FFT processor: a case study in ASIP development

M. Nicola, G. Masera, M. Zamboni
VLSI Lab, Electronic Department,
Politecnico di Torino, Torino, Italy
mario.nicola@polito.it, guido.masera@polito.it

H. Ishebabi, D. Kammler, G. Ascheid, H. Meyr
Institute for Integrated Signal Processing Systems,
RWTH Aachen University, Aachen, Germany
ishebabi@iss.rwth-aachen.de

***Abstract*— The trend for wireless systems is characterized by the necessity to support multiple standards and transmission modes. This calls for user-transparent system reconfigurations between wireless networks and user terminals. System reconfiguration, in turn, requires flexible implementations. In addition to flexibility, the implementations must meet tight constraints with respect to both performance and development time due to the nature of the applications and of the consumer market respectively. These requirements can be met by employing Application-Specific Instruction-set Processors (ASIPs), or by employing FPGA-based solutions. In this paper, we present a case study in ASIP development based on the Language for Instruction Set Architectures (LISA), and contrast the design flow to an FPGA approach. The case study is based on an FFT processor for OFDM-based systems.**

I. Introduction

Flexibility is one of the key requirements in wireless systems beyond 3G (B3G) [1], where both user terminal and the network have to cooperate in user transparent system reconfiguration procedures: encompassing software downloading, capability negotiation, security accomplishment, setting of parameters and quality of service management. Application-specific integrated circuits (ASICs) do not offer the required flexibility but there are other approaches to meet these requirements; the simplest and most flexible one is probably the all-software solution, where one or more programmable processors are in charge of all the required functions and the system can be adapted by simply executing different software. This high degree of flexibility is achieved at a cost of low throughput and low power-efficiency. A much higher throughput can be obtained by employing Field Programmable Gate Arrays (FPGAs). Moreover, FPGA devices offer the capability of reconfiguration. A convenient combination of interconnected processors and FPGAs indubitably constitutes a very powerful and flexible solution for supporting all the processing functions in a software defined radio environment. Another possibility is the use of an ASIP (Application Specific Instruction set Processor), where a software implementation based upon a processor with optimized features is used. ASIPs can achieve a throughput comparable to FPGAs with the flexibility of software solution.

Mapping components of B3G systems to one of the above mentioned solutions implies that a trade-off between flexibility and performance has to be explored because of the different characteristics of the solutions. However, because of the requirement for flexibility and high throughput, only ASIP and FPGA solutions are of interest. Therefore, our intention is to study the suitability and the trade-off potential which are offered by these two solutions for constituent algorithms of B3G systems.

In the PRIMO project [2], several radio access schemes have been analyzed [3], mainly W-CDMA (both single carrier and OFDM approach) and ultra wideband techniques, and most demanding operations and algorithms have been investigated, including synchronization, equalization, channel estimation, multiuser detection, and channel coding. The considered system uses TDD (Time Division Duplex) with OFDM (Orthogonal Frequency Division Multiplexing) designed to obtain both a good performance in highly dispersive channels, and a good spectral efficiency. Each sub-carrier is modulated by using BPSK, QPSK, 16-QAM or 64-QAM, with a multiple access scheme based on spreading in the time domain (MC-DS-CDMA).

In OFDM [4], Fourier transform has a critical role from a performance point of view. Therefore, FFT algorithms were first considered for our studies for the suitability and trade-off potential of ASIPs and FPGAs. For the case study, the OFDM system described in [3] is used. In the PRIMO project an FPGA implementation [5] has been investigated.

In this paper an implementation based on an ASIP is presented in contrast to the FPGA implementation in [5]. The processor used for FFT implementation has been designed using LISATek tools. The rest of this paper is organized as follows: the design flow with FPGAs is presented in section II, followed by a summary of the implementation on FPGA in section III. The LISATek ASIP design flow and our case study with two design concepts for the FFT processor are discussed in sections IV and V respectively. Concluding remarks are finally provided in section VI.

