# Euro DesignCon 2004

# High Performance Network Protocol Processor – Architecture and Tools

Xiaoning Nie, Infineon Technologies AG
Advanced Systems and Circuits
xiaoning.nie@infineon.com

Jens Wagner, Informatik Centrum Dortmund e.V.
Joseph-von-Fraunhofer-Str. 20, 44227 Dortmund
wagner@icd.de

# 1 Abstract

Chips for networking and communications consist increasingly of heterogeneous IP building blocks that provide different trade-offs of flexibility and power-/area-efficiency. This paper addresses building of a 450 MHz protocol processor which is an application specific processor for network protocol processing. The protocol processor has an optimized instruction set and allows programmability of protocol processing functions at high performance and low cost. It is shown that for the considered applications the processor delivers low cycle counts and small code size at high clock rates and small core area. The paper first describes the application domain addressed. The design flow and main features of the instruction set architecture are then described. It is followed by a detailed description of the concept and results of building a complete tool chain.

### Authors Biography

**Xiaoning Nie** received the Diploma degree and the doctorial degree in Electrical Engineering from the University Erlangen-Nuernberg, Germany in 1987 and 1992, respectively. After spending 3 years work on the high-speed optical networks with Alcatel, he joined Infineon Technologies with focus on advanced algorithms and architectures for networking and multimedia SoCs. He has published more than 40 papers and holds over 10 patents in the field of architectures and algorithms for networking, digital signal processing and digital video processing.

**Jens Wagner** has been working in electronics industry for three years and joined the Embedded Systems Group at the University of Dortmund in 1999 as a Ph.D. student. His interests include low level code optimization techniques, retargetable compilers, and hardware/software co-design. He has been working in a number of industrial projects for the technology transfer company ICD as a Project Manager, where he designs customized compilers for application specific processors. Currently, he works in the subject of bit-level data-flow analysis. Wagner holds a B.S. in electrical engineering, a diploma degree in computing from Leipzig University of Applied Sciences / Bolton Institute (U.K.).

# 2 Network Processing Application

Both the number of users and the link bandwidth are steadily increasing in the converging communications and computer networks of today. This trend results in an increasing need for processing performance in all types of network entities (e.g. Routers, Switches, and Gateways) [1]. Furthermore, the usage of more complex protocols and services in the network, e.g. Quality of Service (QoS), Security (IPsec) and Virtual Private Networks (VPN), increases the demands on programmability and functionality of network processing hardware. Traditionally, the fast path of a network entity needs only to support high bandwidth while the slow path handles more complex functions and exception processing. Hence, traditionally the slow path was implemented using General Purpose Processor (GPP) while the fast path which process the most common types of tasks and packets, has been implemented using hardwired, fixed-function hardware. In the last few years the combination of increased requirements on both performance and functionality has led to the birth of a new type of System on Chip (SoC) denoted Network Processor (NP) [4][33]. NP was invented to better tradeoff the following parameters than GPP and ASIC can [35][2][3][26][40]:

- High throughput
- Flexibility through programmability
- Cost efficiency
- Low power

An NP should typically provide functions including [1]

- Packet header parsing
  It has to be able to process nested headers because it has either to deal with multiple protocol layers or to deal with encapsulations for security, mobility and VPN.
- Classification
  It needs to take several header fields for classification.
- Route lookup
- Packet header editing
- Packet fragmentation and reassembling
- Packet storage or enque
- Packet scheduling and deque

It is worth noting that there is increasing emphasis on network processors in the access networks (AN) where high performance, rich functionality and cost efficiency are crucial. In the access network (AN) the NP has typically to handle different types of layer 2 protocols such as Ethernet MAC, ATM, PPP etc. Examples of the AN nodes are those for DSL lines such as DSL access multiplexers (DSLAMs), those for wireless infrastructures such as RNC for the 3GPP access and the access points for WLAN hot spots [29][44][45].

# 3 Network Processor Overview

Normally, NPs are addressed as the solution to the problem that GPPs can hardly keep up with increasing demands on performance for network processing (figure 1). Cost efficient NPs have been built for the access networks [34][33]. The key to cost efficient solutions is to provide no more flexibility as actually required. By flexibility we mean the programmability of the processing engines in use.

It is shown in this paper that Application Specific Instruction-set Processor (ASIP) is the right solution to ensure that the total cost of silicon including code, data memory and peripherials is minimized while providing programmability. The protocol processing ASIP in a network processing silicon is also called network protocol processor (NPP). Sometimes it is also named as protocol engine, micro-engine or pico-engine. The Infineon PP32 is a 32-bit NPP which is now in the third generation. PP32 is a synthesizable core, small in size and can provide as many as 450 networking MIPS in a single CPU.

While the NPP provides an efficient instruction set, a tool chain, especially a compiler, is needed to utilize the more efficient instruction set. Before the NPP is described in more detail we first give an overview on existing network processor (NP) solutions.

## 3.1 NP solutions

In order to provide the performance and flexibility needed there are three main design strategies available:

- In order to increase the number of available clock cycles for each packet, the processor clock frequency must be increased. This can be done using pipelining and/or other micro architectural optimization techniques.
- By increasing the functionality of each instruction, the number of instructions needed to process one packet is reduced. This can be achieved using improved Instruction Set Architectures (ISA), data path acceleration, and/or co-processors.
- By adding more processors (i.e. use a multiprocessor architecture) the processing performance is increased. The multiprocessor approach scales very well and is the only way to reach extreme processing bandwidths.

All NPs use one or several of these approaches to handle the problem.

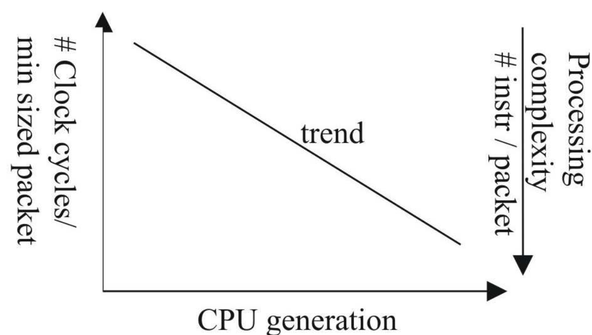



**Fig 1 Network processing to cope with fast growing bandwidth and complexity**

The diversity of architectural approaches in different available NPs is very big [3, 16]. In the following we give an overview on NP architectures according to the application domains [2].

## 3.2 Application domain specific NPs

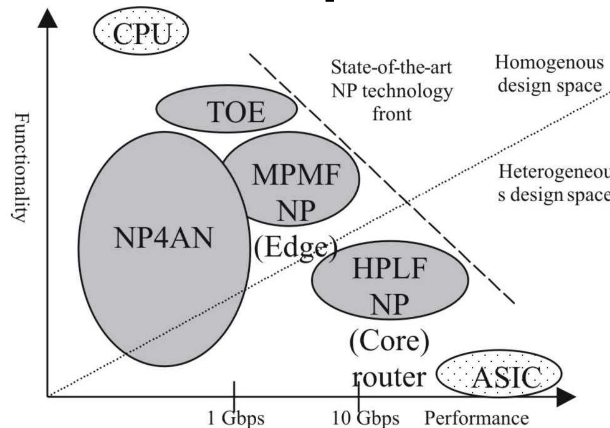



**Fig 2 NP architectures are application area domain specific. Homo-generous multiprocessor NPs provides higher flexibility than heterogeneous but on the other hand the later type of NP can reach higher performance. Hence, MPMF NPs should use a homogenous multiprocessor approach and HPLF NPs a heterogeneous. (MPMF – medium performance medium flexibility, HPLF – high performance low flexibility, NP4AN – NP for access network)**

The concept of NPs was invented to provide both high performance and high flexibility. Different types of network entities do however have different needs. E.g. a core router must support higher data rates than a gateway at the edge but on the other hand a core router needs only a limited amount of flexibility. By examining implementation methods for different types of NPs this paper will generalize and draw conclusions based on current trends.

