Design Space Exploration of Partially Re-configurable Embedded Processors

A. Chattopadhyay, W. Ahmed, K. Karuri, D. Kammler, R. Leupers, G. Ascheid, H. Meyr
Integrated Signal Processing Systems, RWTH Aachen University 52056 Aachen, Germany
anupam@iss.rwth-aachen.de

Abstract

In today's embedded processors, performance and flexibility have become the two key attributes. These attributes are often conflicting. The best performance is obtained from custom designed integrated circuits. In contrast, the maximum flexibility is delivered by a general purpose processor. Among the architecture types emerged over the past years to strike an optimum balance between these two attributes, two are prominent. The first ones are Field Programmable Gate Array (FPGA)-based architectures and the second ones are Application-specific Instruction-set Processors (ASIPs). Depending on the type of application (i.e. stream-like or control-dominated) either one of the abovementioned architecture types is able to deliver high performance or flexibility or both. Consequently, a new design approach with partial re-configurability on the application-specific processor is attracting strong research interest. We call this architecture re-configurable ASIP (rASIP). Currently, the lack of a high-level abstraction of the rASIP limits the designer from trying out various design alternatives because of long and tedious exploration cycles. To address this issue, in this paper, a high-level specification for re-configurable processors is proposed. Furthermore, a seamless design space exploration methodology using this specification is proposed.

1. Introduction

Over the past few years, the increasing demand of performance by complex, cutting-edge applications have created a strong research interest for flexible and high-performance system design. This demand is met by increasing the system complexity to accomodate multiple cores on a single System-on-Chip (SoC). Due to unique blend of performance and flexibility, Application Specific Instruction-set Processor (ASIP) emerged as one of the key component of such SoCs. ASIPs can deliver high throughput, consume low power and at the same time are flexible due to their programmability. However, the flexibility of ASIP is limited to the *soft* changes i.e. the hardware implementation remains unaltered after the fabrication. Often this limitation prevents the ASIP from reaching the optimum performance for a wider range of applications.

Alternatively, in search of higher flexibility, industry and academia have opted for fully re-configurable architectures, which are modelled using a variety of field-programmable devices [1]. To combine the merits of both the architectural alternatives, partially re-configurable processors are gaining prominence in recent years [2] [3]. These architectures contain an ASIP part and an FPGA part. The FPGA part is included to obtain *hard* flexibility, whereas the ASIP part offers *soft* flexibility. This higher flexibility allows the re-configurable ASIP (rASIP) to adapt to changing applications without compromising the performance. A rASIP architecture can be changed between the execution of applications or during the execution of an application resulting into further classification. The rASIPs, which can be configured during the execution of an application are dynamically re-configurable, thereby known as d-rASIPs. The statically re-configurable ASIPs are called s-rASIPs. In this paper, we target s-rASIPs. However, the concept can be extended easily to the domain of d-rASIPs.

The flexibility and performance of the rASIP relies strongly on the organization of the architecture. For example, a rASIP where the re-configurable part does not have direct access to the data memory can face a major bottleneck to boost the performance for memory-intensive applications. A rASIP architecture can be broadly divided into three partially overlapping components namely, the base processor, the ASIP-FPGA coupling and the FPGA architecture. Considering the numerous design alternatives for each of these components, it is understandable that the design of rASIP is a demanding task. This complex task calls for an efficient design methodology. In this paper, this problem is addressed directly by proposing a high-level specification for modelling rASIPs.

Architecture Description Languages (ADLs) [4] [5] [6] have been used successfully during recent years to model ASIPs. From an ADL description, the software toolsuite like compiler, simulator, assembler, linker as well as the Hardware Description Language (HDL) implementation of the processor can be automatically generated. In this paper, ADL LISA [7] is used as the starting point for capturing the rASIP description. This ADL is extended for modelling rASIPs. The existing ADL-based automatic software toolsuite generation is enhanced to cover the new rASIP description. As a case study, a RISC-based rASIP is modelled and its performance improvement in comparison with the base ASIP is shown. In short, the contributions of this paper are to present:

- A high-level specification language for rASIP modelling.
- A fast methodology for rASIP design space exploration.

The rest of the paper is organized as follows. In section 2, some contemporary work in this field is described. The section 3 presents an overview of the rASIP architectures and introduces necessary terminology for the understanding of the rest of the paper. In section 4, the architecture description language LISA is briefly outlined and the new language elements necessary for rASIP modelling are introduced. Section 5 explains the proposed rASIP design flow in detail. In section 6, a case study is elaborated. The paper is concluded and future work is outlined in section 7.

2. Related Work

A detailed discussion of the issues with the rASIP modelling and the associated tools is documented in [8]. Although ADLs are used extensively to perform high-level design space exploration and to model ASIPs [7] [6], the concept of modelling rASIPs in a high-level specification is novel. Exemplarily, an rASIP architecture from industry is chosen and its design methodology is discussed in the following paragraph. Subsequently, few interesting approaches to perform the rASIP design space exploration are discussed.