II. FPGA Design Flow

Figure 1 depicts a general FPGA flow. It starts with a behavior verification of an HDL model, followed directly by the implementation which consists of synthesis, placement and routing. Evaluation of the design is performed after the implementation. Necessary changes to the architecture are applied on the HDL model. Changes can be immediately applied on the design because operators can be spatially (re)arranged to implement (other) functions, hence making FPGAs flexible. This is in turn possible because the structure of FPGAs is regular and fine grained, consisting of logic blocks and

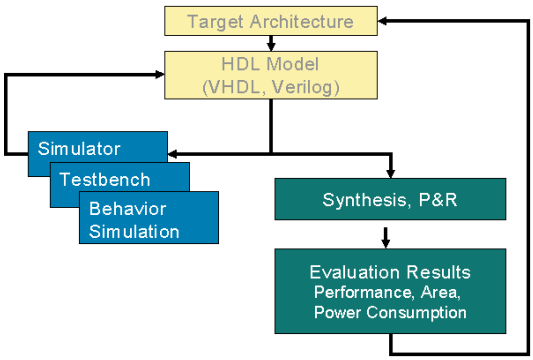

Fig. 1. FPGA development flow

interconnects. A direct consequence of this structure is that homogeneous architectures can be well mapped on FPGAs. Heterogeneous architectures consisting of control and data paths, particularly with large fan-in, lead to long paths when mapped on FPGAs.

Because of the maturity of FPGA development tools, the process of mapping a design onto an FPGA is a straightforward task. However, the design of an FPGA-device for an embedded system (eFPGA) requires a significant effort which involves circuit and physical design. Recently, efforts towards a complete automation of the design of an FPGA-device have been reported [6].

III. The Implementation on FPGA

A. Project Summary

The structure of an FFT implementation is regular, consisting of butterflies, and can therefore be well mapped on FPGAs. In the PRIMO project [2] [5], the radix-4 FFT algorithm was implemented on a Xilinx Virtex E 2000 FPGA by using Xilinx ISE 6.2. Radix-4 was chosen because of already available IP cores. The OFDM system for the case study [3] uses a 256 points FFT, with 320 samples per symbol with 64 samples cyclic prefix. With a 20 MHz sampling rate, these values imply an OFDM symbol duration equal to $\frac{320}{20MHz} = 16\mu s$. The resulting performance constraints are listed in table I.

TABLE I
FFT performance constraints

Number of points	256
Maximum time per transform	$16\mu s$
Data width	real part: 16 bits imaginary part: 16 bits
Arithmetic	fixed point

An FFT transformation is performed in three phases: first the input data is transferred into an internal memory, then the transform is computed and lastly, the result is transferred out. Two Finite State Machines (FSMs) are used to control the operation of the three phases so that the operations are overlapped. Data is transferred into and out of the FPGA by using FIFO buffers.

B. Results

The following results were obtained: design exploration and implementation consumed 4 and 2 man weeks respectively. The implementation can compute a transform in $13.53\mu s$, occupies 272.621 kGates (equivalent), and consumes 834.79 mW.

IV. LISATek ASIP Design Flow

Alternatively, a balance between tight design constraints and flexibility can be achieved by using ASIPs. In contrast to FPGAs, the flexibility of ASIPs is attributed to programmability. Programmability insures the flexibility of the control flow of an implementation, as well as of the schedule of its operations. Specialized instruction sets and application-specific optimizations lead to efficient program execution. Therefore, applications which are both control and data intensive can be efficiently implemented by using ASIPs.

ASIP design requires not only the development of the architecture, but also the development of corresponding software development tools (simulator, compiler, assembler and linker). The design task is typically an iterative process, whereby, starting from an initial architecture (template), refinements are made until design goals are met. This means that, for each iteration, new software development tools are required which match the new architecture. When the design process is finished, production-quality software tools have to be developed. Obviously, this complex process can only be completed with a reasonable effort and within a reasonable time if high-level abstraction models are used. This, however, poses an additional challenge because a link to gate-level synthesis has to be established from the high-level abstraction model.