## 3.3 Heterogeneous NPs

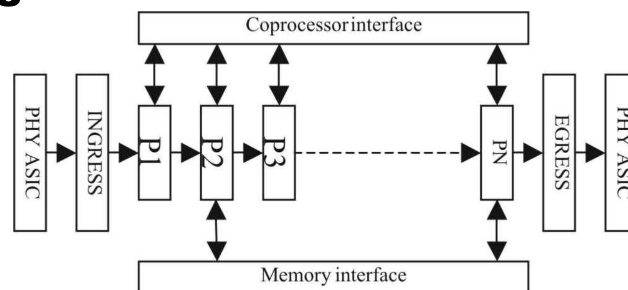



**Fig 3 Typical NPs dedicated for use in core routers benefits from a heterogeneous multi-processor partitioning. Most processors access coprocessors and some also access shared memory resources.**

Core routers have very high demands on processing bandwidth. This is normally supported in NPs by using a heterogeneous multiprocessor approach. Due to problem

with load-sharing, differences in processing latencies etc, heterogeneous NPs normally uses a pipelined architecture as illustrated by figure 3. In heterogeneous NPs each processor is dedicated for the specific task it performs and therefore provides high processing bandwidth and low overhead. Examples are NPs from Xelerated [27] Agere [36], and EZchip [37].

## 3.4 Homogeneous NPs

Edge routers have higher demands on flexibility compared to core routers. Hence, a homogenous multi-processor partitioning is normally used. The NP4AN belongs typically into this category of NP.

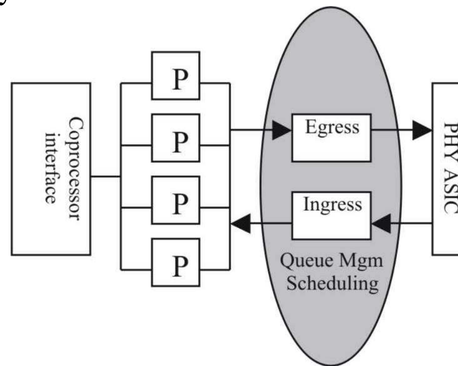



Fig 4 Homogenous multi-processor systems are best suited for NP4AN. Each processor may or may not consist of heterogeneous subsystem.

A homogeneous (non-pipelined) architecture generally offers more flexibility and is easier for programming. It supports load-sharing and thereby avoids some hotspots in processing resources. Examples are NPs from Intel[40], Motorola [38], Infineon [33] and IBM [39]. The Infineon network processor is consisting of a cluster of multiple PP32 in parallel with private memory for each PP32 and shared memory for common access.

In order to support higher bandwidths it is also possible to use a hierarchical multi-processor approach. This means that each processor element P*i* consists of several heterogeneous sub-processors. Normally each processing element possesses multiple threads. Each thread may handle one channel in this type of NP.

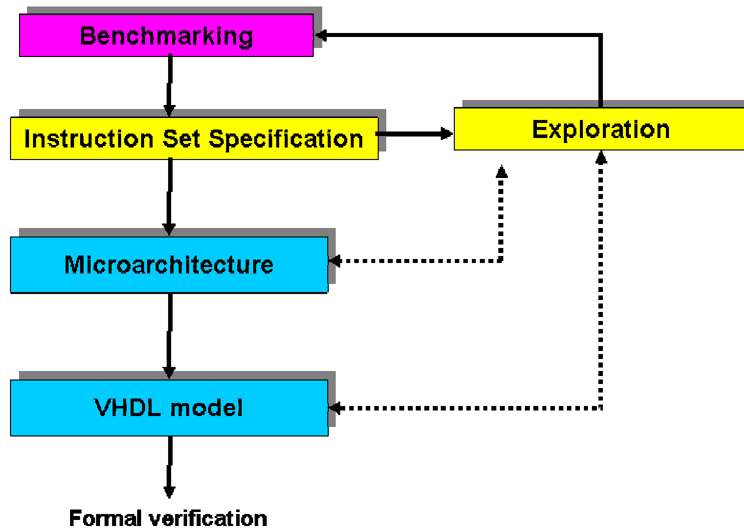
# 4 Designing for Efficiency

Considering NP for AN the cost efficiency is one of the primary issues in addition to performance. This class of applications are in particular characterized by requirements of moderate throughput of processing and high flexibility for multi-services. The design goal has typically to be defined as SoC with maximum area at a given functionality and flexibility

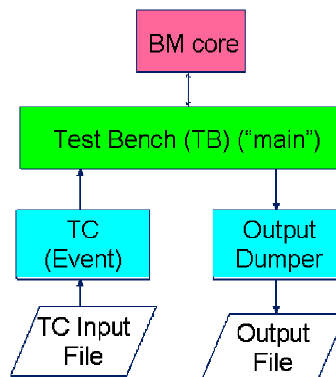
$$E_area = Throughput / Area$$

where throughput is defined as the number of processed minimum-sized packets per second and Area is defined as die size plus memory.

The following design flow is applied to designing the Infineon PP32. It is started with a close look at the application functionalities where the first step is to select a set of relevant benchmarks.




A new methodology has been developed for benchmarking embedded processor cores. There it has been taken into account both the core functionalities and real-time interaction of the processor with the peripherials. The basic structure of the benchmarking scheme is illustrated as follows:




The next steps are the design of instruction set and taking the benchmarks to measure the achievable performance. The first estimation of efficiency is carried out after setup of a micro architecture.

We made use of the Infineon Formal Verification tool for the processor core verification [49]. It is resulted in a high-quality design which is conformed by the emulations at the system levels.

The PP32 core has been synthesized for different applications and the recent design reaches 450 MHz in 0.13µm CMOS technology under worst case condition.

For the tool chain we have had experiences with both C++-based dedicated tool chain and description language-based tool chain generator. In this paper the C++-based dedicated tool chain is reported. A LISA-based tool generation [46] for PP32 will be reported in a separate paper.

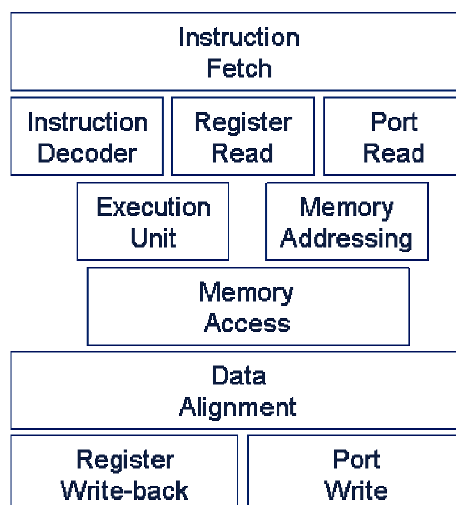
# 5 Protocol Processor Architecture

The PP32 is responsible to handle the fast data path of the protocols. Due to the nature of protocol processing the PP32 is required to efficiently manage

- Context switch between different tasks and/or interrupt service routines
- Bit field manipulations
- Conditional branches and
- Access to coprocessors such as CAM, CRC etc and peripherials.