The Stretch S5000 family of partially re-configurable processors [2] offers a combination of a RISC processor with an Instruction Set Extension Fabric (ISEF), which is essentially an FPGA. The processor comes with a dedicated tool-suite. This tool-suite helps the designer to map an application efficiently to the processor by choosing

978-3-9810801-2-4/DATE07 © 2007 EDAA

the application hot-spot and by re-configuring the field-programmable fabric accordingly. This family of partially re-configurable processors specifically targets video, wireless and biometric applications. However, the approach of Stretch is not generic enough considering that the base processor structure is only partially configurable. The interface between the processor and the programmable fabric is also fixed beforehand, therefore limiting the potential number and organization of custom instructions. Only the base processor structure and interface are provided to the end-user. The complete design space exploration for the rASIP is not feasible.

To allow a generic design space exploration for the complete rASIP in a single framework, an interesting approach is adopted by [9]. Here, the base processor is modelled using an ADL. The retargetable simulator generated from the ADL description is coupled with the FPGA simulator. Using this method, the complete design space exploration can be done. A drawback of this approach is that the re-configurable part and the fixed processor part are written separately using different description formats. This prevents a smooth design space exploration for various degrees of processor-FPGA coupling. Furthermore, no clear methodology is outlined for custom instruction synthesis and integration.

In another attempt to create a generic software development tool chain for re-configurable processors [10], a gcc-based framework is developed. The complete gcc tool chain including the simulator, compiler, assembler and the debugger is extended to support different architectural enhancements e.g. DSP-like instructions, VLIW capabilities and the FPGA instructions. Furthermore, specific latencies for the FPGA instructions can also be specified. This work comes with several limitations, the most notable being the inflexibility of the underlying architecture. The design space exploration is limited to the base architectures supported by gcc. The selection of custom instructions and according modification of the target assembly code is done manually. Moreover, it does not support a seamless and consistent flow for implementation, since HDL code generation for the modified architecture is lacking.

3. rASIP : An Overview

In this section, the various possibilities of rASIP architectures i.e. the rASIP design space is discussed. Consequently, a generic methodology for rASIP design space exploration is proposed.

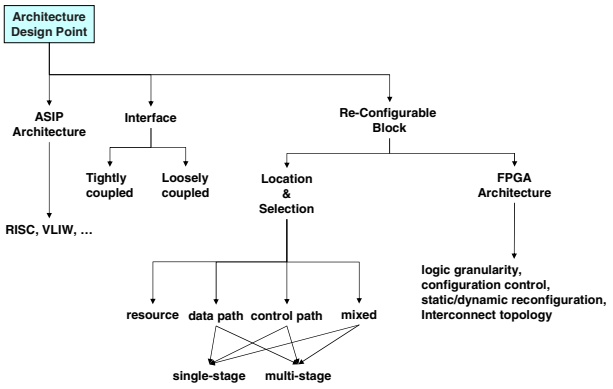

Figure 1. rASIP Design Space

In the figure 1, the rASIP design space is partitioned into three subspaces. The rASIP architecture is partially re-configurable i.e. it contains a base processor which is fixed after the processor is fabricated, a re-configurable part and the interface connecting these two. The base processor can be of different architecture styles e.g. RISC, VLIW. The number of pipeline stages, the pipeline control, the width of the instruction and various other issues are involved with the base processor design. These issues are commonplace with an ASIP design. Regarding the design of the re-configurable block, numerous possibilities can be explored. The granularity of the logic blocks, the interconnect topology and several other aspects determines the performance and flexibility of the rASIP. Furthermore, the selection of elements to belong to the re-configurable block provides a large number of design alternatives. For example, the control path or the decoder of the rASIP can belong to the base processor only. In this case, the instruction encoding for the custom instructions need to be fixed. Whereas, if a part of the decoder is localized in the re-configurable block, then higher flexibility can be obtained. The higher flexibility comes at the cost of higher area and timing due to the irregular structure of the decoder being mapped to a regular FPGA structure. Finally, the interface of the re-configurable part and the base processor is of high importance. It can be tightly coupled meaning the re-configurable block can access processor internal registers, pipeline registers and is guided by several control signals coming from the base processor, which does not belong to the peripherals usually. The rASIP architecture can be loosely coupled, where it interacts with external processor ports only. The interfacing between the re-configurable block and the base processor strongly influences the post-fabrication flexibility. Note, that the subspaces of this rASIP design are overlapping. For example, the design of the base processor cannot be done without caring about the interfacing. Therefore, the proposed specification to cover the complete rASIP description is important for rASIP design space exploration and implementation.