The time and effort of designing a new ASIP can be significantly reduced by using a retargetable modeling approach based on an Architecture Description Language (ADL). The Language for Instruction Set Architectures (LISA)[7] was developed for the automatic generation of consistent software development tools and synthesizable HDL code out of a single processor description. A LISA processor description consists of the instruction-set, the behavior and the timing model of the architecture. Changes in the architecture are easily applied to the LISA model. Since software development tools and HDL-models are automatically generated from the same LISA model, such changes are reflected in the whole software chain and in the HDL-model, insuring consistency across all tools and levels of abstractions. Moreover, the speed and functionality of generated software tools allow for usage after the ASIP development is finished so that no upgrading to production-quality tools is necessary. Because of the automation, the need for an ASIP designer to have expertise in both the hardware and software design domains is eliminated.

ASIP design with LISA follows two loops as shown in figure 2. A LISA processor description is used to generate software development tools. These, together with the target application, are used to verify and evaluate the model with respect to functionality and performance. Evaluation results are then used to guide possible architecture modifications.

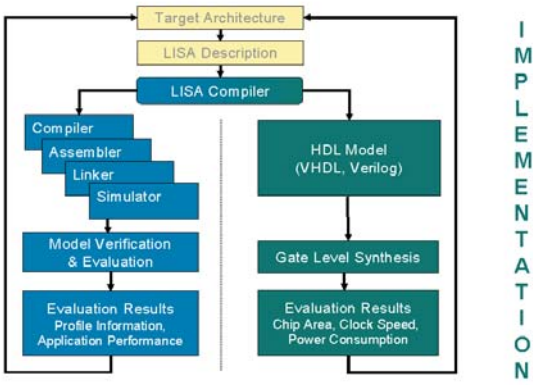

Fig. 2. LISATek based processor architecture design loops

Design parameters (area, timing, energy consumption) can be obtained and utilized through RTL processor synthesis and gate-level synthesis. In this way, architecture alternatives can be rapidly explored. The RTL processor synthesis supports an automated and optimized ASIP implementation in multiple HDLs. The design methodology and the RTL processor synthesis are described in detail in [8] and [9] respectively.

LISATek is a tool suite which is based on the LISA language and covers support for the exploration as well as for the implementation loop. Both the language and the tool suite were developed by the Institute for Integrate Signal Processing Systems (ISS) at RWTH Aachen University [10], and are now commercialized by CoWare [11].

V. A CASE STUDY: FFT PROCESSOR FOR OFDM

In order to explore possibilities obtained by using LISATek tools, a processor suitable for FFT implementation has been designed. The constraints that were used are the same as for the PRIMO project (sect. III, table I). In order to implement the FFT, the radix-2 algorithm [12] has been chosen due to its regularity that makes it a good candidate for a software implementation.

Several solutions have been investigated, by exploiting the reduction of the design time obtained with the use of LISATek tools. Finally, two different approaches can be distinguished:

- an architecture with a highly optimized data path, i.e. a processor with instructions specific for FFT computation
- an architecture with a highly optimized control path, i.e. a processor with instructions for speeding up the control flow of FFT computation

These solutions will be presented in following sections.

A. ASIP with FFT optimized data path

The first investigated implementation is a processor with a highly specialized instruction set for FFT computation. Its main features are complex data support, result bypassing and delayed branch.

In the designed processor, each instruction can elaborate complex data, obtaining complex results. The programmer can choose to elaborate only the real part of data, or the imaginary one, to obtain more flexibility. This is accomplished by using a flag that specifies the data type in the assembly program. The processor will access registers and memory according to the chosen behavior.

With result bypassing, an instruction can use the result of the previous one. The user can disable bypass by using a flag in the assembly program.