## 5.1 The pipeline architecture

Basically PP32 has a 6 stage pipeline. It is a multithreaded architecture with hardware supported threads. The multithreading can effectively reduce the cost of context switches and minimize the effect of memory and coprocessor access latency. Its port read and port write reveal the efficiency of access hardware registers. The memory operations over the 3 pipeline stages memory addressing, memory access and data alignment allow additional improvement of the overall performance.




## 5.2 The instruction set

The PP32 is designed as an ASIP. It has a default instruction set and is subject to extension or tailoring. The default instruction set has 40 instructions including

- Data transfer instructions
- Arithmetic instructions
- Logical instructions
- Branch instructions
- Thread control instructions

PP32 ISA supports up to 4 hardware threads. Run and Stop instructions are provided for thread control. These instructions allow different types of thread transitions controlled by software. The thread switch can also be triggered by hardware signals (interrupts).

PP32 ISA provides rich functionality for bit manipulations. The bit manipulation in PP32 is much more efficient than those provided by other RISC processors because the PP32 bit manipulation is combined with all data transfer instructions. It is a unique feature of PP32 to provide a large set of bit manipulations within the load and store operations.

In a protocol processing SoC there are frequent accesses to hardware registers. The applications are I/O FIFOs or accelerator circuits such as a CAM. Unlike it is usually supported by a memory mapped address the hardware register access from PP32 is supported by the dedicated port instructions and dedicated address space for ports. The ports are not only included in the data transfer instructions with bit manipulation capability. But they are also included in the branch instructions which are especially efficient for polling operations.

# 6 Low-level toolchain

## 6.1 Overview

A domain specific system is designed to fulfil the requirements of a range of very specialized applications. Usually a small number of core benchmarks can be isolated from this range of applications to represent its typical behaviour. These benchmarks will show a number of special characteristics. Typical characteristics in protocol processing are:

- Fast I/O supported by parallel processors or special I/O co-processors ([4], [5], [6], [7])
- Hardware acceleration for transport and processing of bit-packets ([2], [8], [9], [10])
- Specialized hardware for CRC or checksum calculation ([6], [11])

The designed processor is expected to compute these benchmarks very efficiently. Possible aspects for optimization of the processor design are execution time, memory usage, die size or power consumption. When the first hardware model is designed, tool support is required to validate it.

Fig 5 shows a symbolic overview of our assembly level tool-chain ([12]). Because of the small size of the core benchmarks, compiler support is not needed during this first approach. The tool-chain consists of four different instruments: An assembler with integrated linker, a simulator, a debugger and a profiler.

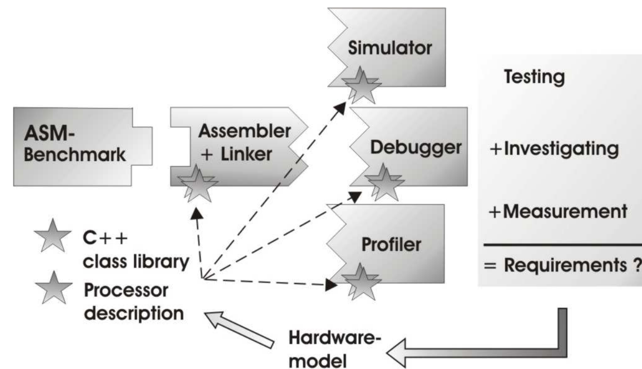



**Fig 5 Low-Level Benchmarking**

The common functionality of all tools is contained in a C++ class library. This library supports not only simple features like assembly file parsing and syntax checking, but also provides access to the hardware description. This library is also used to implement assembly code optimizations. Therefore, it contains data and control flow analysis based on assembly code as well. The C++ Class library is 100% hardware independent and used without changes in many of our projects. Even though the tool-chain requires programmer's interaction for being adapted to new hardware features, this customization can be done at very low costs.

A processor description covers the hardware description and the machine language specification. Changes are thus only necessary in a limited number of source files. After re-compilation, all components of the tool-chain will support the new hardware. The separation of the hardware dependent part of the tool-chain provides the flexibility of changing the hardware description at low costs. In the remaining chapter we will give a closer illustration of each of the components of the tool-chain.

## 6.2 C++ Class Library and Processor Description

The two common libraries are divided into a reusable part and a hardware dependent part, both of which are completely written in C++. Also, tools are used which provide C++ as output (like lex and yacc).

The hardware independent part consists of a Low-Level Intermediate Representation (LLIR) and a number of analysis tools that are mainly used in compilers and optimizers. The LLIR provides easy access to assembly source. It contains functions for file I/O, internal representation and a hierarchical view of the content of an assembly file. Access-functions allow the user to insert, read and delete instructions, parameters, labels, etc.

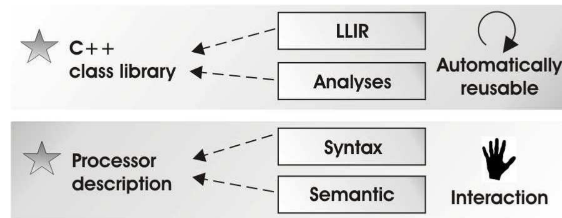



**Fig 6 Common components and their reusability**

The hardware dependent part is also described in C++. The advantage of using C++ is its flexibility. It is believed that C++ as functional language gives the freedom to describe every possible highly optimized pipeline architecture. On the other hand, C++ is a popular language and the code is executable on most platforms and it is relatively easy to integrate it into already produced tools. Fig 1 outlines the common components and their reusability.

The factor of reusability might be even slightly higher than the figure shows. Even between different designs there will be something in common that can be reused in the handwritten description. However, it cannot be reused automatically. Therefore, we still mark the description as a part that needs interaction by the programmer. A closer look at the reusability of the tool-chain is given in [12].

## 6.3 Assembler

A straightforward line assembler is used to transfer benchmark files into binary format. A simple linker extends its functionality. Because of the requirements of the remaining tool-chain, the assembler provides a special feature:
For the use within the low-level tool-chain, the assembler is able to output its result in as a C++ source code. The assembler output in that case will be compileable using a C++ compiler and can be linked against the semantic processor description for easy execution on a host machine. The resulting binary from the C++ output will act on the host machine in such a way that the behavior of the corresponding binary code on the target machine can be determined. The granularity of the used model is both cycle true and bit true.
The C++ output is used to easily create simulator, profiler and debugging tools.

## 6.4 Simulator and Profiler

A simulator enables one machine to execute code with the instruction set of another machine architecture. Most simulators work in an interpretative way. During the run time of the simulator, a data file containing the binary data to be interpreted is loaded and processed step by step. The advantage of this technique is that it can simulate self-modifying code. The disadvantage is that the simulation process itself is relatively slow. In our case the tool-chain uses the opposite technique called compiled simulation.
For compiled simulation, a simulator generator produces a binary that shows the same behavior on the host machine as a given program on the target machine (Fig 7).