The design space exploration and implementation of rASIPs can be naturally sub-divided into two phases. The focus of these two phases are presented in the following.

Pre-fabrication Design Flow: This phase of design happens before the rASIP is fabricated. Here, the complete design space is open. The decisions involving all three design sub-spaces are to be taken in this phase. Finally, the design is implemented partially on fixed and partially on re-configurable hardware.

Post-fabrication Design Flow: This phase of design happens after the rASIP is fabricated. In this phase, the base processor and the interfacing hardware is fixed. The architecture design space is limited to the possible configurations of the re-configurable block only.

4. High Level Language for rASIPs

In this section, a brief overview of the architecture description language LISA is provided, focussing on the parts which are relevant in this context. Following the overview, the necessary additional elements for rASIP modelling are introduced.

4.1 LISA Resource Section

The processor resources (e.g. registers, memories, ports) are declared globally in the *resource section*, which can be accessed from the processor datapath. The datapath of the processor is modelled as a chain of *LISA operations* outside the resource section. Apart from the resource declarations, the resource section of LISA language is used to describe the fundamental structure of architecture. For this purpose the keywords *pipeline*, *pipeline_register* and *unit* are used. The keyword *pipeline* defines the instruction pipeline of the processor with the corresponding order and name of the pipeline stages. The *pipeline_register* is used to define the pipeline registers between the stages. Using the keyword *unit*, the designer can define a set of LISA operations (within a pipeline stage) to form an entity (VHDL) or module (Verilog) in the generated HDL code.

4.2 LISA Operation Graph

In LISA, an *operation* is the central element to describe the timing and the behavior of a processor instruction. The instruction may be split among several LISA operations.

The LISA description is based on the principle that a specific common behavior or common instruction encoding is described in a single operation whereas the specialized behavior or encoding is implemented in its *child* operations. With this principle, LISA operations are basically organized as an n-ary tree. However, specialized operations may be referred to by more than one *parent* operation. The

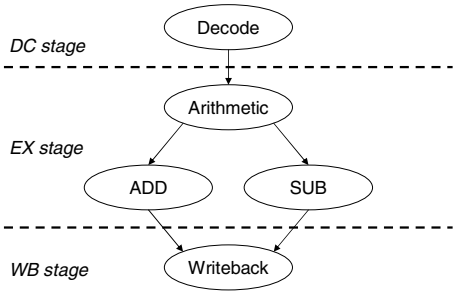

Figure 2. LISA Operation DAG Example

complete structure is a Directed Acyclic Graph (DAG) $\mathcal{D} = \langle V, E \rangle$. V represents the set of LISA operations, E the graph edges as set of child-parent relations. These relations represent either *Behavior Calls* or *Activations*, which refer to the execution of another LISA operation. Figure 2 gives an example of a LISA operation DAG. As shown, the operations can be distributed over several pipeline stages. A chain of operations, forming a complete branch of the LISA operation DAG, represents an instruction in the modelled processor.

Instruction Coding Description : The instruction encoding of a LISA operation is described as a sequence of several *coding fields*. Each coding field is either a terminal bit sequence with "0", "1", "don't care"(X) bits or a nonterminal bit sequence referring to the coding field of another child LISA operation.

Activations : A LISA operation can *activate* other operations in the same or a latter pipeline stage. In either case, the child operation may be activated *directly* or via a *group*. A *group* collects several LISA operations, with the elements being mutually exclusive. The elements are distinguished by a distinct binary coding. All elements belonging to the same group must have same bit-width.

Behavior Description : The behavior description of a LISA operation corresponds to the datapath of the ASIP. The behavior description is a non-formalized element of the LISA language (contrary to formalized elements like *coding*, *activation* etc.), where plain C code can be used. Resources such as registers, memories, signals and pins as well as coding elements can be accessed in the same way as ordinary variables.

4.3 Extensions for rASIP Modelling

Re-configurable unit : The keyword *re-configurable* can be appended to the *unit* definition in order to identify the LISA operations, which will be placed in the re-configurable block. This identification is used during RTL synthesis from LISA to move the complete unit outside the base processor.

Latency : Every LISA operation, in a cycle-accurate description, is assigned to one particular pipeline stage. An operation in one pipeline stage takes one cycle to execute. However, the re-configurable block may run at a different clock-speed. The keyword *latency* is introduced for modelling the ratio of the latency of an operation in the re-configurable block compared to the latency of an operation in the base processor. For every LISA operation an integral latency can be specified, which corresponds to the number of cycles it requires to execute. For operations without any latency specification, a single-cycle latency (similar to the base processor operations) is assumed by default.

