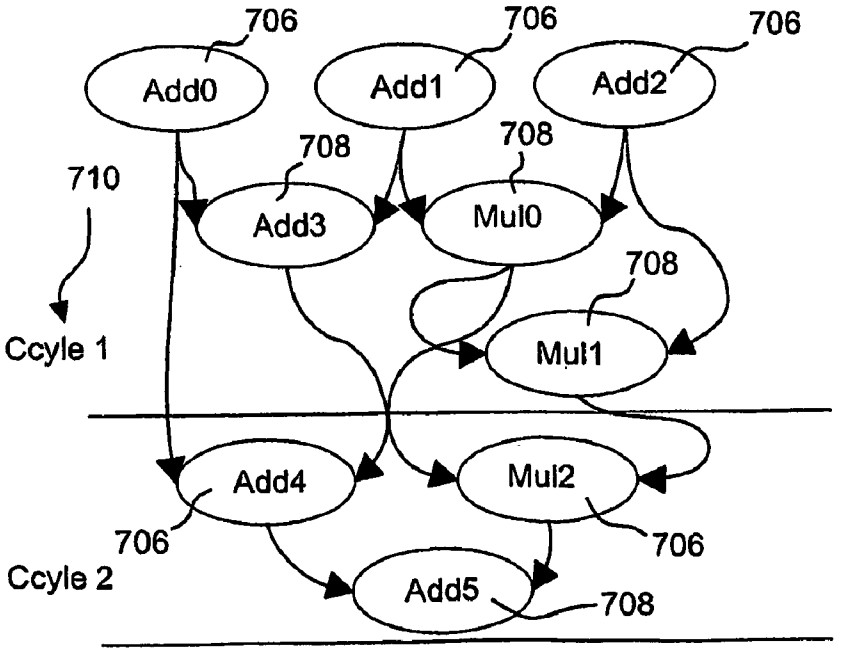

706
706
706
Add0
Add1
Add2
708
708
710
Add3
Mul0
708
Ccyle 1
Mul1
Add4
Mul2
706
706
Ccyle 2
Add5
708

Fig. 7b

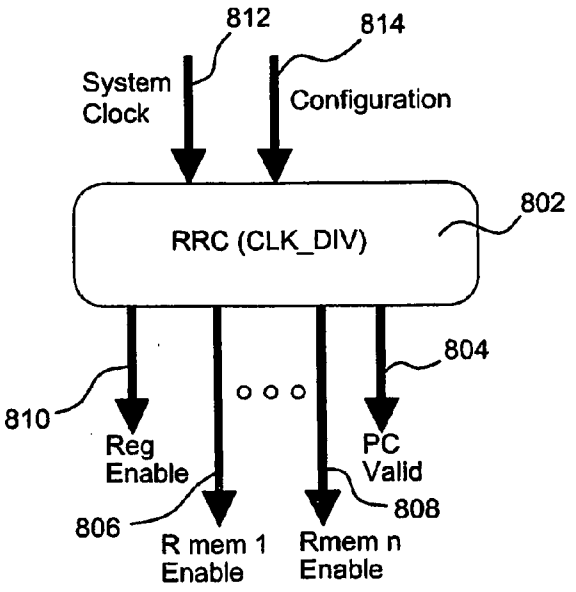

812
814
System Clock
Configuration
802
RRC (CLK_DIV)
804
810
Reg Enable
PC Valid
806
808
R mem 1 Enable
Rmem n Enable

Fig. 8

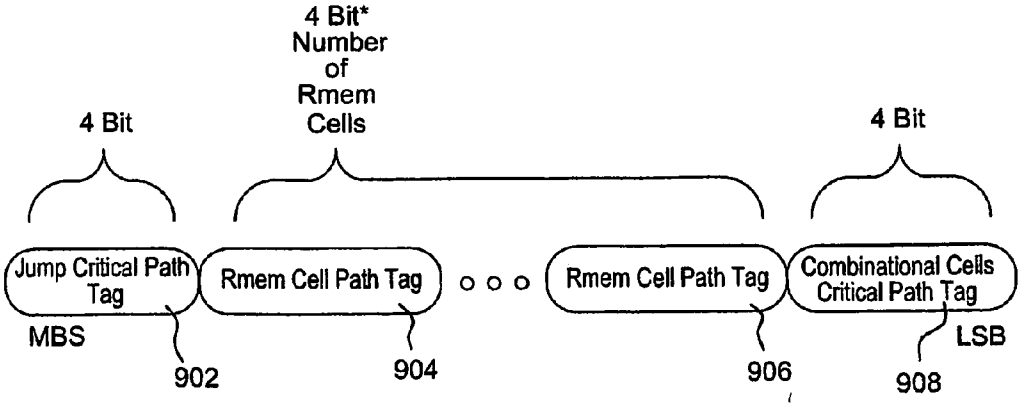

4 Bit*
Number of Rmem Cells
4 Bit
4 Bit
Jump Critical Path Tag
Rmem Cell Path Tag
Rmem Cell Path Tag
Combinational Cells Critical Path Tag
MBS
LSB
902
904
906
908

Fig. 9

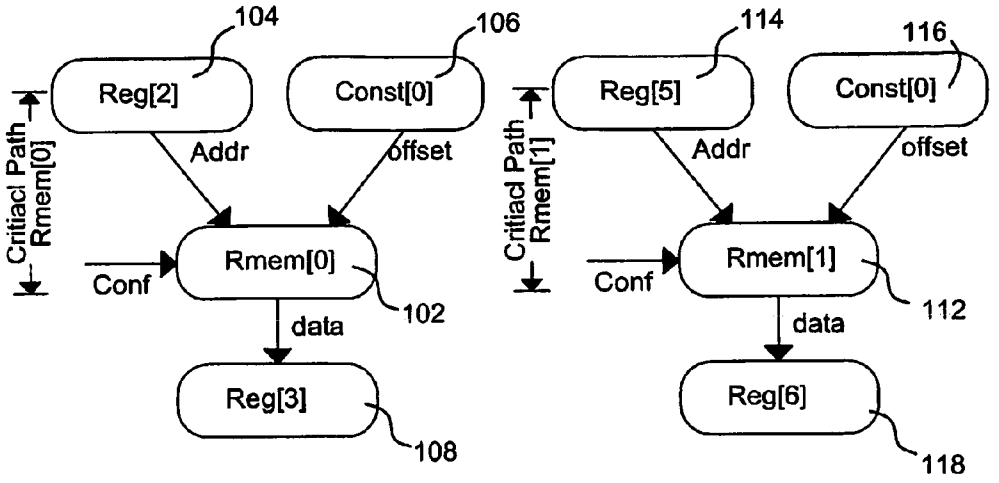
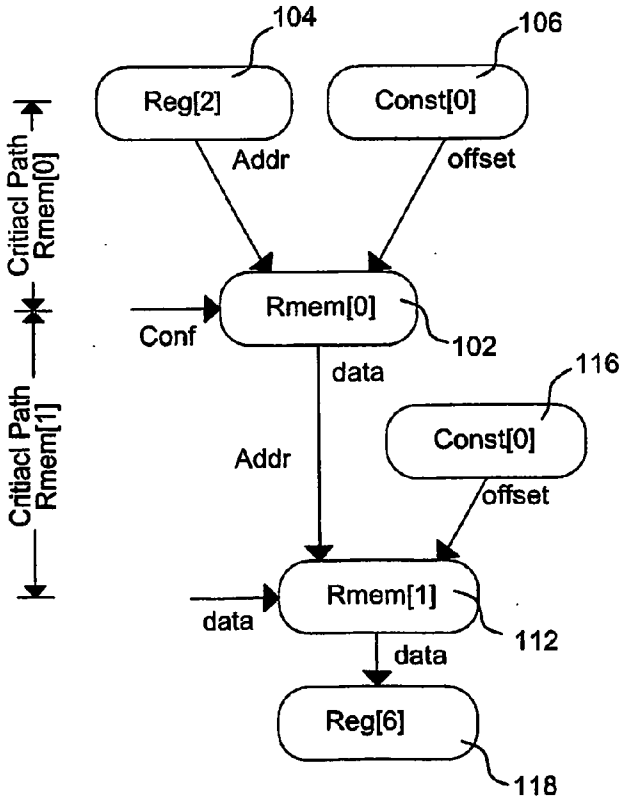

104
106
Reg[2]
Const[0]
Addr
offset
Critiacl Path Rmem[0]
Conf
Rmem[0]
102
data
Reg[3]
108
114
116
Reg[5]
Const[0]
Addr
offset
Critiacl Path Rmem[1]
Conf
Rmem[1]
112
data
Reg[6]
118

Fig. 10


104
106
Reg[2]
Const[0]
Addr
offset
Critiacl Path Rmem[0]
Conf
Rmem[0]
102
data
116
Const[0]
Addr
offset
Critiacl Path Rmem[1]
data
Rmem[1]
112
data
Reg[6]
118

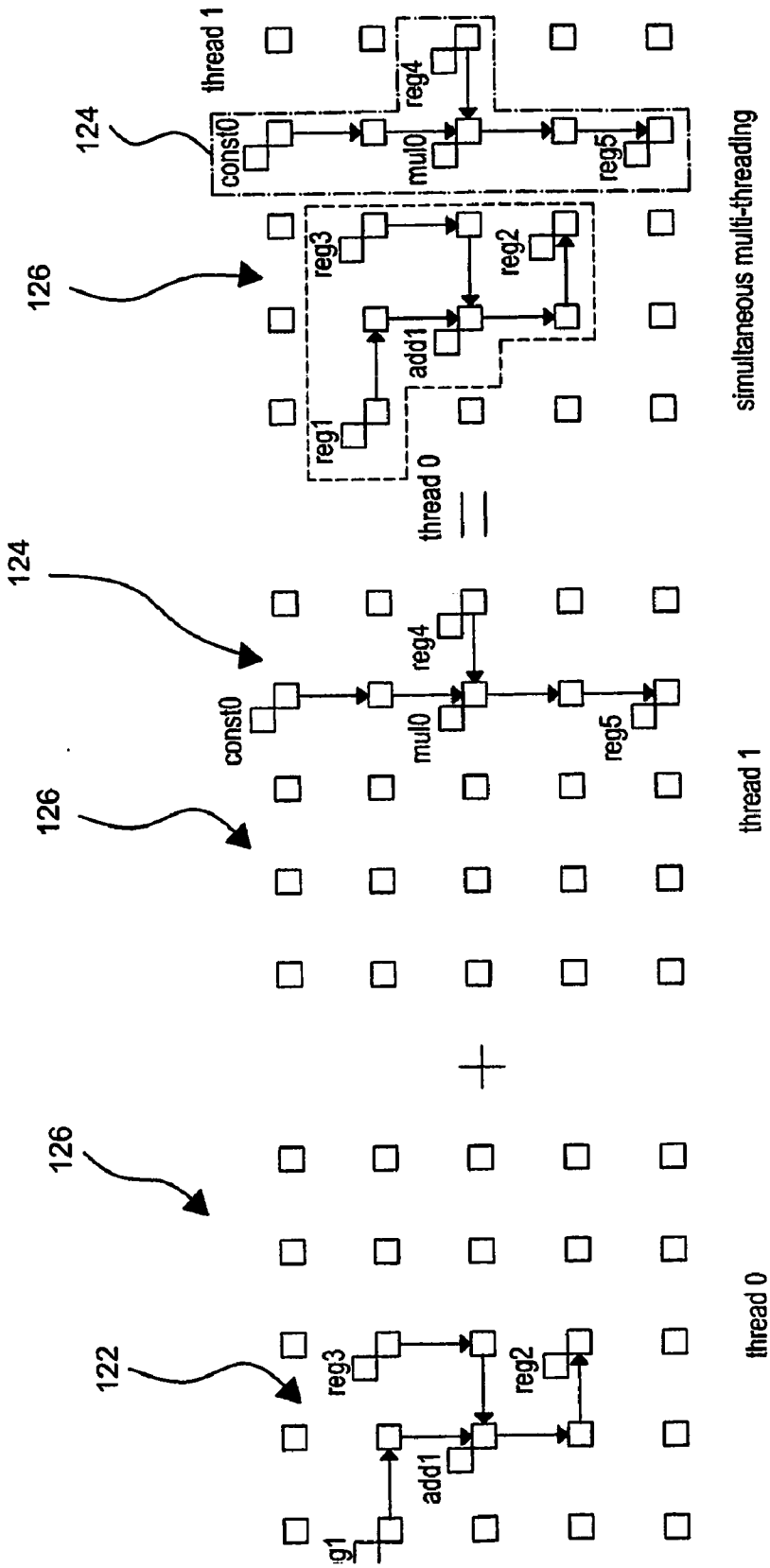

thread 1
124
const0
reg4
mul0
reg5
126
reg3
reg2
add1
reg1
thread 0
simultaneous multi-threading
124
const0
reg4
mul0
reg5
thread 1
126
126
122
reg3
reg2
add1
g1
thread 0

