A Methodology and Tooling Enabling Application Specific Processor Design

Frank Fiedler
CoWare, Inc.
Dennewartstrasse 25-27
52068 Aachen, Germany
frank.fiedler@coware.com

Achim Nohl
CoWare, Inc.
Dennewartstrasse 25-27
52068 Aachen, Germany
achim.nohl@coware.com

Andreas Hoffmann
CoWare, Inc.
Dennewartstrasse 25-27
52068 Aachen, Germany
andreas.hoffmann@coware.com

ABSTRACT

This paper presents a highly efficient processor design methodology based on the LISA 2.0 language. Typically the architecture design phase is dominated by an iterative processor model refinement based on the results of hardware and software simulation and profiling. Thus, traditionally huge teams of hardware and software experts are required to design new programmable architectures. The proposed design flow reduces the design time and enables even non-processor experts to overcome the typical design challenges. The presented design methodology is based on a workbench that automates the generation of all required software tools and furthermore closes the gap between high level modeling and hardware implementation via automatic generation of a Register Transfer Level (RTL) model for the target processor.

A case study demonstrates the design approach discussing the application specific instruction-set processor (ASIP) design for a Fast Fourier Transformation (FFT) algorithm. Several processor types such as SIMD and VLIW with various characteristics have been explored to find an optimal processor implementation for this algorithm.

Programming Languages: LISA 2.0

General Terms: Design, Languages

Keywords: ASIP, SIMD, VLIW

1. INTRODUCTION

Most of today’s SoC designs involve one or more embedded processor cores that execute the software components of a system. While a large number of mixed hardware/software SoC designs are still based on standard off-the-shelf RISC or DSP cores, there is a clear trend towards specialization of processors towards the intended applications, so as to achieve an optimum balance between computational efficiency, re-use opportunities, cost, and flexibility [3]. Within such designs often intended for mobile applications the importance of power consumption is growing. Additionally the life time of devices is decreasing, thus, the efficiency of the architecture design is getting more and more important. For this reason the design of ASIP’s has received high attention in academia and industry.

In opposite to the design of traditional general purpose processors the ASIP design flow is driven by a set of target applications, frequently specified in C/C++, SPW or Matlab. The design generally starts with an successive architecture exploration process that involves stepwise refinement of the architecture and timing abstraction levels based on the simulation and profiling results during architecture development. The abstraction levels range from un-timed high-level language instruction set accurate (ISA) models down to cycle-accurate RTL HDL synthesis models.

Due to the model refinement capabilities of an instruction set language (ISL) the designer can abstract from architecture details and concentrate on the essentials in early design phases which is a big advantage in opposite to the traditional purely HDL based processor design flow.

Tensilica’s customizable Xtensa processor [4] is a popular example for this type of design flow. While the Xtensa approach utilizes a partially predefined RISC core, this paper focuses on an architecture description language which has a higher degree of freedom in modeling the ASIP’s instruction set and the micro-architecture. The architecture design framework presented in this paper is based on the hierarchical processor modeling language LISA 2.0 [2].

The LISA 2.0 language has been developed by the Institute for Signal Processing Systems (ISS) and was licenced to CoWare Inc [1]. CoWare has commercialized the LISA 2.0 based tool-suite under the name “LISATek”. LISA 2.0 allows the designer to create a custom processor model on various abstraction levels. In every design phase the designer can generate a complete set of software development tools (C-compiler, assembler, linker) and an instruction-set simulator to verify and to profile the current state of the architecture implementation. These hardware and software profiling capabilities are essential to the stepwise processor adaption to the application’s needs. Handwriting these tools after each architecture refinement would not allow an iterative exploration process since the manual creation is a lengthy and error-prone process. Besides, hardware implementation capabilities are provided via automatic generation of synthesizable RTL code taking hardware properties such as chip area, clock, etc. into account.