Fullgroup : The keyword *fullgroup* is introduced for reserving the opcode space of additional instructions, which can be introduced during the post-fabrication enhancements. As mentioned earlier, a *group* collects several mutually exclusive LISA operations. The coding width (w) of the group members dictate the maximum possible number (2^w) of operations, which can belong to the group. If a group does not contain all possible operations, then additional operation(s) can be inserted to it. A fullgroup reserves this additional coding space.

Register Localization : This extension is useful during RTL synthesis from the rASIP description. By enabling the register localization option, the designer can move the storage elements, which are used locally within the re-configurable operations to the re-configurable block.

Decoder Localization : This extension is also used during the RTL synthesis. In order to have flexible decoding and custom instruction encoding during post-fabrication phase, the part of the complete decoder, which is relevant for the re-configurable operations can be identified and moved to the re-configurable block. Usually, the full-group keyword is used in conjunction with decoder localization.

5. rASIP Design Flow

As stated before, the proposed rASIP design flow can be divided into two major phases, the first one being the pre-fabrication design flow and the second one being the post-fabrication design flow. In this section, the pre- and post-fabrication design flow is discussed in detail.

5.1 Pre-fabrication Design Flow

The pre-fabrication rASIP design flow is shown in figure 3. As outlined in various literature [11] [12], the design of an application-specific processor often starts from the analysis of the application(s). This analysis can be done using static and dynamic profiling [13] of the application in an architecture-independent manner. The profiling helps to narrow down the architectural design space. With the aid of the profiler, the decisions concerning the memory hierarchy, number of registers, processor architecture (e.g. RISC, VLIW) can be taken. The designer can use the extended LISA language for developing the target rASIP. This design flow, so far, is alike to the ASIP design flow. For rASIPs, the major additional decisions, which need to be taken are categorized in the following. (i) Selection of instructions (or parts of instruction), to be mapped to the re-configurable block. (ii) Decisions concerning the interface between the processor and the re-configurable block. (iii) Decisions concerning the local decoding at the re-configurable block. (iv) Decisions concerning the architecture of re-configurable block.

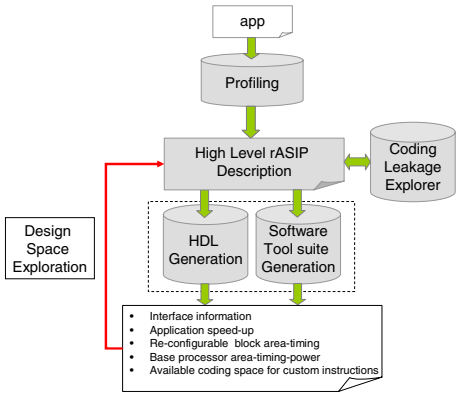

Figure 3. Pre-fabrication rASIP design flow

These decisions are key to play the trade-off between the flexibility and the performance of the rASIP. For example, the availability of local decoding inside the re-configurable block allows flexibility in adding custom instructions. On the other hand, the decoder structure, being irregular in nature, adds to the interconnect cost and power budget of the re-configurable block. The decisions taken during the pre-fabrication design flow constrain the optimization benefits (or in other words, the design space) achievable during the post-fabrication design. For example, the interface between the fixed processor part and the re-configurable block is decided during the pre-fabrication flow. This decision cannot be altered once the processor is fabricated.

Software Tools Generation: The rASIP software tools e.g. simulator, assembler, linker are automatically generated from the extended LISA description. The only software tool, which is currently affected by the language extension, is the processor instruction-set simulator. The existing LISA simulator is extended to support the *latency* of re-configurable LISA operations. The instruction-set simulation guides

the designer to make the pre-fabrication design decisions. Note, that the latencies must be taken into account during the scheduling of assembly instructions.

Coding Leakage Explorer: A stand-alone software tool, named coding leakage explorer, has been developed for rASIP design space exploration. This tool analyzes the coding contribution of different LISA operations. After the analysis, it determines the free coding space. This free coding space is termed as *leakage*. The higher the coding leakage of a particular group of instructions, the more number of special instructions it can accomodate. The coding leakage explorer guides the designer to determine the re-configurable LISA operations and the *fullgroups*. Detail analysis of this tool is beyond the scope of this paper.

RTL Synthesis: The LISA-based RTL synthesis tool is extended to support the rASIP description. In the rASIP description, the re-configurable block is identified by the *unit re-configurable* keyword. The designer can specify *decoder localization* and *register localization* for the re-configurable block. Accordingly, the complete re-configurable block is created and moved out of the processor. This movement generates the processor-re-configurable block interface automatically. The complete interface is stored in the XML format. The storage of the interface is necessary in order to ensure that the interface restrictions are not violated during the post-fabrication rASIP enhancements. The generated HDL description for the re-configurable block can be synthesized using an FPGA synthesis tool in order to get an estimation of the area.

```
RESOURCE {
  …
  REGISTER int R[0..15];
  REGISTER int mul_res;
  PIPELINE PIPE = {DC, EX, WB;};

  UNIT RECONFIGURABLE rec_unit = {REC_OP;};
  UNIT ex_unit = {ADD, SUB;};
  …
}
```

```
OPERATION Arithmetic IN pipe.DC {
  DECLARE {
    FULLGROUP arith_op = {ADD || SUB || REC_OP};
          }
  …
}
OPERATION REC_OP IN pipe.EX {
  DECLARE { LATENCY = 2; }
  BEHAVIOR {
    mul_res = R[src1] * R[src2];
  }
}
```