Fig. 12

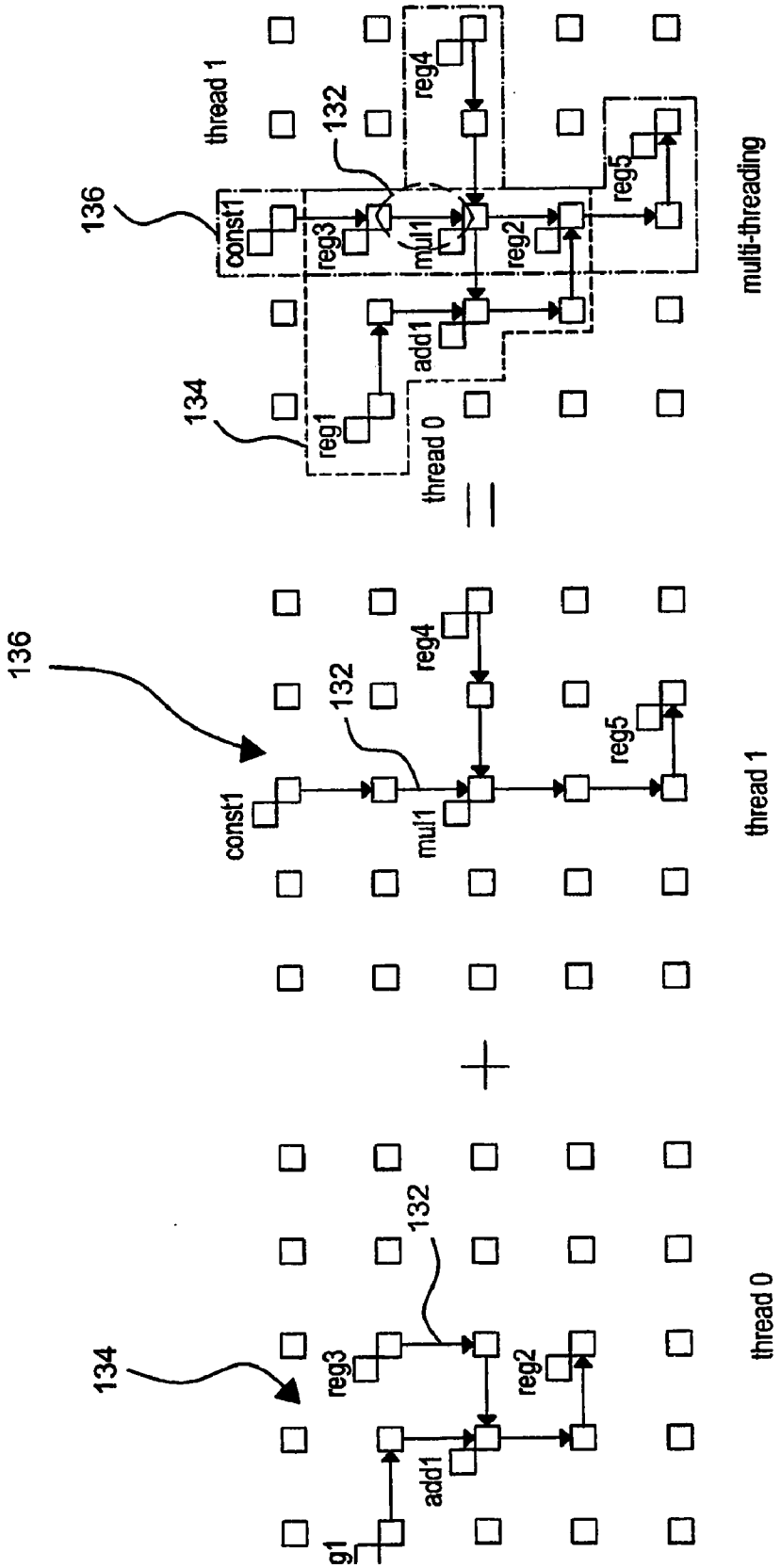

136
thread 1
132
reg4
reg5
const1
reg3
mul1
reg2
add1
134
reg1
thread 0
multi-threading
136
132
reg4
reg5
const1
mul1
thread 1
134
132
reg3
reg2
add1
g1
thread 0

Fig. 13

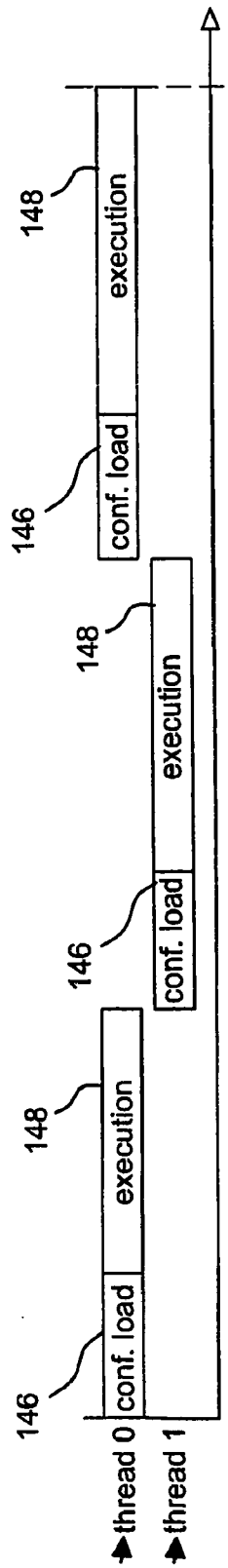


Fig. 14a

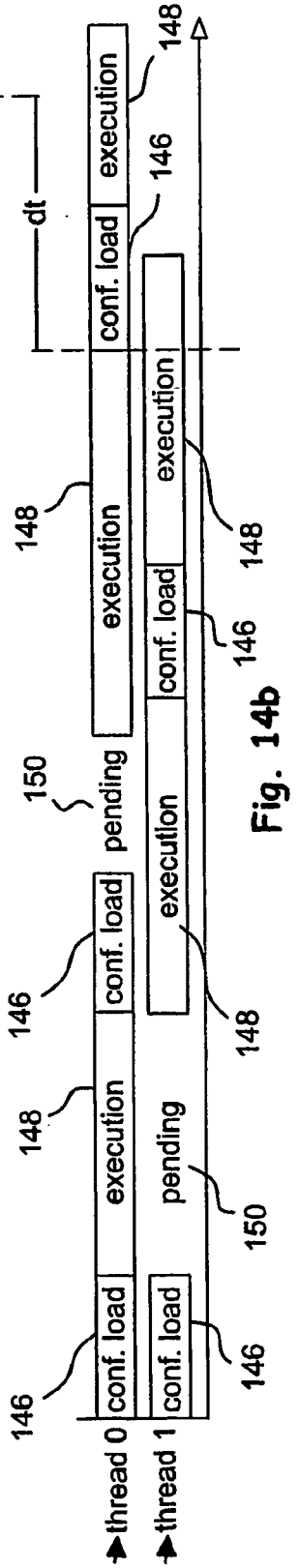


Fig. 14b

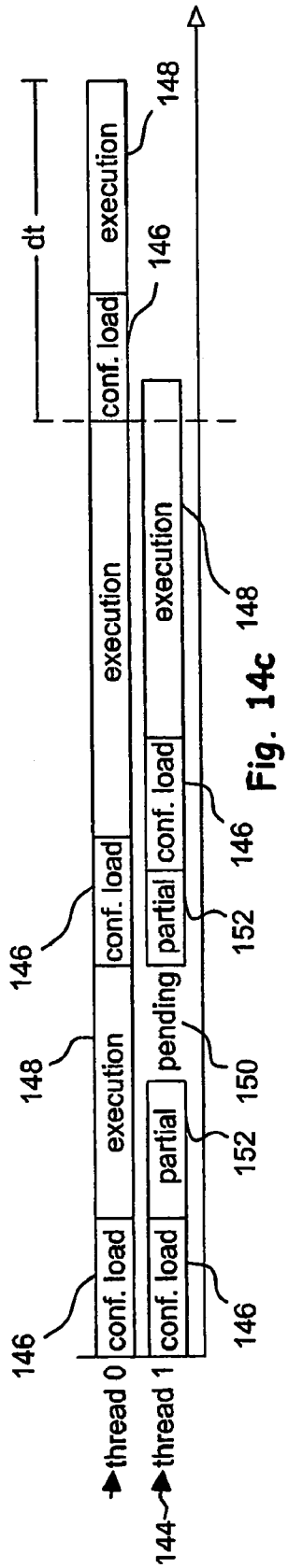
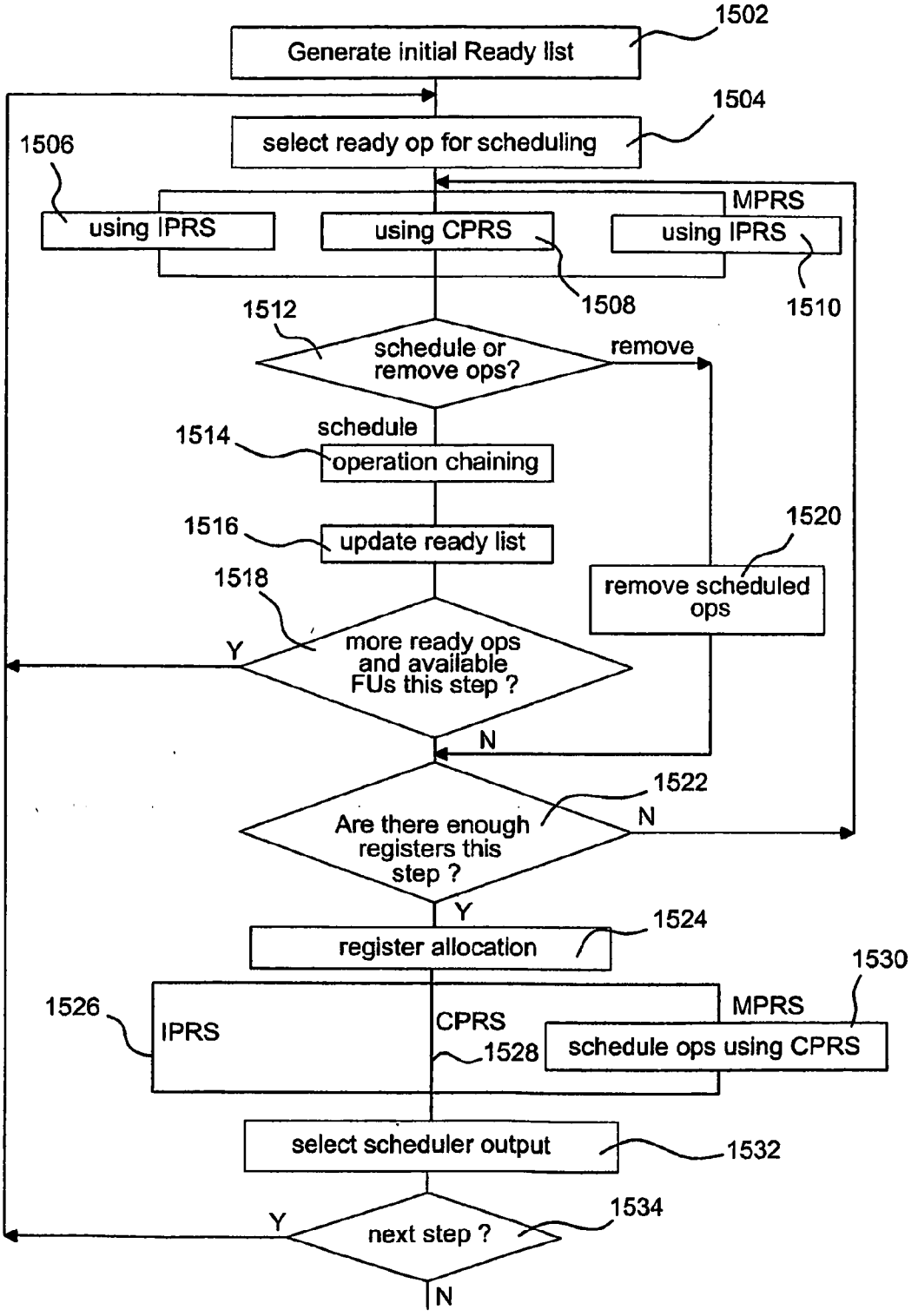


