Reconfigurable Instruction Set Processors from a Hardware/Software Perspective

Francisco Barat, *Student Member*, *IEEE*, Rudy Lauwereins, *Senior Member*, *IEEE*, and Geert Deconinck, *Senior Member*, *IEEE*

Abstract—This paper presents the design alternatives for reconfigurable instruction set processors (RISP) from a hardware/software point of view. Reconfigurable instruction set processors are programmable processors that contain reconfigurable logic in one or more of its functional units. Hardware design of such a type of processors can be split in two main tasks: the design of the reconfigurable logic and the design of the interfacing mechanisms of this logic to the rest of the processor. Among the most important design parameters are: the granularity of the reconfigurable logic, the structure of the configuration memory, the instruction encoding format, and the type of instructions supported. On the software side, code generation tools require new techniques to cope with the reconfigurability of the processor. Aside from traditional techniques, code generation requires the creation and evaluation of new reconfigurable instructions and the selection of instructions to minimize reconfiguration time. The most important design alternative on the software side is the degree of automatization present in the code generation tools.

Index Terms—Reconfigurable instruction set processor overview, reconfigurable logic, microprocessor, compiler.

1 Introduction

Embedded systems today are composed of many hardware and software components interacting with each other. The balance between these components will determine the success of the system. Due to the nature of software, software components are easier to modify than the hardware ones. Thanks to this flexibility, software components, running on programmable processors, provide an easy way to eliminate bugs, to change the application, to reuse components, to differentiate a product, or to reduce the ever more important time to market. However, when compared to pure hardware solutions, software components are slower and consume more power. Hardware components are used when speed and power consumption are critical. Unfortunately, hardware components require a lengthy and expensive design process. Additionally, typical hardware components cannot be modified after they have been manufactured. The task of the system designer is to find an adequate balance between these components, which interact very closely.

The interaction between software and hardware components is very tight, especially in the case of embedded systems, where cost efficient solutions are a must. Application specific instruction set processors (ASIPs) and reconfigurable instruction set processors (RISPs) are a fine example of the interaction of the hardware and software components since they contain specialized hardware and software components interacting very closely. Traditional programmable processors do not have this type of interaction due to the generic nature of the processors. As the processor becomes more specialized, so does the interaction between hardware and software. Designing a system with such a type of processors requires a methodology that encompasses both software and hardware aspects.

An ASIP [1] is a hybrid between a programmable processor and custom logic. Noncritical parts of the application are implemented using the standard instruction set (with the more traditional functional units of the processor). Critical parts are implemented using a specialized instruction set, typically using one or more specialized functional units custom designed for the application. These functional units, thus, implement a specialized instruction set. This specialization permits reduced code size, reduced power consumption, and/or higher processing power. Implementing an application on an ASIP involves the design of the custom functional units and the mapping of the software algorithms to this custom units. Both must be done concurrently to obtain a good solution, but due to the nature of hardware and software development this is not always possible.

A reconfigurable instruction set processor (RISP), the topic of this paper, consists of a microprocessor core that has been extended with reconfigurable logic. It is similar to an ASIP but instead of specialized functional units, it contains reconfigurable functional units. The reconfigurable functional units provide the adaptation of the processor to the application, while the processor core provides software programmability. Fig. 1 presents the floor plan of a hypothetical RISP. RISPs execute instructions, just as normal processors and ASIPs, though the main difference is that the instruction set of a RISP is divided in two sets: 1) the fixed instruction set, which is implemented in fixed hardware, and 2) the reconfigurable instruction set, which is

- *F. Barat and G. Deconinck are with the Department of Electrical Engineering, K.U.Leuven, Kasteelpark Arenberg 10, Leuven-Heverlee 3001, Belgium.*
E-mail: f-barat@ieee.org, Geert.Deconinck@esat.kuleuven.ac.be.
- *R. Lauwereins is with Imec, Kapeldreef 75, Leuven-Heverlee 3001, Belgium. E-mail: lauwerei@imec.be.*

Manuscript received 1 Oct. 2000; revised 1 July 2001; accepted 1 Jan. 2002.
Recommended for acceptance by Luqi.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 116426.

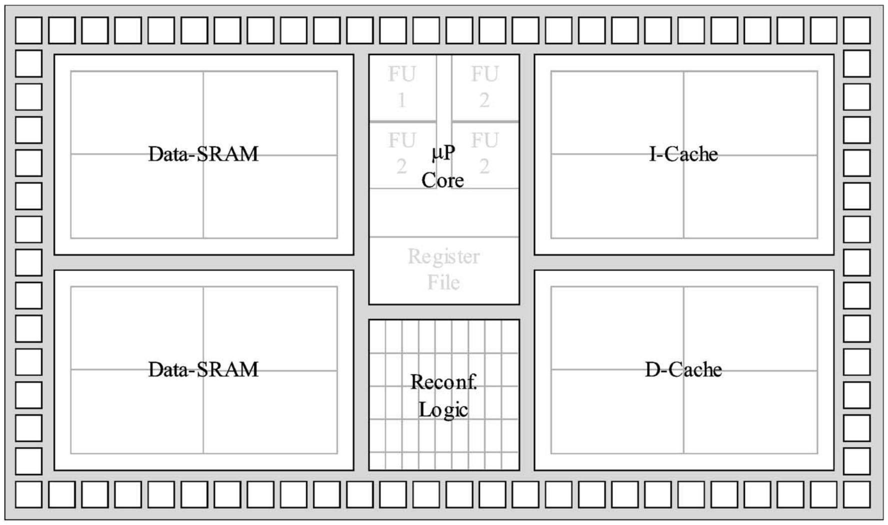

Fig. 1. Example floor plan of a reconfigurable instruction set processor.

implemented in the reconfigurable logic and can change during runtime. This reconfigurable instruction set is equivalent to the specialized instruction set of the ASIP but with the ability to be modified after the processor has been manufactured.

The cost of designing a microprocessor core is reduced by reusing it in many different applications. In ASIPs, the cost for each new generation comes from the redesign of the specialized functional units, which can be quite high. Rapid prototyping techniques are essential for reducing the cost and time required for the redesign. A RISP can be used as a prototyping device for a family of ASIP processors that share the same basic core or fixed instruction set. In this sense, it can be viewed as a field programmable gate array (FPGA) specialized for ASIP design. RISPs provide a better prototyping platform than a pure FPGA for a set of ASIP cores thanks to the specialized elements it contains. As can be seen from Fig. 1, a RISP already contains data and program memory, one or more register files, control logic, and other processor specific elements. These elements are optimized for a processor implementation, which is not the case for FPGA technology. RISP programming tools can also help in the design of ASIPs. New specialized functional units must be designed for an ASIP, as we will see later in Section 3, RISP programming tools should do this automatically. These tools can also be used to instantiate an ASIP, which can be viewed as the nonreconfigurable version of a RISP.

RISPs do not only provide benefits in the prototyping field. In fact, the first models of a product can use a RISP instead of a more expensive ASIP. As the product matures, transition to an ASIP can be done in a convenient manner. This allows the product to evolve without a significant risk and also allows a smaller time to market. We can observe a similar trend in the ASIC and FPGA markets. Additionally, in many cases, it is now more cost effective to use FPGAs instead of ASICs. Furthermore, in some applications, RISPs would be the processor of choice. Evolving standards, unknown applications, and a broad diversity of algorithms are cases where a fixed solution will eventually fail to deliver the required performance. Evolving standards and unknown applications make it very difficult to create specialized hardware for them. Applications with a broad diversity of algorithms require much specialized hardware which may be more expensive than a reconfigurable solution. RISPs offer the flexibility that ASIPs lack.

This paper will describe the design space of RISPs and discuss the problems associated with each design alternative, both hardware and software, and the interactions between them. As with modern microprocessors, RISP cannot be designed focusing on the hardware alone. The success of RISP depends on the quality of the software development tools available. As will be seen, RISPs are not easy to program without adequate tools. Without them, the programmer has not only to program the sequence of instructions the processor will execute, but also the instructions themselves. The hardware design of instructions is a new concept for programmers that can be simplified with tools.

This paper is divided in two main sections. Section 2 describes the hardware aspects of RISPs, while Section 3 presents software techniques that should be incorporated in the programming flow of RISP. Finally, Section 4 gives the conclusions.

2 HARDWARE DESIGN

The hardware design of a reconfigurable processor can be divided in two main topics. The first one is the interfacing between the microprocessor and the reconfigurable logic. This includes all the issues related to how data is transferred to and from the reconfigurable logic, as well as synchronization between the two elements. The second topic is the design of the reconfigurable logic itself. Granularity, reconfigurability, and interconnection are issues included in this topic. Table 1 summarizes the key design issues related to the characteristics of some current reconfigurable processors.