Figure 4. rASIP Description Example

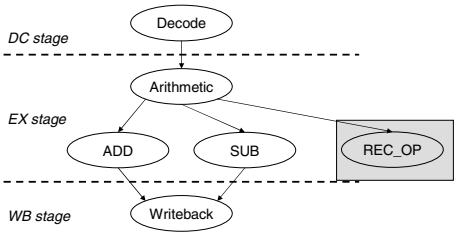

Figure 5. Extended LISA Operation DAG

A rASIP Description Example: Parts of a rASIP description are given in figure 4 for a better understanding of the pre-fabrication design flow. The rASIP description corresponds to the DAG as shown in figure 5. In the DAG, the operation *Arithmetic* activates an operation *REC_OP*. In order to save the coding space for future custom instructions, the group of operations ADD, SUB and REC_OP are put into a fullgroup. The operation REC_OP is re-configurable. In this particular case, the operation REC_OP is performing a multiplication between two General Purpose Registers (GPRs) and is writing the result to the special register *mul_res*. The operation REC_OP has a latency of 2. This indicates that it will take a time corresponding to 2 cycles of the base processor in order to generate the result. There are few points to be noted here. Firstly, although the operation REC_OP is re-configurable, yet it is placed in the EX stage, in order to maintain the pipelining behavior of the overall rASIP architecture. Secondly, from the perspective of the behavioral description, the re-configurable operation is not distinguished from any other operation. This provides flexibility and efficiency in the high-level design, where the designer concentrates on the instruction-set architecture. Thirdly, the re-configurable operation can load/store operands from/to GPRs, special-purpose registers or even pipeline registers and memories(not shown in the example). These decisions directly influence the interface of base processor architecture with the re-configurable block. The proposed modelling style offers absolute freedom in these decisions. Finally, the latency of the re-configurable block can be easily modified. The only thing, which must be taken care of is the conflict in resource access during multi-cycle re-configurable operations. There are several possibilities. For example, the write access to the resources from the base processor can be blocked [14], when it is the target of a re-configurable operation. Alternatively, dedicated special purpose registers can be kept for accessing the re-configurable operations [2]. The high-level rASIP modelling eases the effort in exploring these alternatives.

5.2 Post-fabrication Design Flow

The proposed post-fabrication rASIP design flow is shown in figure 6. In this phase, the rASIP is already fabricated. The major design decision, which needs to be taken, is the selection and synthesis of custom instructions to reduce the application runtime.

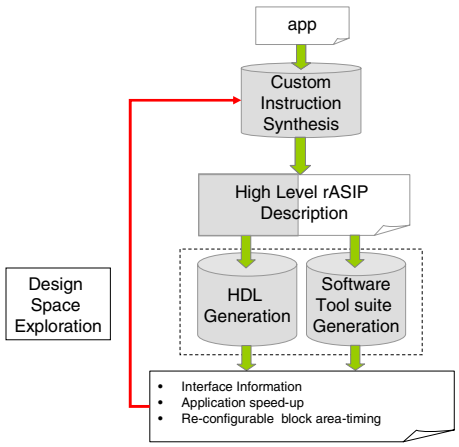

Figure 6. Post-fabrication rASIP design flow

Several custom instruction synthesis tools and algorithms have been presented in the literature [15] [16]. In the proposed flow, the custom instruction synthesis tool (ISEGen) presented in [17] is used. This tool is extended in order to accept generic interface constraints and produce custom instructions in LISA description format. During the custom instruction selection, several hard design constraints must be maintained. Those are listed below:

- The special-purpose or custom instructions must not violate the existing interface between re-configurable block and the base processor. Actually, the custom instruction synthesis tool used in our flow accepts this interface as an input constraint.
- The additional number of instructions must not exceed the number allowed by the free coding space available.
- The overall area of the custom instructions' datapath and corresponding decoding logic should be within the area budget of the existing re-configurable block.
- Additional *internal* storage elements for the re-configurable block can be used as long as the area budget of the re-configurable block is not violated. These internal storage elements can be declared in the LISA resource section, while turning on the *register localization* option.