Fig. 14c


1502
Generate initial Ready list
1504
select ready op for scheduling
1506
using IPRS
using CPRS
MPRS
using IPRS
1508
1510
1512
schedule or remove ops?
remove
schedule
1514
operation chaining
1516
update ready list
1520
remove scheduled ops
1518
more ready ops and available FUs this step ?
Y
N
1522
Are there enough registers this step ?
N
Y
1524
register allocation
1530
1526
IPRS
CPRS
MPRS
1528
schedule ops using CPRS
select scheduler output
1532
1534
Y
next step ?
N

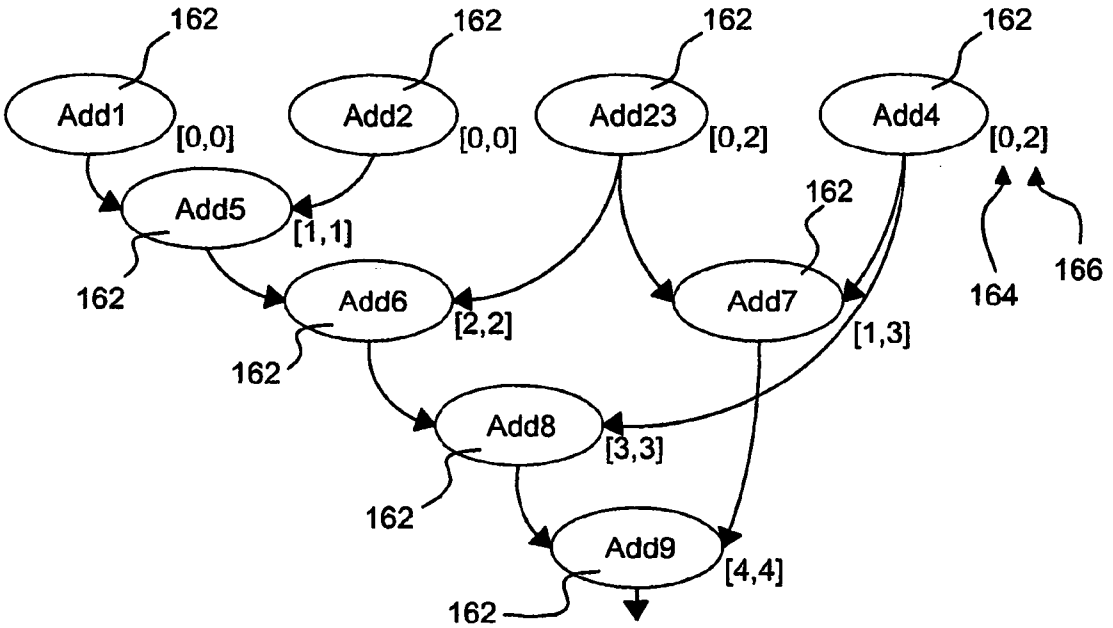

162
Add1
[0,0]
162
Add2
[0,0]
162
Add23
[0,2]
162
Add4
[0,2]
Add5
[1,1]
162
162
Add6
[2,2]
162
Add7
[1,3]
164
166
Add8
[3,3]
162
Add9
[4,4]
162

Fig. 16

RECONFIGURABLE INSTRUCTION CELL ARRAY

[0001] This invention relates to computer processors and in particular to reconfigurable processors.

[0002] The need for new hardware architectures for future semiconductor devices was recognised several years ago as current architectures will not be able to cope with future requirements in data processing and throughput. Typical examples are mobile devices for next-generation networks where a high amount of audio and video data will need to be processed; this adds extra demands on the performance of processors in order to maintain a high throughput at the cost of efficiency in terms of power consumption and silicon area. The hardware also needs to provide high adaptability to upcoming and changing standards.

[0003] Re-programmability and flexibility are key factors in reducing design costs as they permit post-fabrication changes to the system. Having this flexibility programmable through a high-level program description, such as standard C/C++, is important in reducing the design-cycle and in allowing realistic implementations of big systems.

[0004] Current established solutions include Field Programmable Gate Arrays (FPGAs), Digital Signal Processors (DSPs) and Application-Specific Integrated Circuit (ASIC) architectures. However, they all suffer from drawbacks such as very high power consumption in FPGAs, high non-recurrent-engineering costs and low flexibility after fabrication in ASICs and low throughput in DSPs. Very Long Instruction Word (VLIW) DSP architectures offer advantages in parallel processing. However, they are easily restricted by the limited amount of Instruction Level Parallelism (ILP) found in typical programs. Many solutions have been proposed by the research community mainly based on the high-performance offered by reconfigurable computing in terms of flexibility and high amount of parallel processing. Even though these solutions achieve remarkable results in computing-power and flexibility, they either do not provide enough power savings or are too difficult to program.

[0005] Several architectures try to combine a RISC processor with a coarse-grain reconfigurable array like MorphoSys (H. Singh et al. "Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications", IEEE Trans. on Comp., 49(5):465-481, May 2000) and SMeXPP (V. Baumgarte et al., "PACT XAPP-A self-reconfigurable data processing architecture", Proc., ERSA01), however these suffer from difficulties in programming the arrays due to the use of specialised custom low-level languages. They also suffer from other limitations due to the loose coupling between the conventional processors and the arrays, like difficulties in programming and large amounts of data-transfers between the array and processor. The ADRES architecture (Bingfen Mei et al., "ADRES: An Architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix", Proc, FPL'03) closely couples the processor and reconfigurable fabric by sharing the memory and register-file, which simplifies the programming model through the use of a high-level C language. The RaPiD architecture (C. Ebeling et al, "Implementing an OFDM Receiver on the RaPiD Reconfigurable Architecture", Proc., FPL'03) can be programmed using the high-level RaPiD-C language but this language is not compatible with ANSI-C and requires manual scheduling of parallel operations. The approach in the Chameleon/Montium coprocessor (Paul M. Heysters, et al., "Balancing between Energy-Efficiency, Flexibility and Performance", Proc. ERSA'03) and in Elixent's (Bristol, UK) D-Fabrix™ is to interconnect several ALUs to build datapaths using a special C compiler so that time-consuming computations are speeded up. In the case of Elixent™, programming is achieved through Handel-C, which is more like HDL as the designer has to state the parallelism in the code.

[0006] Therefore, none of the proposed reconfigurable hardware infrastructures have become widely adopted since they do not adequately address the need to reduce cost, have a short design time, have low power consumption and high performance.

[0007] In programming environments, threads are uncorrelated, independent tasks working in "parallel", ideally on separate set of data. Throughout the execution of a multi-threaded program there is no pre-defined or known order in which the threads are executed or synchronised with each other. If a known pattern could be constructed from analysing the program then it would be questionable to why one should implement this as a multi-thread (MT) program.

[0008] A multi-threaded program relies on dynamic information which is not known at compile time. Because of that it is extremely difficult to statically analyse the execution flow of a multi-threaded program and make allocation decisions that will satisfy all active supported threads. Each thread is mapped independently without any prior knowledge of other active threads.

[0009] Traditionally multi-threading is associated with the use of completely isolated execution cores. Conventionally MT on a single core can be achieved by time-slicing the hardware resources, often referred to as temporal multi-threading. With the appearance of superscalar computers, VLIW (Very Long Instruction Word) and reconfigurable computing machines, true MT is possible even in a single core. Inherently parallel architectures rely on the use of a large number of computational resources and/or interconnection resources (in the case of reconfigurable computing). Keeping these resources busy is a challenging task. Most parallel architectures suffer from low core utilisation. A known approach to reduce this effect is to employ Simultaneous Multi-Threading (SMT) in a single core. SMT usually comes as an extension to the execution flow control mechanism of a core, allowing sharing most of the core's resources by more than one concurrent thread. The realisation of SMT is very implementation dependent but a general overview for a given class of devices can be defined.

[0010] To achieve resource sharing, SMT requires some duplication of key components of the architecture. These typically include:

[0011] 1. the flow control mechanism

[0012] 2. the architecture state

[0013] Other than achieving higher core utilisation, SMT can hide cache latencies and fetch and decode latencies by using this time in performing useful operations. A stall condition for one thread can still leave other threads running and keeping the resources busy.

[0014] It would be desirable to use a high level language for describing the entire application running on a reconfigurable processor. However, traditional software is sequential in nature. As these high level languages have been developed for conventional CPUs, suitable scheduling algorithms are for sequential code. This poses a challenging task of taking an otherwise sequential code to extract and exploit the available

parallelism. The design flow to automate the implementation of algorithms from a high abstraction level to run on reconfigurable computing system requires an efficient scheduling algorithm that can translate general purpose software code onto the highly parallel reconfigurable device thereby maximising the resource utilisation. Traditional list scheduling can not be directly used on an instruction cell based architecture because:

[0015] 1) It does not deal with dependent instructions' parallelism;

[0016] 2) It does not consider Register allocation in the scheduling algorithm;

[0017] 3) It does not take the time effects of reconfigurable function unit and routing interconnection delay into account, which can change the data path delay in reconfigurable devices.

[0018] It is an object of an aspect of the present invention to provide a processor having reduced cost, short design time, low power consumption and high performance.

[0019] According to a first aspect of the present invention, there is provided a processor for executing program instructions having datapaths of both dependent and independent program instructions, the processor comprising:

[0020] an interconnection network;

[0021] a heterogeneous plurality of instruction cells each connected to the interconnection network;

[0022] a decoding module adapted to receive configuration instructions, each instruction encoding the mapping of at least one of a datapath of dependent program instructions and a datapath of independent program instructions to a circuit of the instruction cells and further adapted to decode a configuration instruction and configure at least some of the interconnection network and instruction cells, thereby mapping the datapath to the circuit of the instruction cells and executing the program instructions.

[0023] Preferably the decoding module is adapted to configure at least some of the interconnection network and instruction cells by connecting at least some of the instruction cells in series through the interconnection network.

[0024] Preferably the decoding module is further adapted to receive configuration instructions encoding the mapping the datapaths of a plurality of program threads to a corresponding plurality of independent circuits of the instruction cells and further adapted to decode a configuration instruction and configure at least some of the interconnection network and instruction cells, thereby mapping the datapaths of the plurality of program threads to the corresponding plurality of circuits of the instruction cells and contemporaneously executing the program threads independently of each other.

[0025] Preferably the processor further comprises a clock module adapted to provide a clock signal having clock cycles and the decoding module is operable to decode the configuration instruction so as to configure at least some of the interconnection network and instruction cells each clock cycle.

[0026] Preferably the clock module is adapted to provide a variable clock cycle.

[0027] Preferably, the clock module is adapted to provide an enable signal.

[0028] Preferably the enable signal is provided to an enable input of an instruction cell.

[0029] Preferably the clock module is adapted to provide a plurality of clock signals.

[0030] Preferably at least some of the plurality of clock signals are separately provided to a plurality of instruction cells.

[0031] Preferably the decoding module is further adapted to decode a configuration instruction so as to configure a clock signal.