Delayed branch is a technique that is able to reduce speed penalties due to conditional jumps in executed program: when a branch is found, the processor will execute the following 2 instructions, regardless of whether the jump is executed or not.

This processor uses a 6-stage pipeline: ISTRF (ISTRuction Fetch), OPF (OPerand Fetch), 3-execution stages (EX, EX2 and EX3), and a ST (STore) stage. This is shown in figure 3, where also the bypass mechanism is outlined. This processor does not have a *Load&Store* architecture, because some instructions support direct access to data memory. Some special

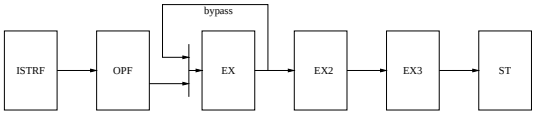

Fig. 3. 6-stage pipeline with bypass

features are implemented to match speed requirements.

1) Specialized resources: This processor uses three different memories: a program memory, a data memory and a coefficient memory. The coefficient memory is a read-only memory that can be used as a LUT (look-up table) to implement non-linear functions, for example, in FFT implementations, to store sinusoidal coefficients. In both data and coefficients memory, complex data are stored in the way depicted in figure 4. To

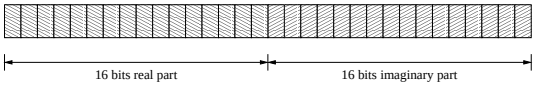

Fig. 4. How data are stored in memories

store complex data, there are two different general purpose register files, made of 32 registers each: one for the real part of data and one for the imaginary part. A programmer can access both real and imaginary parts together, and can access both data and coefficients at the same time.

2) Specialized instructions: An instruction able to calculate a whole butterfly (see [12]) has been implemented. With this solution user can read input data, execute arithmetic operations, store results, and update memory addresses in one clock cycle, because only one instruction has to be fetched. For a butterfly implementation with a throughput of 1 instruction per cycle, a complex multiplication, followed by a sum and a subtraction must be supported. With a straight implementation of the multiplication, 4 multipliers, 2 adders and 2 subtracters are needed. These resources are distributed into the 3 EX

pipeline stages which are shown in figure 3 so that speed penalties do not result.

B. ASIP with FFT optimized control path

Time constraints can be met by employing features, addressing modes and instructions for speeding up the control flow.

1) Maintained features: This processor supports complex data. Memory resources are the same as in the previous version and the pipeline is also very similar (figure 5). Bypass mechanism and delayed branches are supported.

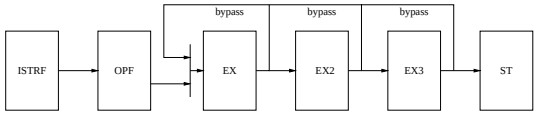

Fig. 5. 6-stage pipeline with bypass

2) New features: In the the following subsections two new features are discussed: Automatic Index Update (AIU) and Zero Overhead Loop (ZOL).

a) AIU: When an instruction accesses a memory location through a register, the processor can automatically increment the content of the register. A programmer can enable this behavior by using a post-increment flag with the desired operand. If this flag is not present, AIU mechanism is disabled. For example:

```
ADD      *R[0], *R[1]++
```

in this line, memory locations which are addressed by `R[0]` and `R[1]` are added together, but, after the memory access, `R[1]` is incremented, while `R[0]` is not. The user can choose the value of the increment by using the `SET_INCREMENT` instruction.