TABLE 1
Characteristics of Some Reconfigurable Processors

	Splash-2 [2]	PRISM-I [3]	PRISM-II [4]	Nano Processor [5]	PRISC [6]	DISC [7]	OneChip [8]	Garp [9]	Chimaera [10]	NAPA [11]	Remarc [12]	OneChip98 [13]	PipeRench [14]
Year	1992	1993	1993	1994	1994	1995	1996	1997	1997	1998	1998	1999	1999
Processor type	Sparc	M68010	Am29050	RISC	R2000	CISC	DLX	MIPS	MIPS	RISC	RISC	S-DLX	Generic
Application type	Streaming	General purpose	General purpose	General purpose	General purpose	General purpose	General purpose	General purpose	Multimedia	General purpose	Multimedia	General purpose	Multimedia
Coupling	Attach	Attach	Copro	RFU	RFU	RFU	RFU	Copro	RFU	Copro	Copr	RFU	Attach
Inst. types	All	All	All	Custom	Custom	All	All	Stream	Custom	All	Custom	Stream	Stream
Duration	Variable	Variable	Variable	Fixed	Fixed	Variable	Variable	Variable	Fixed	Variable	Variable	Variable	Variable
Inst. coding	N/A	Fixed	Fixed	Fixed	Fixed	Fixed	Fixed	Fixed	Fixed	Fixed	Fixed	Fixed	Fixed
Configuration table	No	No	No	No	No	Yes	No	Yes	Yes	No	No	Yes	
Operand coding	N/A	Fixed	Fixed	Fixed	Fixed	Flexible	Fixed	Fixed	Hardwired	Fixed	Fixed	Fixed	Fixed
Shared register file	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes	No
Memory ports	Yes (1)	No	No	Yes (1)	No	Yes (1)	Yes (1)	Yes (1)	No	Yes (2)	No	Yes (1)	Yes (1)
Granularity	Fine	Fine	Fine	Fine	Fine	Fine	Fine	Fine	Fine	Fine	Coarse	Fine	Coarse
Reconfigurable logic Type	Xilinx 4010	Xilinx 3090	Xilinx 4010	Xilinx 3000	Custom	National CLAy31	Xilinx 4010	Custom	Custom	Custom	Custom	Custom	Custom
Dynamically reconfigurable	No	No	No	No	Yes	Partial	No	Yes	Yes	Yes	Yes	Yes	Yes
Configuration Method	Software	Software	Software	Software	Software	Software	Software	Software	Controller	Controller	Controller	Controller	Controller
RFU blocked during reconfiguration	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	No	Yes	Yes	No	No
RFU segmented	No	No	No	No	No	Yes	No	Yes	Yes	Yes	No	No	Yes
RFU with multiple contexts	No	No	No	No	No	No	No	No	No	No	Yes	No	Yes
Can support prefetching?	No	No	No	No	No	No	No	No	No	No	No	Yes	Yes
Relocatable hardware	No	No	No	No	No	Yes	No	Yes	Yes	Yes	No	No	Yes
Instruction caching	No	No	No	No	No	Yes	No	Yes	Yes	No	No	Yes	No
Max. documented speedup	¿200	50	86	240	1.91	23.5	40	43	2.06	N/A	21.2	32	189

This paper focuses on the design of the reconfigurable part of the processor. For a discussion of the design issues related to the nonreconfigurable part of the processors see [1] and [15].

2.1 Interfacing of the Reconfigurable Unit

In this section, the different methods in which the processor communicates with the reconfigurable logic are discussed. The reconfigurable logic will be seen as a piece of hardware in which any circuit of interest to the application domain can be implemented. The internal structure will not be the focus of this section, it will be discussed in Section 2.2.

2.1.1 Coupling the Processor to the Reconfigurable Logic

Reconfigurable instruction set processors belong to the family of reconfigurable processors. Even though it is not the topic of this paper, it is interesting to provide a

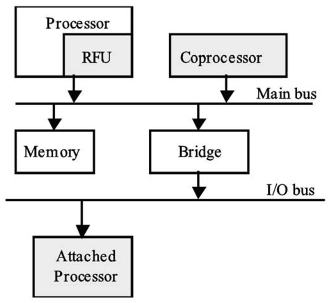

Fig. 2. Coupling schemes of reconfigurable processors (RFU = reconfigurable functional unit).

taxonomy of reconfigurable processors according to the coupling of the reconfigurable logic to the processor. Reconfigurable processors consist of reconfigurable logic coupled with a microprocessor. The position of the reconfigurable logic relative to the microprocessor directly affects the performance of the system and the type of applications that will benefit from the reconfigurable hardware. The benefit obtained from executing a piece of code in the reconfigurable logic depends on two aspects: communication time and execution time [8]. The time needed to execute an operation is the sum of the time needed to transfer the data back and forth, and the time required to process it. If this total time is smaller than the time it would normally take using the processor only, then an improvement is obtained. If the reconfigurable logic was not yet configured to perform that particular instruction, it would also be necessary to add the configuration time.

The reconfigurable hardware can be placed in three main positions relative to the processor [14], as can be seen in Fig. 2. These configurations are:

- **Attached processor**. The reconfigurable logic is placed on some kind of I/O bus (e.g., PCI bus). Example: PRISM-1 [3].
- **Coprocessor**. The logic is placed next to the processor. The communication is done using a protocol similar to the one used for floating point coprocessors. Example: Garp [9].
- **Reconfigurable functional unit (RFU) or Reconfigurable instruction set processor (RISP)**. The reconfigurable logic is placed inside the processor. The instruction decoder issues instructions to the reconfigurable unit as if it were one of the standard functional units of the processor. Example: OneChip98 [13].

With the first two coupling schemes, sometimes called loosely coupled, the speed improvement using the reconfigurable logic has to compensate for the overhead of transferring the data. For example, in PRISM-I, the round trip communication cost is 50 cycles. In streaming applications where a continuous stream of data has to be processed, this communication cost can be hidden by pipelining the transfer and processing of data. Through pipelining, it is possible to achieve high data rates, such as 100MB/s in Splash 2 [2].

Most systems built until recently were of this kind. The main advantages of this approach are: 1) the ease of constructing such a system using standard components (such as Xilinx FPGAs on Splash 2 and PRISM), 2) the fact that large amounts of reconfigurable hardware can be used (the only limitation is the board size), and 3) the possibility of having the microprocessor and the reconfigurable logic working in different tasks at the same time (though this requires complex programming).

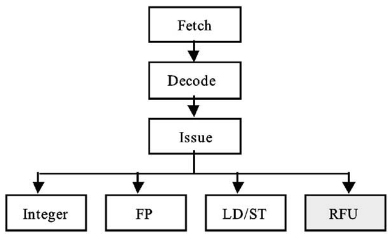

Fig. 3. Pipeline of a RISP.

With the integrated functional unit scheme, sometimes called tightly coupled scheme, the communication costs are practically nonexistant and, as a result, it is easier to obtain an increased speed in a wider range of applications. Unfortunately, design costs for this configuration are higher since it is not possible to use standard components. The amount of reconfigurable hardware is also limited to what can fit inside a chip, which limits the speed increase. We will call this functional unit a reconfigurable functional unit (RFU) and a processor with an RFU, a reconfigurable instruction set processor (RISP).

A reconfigurable instruction set processor can have one or more RFUs. As a normal functional unit, an RFU executes instructions that come from the standard instruction flow. The pipeline of a typical RISP may look like the one in Fig. 3. As mentioned before, this paper is only focused on this type of coupling. Nonetheless, the coprocessor approach is very similar to the RISP approach and, therefore, most of the ideas discussed in this paper also apply to it.

When reconfigurable logic is placed inside the processor, it does not have to be placed always inside a functional unit. Other places of interest are the I/O interface, the decoding logic, or the control logic. By placing the reconfigurable logic next to the I/O pins, it is possible to build custom interfaces to other elements in the system, thus eliminating the need of external glue logic [11].

2.1.2 *Instruction Types*

The design of the interface to the reconfigurable unit depends on the characteristics of the instruction types that are going to be implemented. Two main types of instructions can be implemented on an RFU [14]:

- **Stream based instructions (or block based instructions)**. They process large amounts of data in sequence or by blocks. Only a small set of applications can benefit from this type. Most of them are suitable for a coprocessor approach. Examples: finite impulse response (FIR) filtering and discrete cosine transform (DCT).
- **Custom instructions**. These instructions take small amounts of data at a time (usually from internal

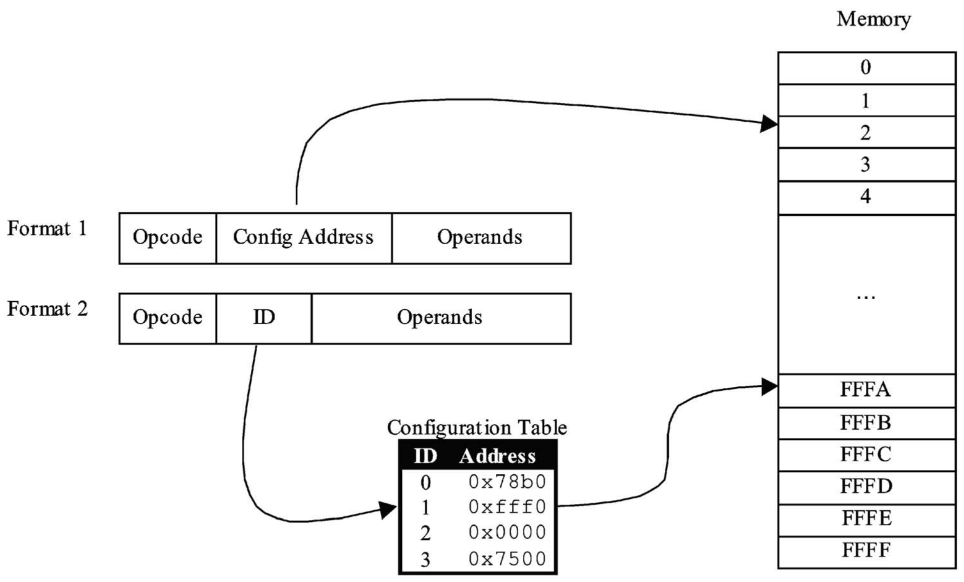

Fig. 4. Fixed instruction selection formats.

registers) and produce another small amount of data. These instructions can be used in almost all applications as they impose fewer restrictions on the characteristics of the application; however, the speedup obtained is usually smaller. Example: bit reversal, multiply accumulate (MAC), variable length coding (VLC), and decoding (VLD).