[0032] Preferably the decoding module is further adapted to decode a configuration instruction so as to configure the clock cycle.

[0033] Preferably the active period of the clock signal is configured.

[0034] Preferably the duty cycle of the clock signal is configured.

[0035] Preferably the processor further comprises a program counter and an interface to a configuration memory, the configuration memory being adapted to store the configuration instructions, and at least one of the instruction cells comprises an execution flow control instruction cell adapted to manage the program counter and the interface so as to provide one of the configuration instructions to the decoding module each clock cycle.

[0036] Preferably the enable signal is provided to an enable input of the execution flow control instruction cell.

[0037] Preferably at least one of the instruction cells comprises a register instruction cell operable to change its output each clock cycle.

[0038] Preferably the enable signal is provided to an enable input of the register instruction cell.

[0039] Preferably the enable signal is provided both to an enable input of the execution flow control instruction cell and to an enable input of the register instruction cell.

[0040] Preferably the processor further comprises at least one input/output port and at least one of the instruction cells comprises an input/output register instruction cell adapted to access the at least one input/output port.

[0041] Preferably the processor further comprises at least one stack and at least one of the instruction cells comprises a stack register instruction cell adapted to access the at least one stack.

[0042] Preferably at least one of the instruction cells comprises a memory instruction cell adapted to access a data memory location.

[0043] Preferably the data memory location comprises a multi-port memory location.

[0044] Preferably a pair or more of memory instruction cells are adapted to access in the same clock cycle a pair or more of data memory locations clocked at a higher frequency than the frequency of the clock cycle.

[0045] Preferably the decoding module is further adapted to decode a configuration instruction so as to configure the clock signal dependent on the configuration of a clock signal decoded from a previous configuration instruction.

[0046] Preferably at least one of the instruction cells comprises a multiplex instruction cell adapted to route data between instruction cells.

[0047] Typically at least one of the instruction cells comprises a combinatorial logic instruction cell.

[0048] Preferably the function of an instruction cell corresponds to a program instruction.

[0049] Preferably the circuit of instruction cells comprises a set of instruction cells including one execution flow control instruction cell.

[0050] According to a second aspect of the present invention, there is provided a compiler for generating configuration instructions for the processor of the first aspect, the compiler comprising:

[0051] a front end module adapted to receive the program instructions and to identify the datapaths of the program instructions; and

[0052] a scheduling module adapted to map the datapaths of both dependent and independent program instructions as circuits of the instruction cells; and

[0053] a mapping module adapted to encode the configuration of circuits of instruction cells as configuration instructions and adapted to output the configuration instructions.

[0054] Preferably the program instructions comprise assembler code output from a high level compiler.

[0055] Alternatively the program instructions comprise source code of a high level language.

[0056] Preferably the scheduling module is further adapted to schedule the program instructions according to their interdependency.

[0057] Preferably at least one of the front end module and scheduling module are further adapted to map access to registers for the storage of temporary results into interconnections between instruction cells.

[0058] Preferably the scheduling module is further adapted to map a datapath of the program instructions that will be executed in a single clock cycle.

[0059] Preferably the scheduling module is further adapted to map the datapaths of a plurality of program threads as a corresponding plurality of contemporaneous circuits of the instruction cells.

[0060] Preferably the scheduling module is further adapted to map the datapaths using operation chaining.

[0061] Preferably the scheduling module is further adapted to use operation chaining while scheduling a single clock cycle.

[0062] Preferably the scheduling module is further adapted to use operation chaining while processing each ready list of a plurality of ready lists.

[0063] Preferably the scheduling module is further adapted to map the datapaths using register allocation.

[0064] Preferably the scheduling module is further adapted to use register allocation while scheduling a single clock cycle.

[0065] Preferably the scheduling module is further adapted to use register allocation while processing each ready list of a plurality of ready lists.

[0066] Preferably the scheduling module is further adapted to map the datapaths using a plurality of scheduling algorithms.

[0067] Preferably the scheduling module is further adapted to map the datapaths by using the output selected from one of the plurality of scheduling algorithms.

[0068] According to a third aspect of the present invention, there is provided a method of executing program instructions, the method including the steps of:

[0069] identifying datapaths in program instructions having datapaths of both dependent and independent instructions; and

[0070] configuring at least some of an interconnection network and a heterogeneous plurality of instruction cells, wherein each instruction cell is connected to the interconnection network, by connecting at least some of the instruction cells in series through the interconnection network thereby mapping the datapaths of both dependent and independent program instructions and executing the program instructions.

[0071] Preferably the method further includes the step of providing a clock signal having clock cycles to at least one of the instruction cells, wherein the step of configuring the instruction cells is performed each clock cycle.

[0072] Preferably the step of configuring further includes the step of configuring the subsequent clock cycle.

[0073] Preferably the active period of the clock cycle is configured.

[0074] Preferably the duty cycle of the clock signal is configured.

[0075] Preferably the step of configuring at least some of the interconnection network and the instruction cells includes the steps of:

[0076] mapping the datapaths of both dependent and independent program instructions as circuits of the instruction cells;

[0077] encoding the configuration of the circuits of instruction cells as configuration instructions; and

[0078] decoding the configuration instructions so as to configure the at least some of the interconnection network and the instruction cells.

[0079] Preferably the step of mapping includes the step of scheduling the program instructions according to their interdependency.

[0080] Preferably the step of mapping includes the step of mapping access to registers for the storage of temporary results into interconnections between instruction cells.

[0081] Preferably the step of mapping includes the step of mapping a datapath of the program instructions that will be executed in a single clock cycle.

[0082] According to a fourth aspect of the present invention, there is provided a computer program comprising program instructions, which, when loaded into a computer, constitute the compiler of the second aspect.

[0083] Preferably the computer program of the fourth aspect is embodied on a storage medium, stored in a computer memory or carried on an electrical or optical carrier signal.

[0084] According to a fifth aspect of the present invention, there is provided a computer program comprising program instructions for causing a computer to perform the method of the third aspect.

[0085] Preferably the computer program of the fifth aspect is embodied on a storage medium, stored in a computer memory or carried on an electrical or optical carrier signal.

[0086] An embodiment of the present invention will now be described with reference to the accompanying Figures, in which:

[0087] FIG. **1** illustrates, in schematic form, a processor architecture in accordance with an embodiment of the present invention;

[0088] FIG. **2** illustrates, in schematic form, examples of (a) island style and (b) multiplexer based interconnects between instruction cells;

[0089] FIG. **3** illustrates a block of C code, VLIW DSP code and sequential assembler code (ASM);

[0090] FIG. **4** illustrates, in schematic form, execution of the assembler instructions of FIG. **3** according to an embodiment of the present invention;

[0091] FIG. **5** illustrates, in schematic form, the programming flow according to an embodiment of the present invention;

[0092] FIG. **6** illustrates, in schematic form, the design methodology according to an embodiment of the present invention;

[0093] FIG. **7** illustrates, in schematic form, a comparison of instruction scheduling between VLIW DSP and that of an embodiment of the present invention;

[0094] FIG. **8** illustrates, in schematic form, an advanced Reconfiguration Rate Controller unit;

[0095] FIG. **9** illustrates, in schematic form, an example construction of the configuration bits for the advanced Reconfiguration Rate Controller;

[0096] FIG. **10** illustrates, in schematic form, independent memory instruction cell operation;

[0097] FIG. **11** illustrates, in schematic form, dependent memory instruction cell operation;

[0098] FIG. **12** illustrates, in schematic form, simultaneous multi-threading of non-conflicting threads;

[0099] FIG. **13** illustrates, in schematic form, simultaneous multi-threading of conflicting threads having a common interconnection resource request;

[0100] FIG. **14***a* illustrates, in schematic form, the execution timing diagram for two threads running under a temporal multi-threading scenario;

[0101] FIG. **14***b* illustrates, in schematic form, the execution timing diagram for two threads running under a simultaneous multi-threading scenario;

[0102] FIG. **14***c* illustrates, in schematic form, the execution timing diagram for two threads running under a simultaneous multi-threading scenario with support for partial execution;

[0103] FIG. **15** illustrates, in schematic form, a flowchart of the operation Chaining Reconfigurable Scheduling process; and

[0104] FIG. **16** illustrates, in schematic form, an input data control flow graph.

[0105] An embodiment of the present invention, referred to herein as the Reconfigurable Instruction Cell Array (RICA), will be described.

[0106] With reference to FIG. **1**, the core elements **1** of the RICA architecture are the Instruction Cells (ICs) **2** interconnected together through a network of programmable switches **4** to allow the creation of datapaths. In a similar way to a CPU architecture, in this embodiment, the configuration of the ICs **2** and interconnects **4** is changeable on every cycle to execute different blocks of instructions. As shown in FIG. **1**, RICA is similar to a Harvard Architecture CPU where the program (configuration) memory **6** is separate from the data memory **8**. In the case of RICA, the processing datapath is a reconfigurable core of interconnectable ICs and the configuration memory **6** contains the configuration instructions **10** (i.e. bits) that via the decode module **11** control both the ICs and the switches inside interconnects. The interface with the data memory **8** is provided by the MEM cells **12**. A number of these cells are available to allow effectively simultaneous (time multiplexed) read and write from multiple memory locations during the same clock cycle. This can be achieved by using multiple ports on the data-memory and by clocking it at a higher speed than the reconfigurable core. Furthermore, some special REG ICs **14** may be mapped to I/O ports **16** to allow interfacing with the external environment.

[0107] The characteristics of the reconfigurable RICA core **1** are fully customizable at design time and can be set according to the application's requirements. This includes options such as the bitwidth of the system, which, in this embodiment, can be set to anything between 4-bits and 64-bits, and the flexibility of the array, which is set by the choice of ICs and interconnects deployed. These parameters also affect the extent of parallelism that can be achieved and device characteristics such as area, maximum throughput and power consumption. Once a chip containing a RICA core has been fabricated, the system can be easily reprogrammed to execute any code in a similar way to a General Purpose Processor (GPP).

[0108] The IC array in the RICA is heterogeneous and each cell is limited to a small number of operations for example:

[0109] ADD—Addition, Subtraction

[0110] MUL—Multiplication (Signed and Unsigned)

[0111] DIV—Divisions (Signed and Unsigned)

[0112] REG—Registers

[0113] I/O REG—Register with access to external I/O ports

[0114] MEM—Read/Write from Data Memory

[0115] SHIFT—Shifting operation

[0116] LOGIC—Logic operation (XOR, AND, OR, etc.)

[0117] COMP—Data comparison

[0118] JUMP—Branches (and sequencer functionality)

[0119] This allows an increase in the overall cell count in the array to do more parallel computations, since the overhead of adding such small cells is merely related to the extra area occupied by the interconnects. The use of heterogeneous cells also permits tailoring the array to the application domain by adding extra ICs for frequent operations. Each IC may have one instruction mapped to it. In a similar way to assembly instructions, cells may have two inputs and one output—this allows creating a more efficient interconnects structure and reduces the number of configuration bits needed. The instruction cells developed for this embodiment support the same instruction sets found in GPPs like the OpenRISC and ARN7™ (Arm Ltd, Cambridge, UK) although instruction cells are not limited to supporting instruction sets from these processors. Hence, with this arrangement the RICA can be made binary compatible with any GPP/DSP system.

[0120] As shown above registers **18** may be defined as standard instruction cells. With this arrangement the register memory elements can be distributed in the array in such a way to operate independently, which is essential to allow a high degree of parallel processing. To program the RICA array the assembly code of a program is sliced into blocks of instructions that are executed in a single variable length clock cycle. Typically, these instructions, that were originally generated for a sequential GPP, would include access to registers for the temporary storage of intermediate results. In the case of the RICA architecture, these read/write operations are simply transformed into wires, which gives a greater efficiency in register use. By using this arrangement of registers the RICA offers a programmable degree of pipelining the operations and hence it easily permits breaking up long combinatorial computations into several variable length clock cycles. Having distributed registers enables allocation of a stack to each register, and hence providing a distributed stack scheme.