At the same time, a memory location can be accessed by bit-reversing the value of the register. The user can select this behavior by using an optional flag `rev`, as shown by this sample code

```
ADD      *R[0]rev, *R[1]rev++
```

In this example, both memory locations are accessed using address bits in reversed order. Two notes should be added about the use of the `rev` flag:

- the content of the register is not affected by the `rev` flag since the value obtained by bit-reversing the address is used for memory access but is not stored in the register
- the user can select the number of bits to be reversed by using the instruction `SET_NOBTBR`

In the previous sample code, if value stored in `R[0]` and `R[1]` are 1 and 3 respectively, and if the number of bits to be reversed is 4, and if the value of the increment is 1, then the instruction accesses memory locations 8 and 12, while final values stored in registers are 1 and 4 due to the ++ flag in the second operand.

b) ZOL: In this processor, a zero overhead loop is implemented by using two general purpose registers containing the start and target address of the loop, and the initial and current value of the loop index respectively. The user can declare a ZOL by performing these operations

- declare a start address by using the `SET_STARTADDR` instruction
- declare a target address by using the `SET_TARGETADDR` instruction
- set an initial value for the index and the number of iterations by using the `SET_INITVALUE` instruction
- activate the loop by using the `SET_ZOL` instruction

Also nested loops can be defined, but two rules have to be respected: up to 4 loops can be present at the same time, and nested loops have to be declared starting from the most external loop to the most internal one. In figure 6, a pictorial

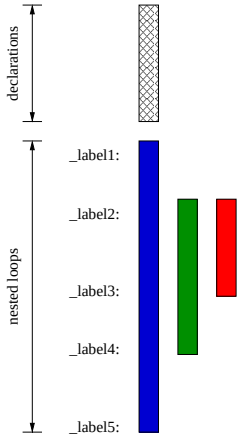

Fig. 6. Nested loops

representation of three nested loops is given. These loops can be declared as follows

```
SET_STARTADDR   R[10], @_label5
SET_TARGETADDR  R[10], @_label1
SET_INITVALUE   R[11], #2
SET_ZOL         R[11], R[10]

SET_STARTADDR   R[12], @_label4
SET_TARGETADDR  R[12], @_label2
SET_INITVALUE   R[13], #2
SET_ZOL         R[13], R[12]

SET_STARTADDR   R[14], @_label3
SET_TARGETADDR  R[14], @_label2
SET_INITVALUE   R[15], #4
SET_ZOL         R[15], R[14]
```

Loops have to be declared before the outer loop (see fig. 6). After a `SET_ZOL` instruction is executed, the zero overhead loop is registered and the processor is able to find and execute it. That is to say, after the `SET_ZOL` instruction, the processor continues to fetch instructions sequentially until the loop is found.

The loop execution is fully automatic, since the processor automatically updates current value of index, re-initializes nested loops, and deletes nested loops when the outer loop is finished. This approach grants better performance compared to the usual zero overhead approach where the loop is declared by introducing an instruction before the first instruction of the loop. In fact, when nested loops are present, the traditional approach needs the loop to be initialized as many times as the external loop is executed, whereas in the proposed approach, a loop is defined just once outside of the external loop, and no instructions have to be inserted in the loop. In FFT radix-2 algorithm this approach saves more than 100 instructions for each block.

C. Results and Comments

The development time for both processors, including exploration, modeling, validation, verification and HW implementation, consumed 3 man weeks.

For RTL simulations, synthesis and power calculations, Synopsys Scirocco, Design Compiler and Prime Power were used. Both processors have been synthesized in a $0.13\mu m$ technology, and are able to meet the time constraint. Because of the memory architecture and because of the pipelining, the speed of algorithm execution depends directly on the number of required instructions in the program.

- The solution with an optimized data path has a 200 MHz maximum clock frequency. So, this processor is able to execute 3200 instructions in $16\mu s$. With supported instruction set, FFT radix-2 algorithm is implemented by using 2056 instructions. The number of gates consumed is 106 kGates. The pre-layout power consumption for the processor core is 95.915 mW.
- The solution with an optimized control path has a 192 MHz maximum clock frequency, so this processor can execute up to 3072 instructions in $16\mu s$. With supported instruction set, FFT radix-2 algorithm is implemented by using 3012 instructions. The number of gates consumed is 110 kGates. The pre-layout power consumption for the processor core is 169.754 mW.

The second processor achieves a lower performance, and the constraint is met with a thin margin. But the mechanism implemented in this processor can improve the performance regardless of the algorithm.