The rest of the paper is organized as follows: Section 2 introduces the role of ASIPs in todays SoCs. Section 3 analyzes the work done in academia and industry in the field of ASIP design. Section 4 outlines the LISA 2.0 language principles and the design flow. Section 5 discusses the design space exploration for a FFT algorithm using the Architecture Description Language (ADL) LISA 2.0. Section 6 elaborates the results of the different ASIP implementations which have been carried out. Section 7 concludes the paper.

2. PROCESSOR LANDSCAPE

Figure 1 shows a classification of ASICs vs. programmable solutions. ASICs have a high energy efficiency measured in MOPS/mW – this means on the one hand they offer very high performance while keeping low power consumption and small area. The drawback is that they have no flexibility to adapt to changes. The other extreme are RISC or DSP microprocessors, which are programmable and thus, offer the required flexibility but cannot cope with the requirements of modern portable devices with respect to power consumption. It is widely accepted to have one controller e.g. ARM/MIPS and one DSP in such systems. However, with more
functions running on the device, it is hardly acceptable to spend more RISC and DSP cores to process things like digital video, multimedia and wireless communication. The truth is a compromise between an ASIC and a programmable solution. These architectures will be very close to ASIC with respect to energy efficiency and size while being programmable. This is – at the first glance – a paradoxon. It is made possible by processors which are very application specific. They only provide limited programmability which is just enough to adapt to changes e.g. in standards. They often feature complex instructions which can execute in some cases things like FFT calculations in one instruction. This is the key to the energy efficiency.

4. LISATEK DESIGN METHODOLOGY

The idea of the LISATek design flow is to define a programmable platform tailored to a specific application domain. This puts a heavy burden on the ASIP designer to compose a capable platform from a huge design space for the target application. The goal of the LISA 2.0 based processor design flow is to guide the designer from the algorithmic specification of the application down to the implementation of the micro-architecture. In every phase of the processor design the designer maintains an abstract model of the target architecture written in the LISA 2.0 language. The LISA 2.0 is aiming at the formalized description of programmable architectures, their peripherals and interfaces. LISA 2.0 is not a completely new language – it is an extension to C. The hardware behavior as well as processor resources like registers are modeled in pure C, whereas LISA 2.0 adds on top of the C-language capabilities to describe an instruction-set with its binary encoding and assembly syntax. Also, LISA 2.0 allows to express timing in processors. An example is a pipelined architecture where instruction execution is spread over multiple cycles. LISA 2.0 is very easy to learn so that a couple of days is sufficient to become familiar with this language.

From such a model, a working software development tool set supporting the evaluation tasks for the current development phase can be generated automatically. The first stage of the design process is concerned with the examination of the application to be mapped onto the processor architecture. Critical portions of the applica-

Figure 1: ASIC vs. Programmable Solutions

Due to the high degree of specialization of these processors, there will be separate processors for different applications like digital video, wireless communication, multimedia, etc. One major benefit of ASIPs is the re-use of proven and stable programmable IP.
Also, there are sample models for different architecture categories available which cover DSPs, micro controllers with specific features like SIMD (single instruction multiple data) which is popular in the multimedia domain as well as the increasingly popular VLIW architectures which comprises massively parallel functionality. It is important to distinguish these sample models from configurable template models where only some parameters may be changed. Taking such models as a basis has the major advantage of the LISA 2.0 language besides its ability to model arbitrarily complex processors is a special template library with memory modules which can be easily parameterized from within the LISA 2.0 model. Using these library elements, caches, MMUs and buses can be easily modeled.

When assigning different parts of the instruction execution to the already defined pipeline stages the developer must care about resource sharing and the length of the critical path in the emerging architecture. This is important for an efficient hardware implementation of the ASIP. Here the required chip area and the gate count are commonly used constraints which directly refer to the power consumption of the resulting hardware. The critical path is important, since its length limits the clock speed which is another important criteria when designing an ASIP.

