The CREC Reconfigurable Computer

Octavian Creț[1], Cristian Vancea[1], Balint Szente[2], Ligiu Uiorean[1] and Florin Rusu[1]

[1] Automation and Computer Science Faculty, Technical University of Cluj-Napoca, Romania
Email: Octavian.Cret@cs.utcluj.ro, vcc@email.ro,
uiorean@codec.ro, usur_nirolf@yahoo.com

[2] Engineering Faculty, "Petru Maior" University of Târgu-Mureş, Romania
Email: bszente@ms.fx.ro

Abstract. *The world of Reconfigurable Computing has known a significant development in the past years. The main research done in this field was oriented towards applications involving low granularity operations and high intrinsic parallelism. CREC is an original, low-cost General-Purpose Reconfigurable Computer whose architecture is generated through Hardware/Software CoDesign. The main idea of the CREC computer is to generate the best-suited hardware architecture for each software application. The Parallel Compiler parses the source code and generates the hardware architecture, based on multiple Execution Units. The hardware architecture is described in VHDL code that is generated by a program, written in ANSI C. Finally, CREC is implemented in an FPGA device. Its great flexibility makes the CREC system interesting for a wide class of applications that mainly involve high intrinsic parallelism, but also other kinds of computations.*

1 Introduction and Background

Programmable architectures heavily and rapidly reuse a single piece of active circuitry for many different functions (for example, processors that perform different instructions in their ALU on every cycle). In *configurable* architectures, the active circuitry can perform any of a number of different operations, but the function cannot change from cycle to cycle (for example, FPGA devices). The main differences between *reconfigurable* and *classical* systems (processors) are [1]:

Instruction distribution – instead of distributing the new instruction to the functional units in each cycle, instructions are locally configured, allowing the compression of their distribution flow.

Spatial routing of intermediate results – intermediate values are being routed in parallel from producers to consumers, instead of forcing the entire communication to take place in time, by means of a central resource (which is subject to "bottleneck").

Several constructive blocks of fine granularity, separately programmable – reconfigurable devices possess a large number of constructive programmable blocks, which enable them to perform several computations per time unit.

Distributed and reusable resources – the resources (such as the memory, the interconnection network and the functional units) are distributed and reusable. Thanks to the independent local access at these resources, one can benefit from the large bandwidth, because the bottleneck effect is no longer an issue.

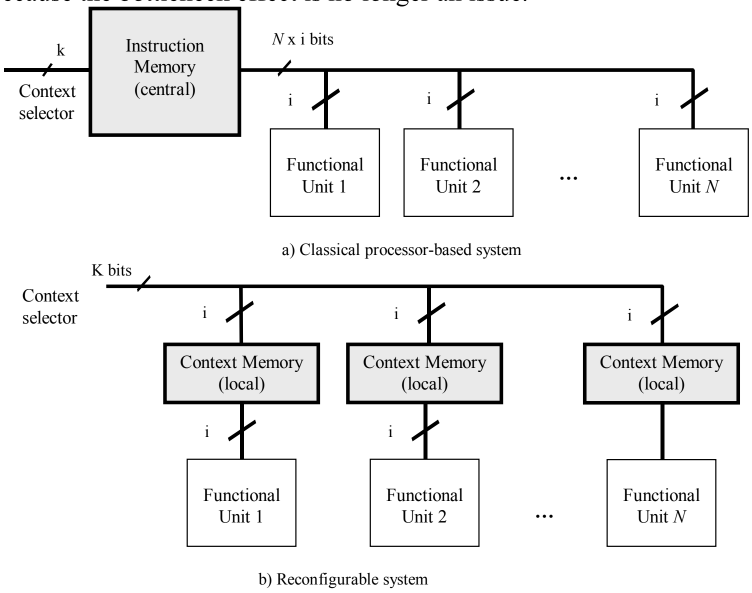

Fig. 1. Generic diagram of instructions broadcasting in classical and reconfigurable systems

General-purpose reconfigurable computers (GPRCs) represent an (quite) old problem. A very important GPRC implementation was the DPGA [1]. The DPGA basically consists of an array of *reconfigurable computing elements* (RCE - similar to the reconfigurable cells that are present into a classical FPGA device), but where the *context memory* of each RCE can store not only one (like in FPGA devices), but several *configuration contexts*. In fact, this is a distinctive feature of all the RC devices: the local storage of the several configuration contexts for each RCE.

The efficiency of Reconfigurable Computers (RCs) has been proven in several types of applications: DSP, image processing, pattern recognition, evolvable hardware, etc. In most projects, the main idea was to integrate a small processor together with a RC unit ([1,4,5,8,9]) inside a single chip, thus achieving a considerable gain of performance for specific applications, sometimes overcoming even the performances of specialized DSP processors.

One of the first proposals for a RC using commercial FPGAs was the Programmable Active Memory from DEC Paris Research Lab [7]. Based on FPGA technology, a PAM (Programmable Active Memories) is a virtual machine controlled by a standard microprocessor that can be dynamically configured into a large number of application-specific circuits. PAM introduced the concept of "active" memory.