The features utilized in the first processor are not flexible. That is to say, algorithms different from FFT can be performed using this processor (e.g. a FIR filter), but performances will be poor since the specialized features are applicable for FFT radix-2 only.

These two solutions show performance-flexibility trade-off possibilities in the ASIP design space. A very high performance can be achieved with a highly specialized data path. Reasonable performance and a high degree in flexibility can be achieved with a highly specialized control path.

Contrasting to the results of the FPGA implementation, the design time for the ASIPs is comparable to that for the FPGA implementation. It is even lower[1]. Both versions of the FFT processor have a lower gate count than the FPGA implementation. The power consumptions cannot be directly compared. This is because the estimate for the FPGA is conservative due to a large overhead. The overhead is caused by power-consuming components of the discrete FPGA device which are not part of the actual design. In addition to that, the power estimate for the ASIPs is based on pre-layout data, and can therefore vary.

VI. Conclusions

A case study for ASIP development with LISA has been presented and contrasted with an FPGA design flow. Two different design concepts were used: utilization of application-specific data path optimizations, and utilization of application-specific control path optimizations. The case study has shown how trade-offs between flexibility and performance in ASIPs can be made. It has further been shown that advanced processor features which have been tailored to a particular application can be utilized to meet tight design constraints without sacrificing flexibility. Particularly, the case study has revealed that the FFT algorithm for OFDM-based systems can be efficiently implemented by using ASIPs with a high degree in flexibility.

References

[1] J. Mitola, *Software radio architecture : object-oriented approaches to wireless systems engineering*. New York: Wiley, 2000.

[2] S. Benedetto, G. Masera, G. Olmo, G. Guidotti, P. Pellegrino, and S. Pupolin, "Primo: Reconfigurable Platform for Wideband Wireless Communications," in *Proc. 7th International Symposium Wireless Personal Multimedia Communications (WPMC) 2004*, Abano Terme, Italy, Sept. 12–15, 2004.

[3] P. Bisaglia, "Proposed Scenario for 4G Cellular Systems," FIRB-UniPD, Tech. Rep., 2004.

[4] L. Hanzo, M. Münster, B. Choi, and T. Keller, *OFDM and MC-CDMA for broadband multi-user communications, WLANs and broadcasting*. Chichester, UK: Wiley, 2003.

[5] B. Cerato and G. Masera, "FFT Processor for OFDM Application," FIRB-PoliTO, Tech. Rep., 2004. [Online]. Available: http://primo.ismb.it/

[6] I. Kuon, A. Egier, and J. Rose, "Design, layout and verification of an fpga using automated tools," in *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*. ACM Press, 2005, pp. 215–226.

[7] Pees, S. and Hoffmann, A. and Zivojnovic, V. and Meyr, H., "LISA – Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures," in *Proceedings of the Design Automation Conference (DAC)*, New Orleans, June 1999.

[8] Hoffmann, A. and Schliebusch, O. and Nohl, A. and Braun, G. and Meyr, H., "A Methodology for the Design of Application Specific Instruction Set Processors (ASIP) Using the Machine Description Language LISA," in *Proceedings of the International Conference on Computer Aided Design (ICCAD)*. San Jose, USA: IEEE/ACM, Nov. 2001.

[9] Schliebusch, O. and Chattopadhyay, A. and Kammler, D. and Ascheid, D. and Leupers, R. and Meyr, H. and Kogel, T., "A Framework for Automated and Optimized ASIP Implementation Supporting Multiple Hardware Description Languages," in *ASP-DAC*, Shanghai, China, Jan 2005.

[10] [Online]. Available: http://www.iss.rwth-aachen.de/

[11] [Online]. Available: http://www.coware.com/

[12] R. Blahut, *Fast Algorithms for Digital Signal Processing*. Reading, MS: Addison-Wesley Publishing Company, 1985.

[1]The design time for both approaches does not include that for physical design