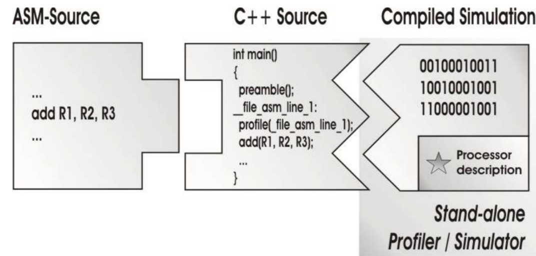



**Fig 7 Compiled Simulation**

The compiled simulation approach is relatively fast (more than one million lines of code per second), but because the assembly program is compiled into the simulator, the program code cannot be changed at run-time. The simulator works as a command line tool. At the granularity of the processor clock, the simulator's result is bit-true. It simulates the complete pipeline, interrupt and I/O behavior of the processor. Because of the exactness of the simulator model, it can be used to validate the VHDL hardware model of the processor.
By adding code for logging events during the runtime of the simulator, the compiled simulation can also be used as a profiler. The full range of profiling information is provided at this level, including number of executions per code line, accumulation of execution per machine instruction and cycles per code section.

## 6.5 Debugger

The compiled simulator is a binary that can be executed on the host machine. Therefore, it can be debugged using a host machine debugger. The approach here is to use the freely available DDD debugger [13]. Unfortunately, this naive approach would show the user the C++ source and the host machine assembly code in the debugger output. Thus, the binary to be debugged is generated by the assembler with modified debug information. Additionally, DDD's configuration script is adjusted to support the impression of debugging the target machine. Both the adjusted DDD and modified debug information in the binary enables the user of the debugger to test his code like on the target machine. Fig 8 shows a screen shot of the DDD-based debugger.

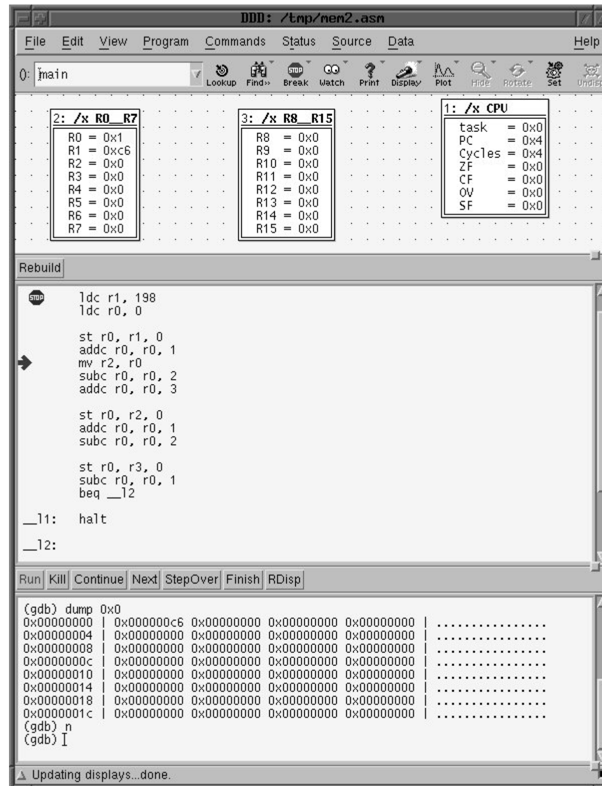



**Fig 8 DDD acts as virtual debugger**

# 7 Challenges of C-Compiler for Protocol Processor

## 7.1 Overview

The development of a compiler is a time consuming and expensive work. For example, the development of the first Fortran-compiler took 18 staff years work ([14]).
Using today's tools and knowledge, the development processes can be fastening up. One goal is the reusability of platform independent parts of a compiler in different projects.
Therefore, compilers are developed using concept, which provides a maximum of modularity.

A compiler is usually divided into front-end and backend. A common technique is the complete reusability of the compiler front-end and the appropriate optimizations. In the first step of the compilation process, a given source code is translated into an intermediate representation (IR). From now, all other components of the front-end are working on this intermediate representation. Virtually, even if the source language changes, most parts of the front-end can keep unchanged because of the constant IR ([15], pg.151).

Especially the expensive part of the front-end optimizations are hardware independent defined on the level of the IR ([16], pg. 165).

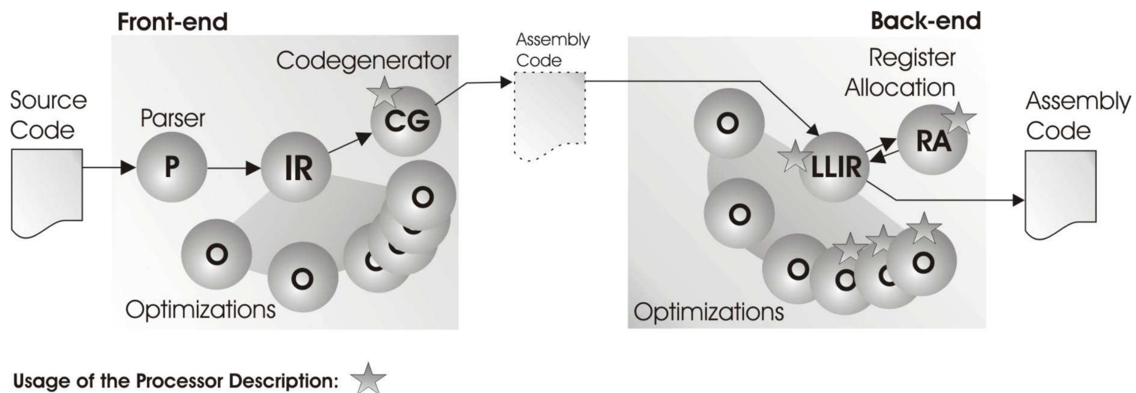



**Fig 9 Structure of the PP32 Compiler**

The back-end is known as the hardware dependent part, which is always changed if the compiler is targeted to a different hardware platform. Because of its poor reusability, the back-end is therefore known as the costly and time consuming part in the development process of a compiler. During the last decade, techniques have been developed to overwhelm the loss of retargetability of compiler back-ends. One costly part of a compiler is the code generator. The code generator is doing the translation from the front-end IR to a machine language. Code generator generators and tools supporting code generators can be found in e.g. [17], [18] and [16].

Code generation often creates code that uses virtual registers. That means simply, an infinite number of registers is allowed. On real hardware the number of registers is restricted. Therefore, a mapping of the virtual registers to the existing registers is needed. This will be done in the register allocation phase of the compiler.

After register allocation, the code is ready to be passed to an assembler and linker. The linker will output a binary, which is executable on the target machine.

## 7.2 Back-end architecture

The application of an interconnecting IR provides a massive enhancement on modularity and reusability. Why one should not use the same advantages on the back-end? Our measurements show that optimization potential after code generation is enormous because of the presence of platform specific operations and data-types.
Only a small set of these features is available in the front-end because of the generality of the used high-level language. Optimizations are able to use the power of the target specific behavior of a processor in the back-end, where all machine resources are available. Therefore, we applied the front-end structure to our back-end design. Fig 9 shows the overall structure of our compiler.