When the architecture meets the design criteria and efficiently implements the application, in a final architecture development step hardware implementation is done. Synthesizable HDL RTL code (currently VHDL and Verilog) for the control and data path of the processor can be derived from the abstract processor model automatically. This includes the entire hardware model structure such as the pipeline, pipeline controller including complex interlocking mechanisms, forwarding, etc. to steer the architecture’s behavior and an implementation of the data path which is directly derived from the behavioral specification in the LISA 2.0 model. Having the RTL generation capabilities in the processor exploration loop allows to easily explore the trade off area vs. timing (clock speed) vs. flexibility. Based on the simulation and synthesis results of the hardware, the abstract LISA 2.0 model might be modified to meet power-, area- and frequency constraints. Due to the fact that RTL and ISS simulator are derived from a sole processor model, they are automatically consistent. It is obvious that deriving both software tools and hardware implementation model from the same architecture specification in LISA 2.0 has significant advantages. Only one model needs to be maintained, even if changes to the micro-architecture or the behavior in the hardware model must be realized.

Once the processor design is finished, a set of production quality software development tools is generated from the LISA 2.0 model. These software tools (C-compiler, assembler, linker) can compete well in terms of functionality and feature richness with state-of-the-art tools from Greenhills, ARM, etc. The generated C-Compiler is an optimizing compiler which is capable of generating code which is close to handwritten assembly code. In addition to these generated software development tools a macro assembler and an archiver are provided for the LISATek product family.

In order to be able to integrate into system simulation environments (SoC) to gather realistic stimuli from the system, LISATek generates processor simulators which couple directly with the following tools: CoWare ConvergenSC, Cadence Incisive, Mentor Graphics Seamless CVE, OSCI reference simulator as well as with any C-based environments. The generated simulators automatically interface with popular busses like AMBA AHB and can be extended to work with proprietary busses easily. The generated instruction-set simulators (ISS) support system simulation on different levels of abstraction from cycle accurate to untimed system simulations. Utilizing the patent pending Just-In-Time-Cache-Compiled (JIT-

**Figure 2: ASIP Exploration and Implementation Phases**

As a starting point for model creation CoWare LISATek provides a library of sample models which contains processors that are already tailored to specific applications. These processors efficiently implement algorithms like turbo decoding and the FFT. Also, there are sample models for different architecture categories available which cover DSPs, micro controllers with specific features like SIMD (single instruction multiple data) which is popular in the multimedia domain as well as the increasingly popular VLIW architectures which comprises massively parallel functionality. It is important to distinguish these sample models from configurable template models where only some parameters may be changed. Taking such models as a basis has the major advantage of directly have compiler support for the architecture due to the existence of an instruction-set. This makes the C- and instruction profiling of the application possible from the very beginning of the architecture development. The simulator which is derived from this model constitutes a virtual machine executing the application directly. The profiling capabilities of the simulator are used to generate execution statistics of the application code. Once the profiling information is gathered, critical portions which require parallelization are identified. Based on the profiling results the instruction-set is adapted until the application profiling meets the given criteria.

At this point in the design phase the designer has to consider different aspects of the micro architecture. The major exploration and optimization point is the pipeline. The designer has to decide how many pipeline stages are required with respect to control flow instructions and the efficient implementation of hardware loops and interrupts. For the performance of the architecture it is important to avoid data hazards during program execution. For this reason data bypassing may be implemented when specifying the pipeline of the processor. This mechanism serves already calculated results to pipeline stages prior in the pipeline. The intention of this mechanism is to bypass data storage in registers or memories. Having an optimal pipeline in an ASIP requires a memory subsystem to support this pipeline with memory data fast enough because otherwise the pipeline has to be stalled working on the application while waiting for the memory data. The memory hierarchy directly contributes to the performance of the memory sub-system, thus, the developer has to consider it carefully. Caches with varying parameters are widely used to enhance the performance of the memory subsystem. Here the cache parameters i.e. the cache size and the cache read and write policies must be determined with respect to the target application. Additionally, the designer has to evaluate the role of a memory managing unit (MMU) and has to check the performance of the utilized bus to ensure the optimal configuration of the utilized memories. A very powerful capability of the LISA 2.0 language besides its ability to model arbitrarily complex processors is a special template library with memory modules which can be easily parameterized from within the LISA 2.0 model. Using these library elements, caches, MMUs and buses can be easily modeled.
CC) simulation technology LISATek simulators run at a very high speed. Moreover, LISATek tools support multi-processor debugging in such a system. Here, the designer can debug one ore more processors with a single graphical debugger. The verification of the ISS vs. the RTL model can be performed using the IBM Genesys [17] test-generation tool. Genesys is a test-generator which has been exclusively developed for validating processors. It works based on a test plan and generates test programs automatically which are run directly on an ISS. Finally the test-program together with the expected result values in the processor are given.