Instructions can also be classified in many other ways, such as whether the instructions have fixed execution time or not, whether they can be pipelined or not, whether they can have internal state, etc. If the type of the reconfigurable instructions closely resembles the type of the fixed instructions supported by the microprocessor, the integration process will be easier. The type of instructions supported depends on the target application domain.

2.1.3 Instruction Coding

Reconfigurable instructions are usually identified by a special opcode. Which reconfigurable instruction is executed is specified using an extra field in the instruction word, the reconfigurable instruction number. This extra field can specify:

- The memory address of the configuration string for the instruction. Example: DISC [7].
- An instruction identifier of small length that indexes a configuration table where information, such as the configuration string address, is stored. Example: OneChip98 [13].

Fig. 4 shows these two formats. The first approach usually needs more instruction word bits but has the benefit that the number of different instructions is not limited by the size of a table, as in the second case. More instructions than those that fit in the configuration table can be used if the contents of table can be changed at runtime. The drawback of this approach is that specialized scheduling techniques might need to be used during code generation. The compiler will have to schedule during the lifetime of the program what instructions are stored in the configuration table. Nonetheless, in typical applications only a fraction of the possible instructions are used (typically 200 out of 2,048 instructions in PRISC [6]).

2.1.4 Operands

The instruction word also specifies the operands to be passed to the RFU. The operands can be immediate values, addresses, registers, etc. They can be the source or destination of the operation. There are several ways to code them:

- **Hardwired**. The contents of all registers are sent to the RFU. The registers actually used depend on the hardware configured inside the RFU. This allows the RFU to access more registers but makes code generation more difficult. The actual selection of registers is done inside the RFU. This is the approach taken in Chimaera [10], where the eight registers from the register file can be accessed simultaneously.
- **Fixed**. The operands are in fixed positions in the instruction word and are of fixed types. Different encoding formats would have different opcodes. This the most common case as seen in Table 1. Example: OneChip98 [13].
- **Flexible**. The position of the operands is configurable. The degree of configuration can be very broad. If a configuration table is used, it can be used to specify the decoding of the operands. Example: DISC [7].

The register file accessed by the RFU can be shared with other functional units (such as the integer functional unit) or dedicated (such as the floating point register file in some architectures). The dedicated register file needs fewer ports than if it was shared. This simplifies its design but complicates code generation because of register heterogeneity.

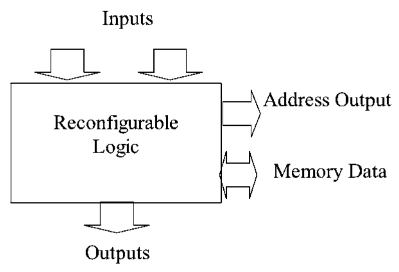

Fig. 5. A reconfigurable functional unit with memory access.

As can be seen from Table 1, most current reconfigurable processors use the same register file for fixed and reconfigurable instructions. This will most likely change when more research is done on reconfigurable superscalar and very long instruction word processors (VLIWs). Splitting the register file allows to reduce the complexity of the hardware by increasing the complexity of the compiler.

2.1.5 Memory Access

By providing access from the RFU to the memory hierarchy, it is possible to implement specialized load/store operations or stream-based operations. In many multimedia applications, data is accessed through complex addressing schemes [16]. Specialized instructions would make these accesses more efficient.

There are two main techniques to implement access to memory on RFUs:

- **The RFU as an address generator**. The RFU logic is used to generate the address of the memory position being accessed. This address is fed into the standard LD/ST unit.
- **The RFU has direct connections to the memory buses**. In this case, the RFU generates the address bits and reads or writes the data from/into the memory. With this approach, it is possible to process the data that is being accessed through the memory.

Fig. 5 presents an RFU with direct access to memory. If the RFU can access memory, it is important to maintain consistency between the RFU accesses and the processor accesses [13] (i.e., caches, prefetch buffers).

2.2 Reconfigurable Logic Design

The focus of this section is the design of the reconfigurable logic itself. The design of the reconfigurable logic will determine, among other things, the number of instructions that can be stored, the size of the configuration stream, the reconfiguration time, and the type of instructions that will give the highest performance.

Reconfigurable logic is composed of three layers: processing layer, interconnect layer, and configuration layer. Fig. 6 presents these layers. The processing layer contains the actual processing elements. The interconnect layer consists of programmable interconnections between the processing elements. Finally, the configuration layer contains memory elements in which the configuration for the other two layers is stored. In current FPGAs, the interconnect layer uses 10 times more area than the configuration layer, which, in turn, uses 10 times more area than the processing layer.

2.2.1 Processing Layer

The processing layer contains the elements that actually perform the calculations. In some cases, the interconnect layer performs the desired operation, such as in shifting operations. The main characteristic of these elements is their granularity (or size), which is normally classified as fine- or coarse-grained.

The building blocks for fine-grained logic are gates (efficient for bit manipulation operations). They are implemented using logic blocks normally composed of lookup tables (LUTs), flip flops, and multiplexers [17].

In coarse-grained RFUs, the blocks are bigger and operate simultaneously on wider buses. They are better suited for bit parallel operations, typically 4 or 8 bits. In many cases, the building blocks are complete ALUs, multipliers, shifters, and the like ([18], [19]).

Granularity directly affects the size of the configuration stream and the configuration time. With fine-grained logic, more information is needed to describe the instruction (e.g., 49,152 bits for a single context in Garp [9]). Coarse-grained logic descriptions are more compact (e.g., 2,048 bits for a single context in Remarc [12]).

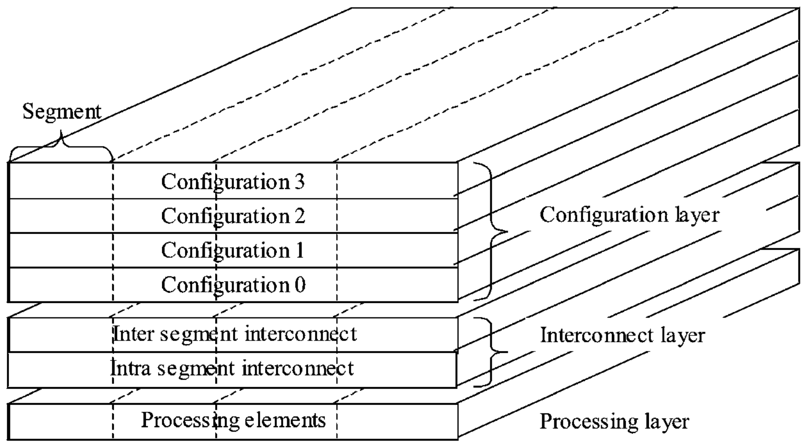

Fig. 6. Layers in the reconfigurable logic of an RFU.

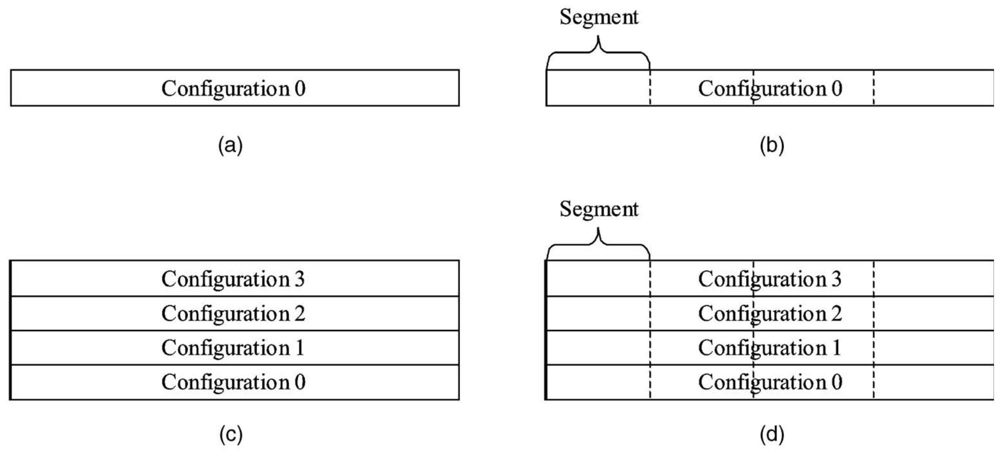

Fig. 7. Different configuration layer options: (a) single context, (b) segmented, (c) multiple context, (d) segmented with multiple contexts.

The optimal granularity depends on the application. When the size of the application data is smaller than that of the logic, the performance decreases due to unused resources. When the application data is bigger than that of the logic, the performance decreases due to an overhead in reconfigurability and an increase in configuration stream size. For bit-oriented algorithms, a fine-grained approach is a better choice if the cost of the higher configuration stream can be tolerated. For computation intensive applications, the coarse-grain approach can be a better solution [12].