Programmable datapaths were proposed by RaPid [8], a linear array of function units composed of 16-bit adders, multipliers and registers connectable through a reconfigurable bus. The logic blocks used are optimized for large computations. They perform these operations much more quickly and consume less space on the chip than a set of smaller cells connected to form the same type of structure.

The GARP project [5] introduced the idea of configuration cache. The GARP system had a cache to hold recently stored configurations for rapid reloading.

The RAW microprocessor chip [9] comprises a set of replicated tiles. Each tile contains a simple RISC processor, a small amount of configurable logic and a portion of memory for instructions and data. RAW achieves performance levels 10x-100x over workstations for many applications.

2 Motivation and Main Concepts

The main idea of the CREC design was to build a low-cost GPRC able to exploit the intrinsic parallelism present in many low-level applications by generating the best-suited hardware for implementing a specific application. The CREC design was introduced for the first time in [2]; the main novelty introduced by CREC consists of the combination, in a very effective way, of several design styles and architectural concepts. The result is a new computational model based on reconfigurable architecture concepts and whose main features are:

1. *Instruction Level Parallelism - ILP*;
2. *Hardware / Software CoDesign*;
3. *DPGA-like architecture*, with *Multiple Execution Units*, whose number is determined dynamically at compilation;
4. *Parallel RISC architecture* and *Parallel Compiler*.

In addition to these aspects, a set of original ideas were introduced, specific to CREC design flow:

1. *Dynamic generation of the hardware architecture*;
2. *Separation of the Direct (Immediate) Operands Memory from the Data Memory*;
3. The choice of *FPGA devices as physical support of the hardware implementation*, thus allowing the static reconfigurability (a new, optimal structure is generated for each application);
4. *Fully scalable architecture* (from the space point of view);
5. *Dynamically reconfigurable architecture*, in DPGA fashion.

The CREC architecture is built without creating a specialized VLSI chip (like in most RC implementations), but only by using existing tools (a VHDL compiler and synthesizer, a C++ or JAVA compiler, FPGA download programs) and FPGA as physical support. Even if the compilation time increases, for applications needing a big execution time, it will be largely recovered at run-time by the parallel execution instead of the classical, sequential one.

This approach opens the way for a new family of computing systems, where there will be no restrictions on the number of EUs (of course, there will always be a restriction given by the FPGA device capacity, but with the technological progress this capacity will continuously increase). A completely integrated development environment (IDE), including all the previously mentioned programs, would significantly reduce the overall compilation time. Such an IDE is under development.

3 Design Flow

CREC is the final product of a Hardware/Software CoDesign process, where the hardware part is dynamically and automatically generated during compilation. The resulting architecture is optimal because it exploits the intrinsic application parallelism. The main steps in the application development are:

1. The application's source code is written in *CREC assembly language*.
2. The source code is compiled using a *parallel compiler*, which allows the implementation of ILP (instruction-level parallelism). The compiler detects and analyses *data dependencies*, then it determines which instructions can be executed in parallel.
3. According to the slice's size, the hardware structure will be generated. The generic architecture already exists in a VHDL source file, so at this moment it is only adjusted. It will be materialized in an FPGA device.
4. An original feature of CREC is the fact that the memory is divided in three parts: *Data Memory* (off-chip), *Instructions Memory* (on-chip, in DPGA style) and *Operands Memory* (on-chip according to the FPGA's capacity).
5. The VHDL file is compiled and the FPGA device is configured.

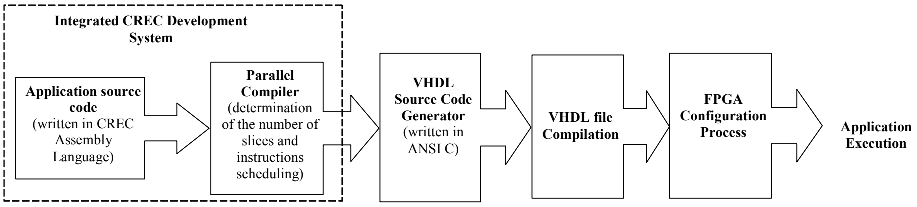
Integrated CREC Development System
Application source code (written in CREC Assembly Language)
Parallel Compiler (determination of the number of slices and instructions scheduling)
VHDL Source Code Generator (written in ANSI C)
VHDL file Compilation
FPGA Configuration Process
Application Execution

Fig. 2. The CREC Design Flow

3.1 The Low-Level Development Environment

The low-level component of the software part is implemented in two variants. The first one is written in ANSI-C and uses the BISON parser, without a user-friendly interface. In the second variant, the development software of the CREC computer is written in JAVA using the JavaCC parser and embedded in an integrated development environment (IDE) consisting of: *the RISC Editor*, *the Instruction Expander, the Parallel Compiler* and *the Test Bench*. These components' functionality is derived from the classical stages of every compiler: lexical analysis, syntactic analysis, semantic analysis and code generation.