Like shown in the description of the low-level tool-chain the design of the compiler is based on an object-oriented framework. All hardware dependencies are covered in a processor description. The processor description reuses the processor description from the low-level tool-chain without change. Additionally, the processor description is extended by two components: The mapping from the source language to the assembly language used in the code generator and the description of the register file and the function frame layout, mainly used in the register allocation. The processor descriptions are done in C++.

If the back-end is considered as a separated structure, it can be seen as a component that reads assembly code, optimizes it and writes back assembly code. Therefore, the back-end can be used as an assembly code optimizer without the front-end. This provides the opportunity to use such a back-end either as a post-pass optimizer for a third party compiler or as a stand-alone tool used for hand optimized assembly code.

There are two different characteristics of back-end optimizations. The first ones are very close to the optimizations in the front-end. These optimizations can be described only using the toolset of the LLIR and the processor description without any processor specific attributes. Therefore, these optimizations are automatically retargetable.
As long as an adequate LLIR and processor description is provided these optimizations will work without change. The second variety of optimizations is using special features of the architecture. Therefore these optimizations will only work on architectures that provide these features.

## 7.3 Low-Level Intermediate-Representation

The LLIR provides the framework to create an optimizing back-end. Fig 10 gives a overview about the internal structure of the LLIR. The LLIR is build of three sets of components: I/O functions and a database to provide a hierarchical access to all components of the assembly program, a set of analyses: control-flow analysis (CFA), data-flow analysis (DFA) and bi-directional data- and value-flow analysis (BDVFA) and as third component the C++ interface to the optimizations.

A wide range of the LLIR is very closely connected to the processor description. For example the parsing of the machine language is depending in many parameters to the related hardware. But these hardware dependencies are still encapsulated by an underlying processor description. Parts, which using the processor description are marked with a star in the figure.

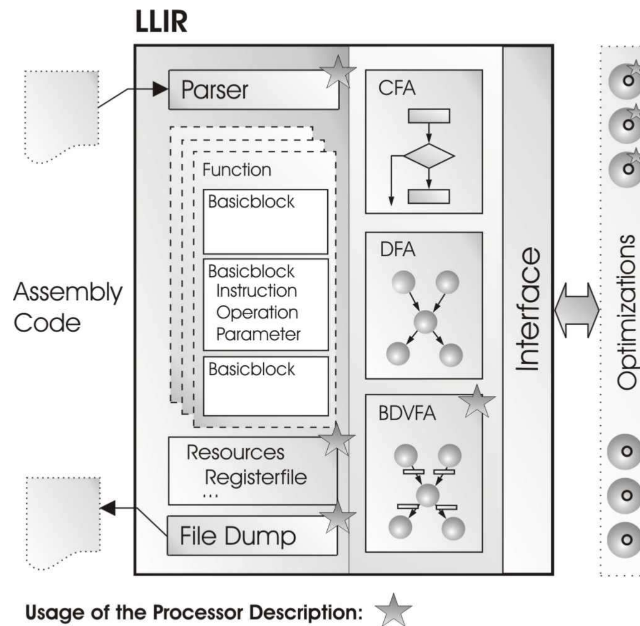



**Fig 10 Internal structure of the LLIR**

The parser and the file dump module are providing the functionality of reading and writing assembly code in different styles. There are the possibilities of having different kind of debug and profile information in the assembly file or assembly code using virtual registers. A special feature is pseudo assembly code that is compileable with a C++ compiler (see section 6.4).

The LLIR stores all in information from the assembly file in a hierarchical database.
If the LLIR notices a function structure in the assembly file it keeps this structure. Otherwise, the whole assembly file is seen as a single function. A function consists of a number of at least one basicblock. Each basicblock starts with a label and may contain a single branch at the end. In our model a basicblocks is a consecutive number of instructions.
Instructions contain a number of operations, which are executed in parallel. There is only a single operation per instruction on many architectures. An operation uses parameters as operands, where operands are either registers or constant numbers.

By using the above terms the processor description is able to define the arity of an operation or which operand is defined or used from an operation. This provides the flexibility to define the data-flow analysis (DFA) and the control-flow analysis (CFA) hardware independent. In the LLIR the DFA calculates where a used parameter is defined and vice versa. The CFA is used to examine the control flow, e.g. is a given instruction reachable from another through the control flow.

## 7.4 Bi-directional Data- and Value-Flow Analysis

The Bi-directional Data- and Value-Flow Analysis (BDVFA) is used to calculated a hardware independent simulation result of an assembly program. The results of this

simulation are used to optimize the simulated assembly program. Because of the independency from the hardware features the optimization that using these results may also be hardware independent.

Bi-directional analysis is frequently described in the last years in literature. Mainly it is used to detect narrow bitwidth operands to save register or data-path width in application specific processors ([20], [21]) or to save operand size to save power and improve the speed of an given program ([19]). BDVFA is also used to detect application specific operands like insert and extract instructions ([9], [10]).

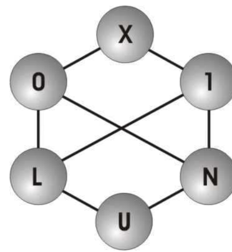



**Fig 11 Lattice of the semi-order used by the BDVFA**

BDVFA bases on multi value logic. The logic value represents the information the analysis knows about a single bit. It distinguishes between X: the value is irrelevant for the result of the program, 0 and 1: the value is constant, L the value is copied from a given location, N the value is copied from a given location and negated and U: the analysis does not know anything about the bit. The value of the bits is stored in the edges in a given data-flow graph (DFG), which represent the operands. A node of the DFG stands for an operation. Therefore, the nodes are simulated and the result is stored in the edges. We define special nodes in the graph. Root nodes are without outgoing edges and leaf nodes are without incoming edges. Fig 11 shows the semi-order if the multi value logic.

In the beginning all bits of all edges are set to U. First, the BDVFA traverses the graph in direction of the edges (top-down analysis) and calculates the new value of the bits. The top-down analysis starts from the leaf nodes. The new value is only stored in an edge if it is enhanced in the meaning of the semi-order. If the value of an edge is enhanced the succeeding nodes are also be simulated. After the top-down analysis reaches its fix-point, the bottom-up analysis starts from the root nodes. The bottom-up analysis calculates whether the value of a bit is dominating the following calculation or not. If the value is not necessary the bit is changed to value X. The BDVFA finishes if the bottom-up analysis reaches its fix-point. Fig 12 shows an example of the analysis of a AND-operation.

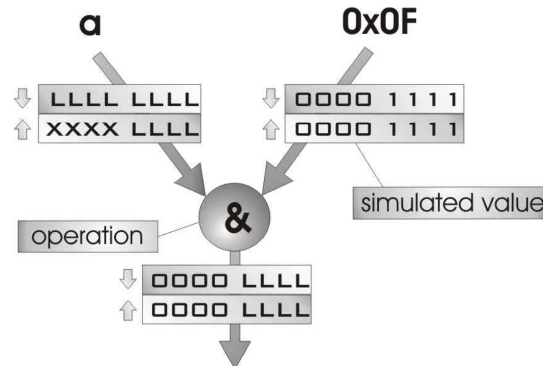



**Fig 12 Example of an AND-Operation in a BDVFA**

The variable a is a known location for the first operand of the AND-operation. Therefore all bits of the edge are set to the value of L in the top-down analysis (marked with down-arrow in the figure). The simulation of the operation gives four constant bits with a value of 0. Hence the bottom-up analysis (marked with up-arrow in the figure) will detect that the most significant four bits of the first operand are not needed to calculate the result. These four bits set to the value of X.