In the post-fabrication design flow, ISEGen generates the custom instructions with potentially high speed-up. The behavior of the custom instructions are generated in form of LISA operations, which can be directly plugged in to the existing LISA model. The software tool suite for the modified rASIP description is generated automatically. For the re-configurable block, the generated HDL description is synthesized with standard FPGA synthesis tools and the synthesis results can be compared with the area and timing budget.

Addition of Custom Instructions : ISEGen embeds the custom instructions in the target application with special compiler directives. These compiler directives are used by the compiler as hints for performing different tasks e.g. the register allocation and latency of custom instructions. The application can be compiled, assembled and linked using the re-targeted software tool suite. The re-targeted rASIP simulator provides an estimation of the application runtime improvement. In the following, the modification of target application by ISEGen tool is explained in detail.

The ISEGen tool utilizes *inline assembly functions calls* to directly call assembly statements replacing the C functions for Custom Instructions (CIs). The inline assembly call defines the usage of arguments, scheduling of the CI and the CI syntax itself. Within the target application, the custom instructions are called like normal function calls, which are replaced by inline assembly functions' assembly syntax portion during compilation. The inline assembly function and the compiler directives used here are part of the LISATek re-targetable compiler framework.

Interface Matching: Interface matching is performed during RTL synthesis from the rASIP during post-fabrication phase. There, the new interface between the FPGA and the base processor is validated against the existing interface.

5.3 Analysis of the Overall Flow

The proposed pre and post-fabrication design flow for rASIPs offers significant advantages compared to the current design flows. Firstly, the proposed flow is completely generic in nature. Secondly, the existing tools like custom instruction synthesis can be seamlessly plugged in to the current flow. Finally, due to the high-level rASIP modelling, the design space exploration is extremely efficient. However, the design flow has one limitation, yet to be covered. Current rASIP architectures [2] [3] often contain an application-specific re-configurable block. The re-configurable block architecture itself presents a wide number of design alternatives. For the available rASIP architectures, these design points are explored in an ad-hoc manner or are pre-designed for a wide range of application domains without allowing the designer flexibility. The rASIP design flow proposed in this paper lacks any kind of re-configurable block exploration and implementation flow. Dedicated, existing commercial FPGAs are used instead. In the proposed rASIP design tool-flow, this paper contributes in three categories. Firstly, a new tool called Coding Leakage Explorer is developed. Secondly, existing ADL LISA and associated tools for LISA-based processor development are extended. Finally, a generic interface with the custom instruction synthesis tool ISEGen is established.

6. Case Study

In this section, first the target processor architecture is discussed. Then a brief introduction about the chosen applications is given followed by the results achieved during the case study.

For this case study, we have chosen LT_RISC_32p5 as base processor. The LT_RISC_32p5 is a 32-bit, 5-stage pipelined RISC based architecture. The processor contains 16 general purpose registers and several special-purpose registers. LT_RISC_32p5 employs general purpose load store instructions, complex multiplication operations and several instructions for aiding DSP applications e.g. add-compare-select.

The target applications for the rASIP are chosen from the domain of cryptography. *DES* is a symmetric 64-bit block cipher algorithm with 64-bit key. *Blowfish* is another symmetric block cipher encryption algorithm that uses a variable-length key, from 32 to 448 bits. *GOST* is a 64-bit block cipher algorithm with a 256-bit key.

As proposed in the design flow, the case study is divided into two phases i.e. the pre-fabrication and post-fabrication phases. The case study starts with the analysis of applications for rASIP design in pre-fabrication phase and then in the post fabrication phase uses the flexibility offered by reconfigurable block to optimize the rASIP. While, *DES* is used to take the pre-fabrication architectural decisions, *GOST* and *Blowfish* are used as the post-fabrication applications with the interface and the base processor remaining fixed.

Pre-fabrication Design Space Exploration : The application DES is chosen as the starting point for rASIP design. The application is subjected to the *Microprofiler* tool [13] for identification of hot spots. The function *des_round* is identified as the hot spot of the DES application. This function is then subjected to the ISEGen tool for identification of custom instructions with initial constraints of a 2-input interface to 2 GPRs of the base processor, a 1-output interface to 1 GPR of the base processor and 32 internal registers, each of 32 bit width. The ISEGen tool generated 5 custom instructions, the behaviors of which are mapped completely in LISA description. The codings of these custom instructions are determined using the *Coding Leakage Explorer*.

During RTL synthesis with these CIs, register localization and decoder localization options are turned on, in order to have flexibility for adding further custom instructions in the re-configurable part. The base processor is synthesized with Synopsys Design Compiler [18] using 0.13 μm process technology of 1.2 V. The re-configurable block is synthesized with Synopsys FPGA Compiler [18] using Xilinx Virtex-II pro [19] (0.13 μm process technology of 1.5 V) as the target device. The synthesis results are given in the table 1.