As can be seen from Table 1, most of the early designs included standard fine-grained FPGA resources (e.g., DISC [7], OneChip [8]). In several of the most modern approaches, the RFU is implemented with nonstandard cell structures (e.g., Garp [9], Chimaera [10]). Many are still fine-grained, though a trend towards coarse-grained architectures can be seen (PipeRench [14], REMARC [12]).

2.2.2 Configuration Layer

The configuration layer is used to store the current configuration of the RFU. It is normally made of SRAM cells. These cells control the behavior of the processing elements and the programmable interconnect. If a change in the behavior of the hardware were needed, a different configuration string would need to be stored.

Some systems, which are called one-time configurable, can only be configured at startup. In this type of RFU (e.g., Nano Processor [5]), the total number of special instructions is limited by the size of the reconfigurable logic. This is equivalent to an ASIP that is customized after being committed to silicon.

If the RFU can be configured after initialization, the RFU is dynamically reconfigurable. In this case, the instruction set can be bigger than the size allowed by the reconfigurable logic. If we divide the application into functionally different blocks, the RFU can be reconfigured for the needs of each individual block. In this manner, the instruction adaptation is done on a per block basis. Most of the reconfigurable processors belong to this kind.

The time taken to reconfigure the logic depends on the size of the configuration string, which itself depends on the granularity of the processing elements as already mentioned in Section 2.2.1. On older systems, all the reconfigurable logic is reconfigured at a time, which leads to high reconfiguration times (i.e., one second in PRISM-I). Modern systems use configuration compression, multiple contexts, and partial reconfiguration to reduce this time (down to one cycle).

In multiple context RFUs, the configuration layer is replicated several times. Since only one context is active at a given time, new configuration data can be loaded to the other contexts. If the configuration data is already loaded, changing the active context is a very fast operation (usually one clock cycle). As the configuration layer typically takes 10 percent of the area, doubling the number of contexts normally means an extra 10 percent area [20].

Since reconfiguring the RFU can take some time, prefetching the instruction configuration data can reduce the time the processor is stalled waiting for reconfiguration. Software tools should do the insertion of prefetching instructions automatically (more in Section 3.3).

Partial reconfiguration is another technique to minimize reconfiguration time and maximize logic usage by reconfiguring a small part of the configuration layer. This is usually done by dividing the RFU in segments (or stripes) that can be configured independently from each other. Segments are the minimum hardware unit that can be configured and assigned to an instruction. They allow a better utilization of the reconfigurable logic by adapting the size of the used logic to the size of the instruction. Segments and contexts can be combined on the same processor. This leads to the configuration layers in Fig. 7.

2.2.3 Interconnect Layer

The interconnect layer connects the processing elements to obtain the desired functionality. The interconnect that connects the elements inside a segment is referred to as intrasegment interconnect. Intersegment interconnect is used to connect different segments.

Intrasegment interconnect design depends on the granularity of the logic and also affects the speed. In fine-grained RFUs, there are different levels of intrasegment interconnect, just like in FPGAs. Local routing connects neighboring elements and global routing, which has longer delays but reaches a larger number of elements, connects distant elements [17]. In coarse-grained architectures, the

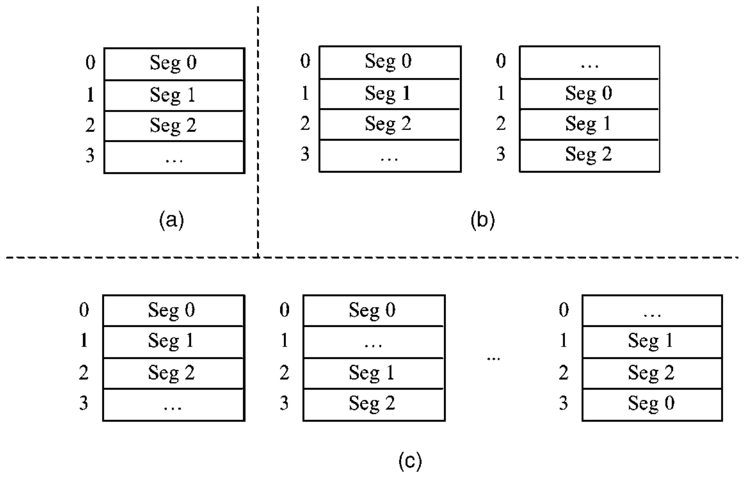

Fig. 8. Intersegment interconnection schemes: (a) fixed, (b) relative, (c) relocatable.

interconnect tends to be done using buses and crossbar switches, which optimize the interchange of parallel data lines [12].

Intersegment interconnect only appears in RFUs that support multiple segments. It is used to transmit data between the different segments and to the register files and memory. There are several kinds of intersegment inteconnect:

- **Fixed**. The position of the segments of an instruction inside the RFU is fixed at compile time.
- **Relative**. The position of the instruction's segments is specified relative to each other. This gives a little slack to position the instruction inside the RFU.
- **Relocatable**. The position of the segments is not fixed at all. They can be placed anywhere on the RFU.

Fig. 8 shows various different configuration options for an instruction of three segments (Seg0, Seg1, and Seg2) and an RFU of four segments. With fixed interconnect, there is only one position where the instruction can be placed (all close together and starting in segment 0). There are two possibilities with relative interconnect if the segments must be kept together. Finally, with relocatable interconnect, there are many possible configurations (only three are presented in the figure).

The type of interconnect determines the complexity and size of the interconnect, the size of the configuration stream used for interconnect description, and the complexity of the code generation tools. From a hardware point of view, fixed interconnect is the simplest and requires the least configuration bits. Relative interconnect is very similar to fixed interconnect in the sense that no extra logic is required and that the size of the configuration stream is very similar. Relocatable interconnect is the most expensive one in terms of area and power. From a software point of view, the best would be relocatable interconnect, since it simplifies the placement task.

2.3 Configuration Controller

Reconfiguration times not only depend on the size of the configuration data, which can be quite large, but also on the configuration method used. In the PRISC processor [6], the RFU is configured by copying the configuration data directly into the configuration memory using normal load operations. If this task is performed by a configuration controller that is able to fetch the configuration data while the processor is executing code, a performance gain can be obtained. By modifying the configuration interface from bit serial to bit parallel, the configuration speed can be easily increased an order of magnitude.

The configuration controller is in charge of managing and loading the instructions that are active on the RFU. It can be hardware or software based. In the simplest case, the one-time configurable RISP, it would read the configuration from some external source once (e.g., ROM), in a manner similar to conventional FPGAs. Since this process only happens once, speed is not an issue.

Fig. 9 presents a complex RFU with a configuration controller. The configuration controller can read new configuration data from its memory port to speed up reconfiguration. As configuration data is normally contiguous in memory, the configuration controller can access external memory using fast memory access modes designed for high throughput. As the reconfigurable logic is divided into segments, this reconfiguration process can be done in parallel to the computations. In many cases, reconfiguration time can easily take more than twice the processing time (three times in [7] while still obtaining a total speedup of more than 20).

A hardware cache can be used to improve reconfiguration times. Each time a reconfigurable instruction is executed, a configuration table is checked to see if the instruction is already configured in the RFU. If the instruction is already configured, it is executed; if it is not configured, the configuration controller loads the configuration data automatically, replacing some segments of the reconfigurable fabric (usually, the least recently used). This is the approach taken in [7]. Hardware caching can be improved by using multiple contexts in the configuration layer and by having relocatable interconnect and homogeneous segments.

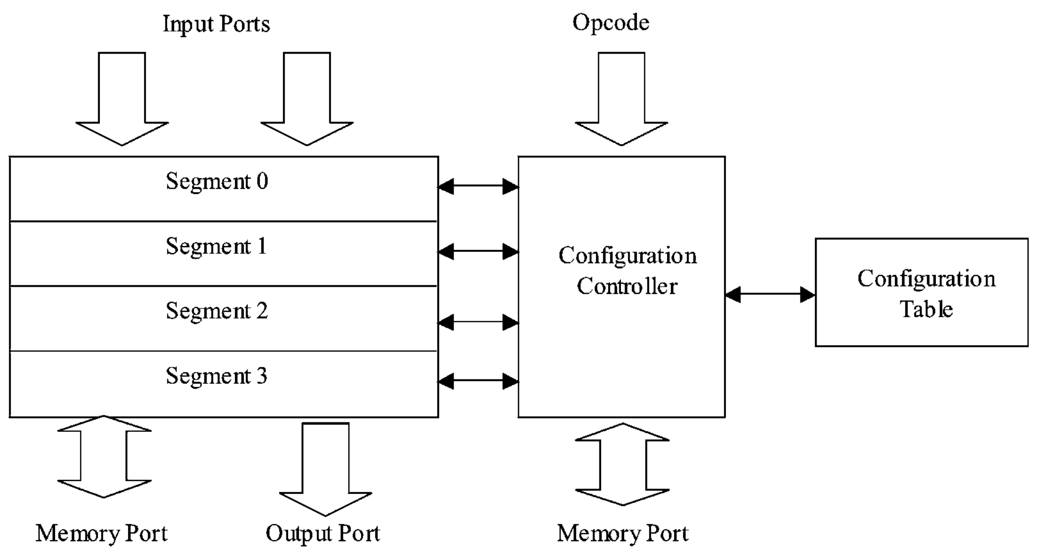

Fig. 9. Detailed view of a reconfigurable functional unit.