The *RISC Editor* is a simple tool for editing CREC code to be analyzed by the *Instruction Expander*. It realizes the lexical and syntactic analysis of the program, offering exact information about the location and type of errors. A supplementary feature of the editor is the syntax help it provides. The *Instruction Expander* generates the explicit form of the instructions that will than be used by the *Parallel Compiler*. It takes a CREC source file and generates the expanded form. This task implies *semantic analysis* of the program, which consists of extracting relevant information about the instructions that will later be used at the *Parallel Compiler* level. The user can alter the expanded form of the instructions (e.g. assign an instruction to a given EU).

The *Test Bench* component permits a step-by-step simulation of a parallel program. This way, the user can follow the values from the EUs and from memory at each clock cycle. As a result of its functionality, the Test Bench can be seen as a debugger.

3.1.1 The Low-Level Parallel Compiler

What particularizes the CREC compiler is the role of the last stage, *code generation*. Besides assigning the instructions to EUs, the *Low-Level Parallel Compiler* also determines the sets of instructions that can be executed in parallel, according to the hardware architecture. The algorithm that realizes this feature using the intrinsic parallelism of instructions is presented below; it has a *linear execution time*, proportional to the number of program's instructions, *O(N)*.

The compiler's task is to divide a CREC assembly language program written in a sequential manner into pieces to be executed in parallel. The compiler generates a file in a specific format that describes the tailored CREC architecture using the expanded form of program's instructions resulted from the previous phase. This file will include the size of the various functional parts, the subset of instructions involved, the number of EUs, etc., together with the sequence of instructions that makes up the program.

Another aspect that makes CREC particular with respect to classic processors, besides its reconfigurable nature, is the parallel nature of the computations: each EU has its own accumulator register. Thus, at each moment during execution, there are *N* distinct EUs, each one executing the instruction that was assigned to it by the compiler, taking into account its nature and the rules associated with instruction scheduling. Some instructions specify the precise EU to execute them, while other instructions can be executed by any EU. For instance, "*MOV R1, 7*" will be executed by EU1, since it works with the register R1, while "*JMP 3*" has no specific EU associated and will be assigned one by the compiler, depending on the availability.

The accumulator register of all the EUs have equal capacities, but *the internal structure of each EU will be different*, according to the subset of instructions (from the CREC Instruction Set) that the EU will actually execute.

The *scheduling algorithm* groups instructions so that they can be executed in parallel. A group of instructions that are executed at the same time is called a *slice*. We will use an example to illustrate the algorithm and its associated notions. Consider the following CREC assembly source code, representing the *bubble sort* algorithm.

```
1.   XOR R1,R1           START BLOCK 1
2.   LOAD [R1]
3.   MOV R2,LDB          END BLOCK 1
4.   CMP R2,2            START BLOCK 2
5.   JB R2,[26]          END BLOCK 2
6.   MOV R1,1            START BLOCK 3
7.   MOV R5,R2           END BLOCK 3
8.   LOAD [R1]           START BLOCK 4
9.   MOV R3,LDB
10.  INC R1
```

```
11.  LOAD [R1]
12.  MOV R4,R1
13.  CMP R3,LDB
14.  CMP R4,R5
15.  CALA R3,[19]        END BLOCK 4
16.  JBE R4,[8]          START & END BLOCK 5
17.  DEC R2              START BLOCK 6
18.  JMP [4]             END BLOCK 6
19.  MOV STB,R3          START BLOCK 7
20.  STORE [R1]
21.  DEC R1
22.  MOV STB,LDB
23.  STORE [R1]
24.  INC R1
25.  RET                 END BLOCK 7
```

Fig. 3. The Bubble Sort algorithm

The scheduling algorithm consists of two parts: a global one that treats the entire program and divides it into *blocks*, and an intra-block one that processes one or more blocks. The result of the first part will be a set of blocks, each consisting of at least one instruction of the program. Then, every block will be processed by the intra-block algorithm, whose result is the parallel program, consisting of *slices*. The formal approach of the algorithm is presented below.

If **P** is the set of instructions from a program, then the set of blocks resulting after the first phase of the algorithm, **B(1)**, **B(2)**, …, **B(k)** is a partition of **P**, where **k** is the number of blocks. The control flow of the program (branches and referenced instructions – targets of branches) determines its set of blocks.

Definition 1: A ***block*** is a set of consecutive instructions that respects at least one of the following conditions:

i) the first instruction is referenced;

ii) the last instruction is a branch (Jump or Call) or return from a procedure (Ret).

Definition 2: An *execution graph* is a particular type of **DAG**, **G (V, E)**, where **V = {B(1), B(2), …, B(k)}** is the set of vertices and **E** is the set of forward direct edges. The set of edges is constructed from the control flow of the program and contains only forward edges. Backward branches are not represented in the graph.