[0121] Special ICs include the execution flow control or JUMP cell **20** which is responsible for managing the program counter **22** and the interface to the configuration memory **6** in a similar way to the instruction controller found in CPUs.

[0122] Another special component is the Reconfiguration Rate Controller (RRC) **24** clock module. The RRC takes a signal from a clock generator **26** and generates an Enable signal that is routed to the enable inputs of the register ICs **18** and the execution flow control IC **20** so as to provide them with a variable clock cycle. The "clock gating" may be implementation specific and there are a number of other well-known techniques (such as asynchronous enable signals) that may be used. In usual CPUs the highest clock frequency at which the processor can be clocked is determined by the longest possible delays in the programmable datapath. For example, if the CPU has a multiplier (which takes a much longer time to execute than operations like addition), then the highest clock frequency has to provide enough time for it to operate. The problem is that if such a clock is used then the processor might end up with instruction cycles where only an adder is used but where the processor would be waiting for a longer time than needed, which limits the overall maximum throughput. In conventional CPUs this problem has been solved by making the CPU clocked at a higher frequency than the one required by the multiplier and at the same time making the multiplier pipelined, and hence requiring multiple cycles to execute.

[0123] In the RICA architecture a similar problem is encountered since the high flexibility provided allows creating data-paths with many levels of calculations and hence many possible delay requirements. If the RICA was to be clocked at the highest frequency dictated by the longest data-path, then there would be a restriction on the maximum achievable throughput.

[0124] The amount of clock cycles the RRC waits for before generating the Enable signal is configurable by the decoder **11** as part of the array's configuration instruction **10**. By combining this with a clock gating technique on the registers and program counter (via the execution flow control cell) it practically achieves variable clock cycles which are programmable as part of the software.

[0125] A further special cell is a multiplexer instruction cell that provides a conditional combinatorial path. By providing a cell that contains a hardwired comparator and a multiplexer, the present invention can provide implementation of the conditional moves identified by the compiler as simple multiplexers. Furthermore, when embodied as RICA, the present invention can also provide implementation of parallel multiple execution datapaths. Such a spanning tree is useful in conditional operations to increase the level of parallelism in the execution, and hence reduce the time required to finish the operation.

[0126] FIG. **2** shows examples of (a) island style and (b) multiplexer based interconnects between instruction cells. With reference to FIG. **2**, the programmable switches **21** perform directional connections **23** between the output and input ports of the ICs **25**. The design of the interconnects may take into account rules, for example that each IC has only one output and two inputs and that in no case will the output of an IC be looped back to its inputs (to avoid combinatorial loops). Different known solutions are available for the circuit design and topology of the switches, such as the island-style mesh found in normal FPGAs as shown in FIG. **2***a*, or the multiplexer-based crossbar as shown in FIG. **2***b*.

[0127] A feature of the RICA architecture is that both the programmable cells and their programmable interconnects can be dynamically reconfigured every clock cycle. The number and type of these cells are parameterisable upon application and implemented on a UMC 0.13 μm CMOS technology library. The reconfigurable fabric of RICA is shown in FIG. **2***a*. The RICA architecture consists of distinct 32 bit (but parameterisable, therefore not limited to 32 bit) programmable function units, general purpose registers, memory cells and an island-style mesh reconfigurable interconnects. The basic and core elements of the RICA architecture are the 32-bit programmable ICs which can be programmed to execute one operation similar to a CPU instruction. For example, an ADD cell can be programmed as a signed/unsigned addition or subtraction function. The ICs are interconnected through an island-style mesh architecture which allows operations chaining in the datapaths.

[0128] With reference to FIG. **3**, the sample C code **30** requires nineteen cycles **32** to execute on a typical sequential processor. However, if the same code is compiled for a VLIW DSPs, such as the TI C6203, then it would execute in fifteen cycles **34** since the VLIW architecture would try to concurrently execute up to 8 independent instructions (6 ALUs and 2 multipliers are available). If 4 simultaneous multiplications and 4 memory accesses were permitted, then the number of cycles would reduce to 8. This is still high taking into account the simplicity of the code and to what could be achieved by using hardwired solutions like FPGAs. The presence of dependent instructions prevents the compiler achieving further reduction in the number of clock cycles.

[0129] With reference to FIG. **4**, the configuration of instruction cells required to execute the previous C code **30** in only two cycles **40**, **42** is shown, if, for example, the architecture provided 14 operational elements that can execute 4×ADD, 4×RAM, 4×MUL and 2×REG simultaneously. This overcomes the limitation faced by VLIW processors and enables a higher degree of parallel processing. As shown in Cycle **1**, **40**, the longest delay path **44** is equivalent to two RAM accesses, one multiplication and some simple arithmetic operations. This is the case only if pseudo-simultaneous RAM accesses can be achieved within a reasonable time in a single variable clock cycle. This is not much longer than critical paths in typical DSPs when compared to how much more instructions are executed in the same cycle. Hence, an architecture that supports such an arrangement is able to achieve similar throughputs as VLIWs but at a lower power consumption, depending on the type of computation.

[0130] An aspect of the present invention provides an automatic tool flow for the hardware generation of RICA arrays, in this embodiment based on the tools available for generating domain-specific arrays. These tools take the characteristics of the required array and generate a synthesizable Register Transfer Language (RTL) definition of a RICA core that can be used in standard System On a Chip (SoC) software flow for verification, synthesis and layout.

[0131] With reference to FIG. **5**, which depicts the programming flow according to an embodiment of the present invention, the standard GNU C Compiler (gcc) is used to compile C/C++ code **50** into assembly format **52** describing which ICs need to be used, with the supposition that instructions are executed in sequence. This assembly is processed by the RICA Compiler to create a sequence of netlists **54** each containing a block of instructions that are executed in one clock cycle. Partitioning into netlists is performed after scheduling the instructions and analysing the dependencies between them. Dependent program instructions are connected serially and independent program instructions are connected in parallel in the netlists. The scheduling algorithm

takes into account IC resources, interconnects resources and timing constraints in the array; it tries to have the highest program throughput by ensuring that the maximum number of ICs is occupied and that at the same time the longest-path delay is reduced to a minimum.

[0132] Depending on the interconnects architecture, the routing part of the compiler can be based on the standard and established techniques such as the ones found in the place and route tool VPR. Furthermore, the compiler also performs crucial optimizations like removing the temporary registers generated by gcc by replacing them with simple wires. The compiler generates a sequence of configurations of instruction cells **56**, that are encoded as configuration instructions **6** for storing in the configuration memory **10**.

[0133] A complete tool chain is useful to allow rapid analysis of the design space for various algorithms and gain the most flexibility from of the hardware. Moreover it is advantageous for the tool chain to be suitable for integration with current design methodologies. FIG. **6** depicts the design methodology of the RICA system.

[0134] There are two levels of flexibility that can be applied to the RICA reconfigurable architecture.

[0135] Before fabrication—The high-level C/C++ code is compiled for a selected number of hardware resources. If the required performance determined by RTL simulation or through the RICA simulator is not met then the high level code can be modified or the mixture of cell resources changed. Adjusting the hardware resources allows the architecture to be tailored to the specific domain where it is to be utilised, thus saving power and resources. This stage has the highest level of flexibility.

[0136] After fabrication—The array resources are fixed. Consequentially, if the algorithm continues to change during or after the fabrication process then the code is simply recompiled for that fixed resources. As often similar operations are required for the updated code when compared to the previous code and a large percentage of the algorithm remains unchanged. This means the generated code is still fairly optimised for the given architecture.

[0137] The programming of the RICA architecture is achieved with a collection of different tools. The use of Instruction-Cells (IC) greatly simplifies the overall effort needed to map high-level programs to the RICA architecture. Having the arrays programmable in a similar way to standard CPUs allows the reuse of existing technologies available for processors, such as optimized compilers and language parsers.

[0138] With reference to FIG. **6**, in Stage **1**, the high-level compiler **604** takes the high-level code **602** and transforms it into an intermediate assembly language format **606**. This step is performed by a standard GNU C Compiler (gcc), which compiles C/C++ code (amongst other front-ends) and transforms it into assembly format describing which ICs need to be used. The output of gcc is written with the supposition that instructions are executed in sequence, i.e. one instruction per cycle; the compiler has no knowledge about the parallelism available on the RICA. However, as is typical for a conventional processor using a machine description definition, gcc is adjusted to take into account some limitations of the target RICA, like the maximum number of available registers and the available Instruction-Cells (which define the allowed operations). The compiler automatically deals with issues like registers allocation, stack handling and prologue/epilogue definitions: Moreover, it performs all the optimisations that will be useful later on like loop unrolling and loop peeling in conjunction with loop fusion.

[0139] In Stage **2**, (RICA compiler **608**), all the optimizations related to the RICA architecture are performed. The RICA compiler process takes the assembly output of gcc **606** and tries to create a sequence of netlists to represent the program. Each netlist contains a block of instructions that will be executed in a single clock cycle. The partitioning into netlists is performed after scheduling the instructions and analysing the dependencies between them. Dependent instructions are cascaded in the netlist while independent ones are running in parallel. The scheduling algorithm also takes into account resources and timing constraints in the array; it tries to have the highest program throughput by ensuring that the maximum number of ICs is occupied and that at the same time the longest-path delay is reduced to a minimum. Finally, it also performs crucial optimizations like removing the temporary registers generated by gcc by replacing them with simple wires.

[0140] Thus the RICA compiler performs several stages of operations to optimise the code for the architecture and generate the configuration instruction streams. It is needed to solve the problem of identifying and extracting the parallelism in the otherwise sequential assembly code.

[0141] The distributed registers in the array allow building data paths with different pipelining options of the same circuit. The change in pipelining plan can affect the maximum throughput or power consumption. Such flexibility in pipelining is usually not available in CPU architectures (it does exists in FPGAs), which creates an opportunity of making the compiler choose the best pipelining arrangement to improve the performance of the compiled C code. This can for example allow choosing a pipelining scheme with reduced power consumption which otherwise would not be possible.

[0142] Stage **3**, Place and Route, is actually begun inside the RICA compiler **608** of Stage **2** since it is part of the loop that performs partitioning and resources scheduling, as interconnect resources can affect the routability of the netlist and the timing of the longest path. However, it is described here as a separate step since we can reuse standard tools, such as the place and route tool VPR **624** to map the netlists **622** into the array **628**.

[0143] Stage **4** Configuration-memory: From the mapped netlists the compiler generates the required content of the configuration memory **26**, in this case the configuration RAM. Simulation and performance analysis can then be performed with conventional EDA tools or functionality of the code can also be tested using the Rica simulator.

[0144] Other tools can be plugged into the design flow to aid the design process such as the RICA compiler being able to output a format **618** suitable for GraphViz **620** that is open source graph visualisation software, to create a visual representation of the dependencies in the code and scheduled steps. As the combination and number of Cells in the RICA can vary depending on the designer's application domain and available silicon estate then a hardware generation utility **642** is used to generate the custom RTL code **644** based on the machine description.

[0145] There are several stages involved in the compilation process. The front end **610** of the RICA compiler is an instruction parser that deals with instructions such as addition, logic, div, shift, multiply. This stage performs lexical analysis of the

modified GCC assembler code and extracts the function, destination register, and source registers along with the cells configuration data. Each new instruction is added to the internal pattern, creating a complete netlist of the instructions based on having infinite amount of resources (register, adders, etc). Appropriate generation of wire names, allocation of registers are performed to unravel the register reuse to ensure that all the information that the scheduler needs are kept. Due to the fixed number of registers on processors compilers utilise techniques, such as graph colouring, to reuse them. The front end section of the RICA compiler also provides a suitable mechanism for the handling of labels from the modified gcc assembler code. This is because the generated scheduled assembly code needs to deal with the labels locations for when branch or jump instructions are met.

[0146] The RICA compiler implements a reconfigurable resource scheduling algorithm (RRS) based on the concepts of operation chaining list scheduling (CLS) algorithm with additional resource and time constraints considerations. It operates to process the available time instants in increasing order (starting at zero) and to schedule as many as possible instructions at a certain instant before moving to the next.