2.4 Interrupts

A major problem present with reconfigurable instructions is that of interrupt processing. Interrupts have to be precise [21]; that is, interrupts should not affect the behavior of the code that is being executed. If reconfigurable instructions only affect registers in the register file, the problem is simplified. On the other hand, reconfigurable instructions that have internal state or write to memory are more difficult to deal with.

If the interrupt is external to the instruction, one approach can be to ignore the interrupt until the instruction is completely finished. Otherwise, the internal state of all RFUs would have to be stored somewhere. VLIW and superscalar processors have the same problem, but, in RISP, it is exacerbated due to the bigger size of the internal state in the RFU.

If the processor is going to be used under a multitasking environment, the internal state of the RFU and which instructions are configured should be stored between context switches. Granting exclusive access to the RFU to one of the processes or tasks can solve these problems [22].

3 Software Tools

The main difference between RISP and standard processors is that the instruction set of RISP changes at runtime; it is not fixed at design time. This fact makes it necessary to change the manner in which code for RISP is generated. Code generation for a RISP involves code generation techniques and hardware design techniques. This process is similar to hardware/software codesign, in which an application is divided into hardware and software components. In the case of RISPs, the entire application is executed in software but with the aid of special instructions implemented in the reconfigurable hardware that must be designed specifically for the application.

Many RISP features depend on the availability of good programming tools in order to obtain good results [23]. More importantly, it is necessary to design the processor and the programming tools simultaneously and check regularly that one does not impose severe restrictions on the other. In this section, we will discuss the main issues that appear in the design of code generation tools for RISP.

The basic element of a RISP code generation suite is a high-level language (HLL) compiler that performs automatic hardware/software partitioning. In some cases, such a complex compiler might not be needed or might not be feasible, but the concepts presented will usually appear in some form. Fig. 10 represents the stages of a generic compiler organization for a RISP ([3], [24], [25]). This figure will be the reference to discuss the problems encountered when writing code for a RISP. White blocks represent traditional compiler blocks and tools. Gray areas represent new techniques for RISP and black areas represent "traditional" hardware synthesis techniques.

This compiler structure can be modified as needed. For example, if there is a significant binary code base that has to be translated to a RISP processor, the front-end of the compiler can be modified to include a parsing mechanism for assembly files. Some systems use this approach since it simplifies the process of creating a compiler for the processor. A normal compiler can be used and then the code is modified to include reconfigurability. This is done, for example, in Concise [32].

The compilation process is quite complex and introduces many new concepts to compiler design. The initial compilation phases are typical of HLL compilers. The source code is analyzed and an internal representation is obtained, normally in the form of some sort of control and data flow graph. The graph is composed of basic blocks, hyper blocks, regions or any other construct that facilitates compilation. From now on, they will be referred to as blocks. The size of the blocks depends on the processor architecture. High-level optimizing transformations are also done during these stages.

Table 2 presents a list of compilers for RISP and their main characteristics. As can be seen, not all phases are implemented in all compilers. The most automated code generation tools have most of the phases outlined in Fig. 10. We will now study each of these phases in more detail.

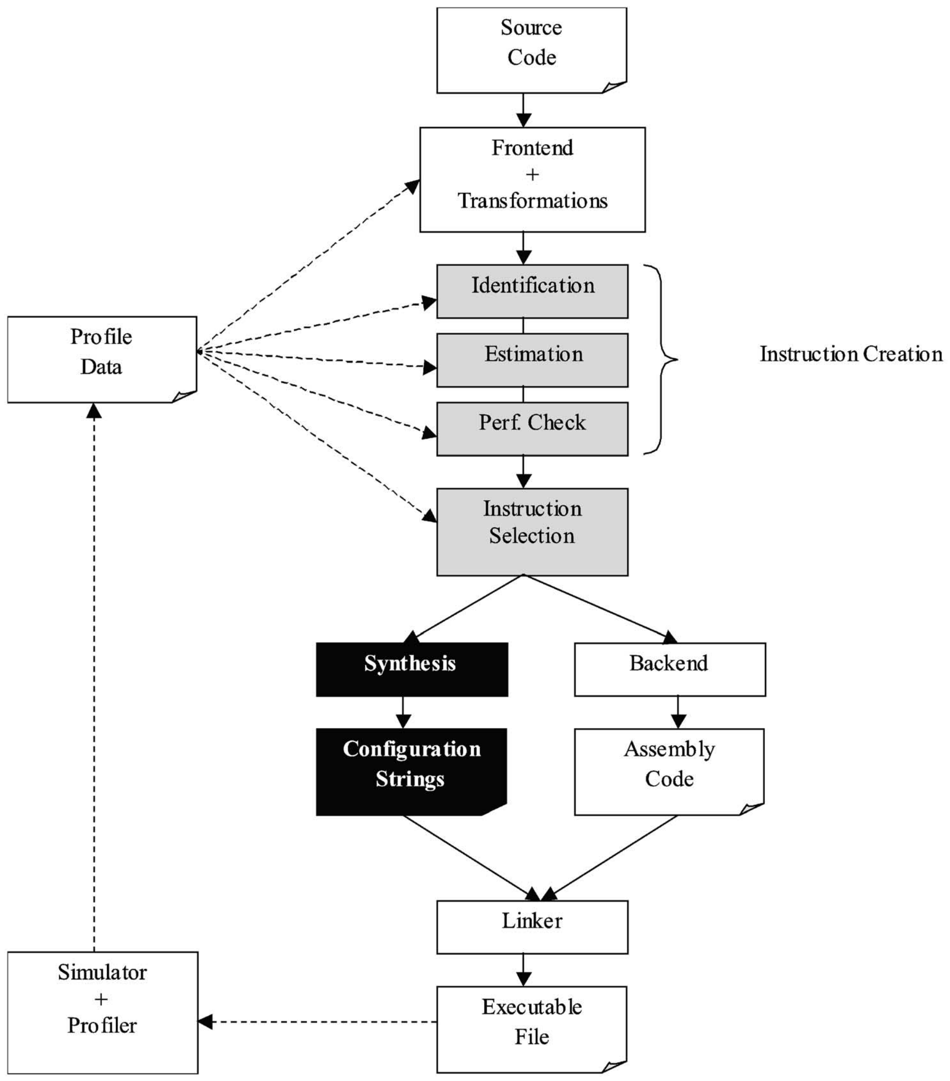

Fig. 10. Generic RISP compiler flow.

3.1 Optimization Focusing

Although not strictly a stage in the compiler, optimization focusing allows faster compilation times by filtering parts of the code so that they do not go through all of the optimization stages. These techniques are not specific to RISP but they are of special relevance to RISP compilation since reconfiguring the processor for an irrelevant part of code could have a drastical impact on performance. There are three main techniques for optimization focusing:

- **Manual identification**. The programmer annotates the code with special compiler directives to identify the places where the compiler should optimize. In C compilers, this is usually done with the pragma directive.
- **Static identification**. By analyzing the code, the compiler is able to recognize candidate code for potential optimization (e.g., loops). This approach is quite limited, since the execution profile of most programs depends on the inputs (i.e., loop bounds unknown at compile time).
- **Dynamic identification or profiling**. The code is initially compiled without optimizations and optimization potential is identified by executing code on real data (the code is profiled). This approach is the most time consuming (usually done by simulation, as seen in Fig. 10), but can achieve the best results. It is important to have a significant and relevant data set in order to get good estimates.

3.2 Instruction Creation

The purpose of the instruction creation phase is to obtain, on a per block basis, new instructions that will locally improve the performance of the processor in a block of code. Most blocks of the original source code will have been filtered by the optimization focusing presented in the previous section. Instruction creation is divided in three closely coupled subtasks:

- **Identification**. The intermediate representation is analyzed. New instructions are created by grouping operators or by performing code transformations. The result of this phase will be a description of the new instructions.

TABLE 2
Comparison of Code Generation Tools of Some Reconfigurable Processors

	PRISC [26]	DISC [27]	NAPA [28]	Garp [29], [24]	Piperench [30], [31]	Chimaera [25]
Year	1994	1996	1998	1998	1999	2000
Coupling	RFU	RFU	Copro	Copro	Attached	RFU
Language	C	C	C	C	Dataflow	C
Optimization focus	Short set vectors,FSM,Hash tables	Intrinsic (manual)	Loops	Inner loops	Manual	Inner loops
Transforms for identification	None	None	None	Hyperblock formation	None	Control localization,SWAR
Instruction identification	Profile	Intrinsic	Manual	Inner loops	Manual	MISO
Instruction identification check	Size	None	None	Loops that fit on the array	None	None
Instruction selection	None	None	None	None	None	None
Configuration scheduling	None	None	None	None	None	None
Synthesis	Special	Commercial	Marge	Special	Special	Special
Profiling info	Yes	No	No	No	No	No
Based on compiler	Mips	Lcc	Suif	Suif	Custom	Gcc
Configuration prefetching	No	No	No	No	No	No

- **Parameter estimation**. The instruction description is processed and important parameters, like instruction latency and size, are estimated.
- **Instruction performance check**. This phase checks that the new instructions improve the execution of the block. The new instructions are kept if, for that particular block, the code with the new instructions has a higher performance than the code without them.