According to these definitions, we obtain the set of *blocks* from Fig. 3 and the *execution graph* from Fig. 4 for the *bubble sort* example.

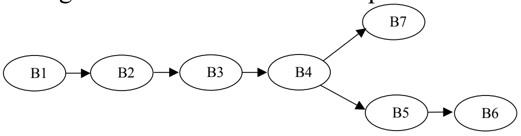

Fig. 4. The execution graph of the Bubble Sort Algorithm

The *blocks* from **Definition 1** have a particular structure. They contain *at most one branch instruction*, situated at their end. This makes easy the compiler's task to determine which instructions can be executed in parallel because it has to take into consideration only the *data dependencies*. This is the task of the intra-block algorithm: to determine the program's *slices*.

According to the classical compiler theory, there are three types of data dependencies between two instructions, **I1** and **I2** that appear in this sequential order:

i) a register defined in **I1** is used in **I2** (flow dependence);
ii) a register defined in **I2** is used in **I1** (anti-dependence);
iii) a register defined in **I1** is defined in **I2** (output-dependence).

For two instructions to be executed in parallel, all these three dependencies have to be held between them. These conditions are also valid for memory locations, but there is more difficult to evaluate them because of the various addressing modes.

Next factors have also to be taken into consideration: the sequential flow of the program and the delay introduced by each instruction. The first one is important for the order of read/write instructions involving the same register. The delay represents the number of cycles after which the target of an instruction modifies its value. In this case, all the instructions have a delay of one cycle.

According to all of the above, we can summarize and define an algorithm for scheduling a sequential program into groups of instructions that can be executed in parallel, whose main steps are:

1. partition the program into **blocks**;
2. build the **execution graph** from the control flow of the program;
3. for every vertex (block) resulted from searching the graph based on topological sort, determine the **slices** that respect the data dependencies between their instructions.

Applying this algorithm to the *bubble sort* example from Fig. 3, we obtain the parallel program scheduling from Fig. 5.

a)

XOR R1,R1		
LOAD [R1]		
MOV R2, LDB		
CMP R2, 2		
JB R2, [26]		
MOV R1, 1	MOV R5, R2	
INC R1	LOAD [R1]	
LOAD [R1]	MOV R3, LDB	MOV R4, R1
CMP R3, LDB	CMP R4, R5	
CALA R3, [19]		
JBE R4, [8]		
JMP [4]	DEC R2	
MOV STB, R3		
DEC R1	MOV STB, LDB	STORE [R1]
INC R1	STORE [R1]	RET

b)

Execution Unit	Slices														
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
EU1	1	2				6	10	11				18		21	24
EU2			3	4	5		8					17		20	23
EU3								9	13	15			19	22	25
EU4								12	14		16				
EU5						7									

Fig. 5. Bubble Sort program scheduling: a) the groups of instructions that can be launched in parallel; b) slides scheduling

3.1.2 The High-Level Parallel Compiler

The efficient programming of a processor implies the use of a high-level language. This is true especially in the case for a complex parallel system like CREC, which needs such a language in order to exploit the characteristics of the hardware architecture. The CREC assembler presented in section 3.1 exploits the instruction-level parallelism. The system is efficient in case of assembly language programs and is used as *optimization method* for the code generated by the high-level compiler.

Assembly language programming is generally efficient from the code optimization and resource management point of view, but it is not practical for complex programs. In addition, programming a parallel system introduces supplementary complications related to the way that parallelism is exploited and to threads tracing. This section presents CREC's high-level language.

The CREC high-level language is aimed to be as close as possible to the C functionality and syntax. A general idea about the language can be formed by examining the example below:

```
#arch 16bit //defines the EU’s accumulator width
// All the program variables will have this width.
#arch 8eu //forces the use of 8 EUs, at most.
main (){//program’s starting point
var r, a, b; //variable definitions
fastvar m;
portin (1,a);//reads from port 1 and stores the value
//in variable a
split { //begin of a parallel zone
         a+=m;
         b=b+m+m;
} //end of the parallel zone - forces synchronization
r=a+b;
portout (1,r);//writes the value of r on the output
              //port 1
}
```

The difference between the *var* and *fastvar* declarations consists in the mode these variables are allocated inside the program. The former declaration denotes a usual variable, stored in the Data Memory; the later one denotes a variable that is permanently stored in one of the EUs, resulting in a faster access to its value.

The zone delimited by *split { }* is specific to CREC language and denotes a *parallel program zone*. All the statements located inside this zone will be executed in parallel, on different EUs. The program's execution advances only when all the statements inside the *split { }* block have finished their execution (this is a synchronization mechanism). This way, it is guaranteed that the following statement will operate on the correct final values.

The parallel program block delimited by the *split { }* zone has a set of constraints. At run-time, the CREC architecture and the program to be executed are already fixed. That's why, it is not possible to use iterative blocks inside the *split { }* block, and execution control blocks neither. The compiler operates in the classical way, building the execution tree for the program. When encountering a *split { }* block, it will create a new tree for each statement.