A major benefit of the LISATek approach is the fact that the designer has neither to be a software nor a hardware expert. So a single person can cover a broad spectrum of development tasks which cannot be covered by the traditional ASIP design approach.

5. FFT ASIP

To demonstrate the strength of the presented design flow and how a highly specialized processor can be designed with the LISATek technology, a case study has been carried out. Different ASIP types with different instruction-sets have been explored to find an optimal processor implementation for the image processing domain. This means the processor is highly optimized for compression algorithms and image transformation.

5.1 Design Space Exploration

In order to find the processor that optimally fits the application, it is mandatory to explore the huge design space (see section 4). The required instruction-set has to be identified based on a high level application profiling. The decisions about the micro-architecture must be made and the critical path must be as short as the implementation constraints define. Also the ASIP must be prepared for later inclusion into a SoC, the interaction with the peripherals must be considered when designing this new processor. Due to the limited scope of this paper, this case study concentrates on the discussion of some very essential high level design steps. The effects of VLIW (parallel instructions), SIMD (single instruction) and special purpose instructions on the ASIP performance and RTL are explored here. The starting point for the design is one of the sample models (figure 3) that come with the LISATek tools.

![Figure 3: Architecture Exploration Starting Point](image)

To take this model as basis makes sense for a very quick design start. This tiny model has already a minimal instruction-set that makes it compiler programmable. Additionally, the micro-architecture etc. is very simple and let the designer easily adjust or tailor the architecture to meet his requirements. The chosen architecture is a 32bit processor with a three stage pipeline, a branch logic, a simple ALU, a multiplier, and a load-store unit to communicate with the data-bus. This processor is later tailored to the execution of a FFT and functionality which is not needed is discarded. A 32bit architecture has been chosen as starting point since the FFT requires three 8bit values for calculation. For this reason 32bit is the minimum to encode these values within one instruction word.

The first step in the exploration phase is to perform a high level profiling of the FFT algorithm. When analyzing the C-code profiling and instruction (assembly-) level profiling results shows that the hot-spot of the target application is the FFT kernel computation.

The C-profiling results (see table 1) clearly indicate that the most time is spent in the FFT and inverse FFT transformation which both make use of the kernel functionality. For this reason the focus is set on the efficient implementation of this kernel computation. The FFT algorithm kernel data flow is shown in figure 4. The FFT kernel is processing two complex samples indexed with i and j. Each sample has real and imaginary part. For the computation the FFT kernel furthermore needs a sin and cosin factor for complex multiplication. The output of the FFT kernel are the two transformed complex samples.

![Table 1: C Profiling Results](image)

<table>
<thead>
<tr>
<th>Section</th>
<th>Calls</th>
<th>Total Steps</th>
<th>Total Steps %</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>6144</td>
<td>35610624</td>
<td>53.62</td>
</tr>
<tr>
<td>IFFT</td>
<td>6144</td>
<td>34292736</td>
<td>51.83</td>
</tr>
<tr>
<td>kernel</td>
<td>196808</td>
<td>58982400</td>
<td>89.15</td>
</tr>
</tbody>
</table>

![Figure 4: FFT Kernel Computation Scheme](image)

The computation itself consists of two phases: Four normalized multiplications are followed by multiple sequential add and subtraction instructions. This means that from input to output a sample is going through a multiplier and two adder stages. A pure software implementation on an un-optimized processor would require the following sequential code (see figure 5) for the FFT execution. This execution costs hundreds of cycles.