[0147] List scheduling (LS) is the most commonly used compiler scheduling technique for basic blocks. The main idea is to schedule as many operations as possible at a certain clock cycle before moving to the next clock cycle. This procedure continues until all operations have been scheduled. The major purpose of this way is to minimise the total execution time of the operations in the DFG. Of course, the precedence relations also need to be respected in list scheduling. All operations, whose predecessors in the DFG have completed their executions at a certain cycle t, are put in the so-called ready list R[t]. The ready list contains operations that are available for scheduling. If there are sufficient unoccupied resources at cycle t for all operations in R[t], these operations can be scheduled. However, if an appropriate resource for each operation in R[t] is not available, a choice has to be made on which operations will be scheduled at cycle t and which operations will be deferred to be scheduled at a later time. This choice is normally based on heuristics, each heuristic defining a specific type of list scheduling. By computing priories in a sophisticated way, they try to schedule the most critical cells first, and to assign them to the right resource. LS usually employ a priority vector to determine what tasks to consider first.

[0148] During the scheduling process **612**, the RRS scheduling algorithm uses a Ready List to be able to keep track of operations. Those operations that are available for scheduling are put in the Ready List. If there are free resources for operations in the Ready List, this operation can be scheduled after taking register allocation and routing information into consideration. However, if there are not enough free resources for operations in the Ready List, the scheduling algorithm must choose which operation to be scheduled at this cycle. The RRS scheduling algorithm selects the highest priority operation determined by mobility and parallelism in the Ready Lists as long as the operation meets resource and clock cycle constraints. Operations with higher priority have smaller mobility and higher parallelism. Mobility is defined as the length of schedule interval. Higher parallelism decides independent instruction to be scheduled first. The compiler technique called operation chaining is adopted in RRS because the available dependent parallelism present in RICA architecture. Since the scheduling and chaining of operations affects the register reuse, the scheduling algorithm also performs register assignment. The resource and time constraints are given in the machine description **616** where pre-scheduling estimated routing times are utilized and post-scheduling the routing time delays from VPR are used. The time constraints include not only the programmable computational unit delay but also the routing delays. A simple example of code before and after scheduling is now presented. The initial assembly code generated by gcc is:

1	Load	r1, X
2	Load	r2, Y
3	Mult	r3, r2, r1
4	Load	r4, A
5	Mult	r2, r4, r1
6	Add	r5, r2, r4
7	Mult	r1, r2, r5
8	Load	r3, B
9	Add	r7, r1, r3

[0149] It takes 9 cycles and 6 registers to implement the code. If the targeted RICA has 3 loads, 2 multipliers and 2 adds, the code generated by the scheduler targeted on RICA only takes 2 variable length clock cycles and 4 registers to finish the code. This scheduled assembly code is:

	Step 1:	
1	Load	wire1, X
2	Load	wire2, Y
3	Mult	r1, wire1, wire2
4	Load	wire3, A
5	Mult	wire4_r2, wire1, wire3
6	Add	r3, wire4, wire3
	Step 2:	
7	Mult	wire5, r2, r3
8	Load	wire6, B
9	Add	r4, wire5, wire6

[0150] The last stage of the RICA compiler is the mapper **614**, which allocates and maps the functional cells in the scheduled step's netlists to the hardware cells layout **622** and creates the appropriate bit streams of configuration instructions **626** for the given hardware architecture based on the resource allocation specifications. This configuration bit stream **626** consists of the configuration for the routing interconnects, which is generated through VPR **624** and the cell function configuration from the compiler. This output can be finally used to program the hardware or used by the simulator to analyse the system performance. The compiler also outputs a human readable form **630** of the configuration data.

[0151] The simulator **632** is an important aspect of the design environment by facilitating rapid exploration of the algorithm and resource allocation at a higher level of abstraction allows the extent of the tradeoffs to be determined before committing to time consuming RTL hardware simulation. The simulator takes the human readable .net file **630** along with simulation template file **634** containing code for each cell's functionality and performance analysis routines and generates a behaviour level C model **636** based on the RICA architecture for that given .net file. The C model is fed into a gcc compiler **638**. A feedback loop exists through the performance characteristics **640** output by the gcc compiler **638** and

the machine description **616** to take into account the routing delays after the netlist have been allocated to the hardware.

[0152] For each scheduled step, the simulator reorders the operations in accordance to their dependency in that step. All these reordered steps along with their connections to each other are then combined to form a set of datapaths representing the whole system. When this is compiled both the functionality can be assessed such as loops, register and memory at each step, Along with total executed steps. Other parameters gained are estimated power and area figures based on having model libraries of each instruction cell.

[0153] With reference to FIGS. **7***a* and **7***b*, The advantages of the present invention are demonstrated. RICA has the same flexibility of coarse-grain FPGA and programmability of DSP. Since it employs coarse-grained reconfigurable architecture, RICA has lower power consumption than generic fine-grained FPGA. VLIW DSP can only execute independent instructions in parallel while RICA can execute both independent and dependent assembly instructions in the same clock cycle because it allows connections in series. This is illustrated in FIGS. **7***a* and **7***b* where it is assumed that the hardware resources include 4 adders and 2 multipliers. With reference to FIG. **7***a*, VLIW processors can only executed independent instructions **702** thus it takes 5 clock cycles **704**. With reference to FIG. **7***b*, RICA can perform independent **706** and dependent **708** instructions under resource constrains so it only consumes two clock cycles **710**. Consequently, RICA allows higher throughput than VLIW DSP. RICA provides a higher throughput if running at the same 'frequency', or if there are the same delays of the computational elements inside both of them. RICA provides a lower maximum frequency than the VLIW DSP, hence potentially a lower maximum throughput for some control-intensive applications.

[0154] In addition, RICA adopts the advanced operation chaining technique to reduce registers and memory usage in order to reduce significantly power consumption. Operation chaining is a known technique that merges two operations one after another in extended clock cycle to reduce the register requirement for the application. Traditional high-level synthesis (HLS) is a translation process which involves taking a behavioural description into a register transfer level (RTL) structural description. Scheduling is a critical task in this process. Scheduling partitions the Control Data Flow Graph (CDFG) into time steps thereby generating a scheduled CDFG model. After the scheduling routine is performed, register allocation is used to minimise the registers in the design.

[0155] An embodiment of an aspect of the present invention will now be described that relates to an advanced Reconfiguration Rate Controller cell.

[0156] With reference to FIG. **8**, an advanced Reconfiguration Rate Controller (RRC) **802** generates separate Enable signals for the program counter **804**, memories **806**, **808** and registers **810** at the appropriate time so as to provide a plurality of clock signals. The amount of clock cycles **812** the RRC waits for before generating the Enable signals for the other parts of, the architecture is programmable as part of the array's configuration instruction **814**.

[0157] With reference to FIG. **9**, in this embodiment of the RRC instruction unit the unit's configuration bits are comprises several tag fields; the jump cell critical path tag **902**, the Rmem cells path tags **904-906** and the combination cells' critical path **908** (which determines the activation of the registers).

[0158] The combinational cells' critical path tag for controlling the registers operates in the same way as described above with reference to FIG. **1**, i.e. the tags determines when to clock the register cells.

[0159] If the jump (execution flow control) cell is not part of the step's longest critical path it is possible to start a pre-fetch and decode of the next step before the longest critical path in the current step finishes. In order to do this extra information is required to determine when the jump cells output value is valid. Consequently, in this example four bits **902** are allocated to the RRC unit's configuration. This corresponds to how many of RICA internal clock cycles it waits for i.e. if the critical path for the jump cell is 20 ns, the clock resolution is 2 ns, the jump configuration field is set to 10.

[0160] For each read interface memory cell provided in the RICA core, there is a corresponding read memory tag (e.g. **904**, **906**) allocated to the RRC configuration instruction. These tags are used to provide timing management to the memory cells thus ensuring data is only read at the appropriate time period during a step's computation.

[0161] With reference to FIG. **10**, the first part of the Rmem tag is used to define the delay time from the start of a step to when it should send the read interface memory instruction cells' **102**, **112** inputs (e.g. the address **104**, **114** and offset **106**, **116**) to memory **108**, **118**.

[0162] Each of these tags also has a bit to indicate the cascading status. When the cascade bit is set high, it means that the read memory is dependent on the previous read interface memory cells access and should only begin its timing delay at the end of the previous read interface memory cells delay time.

[0163] In the case illustrated by FIG. **10**, both Rmem cells cascade bit are set low and both start their delay time at the same time. In the case illustrated by FIG. **11**, however, Rmem [**1**] **112** has a cascade bit set high and will only start its delay after Rmem[**0**] **102** has finished its delay. Furthermore, the output of Rmem[**0**] **102** is mapped directly into a wire to the input of Rmem[**1**] **112**.

[0164] An embodiment of an aspect of the present invention will now be described that relates to Simultaneous Multi-Threading (SMT) on reconfigurable computing architectures.

[0165] This embodiment of the present invention provides SMT in the RICA reconfigurable computing architecture.

[0166] A set of computational units connected via a 2-dimensional multi-hop direct network is used here as an example to demonstrate the effect of simultaneous multi-threading on a single reconfigurable lattice. Other example interconnection strategies may include crossbars or segmented shared-buses.

[0167] Each thread is scheduled and mapped to the array independently to other threads that may coexist in the same program. In temporal multithreading, only one thread is active at any given time instance. In an SMT environment, these threads will occupy different parts of the array and can operate in parallel.

[0168] FIG. **12** shows two threads **122**, **124** that share the same reconfigurable lattice **126** without any conflicts. This is an ideal case which maximises the array utilisation. Attempts to enforce such thread isolation can be made at compile time by restricting one of the threads to one corner of the core while allocating the other one to the opposite side of the core. In most cases two concurrent threads would request one or more common resource at the same time. A form of interconnection resource "conflict" is shown in FIG. **13**. In this

example, the common resource **132** will be assigned to one of the two threads **134** and **136**, while the other one would need to wait.

[0169] A typical thread might have more than one independent data-path in a given configuration. Unless a strict synchronisation is imposed, each independent non-conflicting data-path can operate in parallel to the rest. This could result in a thread being executed partially, i.e. only some of its data-paths. The use of partial execution can further increase core utilisation and as a subsequence increase performance.

[0170] FIG. **14** shows the execution time diagram for two threads, $thread_0$ **142** and $thread_1$ **144** running under various scenarios, such as temporal MT (FIG. **14***a*), SMT (FIG. **14***b*), and SMT (FIG. **14***c*), with support for partial execution. In FIG. **14**, the configuration load **146** and execution **148** states are shown for each thread, as well as pending states **150** and partial execution **152** states.

[0171] An embodiment of the present invention will now be described that provides a new operation chaining reconfigurable scheduling algorithm (CRS) based on list scheduling, which maximizes instruction level parallelism available in distributed high performance instruction cell based reconfigurable systems. Unlike other typical scheduling methods, it uses placement and routing, register assignment and advanced operation chaining compilation techniques to generate higher performance scheduled code. Schedules using this approach achieve at least equivalent throughput to VLIW architectures but at much lower power consumption.

[0172] This embodiment of the present invention provides an efficient operation chaining scheduling algorithm targeted for reconfigurable systems. This Chaining Reconfigurable Scheduling (CRS) algorithm is based on the list scheduling algorithm and adopts the advanced operation chaining technique and considers the effects of register assignment, power consumption, placement and routing delay against the resource and time constraints. The CRS algorithm allows high program throughput and low power consumption by ensuring that the number of dependent and independent instruction cells (ICs) is maximised at each scheduled step and at the same time the total number of clock cycles of the longest-path delay is minimised.

[0173] Differing from the traditional HLS process, this embodiment of the present invention combines the scheduling, routing, instruction cells binding and register allocation together to suit the instruction cell-based architectures. This embodiment of the present invention provides a new efficient multi-objective scheduling optimisation to give both high throughput performance and low power for new reconfigurable architectures.

[0174] Traditional list scheduling can not be directly used on an instruction cell based architecture because:

[0175] 1) Efficient operation chaining is required to deal with dependent instructions' parallelism;

[0176] 2) Register allocation needs to be considered in the scheduling algorithm;

[0177] 3) The scheduling algorithm should also take the time effects of reconfigurable function unit and routing interconnection delay into account, which can change the data path delay in reconfigurable devices.