The program is further optimized in two passes: the first one detects redundant statements, while the second pass is the optimization brought by the low-level compiler.

3.3 The VHDL Source Code Generator

The *VHDL Code Generator* consists in a package of programs written in ANSI C and VHDL, communicating with each other and having as result a set of VHDL files that describe an optimized architecture suited for executing the program and ready for the next phase, *FPGA Configuration Process* (see Fig. 2).

The first task of the *VHDL Code Generator* is to parse the file received from the *Parallel Compiler* and to determine other needed parameters, such as the minimum possible size of the Operand Memory and the Instruction Memory for each EU or the width of the Slice Memory, which multiplied with the number of slices determines the size of this memory; Boolean parameters will specify the need for some parts of the architecture (eg. LOAD BUFFER, STORE BUFFER) or for different signals (nets) that connect the EUs.

Having all needed parameters, the next task is to generate the final VHDL files. For the generation of the VHDL code an ANSI C program is used. This code includes original libraries that contain flexible components. It is strongly dependent on *generics*, user-defined *functions* and *constants*.

In order to outline the flexibility of the *VHDL Source Code Generator*, an example showing the generation of the memories is presented below. To create a random sized memory, the program must decide what BlockRAM to use (a BlockRAM is a special synchronous memory in Virtex FPGA devices). There are different BlockRAM types, of different sizes: 4096x1, 2048x2, 1024x4, 512x8, 256x16 bits. For a 256x24 bits memory, the following configurations can be used: 32 4096x1 bits BlockRAMs, 12 2048x2 bits BlockRAMs, 6 1024x4 bits BlockRAMs, 3 512x8 bits BlockRAMs, 2 256x16 bits BlockRAMs.

The other components that are also taken into consideration when deciding the structure of a memory block are: the number of used FPGA Logic Blocks and the number of tri-state buffers. Each one of these components has an assigned cost (pre-established in the program code) and the *Generator* will chose the structure with the smallest total cost. For example a 256x32 memory might have two possible minimal configurations: 4 BlockRAM modules of 512x8 bits (1x4 BlockRAMs) or 4 BlockRAM modules of 256x16 (2x2 BlockRAMs). The first will be chosen because it has one row of BlockRAMs and does not need extra Logic Cells or tri-state buffers used in address decoding and data output separation (see Fig. 6).

As shown in Fig.2, the result of the *Parallel Compiler* is directed to the *VHDL Code Generator* in a special binary format that specifies a set of parameters such as: the number of slices and EUs (result of the parallel scheduling algorithm), the data width, the sizes of the stacks (Data Stack, Slice Stack), the size of the Data Memory and the slices containing instructions assigned to each EU. Most of these parameters (e.g. memory sizes) cannot be determined by simply parsing the program so they are either specified by the programmer (using special directives of the CREC assembly language), either assigned by default by *the Parallel Compiler*.

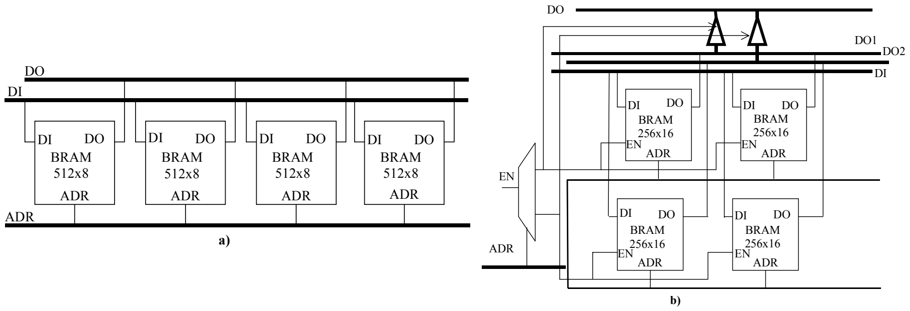

Fig. 6. Two possible implementation of the example memory block

4 The Hardware Architecture

The hardware structure is described using VHDL code, which is generated and optimized by a package of programs. The optimization is done by the VHDL Source Code Generator and consists of eliminating each unnecessary element, signal or bus.

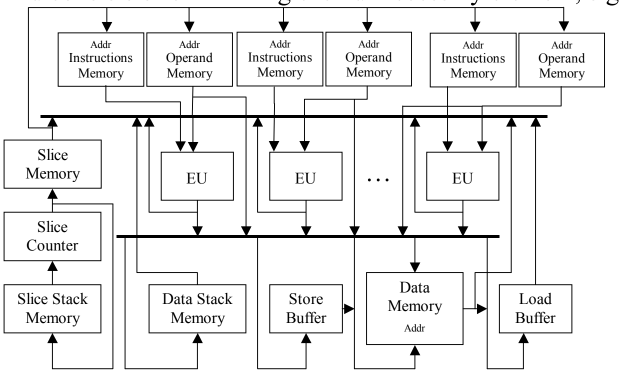

Fig. 7. The general CREC architecture

The architecture's main components are: the *N* EUs; the *N* local configuration memories for the *N* EUs, (in DPGA style), called *Instructions Memories*; a *Data Stack Memory*, used in instructions like PUSH or POP; a *Slice Stack Memory*, used to store the current slice address (CALL, RETURN); a *Slice Program Counter*; an associative memory that maps instructions to the slices that must be executed by each EU, called *Slice Memory*; a *Store Buffer* and a *Load Buffer* (temporary data buffers used to store information to/from Data Memory); a *Data Memory*; *Operand Memories*, which contain the direct operands for the EUs.