For the above mentioned reasons such an implementation would lead to a cycle and code-size expensive software implementation. A better solution is a special purpose FFT hardware unit which is part of the targeted ASIP. Analyzing the data dependencies of the FFT, it turns out that each color of each sample can be processed in parallel. A fixed data-path for this FFT unit would result in a faster and smaller implementation than the software solution (less memory for program code).
5.2 SIMD FFT ASIP

The introduced FFT unit is able to apply the same transformations on all 3 colors (RGB) of all 4 samples (real and imaginary part) in parallel (12 channel SIMD unit – 3 Colors * 2 Samples * 2 Parts).

![Figure 6: Packed Multiply](image)

The parallel FFT hardware kernel will compute the 12 multiplications in parallel followed by two adder stages. Before and after the transformation the 8bit RGB values are extracted (UNPACK) and re-assigned (PACK) to the 32 register in a fixed logic. 

Figure 6 shows a scheme of the packed multiplication performed in the SIMD unit of the ASIP.

All three colors RGB (multiple data) are handled within one multiplication. Thus, three parallel 8bit multiplications are performed in a single step. The broken carry path between the 8bit slices allows the separate calculation. The critical path is going through an 8bit multiplier and two adder stages, this is even less (1/4) than the critical path of a standard 32bit multiplier we used 12 times in a sequence in the software implementation. Hence the complete FFT can be processed in a single cycle instead of hundreds.

The fixed PACK/UNPACK data path is very popular in image processing machines. It is comparable with the Intel multimedia instruction set extensions (MMX) [16]. It makes the expensive shifting and masking operations in software unnecessary. Figure 7 shows the structure of the resulting SIMD ASIP which has been developed.

![Figure 7: SIMD ASIP](image)

Compared with the unoptimized 32bit architecture (starting point of the processor design) this processor has additional special FFT registers and a SIMD FFT unit which is located in the execute stage. This unit comprises the functionality introduced above. At this point it could be checked which impact the number of pipeline stages, the memory structure and all the other criteria mentioned above have on the efficiency of the ASIP, but the full design space exploration capabilities provided by the LISATek ASIP design methodology are out of the scope of this paper.

During the exploration phase two further ASIPs have been developed to implement an effective FFT algorithm, a VLIW architecture and an image processing ASIP which has special instructions well suited for the given problem. To discuss all architecture details, the hardware and software profiling results which caused the design decisions cannot be covered in this paper, thus, only a brief overview on the specialities of these processors is given. Section 6 discusses the results of the different ASIP implementations.

5.3 Alternative ASIPs for FFT

The developed VLIW ASIP has the big advantage of highly parallel application execution, which is possible via multiple slots running simultaneously. In comparison with the SIMD approach this processor is more generally applicable for a broad spectrum of target applications, since it has no special instructions for FFT computation. The drawback is a greater resulting chip area in the implementation phase. The base structure of the VLIW ASIP is the same as the processor which has been the starting point for our exploration. Only the logic of the execute stage exists twice. Due to this fact it is possible to perform two load/store, ALU and multiply operations in parallel.

The third ASIP which has been created is an image processing ASIP which has special instructions for the different phases of the FFT kernel computation. Special units in the execute stage of our ASIP consist of Pack and Unpack instructions. Packing means to compose one 32bit register value by three 8bit values. Unpacking is the opposite, the first, second and third 8bit values are extracted from a given 32bit register value. Additionally, this processor has a special unit for a dual 32bit multiply and accumulate operation which makes it very easy to compute the FFT with this processor.

6. RESULTS

Three very different processor types have been developed to solve the problem of the FFT computation in hardware while having the flexibility of programming these architectures.

This section discusses the allocation results based on the following topics: Cycle counts, compiler support, re-usability and hardware implementation results. Figure 8 illustrates these categories for ev-
ery processor analyzed here. In the left most column of the table the results for the unoptimized 32bit architecture (the starting point of the architecture exploration) are given. These results are not discussed here, they are only shown here to give a hint on how a not optimized processor performs for a special application like the FFT.