Because of the restriction that only improvements of the values of the edges are accepted, the analysis reaches its fix-point in polynomial time. The complexity is based on the following assumptions: The number of the bits per edge is related to the bit-width of corresponding memory |m|. The high of the lattice minus one |l|-1 is the maximum of improvements that can be made to a bit. Both |l-1| and |m| are constant for a given architecture. Therefore the maximum number of improvements per edge is constant and known: |m|*(|l|-1), which gives a complexity of O(|m|*(|l|-1)). The analyzed program can only change the complexity of the analysis by varying the number of edges in the DFG |E|. The number of node simulations is restricted to the number of edge updates and therefore also linear in time. After an edge is improved the target node of this edge has to be simulated and all outgoing edges have to be checked for improvements. In cyclic graphs this may that every edge is checked twice. Hence, the overall complexity of the analysis algorithm is O(2*|E|*|m|*(|l|-1)).

Optimizations will now try to detect defined patterns in the data- and value-flow graph to recognize optimization potentialities to improve the underlying assembly program.

## 7.5 Back-end optimizations

The problem of finding an optimal sequence of code for a given data-flow graph can be transformed into the problem graph pattern matching. For each machine operation a data-flow pattern can be developed. Constructing an isomorphic graph from these patterns can be used to find a machine program according to the data-flow graph. In general there are multiple possibilities to cover a given graph with patterns. According to a cost of the given machine operations model it is hard to find an optimal matching. Nevertheless, graph isomorphism itself is a known NP problem. Therefore, there will be no efficient solutions for this problem in general.

In the area of code selection many solution of graph-pattern matching are documented. Known practical fast solutions are using ILP-Models, Genetic Algorithms or on simplified graphs Dynamic Programming. These solutions combining the phases of code selection and optimization, which may open a wide optimization space. But, all these algorithms work on complete memory cells or registers.

Our solution explores this potential because of its single bit granularity. We using the very fast code selection of dynamic programming and after this we try to find optimization potential by BDVFA combined with searching characteristic pattern in the DFG. The complexity of the pattern recognition varies from O(n) to NP dependent on the pattern itself. Because of the smallness of the patterns the recognition time is in practical negligible to the time the BDVFA needs.

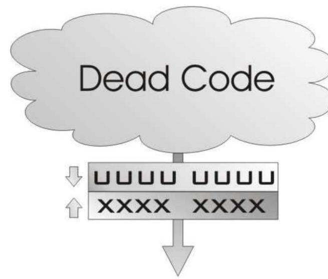



**Fig 13 Example of a dead code elimination**

A very simple pattern is used for dead code elimination. The pattern consists of zero nodes and a single edge. Therefore only every edge have to be tested once which gives a complexity of O(n). If the value of an edge is X in all its bits then the result represented by this edge is not used in the entire graph. Because of the fact that each X values of the different bits may be propagated from many different locations the optimization is much more powerful than an integer based dead code elimination. An academicals example is shown in Fig 13.

```
c = ... ;
a = c & 0xF0;
b = c & 0x0F;
d = a & b;
```

**Fig 14 Academically example of dead code, the calculation of c (marked with dots) is never be used**

Of course this example is unlike in a real world application, but in huge data-flow graphs containing e.g bit-packet or cast operations similar code is more likely. Especially if bit-packet proceeding applications are specified using ANSI-C data-types, the algorithm may detect dead code.

Because of the complex data-flow graphs it is hard to detect such dead code using other known techniques. There can be e.g. 30 nodes in between the definition of a value and its final uses. Because of the NP complexity of graph isomorphism, it exists no fast

algorithm in general. Our technique benefits from the propagation of the simulated values, which is a cheap technique even in huge graphs. This simulation is heuristically, but enables us to define patterns, which are detectable in polynomial time.

Most of the patterns are more complex than the patterns used in dead code elimination. The PP32 supports for example bit-packet insertion. To detect such an bit-packet it is only necessary to check all bits in every edge in O(|E|*|m|) time. After this the sub-graph G' can be exchanged by a single instruction, if the costs of the instructions of G is higher ( O(n) ). Therefore, this can be done in quadratic time. Fig 15 illustrate the optimization.

... = a & 0x0E + 0x71;

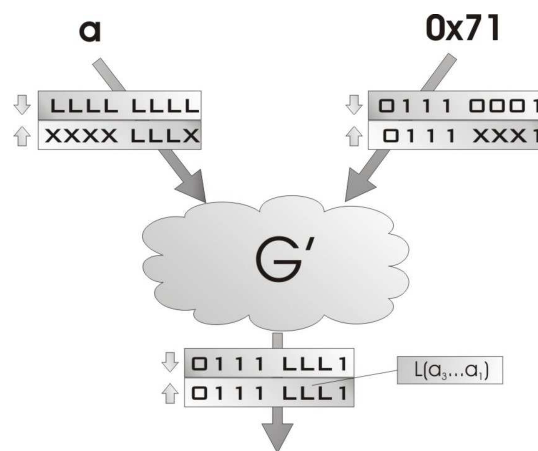



**Fig 15 Bit-packet pattern**

Because of most of the patterns are relatively efficient to cover, the overall time of the complete post-pass optimization differs from 5% to 400% of the run time of the rest of the compiler, dependent on the optimization potential of the benchmark and the optimizations to be used. Mainly, if the potential is high, it can be a reduction of 50% of the over all program code.

# 8 Results and Comments

Both the processor and the compiler evolve with an adaptation to each of the applications. The developed tool-chain has been used as an in-house tool at Infineon Technologies. Large test-suites and stress tests are used to prove correct and stable behavior of the tool-chain. Tests are co-checked by hand and using cross compilation with other platforms.

Smaller benchmarks are provided to show the performance of the processor and the compiler. In this result section, we will present some of the results of measurements during this process.

We took ARM processor as a known candidate of a general purpose processor which is also known to have its strength in C-compiler. A PP32 version with 38 instructions is taken for the comparison.
Result on a benchmark developed from the new benchmarking methodology for an embedded processor in real time applications is presented. It incorporates a typical function in a network processing application (port read, port write, push packet, pop packet).

In the following Figure of the total cycle number after 5 arriving packets and sending them. It is seen that a PP32 as a specialized processor has a clear advantage in performance. Recall that the area efficiency is

$$E_area = Throughput / Area$$

Or

$$E_area = Clock_rate / (Cycle_counts * Area)$$

Due to the small size and high clock rate of PP32 the area efficiency is even better.