Re-configurable Block			Base Processor	
Area	Minimum Clock Delay (ns)	Latency	Area (Gates)	Clock Delay (ns)
1653 LUTs 512 Registers	12.09	4	88453	4.0

Table 1. Pre-fabrication Synthesis Results

As the synthesis results demand the FPGA latency to be at least thrice that of the base processor, a latency of 3 is introduced during the simulation of the modified DES application. The latency information can be fed to the inline assembly function (using compiler directives) to obtain a scheduled assembly application. However, the current compiler directives do not allow to specify the dependencies between the internal registers of the FPGA, possibly resulting in a data hazard. Therefore, the scheduling of the custom instructions is performed manually. It is observed that the latency of CIs can be completely hidden by efficient scheduling. The initial simulation results of the DES application show upto 3.5 times runtime speed-up (table 2). After these simulations, it is observed that the *des_round* function (the hot-spot function of DES) performs several accesses to data memory containing S-box. Each S-box contains 64 32-bit elements. These memory contents are known prior to the hot-spot execution. Existing studies [20] on exploiting such knowledge show that the runtime improvement can be stronger by including scratchpad memories within the CI. To experiment with such extensions, local scratchpad memory resource is appended to the rASIP description. A special instruction to transfer the S-boxes from data memory to the scratchpad memory is included. ISEGen is then configured to have upto 4 parallel scratchpad accesses. Each of these configurations produced different set of custom instructions. Since the scratchpad access is local to the re-configurable block, the interface constraints are not modified due to the access. The only modification required is for allowing the data transfer from the data memory to the scratchpad memory. The complete simulation and synthesis results for custom instructions with scratchpad access are given in table 3.

Considering no significant change in speed-up with other interface constraints (e.g. 3-input,1-output and 4-input,1-output), it is decided

Latency	Without CIs	With CIs	Speed-up
4	1563266	625306	2.5
Hidden	1563266	453397	3.5

Table 2. Simulation Cycles : DES

Number of Parallel Scratchpad Access	Speed-up	Minimum Clock Delay (ns)	Area (LUTs)
1	4.4	10.7	1342
2	4.2	9.72	1665
3	5.9	9.05	1638
4	6.0	9.05	1616

Table 3. CIs with Scratchpad Access : DES

to keep the 2-input, 1-output interface setting for the rASIP fabrication. Since scratchpad memories improved the runtime performance considerably, we decided to keep it. Number of parallel accesses can be increased or decreased post-fabrication depending on the available re-configurable block area. Note that these memories can be physically implemented as EPROMs, SRAMs or flexible hardware tables on FPGA. The trade-off between these alternatives are to be explored in future.

Post-fabrication Design Space Exploration : In keeping trend with cryptographic applications, the hot-spot of Blowfish application does also contain accesses to pre-calculated memory elements. Those elements are loaded to the scratchpad memory. ISEGen identified various set of custom instructions for Blowfish. The interface restrictions from the pre-fabrication design as well as various scratchpad access configurations are fed to the ISEGen. The generated set of instructions are then appended to the rASIP description. The syntheis and simulation results (refer table 4) demonstrate the prudency of our pre-fabrication decisions.

Number of Parallel Scratchpad Access	Speed-up	Minimum Clock Delay (ns)	Area (LUTs)
0	2.7	13.78	1456
1	3.3	8.72	1009
2	3.4	13.78	1221
3	3.5	8.43	1473
4	3.8	9.05	939

Table 4. Simulation, Synthesis Results : Blowfish

For the GOST application, the hot-spot function is found to be relatively small, thereby providing little opportunity to speed-up. Even then, the improvement is strongly different between no scratchpad access and scratchpad access. The results are summarised in table 5. Interestingly, 2 parallel scratchpad accesses resulted in poor speed-up compared to 1 parallel access. It is observed that the ISEGen left some base processor instruction out due to GPR I/O restrictions (2-input, 1-output). The base processor instructions with a subsequent custom instruction incurred exta *nop*s due to data dependency. This serves as an example of how the interface restriction can control the speed-up. By allowing increased number of scratchpad accesses a bigger data-flow graph could be accomodated in the CI, thereby avoiding the GPR restriction. Similar effect is visible for DES (table 3), too. However, 4 scratchpad accesses masked the effect of sub-optimal GPR I/O decision.

The strong improvement in the application runtime shows the importance of flexibility, which could be offered by rASIP in comparison with the ASIP. Note that the custom instructions selected for the DES applications are different from the custom instructions selected for the Blowfish or GOST application, stressing the importance of post-fabrication flexibility. The results also reflect that the improvement is strongly dependent on the application and a prudent selection of the pre-fabrication design constraints. Finally, the complete design space exploration, starting with a LISA description of the base processor, took few hours by a designer. This is a manifold increase in design productivity, while maintaining the genericness.