[0178] FIG. **15** shows a flowchart of an embodiment of the scheduling algorithm.

[0179] The scheduler generates **1502** a ready list and selects **1504** an operation from the list for scheduling. Three scheduling approaches are used to deal with the priority allocation mechanism. These are IPRS **1506**, CPRS **1508** and MPRS **1602**. In the MPRS case, IPRS is used at this step **1510** of the algorithm combined with CPRS after register allocation at step **1530**.

[0180] The part of the algorithm from **1506** to **1520** is used for either scheduling or removing operations. If scheduling is being performed **1512**, then operation chaining is performed **1514** and the ready list is updated **1516**. If **1518** more operations are available in the ready list and functional units are available at this step, the next operation is selected **1504** from the ready list.

[0181] If at step **1512**, the part of the algorithm from **1506** to **1520** is determined to be used for register removal, then the removal of scheduled operations is performed **1520** instead of steps **1514** to **1518**.

[0182] If **1522** there are not enough registers at this configuration step (or clock cycle) then the removal of operations is performed by looping back to steps **1506** to **1510** and, for example, a flag is set to indicate to the decision test **1512** that register removal is to be performed.

[0183] If at step **1522** there are enough registers, then register allocation **1524** is performed, followed by no action in the case of IPRS **1526** and CPRS **1528**, or followed by further scheduling of operations using CPRS in the case of MPRS at step **1530**.

[0184] Using the three approaches for scheduling, it is possible to schedule the whole or sections of the code using the most optimal technique to generate the configuration bits for the target RICA architecture. The optimal scheduler output is selected **1532** before the scheduler proceeds **1534** to the next configuration step.

[0185] In order to illustrate a comparison of the CRS algorithm with simple list scheduling, a further example of the CRS algorithm is given below. It consists of a simple list scheduling algorithm with the additional new lines marked with a @.

```
Method CRS Scheduling
Input:      Assembly Code representing operations to schedule,
              Space machine description (resource, clock
            cycle),
              Routing delay generated by VPR algorithm;
Output:     Fast and parallel Netlist
Begin
Step 1:     construct control flow graph (CFG)
Step 2:     construct data flow graph (DFG)
Step 3:     rename to eliminate anti/output dependences
Step 4:      assign priority to instructions based on cells
            flexibility
Step 5:     iteratively select & schedule instructions
Cycle =1;
Ready [Cycle] = Roots of DFG;
Ready [Cycle+1] =ϕ;
Scheduled [Cycle] =ϕ;
@Available_Register[Cycle]={the output registers but not
          used as input registers of basic block}
While (Ready Lists ≠ϕ) //All Ready[i] lists
{
  If (Ready [Cycle] =ϕ) then
  {
@       Initialise available hardware resource;
          Cycle = Cycle + 1;
  }
  While (Ready [Cycle] ≠ϕ) {
@         Remove an op from Ready in priority order and a
          series low power optimisation techniques are used
          to select an op
    If (∃ free issue units of type(x) on cycle
```

-continued

```
@          && operation can be chained in the same cycle) {
             S (op) = Cycle;
               F (op) = FinishTime (op);
               Scheduled [Cycle] = Scheduled [Cycle] ∪ {op};
               For (each successor s of op in DFG) {
                 If (s is ready) then
                     Ready [Cycle] = Ready [Cycle] ∪ s;
               }
      }
      else
               Ready [Cycle+1] = Ready [Cycle+1] ∪ {op}
  }
@If    (op uses register and each successor of op in
          Scheduled [Cycle])
@then {//release register
        Remove Register that saves the output of op and Put
          it in available register list}
@ N = the number of registers are needed in this clock
          cycle
@ A = the number of registers are available in this cycle
@ if (N<A)
@          Assign Register to save the output of op
@ else {
@            Remove some scheduled ops in terms of priority
      until N ≦A;
@       Assign Register to save the output of op;}
Θ Further scheduling using CPRS (this step is only used
      for MPRS)
@Calculate the longest critical path and variable clock
      cycles
}
@ Resource binding - using hamming distance.
End Method
```

[0186] As the design input of the scheduler is the assembly code or other intermediate representation, a dependence graph (DG) is generated by removing the registers to represent instruction dependences. Since a compiler for conventional processors implements an advanced register allocation algorithm to reuse the fixed number of physical registers, DG may have false data dependence called output dependence and anti-dependence.

[0187] An operation B is true dependent on operation A if B reads a value that A produces. It is also called read after write (RAW). An operation B is anti-dependent on operation A if B writes a value to a register that is read by A, thereby destroying the value read by A. It is also called write after read (WAR). An operation B is output dependent on operation A if both A and B write the value to the same register. It is also called write after write (WAW).

[0188] Register renaming is a technique for removing the false dependencies by changing the name of the register which causes the WAR/WAW conflict. However, this technique might cause register spilling. To overcome this problem, additional load and store instructions need to be used to store the value that is expelled because of the lack of registers. This will decrease performance and increase power consumption because of increased memory accesses. Since the function units in RICA are sequential chained in the same clock cycle, the operation chaining technique reduces registers and memory usage.

[0189] If the design entry in the CRS scheduling algorithm is the compiled and scheduled assembly code, the scheduler must handle the additional complexity such as removal of unnecessary registers. Register allocation is performed after scheduling in high-level synthesis. Since the scheduling and chaining of operations affect the register allocation, the CRS scheduling algorithm also considers the register assignment.

[0190] The control flow graph is a fundamental data structure that is used to generate the correct data dependence between two basic blocks and find the instruction parallelism for different basic blocks. Firstly, the code is partitioned into a set of basic blocks, which begin at a label or after jump instruction. Basic blocks consist of CFG's nodes. After that, the CFG's edges will be added from the current block to all possible target basic blocks of the branch. The input/output ports of CFG nodes are related with the input/output registers of basic block. The input registers save the values which have been written by another basic block and will be read in this basic block. The output registers save the values which are calculated by this basic block and will be read in other basic blocks. The available registers in the CLS scheduling only include the output registers because the input registers may be used by another basic block and should keep the same value.

[0191] Three methods are adopted in the CRS scheduling algorithm, and the output code with the minimal number of clock cycles (highest throughout) or lowest power consumption is selected as scheduler's output.

[0192] The first method is called the critical-path reconfigurable scheduling (CPRS). In this context, the longest path from a node to an output node of the DFG is called its critical path. The maximum of all critical path lengths gives a lower bound value of the total time necessary to execute the remaining part of the schedule. Operations with higher priority have smaller mobility that is defined as the length of schedule interval (mobility=as late as possible (ALAP)—as soon as possible (ASAP)). Therefore, in CPRS scheduling, nodes with the greatest critical-path lengths are selected to be scheduled at cycle t.

[0193] The second method is called independent instruction priority reconfigurable scheduling (IPRS). In IPRS, nodes are selected to be scheduled at cycle t if they minimally increase the critical path of this clock cycle compared to other available nodes.

[0194] Finally, the third method called mixed priority reconfigurable scheduling (MPRS) is also used in the CRS scheduler, which combines the CPRS and IPRS algorithm. In MPRS, nodes are firstly scheduled using IPRS method, and the CPRS algorithm is executed again after register allocation (A line marked with a □ in FIG. **1** is adopted only for MPRS). The CPRS algorithm will generate the minimal execution time for fixed clock system. Since the CRS algorithm adopts variable clock cycles, the CPRS scheduling method may result in longer execution time compared to the IPRS scheduling. As the CRS algorithm combines with register allocation, the finite amount of register may limit IPRS method. The MPRS scheduling may generate better scheduling than the CPRS and IPRS scheduling.

[0195] These scheduling techniques will be illustrated using the example shown in FIG. **16**. Assuming the architecture provides 5 operational elements that can execute 3×ADD and 2×REG simultaneously and the addition is executed in one clock cycle, the ASAP and ALAP times for each adder **162** are shown in square brackets, e.g. **164**, **166**, in FIG. **16**. Table 1 below compares three different CRS algorithm to see how they impact on the number of clock cycles needed to run the function of FIG. **16**.

TABLE 1

Comparisons of Schedulers (3 adders and 2 registers)

Execution	CPRS		IPRS		MPRS	
Step	Cycles	SC.	Cycles	SC.	Cycles	SC.
1	2	[1, 2, 5]	1	[1, 2]	2	[1, 2, 5]
2	2	[3, 6]	1	[5, 3]	2	[3, 6]
3	2	[4, 8, 7]	1	[6]	2	[4, 8, 7]
4	1	[9]	2	[4, 8, 7]	1	[9]
5			1	[9]		
ET	7		6		7	

[0196] In Tables 1 to 3, ET refers to the total clock cycles of execution times and SC refers to scheduled cells. Table 1 shows that the CPRS scheduling generates longer ET than the IPRS scheduling, and the same ET as the MPRS scheduling. However, the ET of the different algorithms is heavily dependent on machine description. Tables 2 and 3 below give the different scheduling results based on the different machine-description.

TABLE 2

Comparisons of Schedulers (4 adders and 2 registers)

Execution	CPRS		IPRS		MPRS	
Step	Cyc	SC.	Cyc	SC.	Cyc	SC.
1	2	[1, 2, 5, 3]	1	[1, 2]	1	[1, 2, 5, 3]
2	2	[6, 4, 8, 7]	2	[5, 3, 6]	1	[6, 4, 8, 7]
3	1	[9]	3	[4, 8, 7, 9]	2	[9]
ET	5		6		5	

TABLE 3

Comparisons of Schedulers (5 adders and 5 registers)

Execution	CPRS		IPRS		MPRS	
Step	Cyc	SC.	Cyc	SC.	Cyc	SC.
1	3	[1, 2, 5, 3, 6]	2	[1, 2, 3, 4, 5]	2	[1, 2, 3, 4, 5]
2	3	[4, 8, 7, 9]	3	[6, 7, 8, 9]	3	[6, 7, 8, 9]
ET	6		5		5	

[0197] As shown in Tables 1 and 2, the CPRS algorithm results in longer (Table 1) or shorter (Table 2) ET than the IPRS algorithm, which is caused by the number of registers. Otherwise, the IPRS algorithm generates the shorter ET time if there are enough registers (Table 3). As the functional resources only include adders in this example, the results using the MPRS algorithm is either the same as CPRS (Table 1-2) or the same as IPRS (Table 3). However, when applied to a greater variation in resource as in a reconfigurable instruction cell based machine, it generates different results. The compiler will schedule the input code using all methods and select an appropriate output code for their requirement.

[0198] For each basic block, a data flow graph (DFG), which represents data dependence among the number of operations, is used as an input to the CRS scheduling algorithm. One limitation of running a design on processors is the number of operations that can be executed in a single cycle. To maximise the number of operations executed in a single cycle, operation chaining and variable clock cycle are added to the CRS scheduling algorithm. It reduces the total number of registers usage, decreases the power consumption by reducing memory access, and increases throughput by variable clock cycles. When chaining an operation, scheduling algorithms must consider not only the operation's impact on the critical path but also the programmable routing delay. The scheduling algorithm in the tool flow includes pre-scheduling and post-scheduling. The pre-scheduling algorithm considers the operation and estimated routing time effects on the critical path. The Netlist after pre-scheduling will be fed into a standard placement and routing tool such as VPR. The corresponding routing information given by VPR tool or other routing tools is used to provide more accurate timing information to post-scheduling algorithm in order to generate the Netlist that met design requirement. The CRS algorithm is used in the first phase for performance optimization. After that, the resource binding is generated by using Hamming distance as our power cost model to estimate the transition activity in instruction cell configuration bus. Hamming distance is the number of bit differences between two binary strings. The resource binding scheme with the minimum Hamming distance is chosen to reduce the power consumption.