Fig. 7 shows the linkage between the basic hardware elements. *The links between EUs are point-to-point*, but the Data Memory, the Slice Counter and the Slice Stack Memory are accessed via Address, Data and Control busses. The Direct Operands Memory can be accessed only by its corresponding EU.

3.3.1 The Instruction Set

Each instruction is encoded on the same number of bits, like in RISC architectures. Although CREC is a RISC processor, its EU has a relatively large instruction set, making it attractive for a wide range of applications. The instruction set is divided in the *Data Manipulation* and the *Program Control* Groups. The Data Manipulation Group contains the specific instructions for manipulating the value of the EU's accumulator. The Program Control Group contains the instructions for altering the program execution. Each instruction performs operations on unsigned numbers.

The EU can perform the following instructions: *Addition* with/without carry and *Subtraction* with/without borrow and *Compare*; Logical functions: *And*, *Or*, *Xor*, *Not* and *Bit test*; *Shift arithmetic* and *logic* left/right; *Rotate* and *rotate through carry* left/right; *Increment*/*decrement* the accumulator and *negation* (2's complement); Slice counter manipulation: *Jump*, *Call* and *Return*; *Data movement*; Stack manipulation: *Push* and *Pop*; *Input from* and *Output to port*; *Load* from and *Store* in the Data memory.

Each program control instruction is conditioned, thus offering a great flexibility. This way, the source code can be optimized, because most of the Compare and Jump statements - typically the most frequently used instruction pairs - can be replaced by conditional instructions (for example, conditioned *move* and other instructions).

3.3.2 The Execution Unit

The main part of the CREC processor is the scalable EU. The word length of the EU is *n**4 bits. At the current state of the implementation, the parameter *n* is limited to 4, so the word length can be up to 16 bits. The complete structure of the EU is presented in fig. 8.

There are two variants of the CREC EU implementation, but from the functional (behavioral) point of view they are absolutely the same. In the first one, each subunit is strongly optimized for the Xilinx VirtexE FPGA family, occupying the same number of Virtex Slices (2**n* Slices) and using the dedicated Fast Carry Logic. This leads to a platform-dependent solution, but there was the need to increase the performance of the EU and to obtain almost equal propagation times. The second variant uses a general VHDL code, not optimized for any FPGA devices family. This increases CREC's portability, but the architectural optimizations become the VHDL compiler's task, reducing the designer's control over the generated architecture. In the next sections we will present the Virtex-optimized architecture.

The EU consists of six major parts: *Decoding Unit* (decodes the instruction code); *Control Unit* (generates the control signals for the Program Control Group); *Multiplexer Unit* (allows to select the second operand of binary instructions); *Operating Unit* (implements the data manipulating operations); *Accumulator Unit*; *Flag Unit* (contains the two flags: Carry and Zero).

The *Operating Unit* has a symmetrical organization. At the right side are the binary operation blocks, and at the left side are the unary operation blocks. Its four blocks are: the *Logic Unit*, the *Arithmetic Unit*, the *Shift Left Unit* and the *Shift Right Unit*.

The Execution Unit is customisable. For example, if an EU will not execute any Logical Instruction, then this part is simply cut out, resulting in a gain of space. The structure of the Arithmetic Unit has a set of advantages: it uses only one level of slices, it is cascadable and the number of slices increases linearly with the width. The implementation of the Shift cell is paired only with the NOT function. Eight shift operations can be realized: SHL, SAL, ROL, RCL, SHR, SAR, ROR and RCR.

All four units use the same number of Virtex slices. For this reason, the size of the Operating Unit is growing *linearly* with the word length. But the operating time will not decrease significantly, because the number of CLB levels is constant. Thus, the EU can be easily pipelined for higher frequency operation. For example, a complete EU (with all the subunits generated) having an 8-bit accumulator will consume 17 CLBs, but the same EU with a 16-bits accumulator will consume 33 CLBs.