### Results

<table>
<thead>
<tr>
<th>Cycles</th>
<th>SIMD FFT ASIP</th>
<th>VLIW ASIP</th>
<th>Image Proc. ASIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total App/FFTKernel*</td>
<td>294 / 70.4M</td>
<td>14 / 11.4M</td>
<td>146 / 37.1M</td>
</tr>
<tr>
<td></td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
</tbody>
</table>

**General Compiler Support**

<table>
<thead>
<tr>
<th>Specialization Compiler Support</th>
<th>FFT only</th>
<th>all</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES, via intrinsic</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>YES, automatically</td>
<td>YES</td>
<td>YES</td>
</tr>
</tbody>
</table>

**Specialization Re-usability**

<table>
<thead>
<tr>
<th>First iteration of RTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gates 33K/21K</td>
</tr>
<tr>
<td>Clock 166Mhz</td>
</tr>
<tr>
<td>YES</td>
</tr>
<tr>
<td>152Mhz</td>
</tr>
<tr>
<td>236Mhz</td>
</tr>
<tr>
<td>127Mhz</td>
</tr>
</tbody>
</table>

* Registerfile: 32 x 32bit registers, can be easily reduced to 16 x 32bit

**Figure 8: Architecture Exploration Results**

The required cycle count for the FFT calculation (total application) ranges from 11.4 (14) million cycles for the SIMD ASIP to 47.3 (182) million cycles for the image processing ASIP with the special instructions for the FFT. The efficiency of the SIMD implementation is caused by the fact that only this processor is able to perform the FFT computation in a single cycle which is tremendously fast.

For all processors, except for the SIMD ASIP, full compiler support is available, thus, the programming of them is very comfortable. The SIMD processor requires inline assembly (intrinsic) for the special SIMD instruction. Instructions with more than one output (the SIMD has three) cannot be supported by a compiler. For all other instructions of the SIMD compiler support is available.

A big benefit of programmable architectures is the re-usability. The VLIW architecture is not specialized for this problem, so it is reusable for all applications. The image processing ASIP contains special instructions which can be used for all RGB operations, but the SIMD approach is so specialized that this processor is only intended to run the FFT application. This result demonstrates the trade-off between re-usability and efficiency.

In order to get an idea of the later hardware implementation, a first iteration of RTL cycle generation is carried out for all of these processors. The gate count for the different processors ranges from 30k gates for the SIMD to 94k gates for the VLIW ASIP. While the maximum clock speed has been 236MHz for the VLIW processor, the lowest speed has been 127MHz for the image processing ASIP.

Three different architectures have been examined for the given algorithm inclusively the design space exploration as introduced above. The entire design flow for all of these processors have been performed beginning from the functional description of the problem down to the hardware implementation within two man-weeks. This time also includes the creation of architecture simulators and production quality software development tools. This extremely short development time demonstrates how effective the LISATek design methodology is.

### 7. CONCLUSION

This paper introduced a highly efficient methodology to design ASIPs. The presented design approach is based on the LISA 2.0 language and enables even non experts to cover all processor development tasks from the abstract functional specification of the target architecture down to the hardware implementation. Using LISA 2.0 allows to do architecture exploration and implementation on a very high abstraction level.

It has been highlighted within the scope of this paper that the importance of ASIPs in today’s industry is enormously growing. Due to extremely short product life times and thus, development cycles for new systems, the efficiency of the architecture design tools directly contributes to the success of the companies. The LISA 2.0 based architecture development approach can cope with these requirements which was demonstrated by a case study which focused on the ASIP design for the FFT algorithm.

In this study three different types of ASIPs have been explored for the FFT computation. It has been possible to provide for every of the processors an architecture simulator, software development tools (C-compiler, macro-assembler, assembler, linker, archiver) and an RTL hardware model within two man-weeks.

The usage of this methodology in big IP vendors and the chip industry has shown that the overall performance gain in ASIP design is at least 50 percent [15]. This number does not include the amount of time which would be necessary to do architecture exploration using the traditional processor design approach.

### 8. REFERENCES