This instruction creation phase performs an extensive search and produces a large number of new instructions. To reduce the number of instructions output to the next stage, pruning techniques are used. Pruning is performed by the optimization focusing during identification, as already mentioned, and by the performance check.

The main output of this phase is a list of instructions per block that will be used by the backend during code generation. If the instructions are very specialized, the backend might not be able to use them directly. In this case, the data flow graph (DFG) is labeled with the usage of these new instructions (this is a form of early instruction selection).

The next sections will discuss these phases in more detail. Section 3.2.4 will present alternative approaches for instruction creation.

3.2.1 Identification

The purpose of the instruction identification phase is to find instruction candidates to be implemented in the RFU. The method used to identify the instructions depends on the intermediate representation used. It is thus very important to use the correct representation.

Automated instruction identification is complex, but desirable. It enables the acceleration of almost every program, independently of the programmer's expertise. There are two main types of automated instruction identification techniques:

- **Dataflow/general techniques**. These techniques look for patterns or groups of operations that can be combined to create a new more complex instruction.
- **Ad hoc/customized techniques**. These are specialized techniques. They look for a special construct in the application and create a new instruction specifically for that. For example, we could design a special technique to optimize switch statements in C code.

An example of a general technique is the study of the DFG for groups of instructions that can be combined into one. This usually requires that the group has a limited number of inputs and one output. These patterns, usually called MISO (multiple inputs single output), can be implemented as a single instruction instead of several instructions [25], leading to an improvement in power consumption and performance. In Fig. 11, part of a data flow graph is compressed into a single instruction with two inputs and one output.

A special case of dataflow techniques is the replication of standard instructions. If part of the application is addition intensive, several addition instructions can be implemented. This only applies to architectures with instruction level parallelism, such as superscalar and VLIW processors.

More specialized techniques can be built in an ad hoc manner. For example, switch statements can be directly implemented by an instruction. This is done in [3], where a

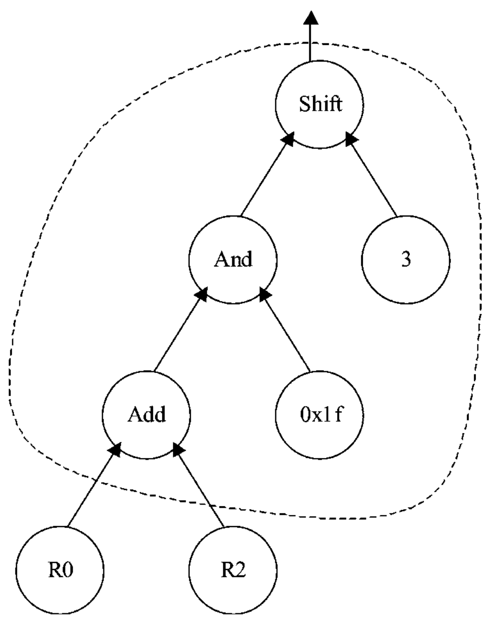

Fig. 11. Generation of a complex instruction.

set of techniques map certain C constructs onto specialized instructions. Subword parallelism is another target of specialized instructions [25]; for example, packing and unpacking instructions found in multimedia extensions can be easily mapped onto reconfigurable instructions.

Instruction identification can benefit from code transformations. A clear example is the hyper-block formation in the Garp processor [24]. These transformations can increase the instruction-level parallelism available, increasing the opportunities to find new instructions.

3.2.2 Parameter Estimation (Instruction Characterization)

Later stages in the compilation process need a model of the instruction to see whether or not it is worth using the instruction. Typical parameters are reconfiguration time, number of segments used, execution latency, and power consumption. These parameters can be precisely obtained by synthesizing the instruction. Unfortunately, synthesis is usually a time consuming task, making it undesirable to synthesize instructions that will not be used. Therefore, a fast method for estimating the working parameters of an instruction is applied.

To reduce compilation time, this phase produces estimates of the required parameters without actually doing the complete synthesis. The quality of the estimator depends on the complexity of the reconfigurable logic and on the effort used.

This estimation phase also performs a pruning of instructions. Instructions that are estimated not to fit inside the RFU are discarded.

3.2.3 Performance Check

After instructions have been identified and characterized, it has to be checked if they improve the code produced. The easiest way to do this is to compile the corresponding block with the set of instructions currently active. This compilation will give an estimate on the number of cycles (or power) consumed. This result is then compared with the general version of the block. If a decrease is obtained, the instruction is accepted. If not, it is rejected.

This phase will produce several sets of reconfigurable instructions for each block with different values of speed/code size or speed/power per set. All possible sets go to the next stages to allow for global optimization.

It is important to note that the performance measurements are done assuming that the RFU has the correct configuration. Reconfiguration times will be accounted for in the instruction selection/configuration scheduling phase.

3.2.4 Simplifications to Instruction Creation

As we have seen, instruction creation is a complex problem. For this reason, in many cases, designers have taken the choice to simplify it. There are two alternative methods for instruction creation:

- **Library programming**. Reconfigurable instructions are designed manually and placed into a library that the programmer can explicitly use.
- **Programmer identification**. The programmer indicates in the code the parts that should be implemented in hardware.

In the first method, the programmer uses reconfigurable instructions in the library just like normal functions. The compiler recognizes these function calls as instructions instead of functions and treats them as such. Although the creator of the library has to be proficient in the design of these instructions, there is no need for the programmer to know about this task. These designed instructions can be more efficient than those generated automatically, though their use is limited to what the designer intended, as occurs in ASIP design.

This approach is identical to the one used for multimedia extensions in modern microprocessors. If source code equivalent to the instruction is available, it allows the application to be compiled for a different processor, albeit with a performance penalty. It is also important to have this equivalent code in the library so that it can be used instead of the reconfigurable instruction when it is not efficient to use it in a particular case (i.e., because of reconfiguration time).

This method eliminates instruction identification and estimation since identification is done directly through the function call construct. Estimation and synthesis are already done by the library designer. The performance check can still be used, depending on the existence of code equivalent to the instruction.

The second method forces a piece of code to be treated as a new instruction. The compiler substitutes the code by its equivalent instruction. It would be a special case of optimization focusing. The compiler can still choose to create the instruction or not, depending on the obtained change in performance.

It is desirable to have these two other methods in a compiler as a means to increase the level of flexibility for the programmer.

3.3 Instruction Selection/Assignment (Configuration Scheduling)

During the instruction selection phase, the compiler chooses which of the new instructions will be used for each block. This selection is done trying to optimize the application globally, with execution time as the typical metric. There are two main cases for instruction selection: one time reconfigurable RISP or dynamically reconfigurable RISP.

3.3.1 One-Time Reconfigurable RISP

One-time reconfigurable RISPs are similar to ASIPs. They have an instruction set that is fixed during the entire application. In fact, the process of selecting which instructions are to be used is the same in both types of processors. Total execution time is minimized by a proper selection of instructions. This selection process is constrained by the number of instructions that can be implemented. In an ASIP, there is an area limit, while, in a RISP, the limit comes from the size of the RFU.

A simple technique to estimate the total execution time is based on block frequency estimates. Total execution time is the time needed to execute each block multiplied by the number of times the block is executed. Obtaining block frequency estimates is a very complex analysis that can be done using static and dynamic information.

As with optimization focusing, static analysis is performed studying the graph representation. The compiler can make estimates on the number of iterations of loops, control flow paths and several other aspects that can lead to a conservative decision. Better results can be obtained by using profiling. In this manner, a dynamic analysis is performed on the program, allowing a better estimate of block frequencies.

3.3.2 Dynamically Reconfigurable RISP

For RISPs in which reconfiguration takes place at runtime, the instruction selection process is more complex. It does not suffice to know the block frequencies, it is also necessary to know the control path taken. Instead of selecting a globally optimal instruction set, the compiler has to select locally optimal instruction sets. The locality of these instruction sets is also not defined and the compiler must find them itself, which is equivalent to scheduling the contents of the RFU along all control paths of the application.

Since the compiler cannot optimize all control paths, it has to estimate the most common path and optimize it. Profiling techniques can be used for this. This phase analyses the control graph of the program and finds an optimal schedule of instructions inside the RFU. This optimal schedule has to deal with two main aspects: 1) reconfiguring the RFU takes time and resources and 2) the performance of the code depends on what instructions are configured on the RFU. The optimal schedule will reconfigure the RFU the least number of times and the needed instructions will usually be already on the RFU [33]. In this manner, execution time and power consumption are reduced to the minimum.

In some RISP compilers, such as [25], selection is not done at all. For each block, the instructions that optimize the block are selected. This can lead to solutions in which the processor spends most of its time reconfiguring the RFU.

3.4 Instruction Synthesis/Compilation

Instruction synthesis takes the description of an instruction and generates the configuration string needed for the RFU. The techniques for synthesis used depend on the granularity of the reconfigurable logic.

For fine-grained logic, this process is identical to the compilation (or synthesis) of hardware with a hardware description language, such as VHDL or Verilog, onto a standard FPGA. In fact, the basic techniques are the same. The main differences deal mainly with the fact that the interfaces with the rest of the processor are already fixed. The input language used to describe the instruction can be specific to the RISP in order to obtain the best performance [31].