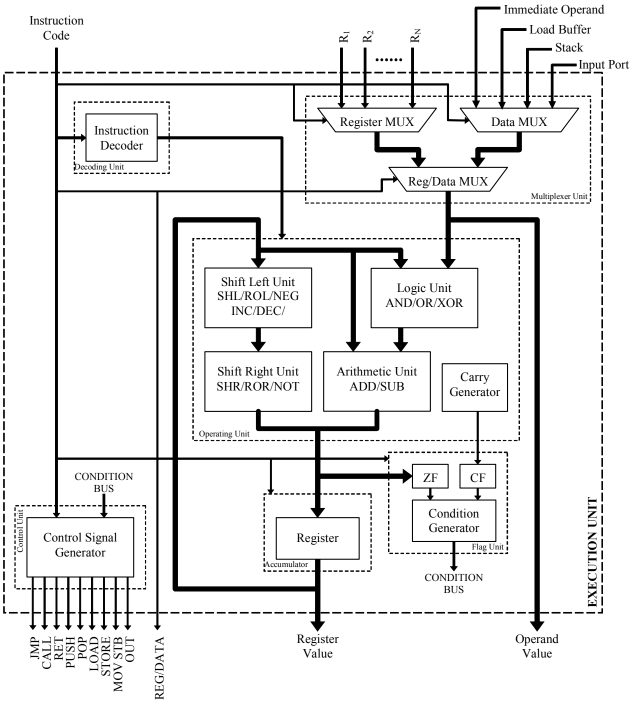

Fig. 8. The basic CREC Execution Unit

The output of the *Flag Unit* is a 6-bits wide Condition Bus for the six possible condition cases: *Zero*, *Not Zero*, *Carry*, *Not Carry*, *Above* and *Below or Equal*. This bus validates the conditioned Program Control instructions.

The *Multiplexer Unit* is built on two levels for optimal instruction encoding. At the first level, the Register and the Data MUXs have the same selector. The second 2:1 multiplexer selects the input operand for the instruction. This Unit is also customizable: only those input lines would be implemented, which are actually used by the EU. In Virtex FPGAs, only 8-to-1 multiplexers can be implemented on a single level of CLBs. For this reason, the multiplexer is optimized for up to 8 inputs. For CREC architecture with more than 8 EUs, the multiplexers are implemented on two levels of logic. This disadvantage can be overcome by using FPGAs containing wider multiplexers. The most important aspect is that this unit's size increases linearly with the increase of the word length and the number of EUs. For example, the CREC with eight 8-bit EUs uses 96 CLBs and with eight 16-bit EUs it consumes 160 CLBs.

The *Accumulator Unit* stores the primary operand also the result of each Data Manipulation instruction. The *Control Unit* generates the validation signals for the Program Control instructions taking into account the Condition Bus. The condition code is compared against the value of the flags generated by the previous non-program control instruction. The *Decoding Unit* generates the appropriate signals for the four functional parts of the Operating Unit and for the Carry Generator. It is a fast Combinatorial Logic Circuit, thus obtaining a greater speed.

In conclusion, an EU uses only a fraction of the FPGA device. For example, an EU with 8-bits wide registers occupies approximately 4%, and an EU with 16-bits wide registers occupies approximately 6% of the available CLBs in a Virtex600E FPGA.

The *Slice Memory* is an associative memory that stores the general slice word. This word is composed of the slice words for each EU and contains two fields: the address of the instruction to be executed by the EU (a pointer in the Instructions Memory – IM), and a pointer in the Operands Memory – OM. According to the number of instructions that each EU performs in the current application, the widths of these two fields are variable and different for each EU. This way, the Slice Memory's word width is variable with each application and will always be kept to a minimum. This memory is implemented in the Virtex BlockRAM memories.

The *Instructions Memory* and the *Direct Operands Memory* are *distributed* memory blocks, implemented in the Virtex Look-up Tables configured as ROMs. Due to their relatively small size, a BlockRAM-based implementation would be inefficient. Their size depends on the number of instructions and on the number of direct operands that each EU works with during the application execution. The size will be optimized according to these parameters.

The general *Data Stack Memory* is implemented as RAM. Its size is variable and will be estimated according to the number of PUSH / POP instructions present in the application source code. The *Slice Stack Memory* is used by the general instructions CALL and RETURN. Here is stored the slice number for a procedure call/return from procedure call. The *Data Memory* is normally implemented outside the FPGA chip, but for simulation purposes we created this block inside the Virtex chip. This memory contains the usual data (constants or variables used in the application).

5 Experimental Results

At this point of the project, the software part of the CREC system is finished, but can still be subject to modifications, as new features will be added.

Both the low and high-level parallel compilers are functional, in several variants. They have been tested and verified for some classical benchmark applications. The tests were performed on classical general-purpose algorithms. Some of the results are shown in Table 1. The algorithms were executed on CREC architectures having a different number of EUs; the reference architecture is the classical one, where an instruction is executed in each cycle. The performance indexes show how many times faster a given algorithm is executed on an optimized CREC system than in the case of classical execution flow (with one instruction executed *per* cycle). The efficiency of the CREC system is obvious for all kinds of algorithms. Higher improvements are obtained for DSP-like applications.