Number of Parallel Scratchpad Access	Speed-up	Minimum Clock Delay (ns)	Area (LUTs)
0	1.02	10.45	1803
1	1.6	9.05	1554
2	1.5	8.23	1513
3	1.7	9.05	1575
4	1.8	8.43	1455

Table 5. Simulation, Synthesis Results : GOST

7. Conclusion and Future Work

Due to the high flexibility, partially re-configurable processors are attracting significant research interest in today's processor design community. In this paper, a specification-driven design framework for rASIPs is proposed. We have separated the complete design space exploration framework into two phases, namely the pre-fabrication phase and the post-fabrication phase. The proposed design flow integrated existing and new tools for a seamless design space exploration in both the phases.

In future, we will concentrate on the design space exploration and implementation of the re-configurable blocks in the rASIP. Furthermore, the modelling and exploration for d-rASIPs will be targeted.

8. REFERENCES

[1] R. Hartenstein, M. Herz, T. Hoffmann and U. Nageldinger, "KressArray Xplorer: a new CAD environment to optimize Reconfigurable Datapath Array," in *Proceedings of the 2000 conference on Asia South Pacific design automation.*

[2] Stretch, *http://www.stretchinc.com.*

[3] B. Mei, A. Lambrechts, D. Verkest, J. Mignolet and R. Lauwereins, "Architecture Exploration for a Reconfigurable Architecture Template," in *IEEE Design & Test*, 2005.

[4] A. Fauth, J. V. Praet, and M. Freericks, "Describing Instruction Set Processors Using nML," in *Proc. of the European Design and Test Conference (ED&TC)*, 1995.

[5] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau, "EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability," in *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, 1999.

[6] Target Compiler Technologies, *http://www.retarget.com.*

[7] A. Hoffmann, T. Kogel, A. Nohl, G. Braun, O. Schliebusch, O. Wahlen, A. Wieferink, and H. Meyr, "A Novel Methodology for the Design of Application Specific Instruction-Set Processor Using a Machine Description Language," in *IEEE Transactions on Computer-Aided Design of Integrated Cicuits and Systems (TCAD) vol. 20 no. 11*, pp. 1338–1354, IEEE, 2001.

[8] F. Barat, R. Lauwereins, and G. Deconinck, "Reconfigurable instruction set processors from a hardware/software perspective," *IEEE Trans. Softw. Eng.*, vol. 28, no. 9, pp. 847–862, 2002.

[9] C. Mucci, F. Campi, A. Deledda, A. Fazzi, M. Ferri, M. Bocchi, "A Cycle-Accurate ISS for a Dynamically Reconfigurable Processor Architecture," in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, 2005.

[10] A. L. Rosa, L. Lavagno and C. Passerone, "A Software Development Tool chain for a Reconfigurable Processor," in *Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*.

[11] H. Ishebabi, D. Kammler et al., "Design of Application Specific Processors for the Cached FFT Algorithm," in *31st International Conference on Acoustics, Speech, and Signal Processing*, 2006.

[12] K. Karuri, R. Leupers, G. Ascheid, H. Meyr and M. Kedia, "Design and Implementation of a Modular and Portable IEEE 754 Compliant Floating-Point Unit," in *Design, Automation & Test in Europe (DATE)*, 2006.

[13] K. Karuri, M. A. Al Faruque, S. Kraemer, R. Leupers, G. Ascheid and H. Meyr, "Fine-grained Application Source Code Profiling for ASIP Design," in *42nd Design Automation Conference*, 2005.

[14] E. M. Panainte, S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels and G. Kuzmanov, "The MOLEN Polymorphic Processor," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1363–1375, 2004.

[15] P. Biswas, S. Banerjee, N. Dutt, L. Pozzi and P. Ienne, "ISEGEN: an Iterative Improvement-based ISE Generation Technique for Fast Customization of Processors," *IEEE Transactions on VLSI Systems*, vol. 14, no. 7, 2006.

[16] K. Atasu and L. Pozzi and P. Ienne, "Automatic Application-specific Instruction-set Extensions under Microarchitectural Constraints," in *DAC 2003: Proceedings of the 40th conference on Design automation*, 2003.

[17] R. Leupers, K. Karuri, S. Kraemer and M. Pandey, "A Design Flow for Configurable Embedded Processors based on Optimized Instruction Set Extension Synthesis," in *Design, Automation & Test in Europe (DATE)*, 2006.

[18] Synopsys, *http://www.synopsys.com.*

[19] Xilinx, *Virtex-II pro http://www.xilinx.com/products/silicon_solutions/fpgas/.*

[20] P. Biswas, V. Choudhary, K. Atasu, L. Pozzi, P. Ienne and N. Dutt, "Introduction of Local Memory Elements in Instruction Set Extensions," in *DAC '04: Proceedings of the 41st annual conference on Design automation*, 2004.