Coarse-grained logic, on the other hand, borrows techniques from the compiler domain [19]. A coarse-grained architecture can be viewed as a very complex clustered VLIW processor. The processing elements can be viewed as the functional units in the VLIW, the segments can be considered as clusters of functional units connected by intra segment interconnect, and the communication paths between the different clusters can be the inter segment interconnect. The configuration memory would store the VLIW code. The length of the instruction word would be much greater than in common VLIWs.

3.5 Retargetable Backend

The backend of the compiler performs platform specific optimizations and outputs assembler code. In the case of RISP, the backend needs to adapt to the new instructions created in the previous phases; the backend needs to be retargetable.

There are two main types of retargetability that can be incorporated on the compiler. Reconfigurable instructions can be tied to a specific part of the code (this is usually the case with custom instructions), simplifying code generation. The DFG is marked with information concerning where to use such specific instructions. The backend will only have to schedule the instruction and assign a set of operand registers, which is a major benefit when compared to ASIPs. In compilers for ASIPs, the specialized instructions have to be explicitly used by the programmer since the compiler is not able to use them automatically.

The backend needs to be truly retargetable for those instructions that are not tied to a specific part of the code, such as new addition instructions or a specialized "multiply by 5" instruction. This would be the behavior of a traditional retargetable compiler [34], but the RISP backend needs to be retargetable at a finer level because the instruction set changes over time. Therefore, the compiler should have different instruction sets and change between them depending on which part of the code is being generated. These sets are obtained in the previous stages.

For processors in which the reconfigurable logic runs asynchronously to the processor core, the backend needs to insert special synchronization instructions. An example is the Garp compiler [29].

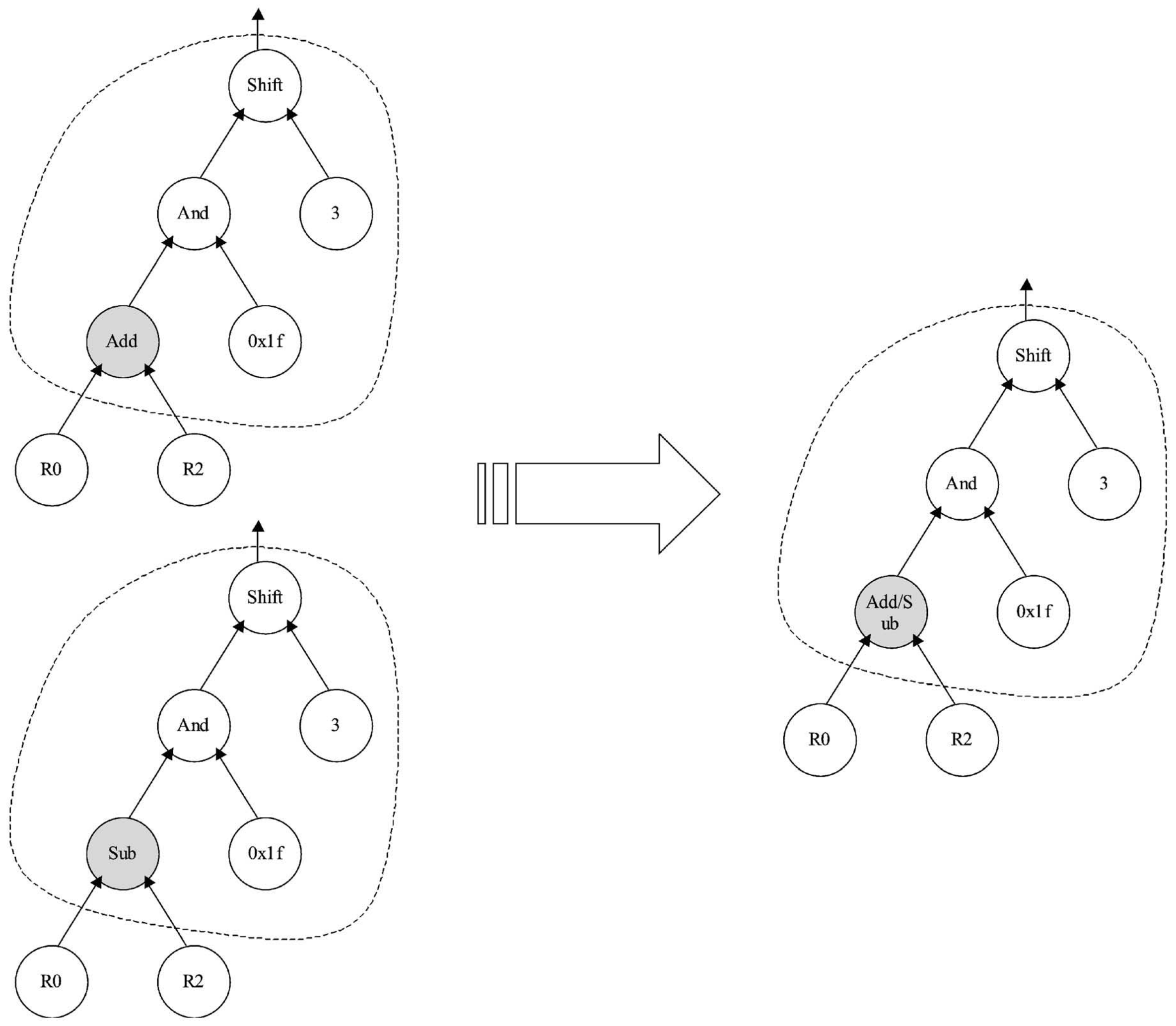

Fig. 12. Instruction merging.

3.6 Code Transformations

During the compilation process, many code transformations can be done in order to increase the performance of the code. Of particular interest are transformations that reduce the number of times that the RFUs need to be reconfigured since this is usually the most expensive part.

If the architecture supports prefetching, then prefetch instructions can be inserted after instruction selection [35]. This will modify the schedule by allowing the RFU to be reconfigured while some other code is being executed. This can easily reduce execution time by two [35]. Prefetching operations can be inserted at compilation time or at runtime using profiling techniques. If they are inserted at runtime, it is possible to adapt the program to the data set being handled.

For complex control paths, an alternative to just reconfiguring the RFU as needed is to check the state of the RFU explicitly in the code. Thus, if the next block would benefit from a certain instruction, and this instruction is configured in the RFU, optimized code is used. If the RFU does not have the correct configuration, code with the fixed instruction set is used instead.

Another type of transformation is instruction merging. Instruction creation is done locally. This local analysis might skip opportunities to reduce the number of reconfigurations in the processor. In Fig. 12, two different instructions are merged into a single instruction that can perform addition or subtraction in the first node. This reduces the number of different instructions that have to be placed in the RFU and, thus, reduces the reconfiguration time. No work has been found on the literature regarding this stage, but it is likely that clustering techniques developed for high-level synthesis will prove useful. Merging two or more instructions can result in increased latency, size, or other parameters. If this is the case, a performance check should be performed with these new instructions.

During the instruction selection phase, code transformations can be done to increase the locality of the configurations. These transformations would modify the code so that uses of an instruction are closer in time. Loop transformations look specially suited for this task [36].

4 Conclusions

This paper has presented the different design alternatives for reconfigurable instruction set processors. As has been seen, RISP design is not an easy task. The design of the reconfigurable logic is bound to be completely different from that of standard FPGAs. The coupling of the processor to this logic will pose many new problems, such as operand passing, synchronization, etc. But, the most important aspect will be the reconfigurability. Reconfiguration times will have to be minimized in order to obtain a satisfactory performance. The solution to this problem will be in the software tools. Code generation for RISP involves many new issues, and correctly managing the reconfiguration delay is going to be toughest one. As a matter of fact, the viability of RISP is mostly based on software technology and not in hardware technology.

On the other hand, once these problems are solved, RISPs will become a very good alternative to ASIPs and general purpose processors. They provide hardware specialization for compute intensive tasks with, at the same time, great flexibility. This factor, coupled to the reduction in power consumption that could be obtained by this specialization will change the way processors are built, in a similar way to what is now happening with VLIW processors.

Acknowledgments

This work is supported in part by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT) project Irmut, FWO project G.0036.99, IMEC and MEDEA+ A.502 (MESA).

References

[1] M.F. Jacome and G. de Veciana, "Design Challenges for New Application-Specific Processors," *IEEE Design & Test of Computers,* vol. 17, no. 2, pp. 40–50, Apr. 2000.

[2] D. Buell, W.J. Kleinfelder, and J.M. Arnold, *Splash 2: FPGA's in a Custom Computing Machine.* IEEE Computer Society Press, 1996.

[3] P.M. Athanas and H.F. Silverman, "Processor Reconfiguration through Instruction-Set Metamorphosis," *Computer,* pp. 11–18, Mar. 1993

[4] M. Wazlowski, L. Agarwal, A. Smith, E. Lam, P. Athanas, H. Silverman, and S. Ghosh, "PRISM-II Compiler and Architecture," *Proc. Workshop FPGAs and Custom Computing Machines (FCCM '93),* pp. 29–16, 1993.

[5] M.J. Wirthlin, B.L. Hutchings, and K.L. Gilson, "The Nano Processor: A Low Resource Reconfigurable Processor," *Proc. Workshop FPGAs and Custom Computing Machines (FCCM '94),* pp. 23–30, 1994.