Algorithm	CREC Optimized		CREC architecture
	Worst	Best	
Bresenham's line	3.00	3.00	CREC-10
Bresenham's circle	4.08	4.60	CREC-12
Bubble sort	1.12	1.67	CREC-3
Quick sort	2.66	3.00	CREC-5
Map coloring	1.53	1.70	CREC-8
Integer square root	1.61	1.64	CREC-4

Table 1. Performance evaluation of the CREC system

The hardware part of the CREC system has been implemented in some earlier versions (with a smaller, less complex instruction set and less architectural optimizations). The targeted FPGA device was a Xilinx VirtexE 600. The physical implementation was made on a Nallatech Strathnuey© + Ballyderl© board, containing a Virtex 600E FPGA. Several CREC architectures were simulated and downloaded in this development system. A preliminary version of the CREC system having 4 EUs (CREC-4) with 4-bits wide registers occupies 4% of the CLBs and 5% of the BlockRAMs in the VirtexE600. A CREC architecture having 4 EUs with 16-bits wide registers occupies 18% of the CLBs and 20% of the BlockRAMs in the same device. The operating clock frequency is of 100 MHz.

6 Conclusions and Further Work

In this paper we presented CREC, a general-purpose low-cost reconfigurable computer that combines hardware and software CoDesign. CREC exploits the intrinsic parallelism present in many low-level applications by generating the best-suited hardware for implementing a specific application. In this way, there will be a different hardware architecture for each application. Then, this architecture is downloaded into a FPGA device, and then the application is executed by it. CREC structure is basically composed of two main parts: the parallel compiler and the hardware architecture.

The CREC computing system has been proven to be a very effective, low-cost reconfigurable architecture, allowing the execution of parallel applications with considerable performance improvements, not only for DSP-like algorithms, but also for all kinds of applications. The main CREC advantages are:

- All the advantages of RISC processors and RCs are available for CREC too;
- Instructions execution is done in parallel;
- The instruction set can be modified according to the desired application;
- Space utilization in the FPGA chip is optimal;
- The architecture is also optimized for speed, scalable and parameterizable;
- The FETCH operation is much faster due to the use of the Instructions Memory;
- No *branch-prediction* optimizations are necessary;

- The size of the Instructions Memory is flexible and can be adjusted according to the complexity of the operations that the application has to perform;
- The whole system is portable (it is an *intellectual property system*).

Further research will consist, in the first phase, of extending the functionalities of the parallel compiler, then to develop the high-level compiler for CREC applications. In the future, CREC will be used to perform more research on hardware distributed computing, using the FPGAs configuration over the Internet (application that is already implemented and tested). Another usage of CREC systems will be for developing and testing new parallel algorithms for hardware distributed computing.

Acknowledgements

This research was supported by the Romanian Ministry of Research, Education and Youth, National University Research Council, under the AT grant, theme 140 / 2003.

References

[1]. DeHon A. “Reconfigurable Architectures for General-Purpose Computing”. PhD. Thesis. Massachusetts Institute of Technology (1996)

[2] Creț O., Pusztai K., Vancea C. Szente B., “CREC: a Novel Reconfigurable Computing Design Methodology”. *Proceedings 17th International Parallel and Distributed Processing Symposium (IPDPS)*, Nice, France, (2003), 175.

[3] Creț O., Pusztai K., Vancea C., Szente B., Uiorean L., Dărăbant A.S. “A Hardware/ Software Codesign Method for General Purpose Reconfigurable Computing”. *Proceedings 14th International Conference on Control Systems and Computer Science,* Bucharest, Romania, Editura Politehnica Press, (2003), 57-63.

[4]. Singh, H. Lee, M.-H., Lu, G. Kurdahi, F. Baghrzadeh, N., Chaves Filho, E. “MorphoSys: an Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications”. *IEEE Transactions on Computers* 49(5), (2000)

[5]. Hauser J., Wawrzynek J. “Garp: a MIPS Processor with a Reconfigurable Coprocessor”. *Proceedings IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, (1997)

[6] Wolinski C., Gokhale M., McCabe K. “A Reconfigurable Computing Fabric”, Technical Report (LAUR), Los Alamos National Laboratory, (2002)

[7] Vuillemin J., Bertin P. et al. “Programmable Active Memories: Reconfigurable Systems Come of Age”, *IEEE Transactions on VLSI Systems* 4(1):56–69 (1996)

[8] Green C., Franklin P. “RaPiD – Reconfigurable Pipelined Datapath”. In R. W. Hartenstein and M. Glesner (eds.), *Field-Programmable Logic: Smart Applications, New Paradigms, and Compilers. Proceedings 6th International Workshop on Field-Programmable Logic and Applications*, Darmstadt, Germany, (1996) 126-135

[9] Waingold E., Taylor M. et al. “Baring it all to Software: Raw Machines”. *IEEE Computer*, pp.86–93, (1997)

[10] Bernstein D., Rodeh M. “Global Instruction Scheduling for Superscalar Machines” *Proceedings ACM SIGPLAN Conference on Programming Language Design and Implementation*, (1991) 241-255