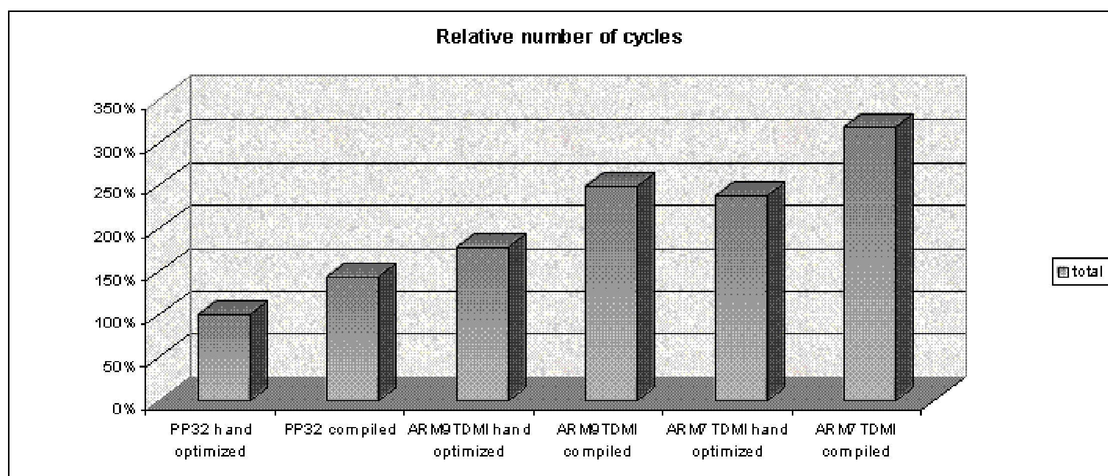



**Fig 16 Relative number of cycles compared to PP32 hand optimized code**

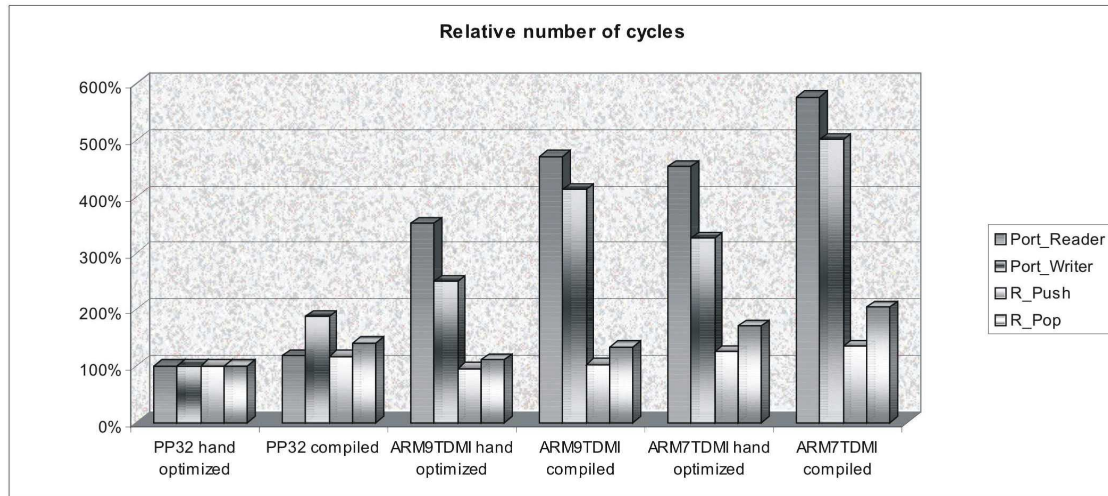



**Fig 17 Relative number of cycles compared to PP32 hand optimized code**

Fig 17 shows the speed in cycles compared to other architectures with respect to each component of the benchmark function.

The PP32 uses its powerful thread switching and interrupt mechanism and a number of special purpose operations. The PP32 compiler can take pace as long as its assumptions on the application data-flow structure are not to pessimistic. The hand optimized code uses parameter passing by registers, which are not supported by the compilers. Therefore there is a noticeable overhead to hand optimized code. Additionally the PP32 hand optimized code uses the predicated execution better than the PP32 compiler. The PP32 supports zero overhead function calls by using shadow register sets and bit-packet transport operations on ports which speeds up the code compared to the ARM. The ARM architecture performs better if it takes advantage of its more powerful predicated executions, which reduces the number of jumps and pipeline stalls.

Fig 18 shows the compactness of the resulting code. We count the number of 32 Bit instruction words used in program memory to implement the given benchmarks. The compiled code for the PP32 platform is set to 100%. All other numbers are relative to this code. To prove the quality of the compiler we show its relative performance to handwritten code. PP32 has better code compactness if multithreading and interrupt mechanism can be used. The ARM code is extremely efficient and compact if a block move of registers are needed.

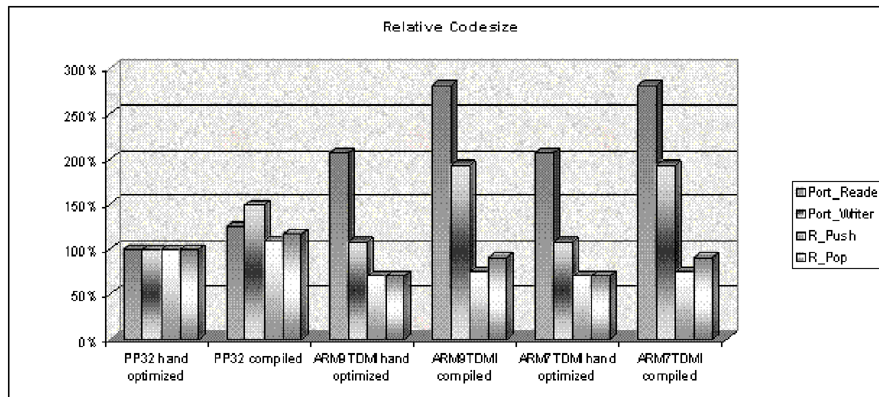



**Fig 18 Code size relative to PP32 hand optimized code**

The ARM architecture is able to save multiple registers on stack using a single operation, where the PP32 need to apply several operations. The ARM's more powerful predicated execution also contributes to the more compact results in Push and Pop Functions. The ARM provides also a more efficient stack access. Indeed if push and pop operations dominate in the application there is a need for more specialized instructions. The more application specific processors can also be supported by high-level languages and an adapted compilers.

# 9 Acknowledgement

The authors wish to thank Lorenzo Digregorio, Jinan Lin, Stefan Reif, Axel Koehnen for their contribution to the development of the PP32 core at Infineon Technologies. Jinan Lin, Michel Sarkis and Robert Pyka are appreciated for their contribution to benchmarks. The work of Jens Wagner was partially supported by the grant group 73 of the ministry of education of the state North Rhine-Westphalia. The PP32 compiler is developed by the Embedded System Group of the Informatik Centrum Dortmund (ICD).

# 10 References

[1] Special Issue on: "Internet Technology and Convergence of Communications Services", Proc. of the IEEE, Vol. 90, Nr. 9, September 2002

[2] X. Nie, U. Nordqvist, L. Gazsi, D. Liu: Network Processors for Access Network (NP4AN): Trends and Challenges, to be presented on IEEE Intl SoC Conference Sept 2004, Santa Clara.