[6] R. Razdan and M.D. Smith, "A High-Performance Microarchitecture with Hardware-Programmable Functional Units," *Proc. 27th Int'l Symp. Microarchitecture (MICRO 27),* pp. 172–180, Nov. 1994.

[7] M.J. Wirthlin and B.L. Hutchings, "A Dynamic Instruction Set Computer," *Proc. Workshop FPGAs and Custom Computing Machines (FCCM '95),* pp. 99–107, 1995.

[8] R.D. Wittig and P. Chow, "OneChip: An FPGA Processor with Reconfigurable Logic," *Proc. Workshop FPGAs and Custom Computing Machines (FCCM '96),* pp. 126–135, 1996.

[9] J.R. Hauser and J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor," *Proc. Workshop FPGAs and Custom Computing Machines (FCCM '97),* pp. 12–21, 1997.

[10] S. Hauck, T. Fry, M. Hosler, and J. Kao, "The Chimaera Reconfigurable Functional Unit," *Proc. Workshop FPGAs and Custom Computing Machines (FCCM '97),* pp. 87–96, 1997.

[11] C.R. Rupp, M. Landguth, T. Garverick, E. Gomersall, and H. Holt, "The NAPA Adaptive Processing Architecture," *Proc. Workshop FPGAs and Custom Computing Machines (FCCM '98),* pp. 28–37, 1998.

[12] T. Miyamori and K. Olukotun, "REMARC: Reconfigurable Multimedia Array Coprocessor," *Proc. Sixth Int'l Symp. Field-Programmable Gate Arrays (FPGA '98),* Feb. 1998.

[13] J.A. Jacob and P. Chow, "Memory Interfacing and Instruction Specification for Reconfigurable Processors," *Proc. Seventh Int'l Symp. Field-Programmable Gate Arrays (FPGA '99),* pp. 145–154, 1999.

[14] S.C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R.R. Taylor, and R. Laufer, "PipeRench: A Coprocessor for Streaming Multimedia Acceleration," *Proc. 26th Int'l Symp. Computer Architecture (ISCA '99),* pp. 28–39, May 1999.

[15] J.L. Hennessy and D.A. Patterson, *Computer Architecture A Quantitative Approach,* second ed. Morgan Kauffmann, 1996.

[16] C. Liem, P. Paulin, and A. Jerraya, "Address Calculation for Retargetable Compilation and Exploration of Instruction-Set Architectures," *Proc. Design Automation Conf.,* pp. 597–600, June 1996.

[17] S. Brown and J. Rose, "Architecture of FPGAs and CPLDs: A Tutorial," *IEEE Design and Test of Computers,* vol. 13, no. 2, pp. 42–55, 1996.

[18] S. Shrivastava and V. Jain, "Rapid System Prototyping for High Performance Reconfigurable Computing," *Proc. 10th Int'l Workshop Rapid System Prototyping (RSP '99),* pp. 32–37, July 1999.

[19] D.C. Cronquist, P. Franklin, S.G. Berg, and C. Ebeling, "Specifying and Compiling Applications for RaPiD," *Proc. Workshop FPGAs and Custom Computing Machines (FCCM '98),* pp. 116–127, 1998.

[20] A. De Hon, "DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century," *Proc. Second Int'l Symp. Field-Programmable Gate Arrays (FPGA '94),* 1994.

[21] M. Moudgill and S. Vassiliadis, "Precise Interrupts," *IEEE Micro,* vol. 16, no. 1, pp. 58–67, Feb. 1996.

[22] A.A. Chien and J.H. Byun, "Safe and Protected Execution for the Morph/AMRM Reconfigurable Processor," *Proc. Workshop FPGAs and Custom Computing Machines (FCCM '99),* pp. 209–221, 1999.

[23] K. Nelson and S. Hauck, "Mapping Methods for the Chimaera Reconfigurable Functional Unit," technical report, Northwestern Univ., 1997.

[24] T.J. Callahan and T. Wawrzynek, "Instruction Level Parallelism for Reconfigurable Computing," *Proc. Eighth Int'l Workshop Field-Programmable Logic and Applications (FPL '98),* Hartenstein and Keevallik, eds., Sept. 1998.

[25] Z.A. Ye, N. Shenoy, and P. Banerjee, "A C Compiler for a Processor with a Reconfigurable Functional Unit," *Proc. Eighth Int'l Symp. Field-Programmable Gate Arrays (FPGA '00),* 2000.

[26] R. Razdan, K. Brace, and M.D. Smith, "PRISC Software Acceleration Techniques," *Proc. Int'l Conf. Computer Design: VLSI in Computers and Processors (ICCD '94),* pp. 145–149, Oct. 1994.

[27] D.A. Clark and B.L. Hutchings, "Supporting FPGA Microprocessors Through Retargetable Software Tools," *Proc. Workshop FPGAs and Custom Computing Machines (FCCM '96),* pp. 195–203, 1996.

[28] M.B. Gokhale and J.M. Stone, "NAPA C: Compiling for a Hybrid RISC/FPGA Architecture," *Proc. Workshop FPGAs and Custom Computing Machines (FCCM '98),* pp. 126–135, 1998.

[29] T.J. Callahan, J.R. Hauser, and J. Wawrzynek, "The Garp Architecture and C Compiler," *Computer,* vol. 33, no. 44, pp. 62–69, Apr. 2000.

[30] S.C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R.R. Taylor, "PipeRench: A Reconfigurable Architecture and Compiler," *Computer,* vol. 33, no. 4, pp. 70–77, Apr. 2000.

[31] M. Budiu and S.C. Goldstein, "Fast Compilation for Pipelined Reconfigurable Fabrics," *Proc. Seventh Int'l Symp. Field-Programmable Gate Arrays (FPGA '99),* pp. 195–205, 1999.

[32] B. Kastrup, A. Bink, and J. Hoogerbrugge, "ConCISe: A Compiler-Driven CPLD-Based Instruction Set Accelerator," *Proc. Workshop FPGAs and Custom Computing Machines (FCCM '99),* pp. 92–101, 1999.

[33] K. Bazargan, R. Kastner, and M. Sarrafzadeh, "3-D Floorplanning: Simulated Annealing and Greedy Placement Methods for Reconfigurable Computing Systems," *Proc. 10th Int'l Workshop Rapid System Prototyping (RSP '99),* pp. 38–43, July 1999.

[34] G. Goossens, J. Van Praet, D. Lanneer, W. Geurts, A. Kifli, C. Liem, and P.G. Paulin, "Embedded Software in Real-Time Signal Processing Systems: Design Technologies," *Proc. IEEE,* vol. 85, no. 3, pp. 436–454, Mar. 1997.

[35] S. Hauck, "Configuration Prefetch for Single Context Reconfigurable Coprocessors," *Proc. Sixth Int'l Symp. Field-Programmable Gate Arrays (FPGA '98),* Feb. 1998.

[36] D.F. Bacon, S.L. Graham, and O.J. Sharp, "Compiler Transformations for High-Performance Computing," *ACM Computing Surveys,* vol. 26, no. 4, pp. 345–420, Dec. 1994.

Francisco Barat received the engineering degree in telecommunications from the Polytechnic University of Madrid, Spain, in 1999. That same year he joined the Katholieke Universiteit Leuven, Belgium, where he is currently pursuing a PhD degree in applied sciences. His current research interests are in the field of multimedia embedded systems focusing on microprocessor architectures and compilation techniques. He is a student member of the IEEE and the ACM.

Rudy Lauwereins obtained the MEng and PhD degrees in electrical engineering from the Katholieke Universiteit Leuven, Belgium, in 1983 and 1989, respectively. In 1991, he was a visiting researcher at the IBM Almaden Research Center, San Jose, CA on a postdoctoral NFWO research fellowship. In 1998, he acted as advisor to the team that won the Northern European section of the Texas Instruments DSP Solutions Challenge. In 2000-2001, he was elected by students as the best teacher of the engineering faculty's Master's curriculum and got a nomination for the same prize in 2001-2002. He became a professor at the Katholieke Universiteit Leuven, Belgium, in 1993, and vice-president of Institut Mémoires de l'édition Contemporaine in 2001. His main research interests are in computer architectures, implementation of interactive multimedia applications on dynamically reconfigurable platforms, and wireless communication. In these fields, he has authored and coauthored more than 250 publications in international journals and conference proceedings. He is a senior member of the IEEE.

Geert Deconinck received the MSc degree in electrical engineering and the PhD degree in applied sciences from the K.U.Leuven, Belgium, in 1991 and 1996, respectively. He is a visiting professor at the Katholieke Universiteit Leuven (Belgium) since 1999 and a postdoctoral fellow of the Fund for Scientific Research—Flanders (Belgium) since 1997. He is working in the research group ELECTA (Electrical Energy and Computing Architectures) of the Department of Electrical Engineering (ESAT). His research interests include the design, analysis, and assessment of computer architectures to meet real-time, dependability, and cost constraints for embedded distributed applications. In this field, he has authored and coauthored more than 75 publications in international journals and conference proceedings. In 1995-1997, he received a grant from the Flemish Institute for the Promotion of Scientific-Technological Research in Industry (IWT). He is a certified reliability engineer (ASQ), a member of the Royal Flemish Engineering Society, a senior member of the IEEE, and of the IEEE Reliability, Computer, and Power Engineering Societies.

▷ **For more information on this or any computing topic, please visit our Digital Library at** http://computer.org/publications/dilb.