[0199] Benchmark tests are conducted using different CLS scheduling algorithms (CPRS, IPRS and MPRS) and may be targeted on the reconfigurable instruction cell based architecture of the present invention to demonstrate the performance of each CLS algorithm. Table 4 below shows the execution time and energy consumption where the benchmarks are running on the architecture at the same frequency (125 MHz) and the same hardware resources are used. The power consumption values for our reconfigurable architecture are obtained after post-layout simulation by the Synopsys PrimePower using UMC 0.13 μm technology.

TABLE 4

Comparison of Different Scheduling Methods

Benchmark	Method	Execution Times (us)	Energy (uJ) consumption
2D-DCT	CPRS	2.352	0.1309
	IPRS	2.192	0.108
	MPRS	2.192	0.112
FIR	CPRS	1.998	0.263
	IPRS	1.98	0.248
	MPRS	1.98	0.249
IIR	CPRS	0.194	0.015
	IPRS	0.186	0.0154
	MPRS	0.188	0.0156
Min-Error	CPRS	9.286	1.59
	IPRS	9.086	1.519
	MPRS	9.286	1.529
OFDM	CPRS	42.496	0.7980
	IPRS	43.224	0.8003
	MPRS	41.6	0.7905
Viterbi	CPRS	501.488	0.2296
	IPRS	452.162	0.225
	MPRS	452.1	0.227
Dhry-stone	CPRS	283.046	1.004
	IPRS	281.08	0.878
	MPRS	282.06	0.925

[0200] From Table 4, we can see that for all the benchmarks the CPRS, IPRS and MPRS algorithms achieve slightly different execution times and power consumption values. Here, scheduled code can be selected by the end user's design criteria, e.g. power, throughput. In order to provide the high throughput, the output code with the lowest execution time is selected. The optimal solution is heavily dependent on applications; it may be generated by CPRS, IPRS or MPRS algorithm. This has been illustrated in Table 4 below. In most benchmarks, the CPRS scheduling algorithm provides less register usage but however it has higher energy consumption compared to other scheduling algorithms. From scheduling and simulation analysis, the code generated by the CPRS scheduling algorithm has longer combinational logic path than other algorithms, which results in high power consumption. However, the CPRS algorithm provides less energy consumption in some cases. Once again, energy consumption is dependent on applications. The compiler will schedule the input code using all methods and select an appropriate output code for their requirement. The CLS scheduling algorithm provides the potential lower energy consumption and a similar throughput compared to others DSP/VLIW processor

[0201] Aspects of the present invention provide an instruction cell-based reconfigurable computing architecture for low power applications. For the development of RICA, a top-down approach was adopted that revealed the key design decision for a flexible, easy to program, low power architecture. These features make RICA an architecture that inherently solves the main design requirements of modern low power devices.

[0202] By designing the silicon fabric in a similar way to reconfigurable arrays but with a closer equivalence to high level program software the present invention achieves the same high performance as coarse-grain FPGA architectures, and the same flexibility, low cost and programmability as DSPs. Although the RICA architecture is similar to a CPU, the use of an IC-based reconfigurable core as a datapath gives important advantages over DSP and VLIWs, such as more support for parallel processing. The reconfigurable core can execute a block containing both independent and dependent assembly instructions in the same clock cycle, which prevents the dependent instructions from limiting the amount of ILP in the program. Other improvements over DSP architectures include reduced memory access by eliminating the centralized register file and the use of distributed memory elements to allow parallel register access.

[0203] Results show that RICA significantly outperforms FPGA based implementation while providing straightforward high-level reprogrammability. In initial performance results it also delivers up to 6 times less power consumption when compared to leading VLIW and low-power DSPs processors.

[0204] Further modifications and improvements may be added without departing from the scope of the invention herein described.

1. A processor for executing program instructions having datapaths of both dependent and independent program instructions, the processor comprising:

an interconnection network;

a heterogeneous plurality of instruction cells each connected to the interconnection network;

a decoding module adapted to receive configuration instructions, each instruction encoding the mapping of at least one of a datapath of dependent program instructions and a datapath of independent program instructions to a circuit of the instruction cells and further adapted to decode a configuration instruction and configure at least some of the interconnection network and instruction cells, thereby mapping the datapath to the circuit of the instruction cells and executing the program instructions.

2. The processor of claim **1** wherein the decoding module is adapted to configure at least some of the interconnection network and instruction cells by connecting at least some of the instruction cells in series through the interconnection network.

3. The processor of any of claims **1** and **2** wherein the decoding module is further adapted to receive configuration instructions encoding the mapping the datapaths of a plurality of program threads to a corresponding plurality of independent circuits of the instruction cells and further adapted to decode a configuration instruction and configure at least some of the interconnection network and instruction cells, thereby mapping the datapaths of the plurality of program threads to the corresponding plurality of circuits of the instruction cells and contemporaneously executing the program threads independently of each other.

4. The processor of any previous claim wherein the processor further comprises a clock module adapted to provide a clock signal having clock cycles and the decoding module is operable to decode the configuration instruction so as to configure at least some of the interconnection network and instruction cells each clock cycle.

5. The processor of claim **4** wherein the clock module is adapted to provide a variable clock cycle.

6. The processor of any of claims **4** and **5** wherein the clock module is adapted to provide an enable signal.

7. The processor of claim **6** wherein the enable signal is provided to an enable input of an instruction cell.

8. The processor of any of claims **4** to **7** wherein the clock module is adapted to provide a plurality of clock signals.

9. The processor of claim **8** wherein at least some of the plurality of clock signals are separately provided to a plurality of instruction cells.

10. The processor of any of claims **4** to **9** wherein the decoding module is further adapted to decode a configuration instruction so as to configure a clock signal.

11. The processor of claim **10** wherein the decoding module is further adapted to decode a configuration instruction so as to configure the clock cycle.

12. The processor of any of claims **10** and **11** wherein the active period of the clock signal is configured.

13. The processor of any of claims **10** to **12** wherein the duty cycle of the clock signal is configured.

14. The processor of any previous claim wherein the processor further comprises a program counter and an interface to a configuration memory, the configuration memory being adapted to store the configuration instructions, and at least one of the instruction cells comprises an execution flow control instruction cell adapted to manage the program counter and the interface so as to provide one of the configuration instructions to the decoding module each clock cycle.

15. The processor of claim **14** wherein the enable signal is provided to an enable input of the execution flow control instruction cell.

16. The processor of any previous claim wherein at least one of the instruction cells comprises a register instruction cell operable to change its output each clock cycle.

17. The processor of claim **16** wherein the enable signal is provided to an enable input of the register instruction cell.

18. The processor of any of claims **16** and **17** wherein the enable signal is provided both to an enable input of the execution flow control instruction cell and to an enable input of the register instruction cell.

19. The processor of any previous claim wherein the processor further comprises at least one input/output port and at least one of the instruction cells comprises an input/output register instruction cell adapted to access the at least one input/output port.

20. The processor of any previous claim wherein the processor further comprises at least one stack and at least one of the instruction cells comprises a stack register instruction cell adapted to access the at least one stack.

21. The processor of any previous claim wherein at least one of the instruction cells comprises a memory instruction cell adapted to access a data memory location.

22. The processor of claim **21** wherein the data memory location comprises a multi-port memory location.

23. The processor of any of claims **21** and **22** wherein a pair or more of memory instruction cells are adapted to access in the same clock cycle a pair or more of data memory locations clocked at a higher frequency than the frequency of the clock cycle.

24. The processor of any of claims **4** to **23** wherein the decoding module is further adapted to decode a configuration instruction so as to configure the clock signal dependent on the configuration of a clock signal decoded from a previous configuration instruction.

25. The processor of any previous claim wherein at least one of the instruction cells comprises a multiplex instruction cell adapted to route data between instruction cells.

26. The processor of any previous claim wherein at least one of the instruction cells comprises a combinatorial logic instruction cell.

27. The processor of any previous claim wherein the function of an instruction cell corresponds to a program instruction.

28. The processor of any of claims **14** to **27** wherein the circuit of instruction cells comprises a set of instruction cells including one execution flow control instruction cell.

29. A compiler for generating configuration instructions for the processor of any previous claim, the compiler comprising:

a front end module adapted to receive the program instructions and to identify the datapaths of the program instructions; and

a scheduling module adapted to map the datapaths of both dependent and independent program instructions as circuits of the instruction cells; and

a mapping module adapted to encode the configuration of circuits of instruction cells as configuration instructions and adapted to output the configuration instructions.

30. The compiler of claim **29** wherein the program instructions comprise assembler code output from a high level compiler.

31. The compiler of claim **29** wherein the program instructions comprise source code of a high level language.

32. The compiler of any of claims **29** to **31** wherein the scheduling module is further adapted to schedule the program instructions according to their interdependency.

33. The compiler of any of claims **29** to **32** wherein at least one of the front end module and scheduling module are further adapted to map access to registers for the storage of temporary results into interconnections between instruction cells.

34. The compiler of any of claims **29** to **33** wherein the scheduling module is further adapted to map a datapath of the program instructions that will be executed in a single clock cycle.

35. The compiler of any of claims **29** to **34** wherein the scheduling module is further adapted to map the datapaths of a plurality of program threads as a corresponding plurality of contemporaneous circuits of the instruction cells.

36. The compiler of any of claims **29** to **35** wherein the scheduling module is further adapted to map the datapaths using operation chaining.

37. The compiler of any of claims **29** to **36** wherein the scheduling module is further adapted to use operation chaining while scheduling a single clock cycle.

38. The compiler of any of claims **29** to **37** wherein the scheduling module is further adapted to use operation chaining while processing each ready list of a plurality of ready lists.

39. The compiler of any of claims **29** to **38** wherein the scheduling module is further adapted to map the datapaths using register allocation.

40. The compiler of claim **39** wherein the scheduling module is further adapted to use register allocation while scheduling a single clock cycle.

41. The compiler of any of claims **39** and **40** wherein the scheduling module is further adapted to use register allocation while processing each ready list of a plurality of ready lists.

42. The compiler of any of claims **29** to **41** wherein the scheduling module is further adapted to map the datapaths using a plurality of scheduling algorithms.

43. The compiler of claim **42** wherein the scheduling module is further adapted to map the datapaths by using the output selected from one of the plurality of scheduling algorithms.

44. A method of executing program instructions, the method including the steps of:

identifying datapaths in program instructions having datapaths of both dependent and independent instructions; and

configuring at least some of an interconnection network and a heterogeneous plurality of instruction cells, wherein each instruction cell is connected to the interconnection network, by connecting at least some of the instruction cells in series through the interconnection network thereby mapping the datapaths of both dependent and independent program instructions and executing the program instructions.

45. The method of claim **44** further including the step of providing a clock signal having clock cycles to at least one of the instruction cells, wherein the step of configuring the instruction cells is performed each clock cycle.

46. The method of any of claims **44** and **45** wherein the step of configuring further includes the step of configuring the subsequent clock cycle.

47. The method of any of claims **44** to **46** wherein the active period of the clock cycle is configured.

48. The method of any of claims **44** to **47** wherein the duty cycle of the clock signal is configured.

49. The method of any of claims **44** to **48** wherein the step of configuring at least some of the interconnection network and the instruction cells includes the steps of:

mapping the datapaths of both dependent and independent program instructions as circuits of the instruction cells;

encoding the configuration of the circuits of instruction cells as configuration instructions; and

decoding the configuration instructions so as to configure the at least some of the interconnection network and the instruction cells.

50. The method of claim **49** wherein the step of mapping includes the step of scheduling the program instructions according to their interdependency.

51. The method of any of claims **49** and **50** wherein the step of mapping includes the step of mapping access to registers for the storage of temporary results into interconnections between instruction cells.

52. The method of any of claims **49** to **51** wherein the step of mapping includes the step of mapping a datapath of the program instructions that will be executed in a single clock cycle.

53. A computer program comprising program instructions, which, when loaded into a computer, constitute the compiler of any of claims **29** to **43**.

54. The computer program of claim **53** embodied on a storage medium, stored in a computer memory or carried on an electrical or optical carrier signal.

55. A computer program comprising program instructions for causing a computer to perform the method of any of claims **44** to **52**.

56. The computer program of claim **55** embodied on a storage medium, stored in a computer memory or carried on an electrical or optical carrier signal.

* * * * *