[3] X. Nie, L. Gazsi, F. Engel, G. Fettweis: A New Network Processor Architecture for High-Speed Communications, IEEE Workshop on Signal Processing Systems (SiPS), 1999

[4] Pierre Paulin: Network Processors: A Perspective on Market Requirements, Processors Architectures, and Embedded S/W Tools, Design Automation & Test in Europe (DATE), 2001

[5] J. Freeman: An Industry Analyst's Perspective on Network Processors, in: P. Crowley, M. A. Franklin, H. Hadimioglu, P. Onufryk (eds.): Network Processors: Design Principles and Practice, Morgan Kaufmann Publishers, 2002

[6] U. Nordqvist: A Programmable Network Interface Accelerator}, Institute of Technology, Linköpings Universitet, Ph.D. Thesis, 2002

[7] T. Wolf, M. Franklin: CommBench - A Telecommunications Benchmark for Network Processors}, In Proc. of the 2001 International Conference on Architecture of Computing Systems (ARCS), Karlsruhe, Germany, 2002

[8] T. Wolf: Design of an Instruction Set for Modular Network Processors, 1000, IBM Research Report

[9] J. Wagner, R. Leupers: C Compiler Design for a Network Processor, IEEE Transactions on Computer Aided Designs of Integrated Circuits and Systems, Vol.~20, No.~11, 2001

[10] J. Wagner, R. Leupers: Compiler Backend Optimizations for Network Processors with Bit Packet Addressing}, in: P. Crowley, M. A. Franklin, H. Hadimioglu, P. Onufryk (eds.): Network Processors: Design Principles and Practice, Morgan Kaufmann Publishers, 2002

[11] G. Memik, W. H. Mangione-Smith: A Flexible Accelerator for Layer 7 Networking Applications}, Proc. of 39th IEEE/ACM Design Automation Conference (DAC), 2002

[12] J. Wagner, R. Leupers: A Fast Simulator and Debugger for a Network Processor, Embedded Systems Europe Conference, 2002

[13] A. Zeller, D. Lütkehaus: DDD - A Free Graphical Front-End for UNIX Debuggers, SIGPLAN Notices, volume 31, number 1, 1996

[14] Alfred V. Aho and Jeffrey Ullman and Ravi Sethi: Compilers: principles, techniques and tools, Addison--Wesley, 1986

[15] Andrew W. Apple and Maia Ginsburg: Modern Compiler Implementation in C}, Cambridge University Press, 1998

[16] Rainer Leupers: Code Optimization Techniques for Embedded Processors, Kluwer Academic Publishers, 2000b

[17] A.V. Aho, M. Ganapathi, S.W.K Tjiang: {\it Code Generation Using Tree Matching and Dynamic Programming}, ACM Trans.\ on Programming Languages and Systems 11, No.\ 4, 1989

[18] C.W. Fraser, D.R. Hanson, T.A. Proebsting: Engineering a Simple, Efficient Code Generator Generator, ACM Letters on Programming Languages and Systems, vol.\ 1, no.\ 3, 1992

[19] D. Brooks, M. Martonosi: Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance, HPCA-5, January 1999. Princeton University

[20] M. Budiu, M. Sakr, K. Walker, S. C. Goldstein: BitValue Inference: Detecting and Exploiting Narrow Bitwidth Computations}, European Conference on Parallel Processing 2000

[21] M. Stephenson, J. Babb, S. Amarasinghe: Bitwidth Analysis with Application to Silicon Compilation, Proceedings of the SIGPLAN '00, Conference on Program Language Design and Implementation, Vancouver, Canada, June 2000

[22] S.C. Goldstein, H. Schmidt, M. Moe, M. Budiu, S. Cadambi, R.R. Taylor, R. Laufer: PipeRench: A Coprocessor for Streaming multimedia Acceleration, ISCA, 1999

[23] A. Inoue, H. Tomiyama, H. Okuma, H. Kanbara, and H. Yasuura: Language and Compiler for Optimizing Datapath Widths of Embedded Systems, IEICE Trans. Fundamentals, vol. E81--A, no. 12, pp. 2595--2604, Dec. 1998

[24] G. Memik, W. H. Mangione-Smith: Evaluating Network Processors using NetBench, ACM Transactions on Embedded Computing Systems (TECS), 2003

[25] Y. Hoskote, V. Erraguntla, D. Finan, J. Howard, D.Klowden, et al.: A 10GHz TCP offload accelerator for 10Gbps Ethernet in dual-VT CMOS, IEEE International Solid-State Circuits Conference, 2003, Digest of Technical Paper, 14.7.

[26] B. Wheeler, L. Gwennap: A Guide to Network Processors, Fifth Edition http://www.linleygroup.com/Reports/npu_guide.html

[27] Xelerated, on the www, http://www.xelerated.com

[28] L. Kencl: Load Sharing for Multiprocessor Network Nodes, Ph.D thesis no 2725, EPFL, Swiss Federal Institute of Technology Lausanne, Jan. 2003.

[29] R. Neogi, K. Lee, K. Panesar, J. Zhou, "Design and performance of a network-processor-based intelligent DSLAM", Network, IEEE, Volume: 17 , Issue: 4, Jul. 2003, pp. 56 – 62

[30] X. Nie, “Einheit zur Verteilung und Verarbeitung von Datenpaketen “ , German patent DE 10059026 A1, 2000

[31] EEMBC, the Embedded Microprocessor Benchmark Consortium, http://www.eembc.hotdesk.com/

[32] NPF, Network Processing Forum, http://www.npforum.org/
[33] Linley Gwennap, The Linley Wire: Volume 4, Issue 7
[34] http://www.wintegra.com/

[35] N. Shah and K. Keutzer, “Network Processors, origin of species”, Proceedings of ISCIS XVII, 2002.

[36] http://www.agere.com

[37] http://www.ezchip.com

[38] Motorola: Product brief: C-5 Network Processor, http://e-www.motorola.com/files/netcomm/doc/fact_sheet/C5NP-PB.pdf

[39] J. R. Allen, et al.: IBM PowerNP network processor: Hardware, software, and applications, IBM Journal of Research and Development, Volume 47 Issue 2, 2003

[40] M. Venkatachalam, P. Chandra, R. Yavatkar, “A highly flexible, distributed multiprocessor architecture for network processing”, White-paper, http://www.intel.com/research/mrl/Library/Network_Processing_WhitePaper_121602.pdf

[41] S. Sheafor, C. Lindsay, "Vitesse Network Processors: Optimizing Architecture for Bandwidth and Flexibility", Hootchips 12, 2000, http://www.hotchips.org/archive

[42] W. Bux, et al, “Technologies and Building Blocks for Fast Packet Forwarding”, Communications Magazine, IEEE, Volume: 39, Issue: 1, Jan. 2001.

[43] T.Wolf and J.Turner, “Design Issues for High Performance Active Routers,” in Proc. 2000 Int'l Zurich Sem. Broadband Commun. Zurich, Switzerland, Feb. 15–17, 2000, pp. 199-205.

[44] J.-H. Park, “Wireless Internet access for mobile subscribers based on the GPRS/UMTS network”, IEEE Comm magazine, April 2002.

[45] C. Bouchat, “QoS in DSL access”, Volume: 41, Issue: 9, Year: Sept. 2003.

[46] A. Hoffmann, G. Braun, H. Meyr et al, “.A Methodology for the Design of Application Specific Instruction Set Processors (ASIPs) Using the Machine Description Language LISA”, Proceedings of the ICCAD, November 2001

[47] Dezsö Sima et al., Advanced computer architectures – a design space approach, Addison-Wesley 1997

[48] F. Cathor, et al, Custom memory management methodology – exploration of memory organization for embedded multimedia system design, Kluwer academic publishers, 1998

[49] R. Brinkmann, P. Johannsen, K. Winkelmann, Application of property checking and underlying techniques: Infineon’s Circuit Verification Environment in: R. Drechsler, Advanced Formal Verification, de Kluwer, 2003